Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

An Approach for Identifiying False Positive Warnings in SAST Tooling

Lukas Krawczyk

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Markus Haug, Dr. Ana Cristina Franco da Silva, Dr. Daniel Graziotin
Commenced:	February 1, 2022
Completed:	August 1, 2022

Abstract

In this paper, we present an approach to reduce the number of false positive results in static analysis of cryptographic libraries. To achieve this, we use an existing path-sensitive algorithm to eliminate RCE vulnerabilities and adapt it to recognize a group of vulnerabilities found in cryptographic libraries. We implement a prototype of our approach in Java using the Soot API for control flow graph and call graph generation. The prototype is then evaluated from two perspectives: accuracy and performance. We use a cryptographic benchmark to evaluate accuracy and a set of randomly chosen executable Java programs to evaluate performance. We summarize our results in a concluding chapter.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

CFG Control Flow Graph. 23

CG Call Graph. 24

ECB Electronic Code Book. 21

FP False Positive. 11

HTTP Hypertext Transfer Protocol. 21

HTTPS Hypertext Transfer Protocol Secure. 21

IV Initialization Vector. 21

JCA Java Cryptography Architecture. 20

MAC Message Authentication Code. 20

PBE Password-based Encryption. 21

PRNG Pseudo-random Number Generator. 21

RCE Remote Code Execution. 29

SAST Static Application Security Testing. 17

SSL Secure Sockets Layer. 21

TP True Positive. 19

1 Introduction

One of the backbones of the internet are web-applications. These become increasingly important as companies offer their services through the Internet. As convenient and powerful as they are, they present a significant security challenge to overcome. Through exposed vulnerabilities, malicious parties may gain access to company secrets and user data or gain access to critical functionality. To reduce this risk, one has numerous options. One of the more efficient ways to achieve this goal is to perform automated analysis during the implementation phase of the application's life cycle. The so-called Static Application Security Testing (SAST) tools try to achieve exactly that. However, to function as intended, these tools must be performant and have a low rate of falsely reported vulnerabilities, that is, vulnerabilities that are not really vulnerabilities.

In this paper, we will try to reduce the FP rate of existing SAST tools. We will focus our efforts on SAST tools that specialize in cryptographic libraries. We start with introducing and explaining the core concepts of this paper in Chapter 2 and mention related works and existing approaches in Chapter 3. We then outline our study design and explain our approach in Chapter 4. There, we will also provide a general view of the algorithms, tools, and libraries involved in our implementation. In Chapter 5 we will go into more depth about our prototype and talk about the various components of our program. The evaluation of our implementation will be the subject of Chapter 6. We will evaluate for performance and accuracy. Finally we will summarize our work and provide an outlook on future potential expansions in Chapter 7.

2 Background

This chapter is supposed to provide the necessary definitions and explanations of the technical terms and concepts used in this thesis. Additionally examples are provided to better illustrate these concepts and further their understanding.

2.1 Static Code Analysis

In order to find bugs or vulnerabilities early on in a programs life cycle, code analysis is one of the most efficient tools. To avoid spending human resources on this task, one uses automated code analysis tools. These programs are designed for the express purpose of analyzing other programs. Such an analysis can be done statically, i.e. the analysis tool uses only the source code, the byte code, or the binaries as input in the analysis, or dynamically, i.e. the code to be analyzed will be executed and the analysis runs parallel to that execution. Dynamic analysis has access to more information than static analysis (program stack, values of variables during runtime etc.) but is more resource heavy and complicated to perform.

Since about half of all vulnerabilities of a program are introduced in the initial implementation phase, employing analysis tools at this stage seems particularly fruitful [Bar+10]. We will now take a closer look at static analysis tools and cover some examples.

2.2 SAST Tools

SAST tools offer automated static analysis of the source code to expose vulnerabilities in the code. As they perform a static analysis, they do not need to compile and run the code in question, parallel to the analysis. Most SAST tools build an abstract representation of the source code provided and use it to search for the specific error patterns of vulnerabilities that they know of. They usually offer the user of the tool a way to edit these patterns or add new error patterns to the existing ones in order to cover new vulnerabilities or fine tune the analysis [Bar+10].

As convenient as this sounds, they need to satisfy two metrics in order to be efficient tools of code analysis. These are performance and accuracy.

Performance is relatively self-explanatory and signifies the amount of work done in a certain time span. Accuracy, on the other hand, needs a bit more explanation. Accuracy involves both maximizing the amount of True Positives (TPs) found and minimizing the amount of FPs. By TP we understand a genuine vulnerability contained in the source code, while a FP is a harmless code passage falsely reported as a vulnerability by the SAST tool.

In practice, most SAST tools need to find a compromise between performance and accuracy. For instance, a path-sensitive static analysis can distinguish between the different paths a program might take as a result of a conditional statement and might ignore infeasible paths. As a result, vulnerabilities that occur solely on such infeasible paths will not be reported, which will decrease the FP rate and increase accuracy. Path-sensitive analysis, however, is costly in terms of performance and, as such, rarely used in SAST.

Popular examples of SAST tools are *SonarQube* [SQ22] or *Coverity* [COV22]. These tools are more general tools in that they are not specialized to detect certain types of vulnerabilities or work with certain types of APIs. Throughout this thesis, we will focus on SAST tools developed to detect misuses of cryptographic libraries. To better understand these tools, we will dedicate the next section to explaining these types of library and their categories of misuses.

2.3 Cryptographic Libraries

Cryptographic APIs provide cryptographic algorithms and methods for usage in web applications. Examples of cryptographic functionality are key encryption algorithms, hash functions, Message Authentication Code (MAC) algorithms, key/password storage, secure communication etc.

Whenever cryptographic APIs are used in code, we are dealing with sensitive data and must therefore take special care not to leak or compromise that data. Any would-be attackers that are interested in this data, will certainly exploit any vulnerabilities in our cryptographic methods. Additionally, correct usage of cryptographic methods necessitates some sort of expert knowledge making it easy to mishandle said methods. It is for these reasons, why specialized analysis of source code that imports cryptographic functionality is so important.

Examples of SAST tools specialized in analysis of cryptographic libraries are *CogniCrypt* [KNR+17] and *Cryptoguard* [Fra20; RXA+19]. For this thesis we will use *CogniCrypt* to generate reports of detected vulnerabilities. We will use these reports as a basis and try to improve them by eliminating specific groups of FPs warnings reported by *CogniCrypt*.

For our implementation, we will work with methods and classes provided by the Java Cryptography Architecture (JCA), the standard cryptographic library used for Java programs.

2.4 Categorizing Crypto Misuses

A helpful tool for evaluating SAST tools focused on cryptographic APIs is Crypto-API-Benchmark [ARY19]. Crypto-API-Benchmark provides a set of classes covering a wide variety of Cryto-API use cases that test the accuracy of these tools. For this reason, every use case consists of a variety of different variants such as interprocedural or path-sensitive applications. Crypto-API-Benchmark also provides a taxonomy of cryptographic misuses that we will use as a reference in this thesis.

These categories, as listed in [ARY19], are as follows:

1. **Cryptographic keys:** Cases where predictable and insecure keys are used for encryption. Concerns the usage of the `javax.crypto.spec.SecretKeySpec` API.

2. **Passwords in Password-based Encryption (PBE):** Cases where predictable passwords are used in PBE. Concerns the usage of the `javax.crypto.spec.PBEKeySpec` API.
3. **Passwords in KeyStore:** Cases where predictable and constant passwords are used when storing keys or certificates using the `java.security.KeyStore` API.
4. **Hostname Verifier:** Cases where the verification of a hostname is omitted. Concerns the usage of the `javax.net.ssl.HostnameVerifier` API, specifically where the `verify()` method returns `true` without executing a verification routine.
5. **Certificate Validation:** Cases where connections to clients or servers are established without certificate validation. Concerns the usage of the `javax.net.ssl.X509TrustManager` interface.
6. **Secure Sockets Layer (SSL) Sockets:** Cases where hostname verification is omitted when using SSL Sockets from `javax.net.ssl.SSLSocket`.
7. **Hypertext Transfer Protocol (HTTP):** Cases where HTTP instead of Hypertext Transfer Protocol Secure (HTTPS) is used to retrieve a web page. Concerns the usage of `java.net.URL`.
8. **Pseudo-random Number Generator (PRNG):** Cases where predictable random numbers are generated by using the `java.util.Random` API instead of using the secure `java.security.SecureRandom` API.
9. **Seeds in PRNGs:** Cases where constant or static seeds are input to generate predictable random numbers with the `java.security.SecureRandom` API.
10. **Salts in PBE:** Cases where constant or static salts are used in PBE. Concerns the usage of the `javax.crypto.spec.PBEParameterSpec` API.
11. **Mode of Operation:** Cases where the insecure Electronic Code Book (ECB) mode is used to encrypt plaintext with the help of a blockcipher. Concerns usage of the `javax.crypto.Cipher` API. An example of this particular vulnerability type is given in Listing 2.1.
12. **Initialization Vector (IV):** Cases where a predictable constant IV is used in encryption and decryption. Concerns usage of the `crypto.spec.IvParameterSpec` API.
13. **Iteration Count in PBE:** Cases where insufficiently high iteration counts are used in PBE. Concerns the usage of the `javax.crypto.spec.PBEParameterSpec` API.
14. **Symmetric Ciphers:** Cases where insecure ciphers are used in symmetric encryption. Concerns usage of the `javax.crypto.Cipher` API. An example of the *Symmetric Ciphers* vulnerability is given in Listing 2.2.
15. **Asymmetric Ciphers:** Cases where insufficiently large key sizes are used in asymmetric encryption. Concerns usage of the `java.security.KeyPairGenerator` API.
16. **Cryptographic Hash Functions:** Cases where insecure hash functions are used in hashing. Concerns the usage of the `java.security.MessageDigest` API. An example of this vulnerability is given in Listing 2.3.

Listing 2.1 Example of the *Mode of Operation* vulnerability. The instance of the Cipher class is initialized with the insecure ECB mode (line 15).

```
01  package crypto.examples;
    ...
11  public class ModeOfOperationExample {
12    public static void main(String[] args) throws NoSuchAlgorithmException,
        NoSuchPaddingException,
        InvalidKeyException {
13          KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
14          SecretKey secretKey = keyGenerator.generateKey();
15          Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
16          cipher.init(Cipher.ENCRYPT_MODE, secretKey);
17    }
18  }
```

Listing 2.2 Example of the *Symmetric Ciphers* vulnerability. The instance of the Cipher class is initialized with the insecure cryptographic cipher *Blowfish* (line 15). Additionally, the KeyGenerator object is initialized with the same insecure cryptographic cipher (line 13).

```
01  package crypto.examples;
    ...
11  public class SymmetricCiphersExample {
12    public static void main(String[] args) throws NoSuchAlgorithmException,
        NoSuchPaddingException,
        InvalidKeyException {
13          KeyGenerator keyGenerator = KeyGenerator.getInstance("Blowfish");
14          SecretKey secretKey = keyGenerator.generateKey();
15          Cipher cipher = Cipher.getInstance("Blowfish");
16          cipher.init(Cipher.ENCRYPT_MODE, secretKey);
17    }
18  }
```

Listing 2.3 Example of the *Cryptographic Hash Functions* vulnerability. The instance of the MessageDigest class gets initialized with the insecure hash function SHA1 (line 9).

```
01  package crypto.examples;
    ...
06  public class CryptographicHashFunctionsExample {
07      public static void main (String[] args) throws NoSuchAlgorithmException{
08          String value = "hashThis";
09          MessageDigest messageDigest = MessageDigest.getInstance("SHA1");
10          messageDigest.update(value.getBytes());
11          System.out.println(messageDigest.digest());
12      }
13  }
```

Our approach aims to reduce FPs that relate to categories 11, 14 and 16. Specifically we aim to eliminate FPs that appear as a result of path-insensitive static analysis.

We will now further elaborate on the path sensitivity in static analysis by using an example of a FP vulnerability in a block of path-sensitive code.

2.5 Path-Sensitivity

Most applications these days heavily utilize conditional statements, like if-else or switch, and loop constructs, like while- or for-loops. These constructs make it so that an otherwise linear program execution can branch out into multiple possible paths. Taking also into account that conditionals and loops can be nested into other conditionals or loops, one can see how using these features can easily exponentially increase the number of possible paths, and, therefore, the complexity of a thorough code analysis.

To reduce complexity and improve performance SAST tools will rarely, if at all, try to figure out if every seemingly possible path is truly reachable given all possible values a program can have at that point in time. Instead, most SAST tools will treat all paths as feasible, that is, if a security risk is present on a path that would be impossible to reach during execution, the SAST tool in question would report it anyway. A path-sensitive analysis would recognize the unreachable path and omit any vulnerability present on that path. Therefore, path-sensitive analysis reduces the rate of FPs.

An example of a program, where a path-insensitive SAST tool like *CogniCrypt* would wrongly report a vulnerability on an unreachable path, is given in Listing 2.4. This code uses the `java.security.MessageDigest` API used for hashing. The code creates a `MessageDigest` object and assigns it the secure hash-function SHA-256 (line 12). The object will then receive a string in the form of a byte array (line 13) and then print the hashed string (line 18). The fact that the `MessageDigest` object is assigned the insecure hash function SHA1 beforehand (line 10) is irrelevant, since it is guaranteed that the code will always evaluate the if-statement to true (line 11) and enter the associated branch, wherein the previously assigned hash function is overwritten by the new assignment.

We will now look at two constructs crucial for the sort of path-sensitive analysis we want to implement in this thesis.

2.6 Control-Flow-Graph and Call-Graph

Control Flow Graphs (CFGs) are used to illustrate all possible paths through a program. They are essential constructs for any data-flow analysis, i.e. an analysis tracing the path of critical values throughout the code. A CFG is a directed graph whose nodes represent basic code blocks, that is, blocks of code that do not contain any jump commands. Edges signify jumps or paths from one basic block of code to another.

Listing 2.4 Example of a FP due to path-insensitive static analysis. Path-insensitive analysis won't be able to tell whether SHA1 or SHA-256 will be used as hash-function.

```
01  package crypto.examples;
    ...
06  public class PathSensitivityExample {
07      public static void main (String [] args) throws NoSuchAlgorithmException {
08          String message = "abcdef";
09          boolean alwaysTrue = true;
10          MessageDigest messageDigest = MessageDigest.getInstance("SHA1");
11          if(alwaysTrue)
12              messageDigest = MessageDigest.getInstance("SHA-256");
13          messageDigest.update(message.getBytes());
14          hashAndPrint(messageDigest);
15      }
16
17      public static void hashAndPrint(MessageDigest messageDigest) {
18        System.out.println(messageDigest.digest());
19      }
20  }
```

If a node has multiple outgoing edges, it means that the execution of the code will branch out at this point and multiple paths could possibly be traversed. This happens because of a conditional statement, i.e. a `if-else`-statement or a loop. Each outgoing edge represents one possible condition. A node in the CFG with multiple incoming edges, signifies the merging of multiple paths.

One CFG alone is usually not used to represent the control flow of the entire application, but rather the control flow throughout a single method. An example of a simplified CFG is shown in Figure 2.1. The CFG depicted here represents the main method in Listing 2.4. The line number in each node signifies the respective starting point of that node.

A data-flow analysis will also need to know how the methods represented by CFGs are connected. For this purpose, a Call Graph (CG) is used. A CG is a directed graph that represents the call hierarchy of a program. Its nodes represent methods. Incoming edges are specific calls made to that method and outgoing edges are method calls made within the method represented by that node. An example of a simplified CG is shown in Figure 2.2. As with the CFG example, this CG is constructed from the example code Listing 2.4. The main method for that code sample is the entry point of the graph.

Using these two constructs, an analysis that accounts for path-sensitivity can be performed. In Chapter 4 we will introduce an algorithm that uses these graphs to perform a path-sensitive analysis for finding taint vulnerabilities in PHP code. To understand this algorithm, we have to elaborate on the concept of taint vulnerabilities.

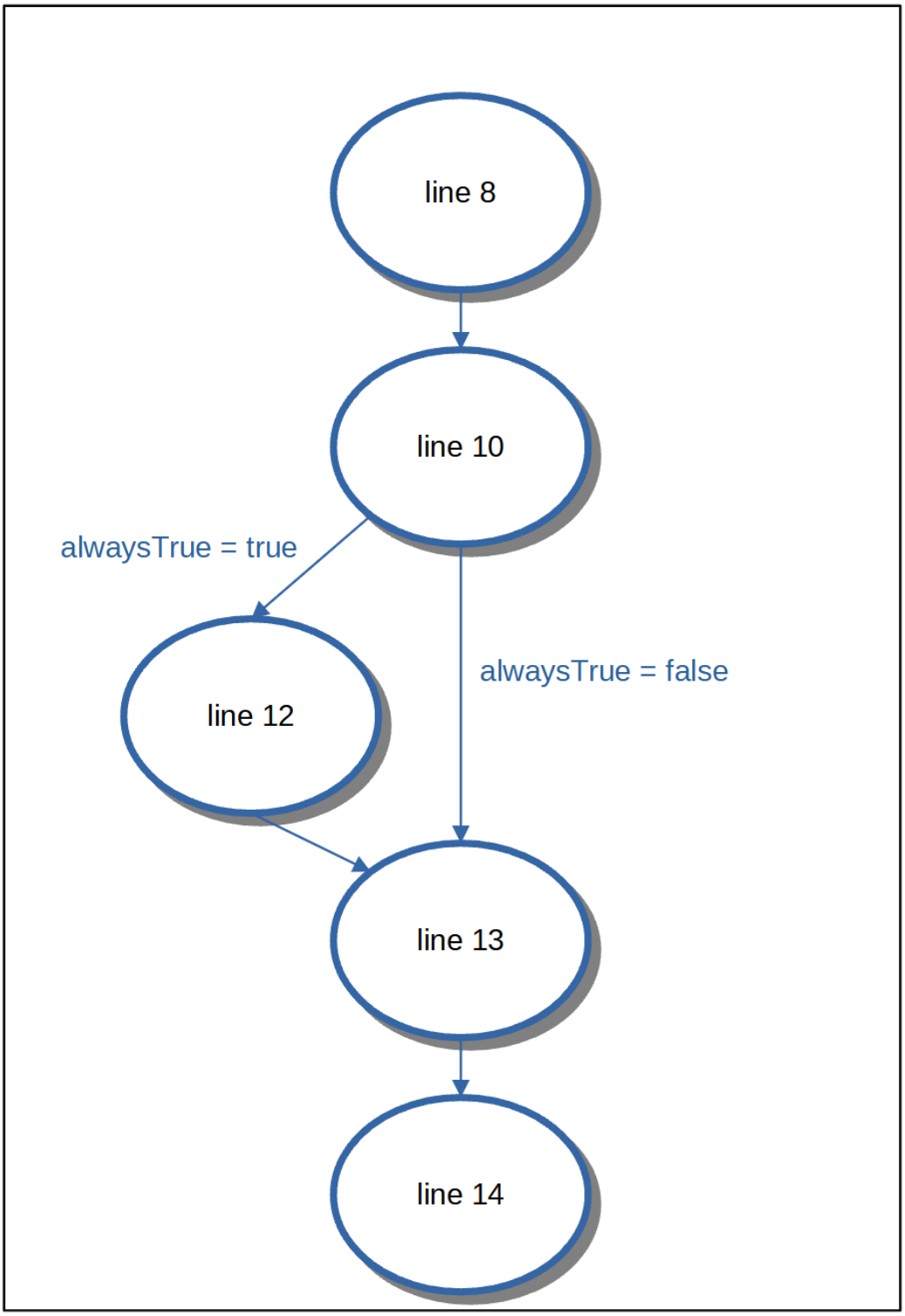

Figure 2.1: Example Control Flow Graph

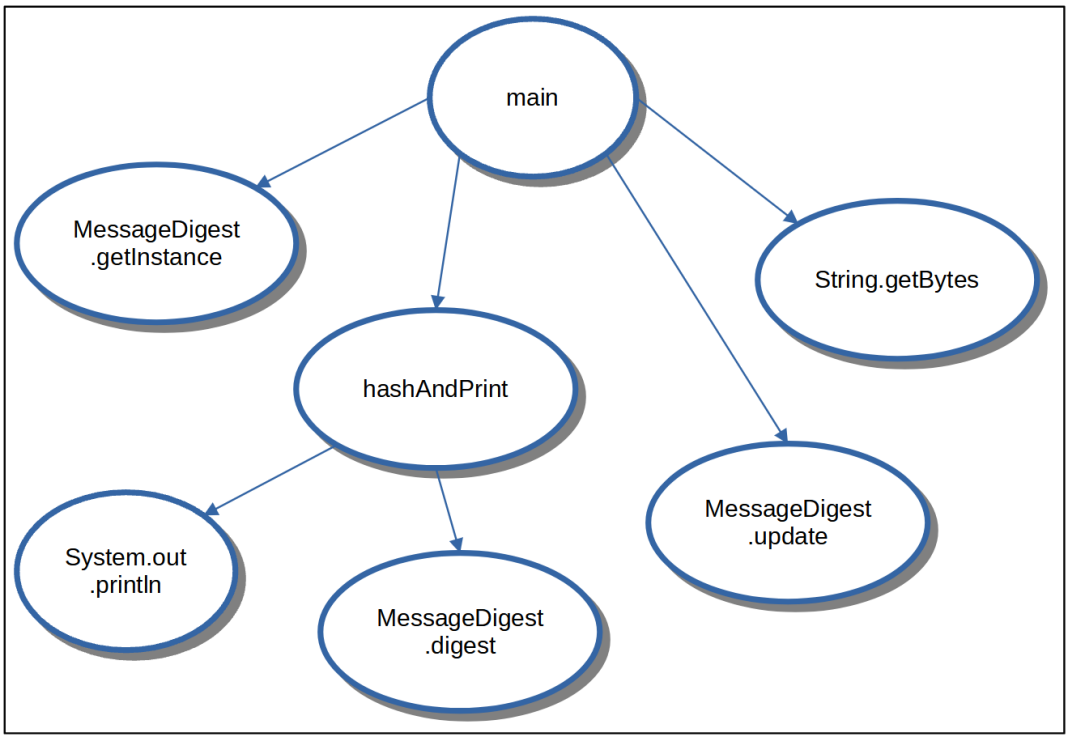

Figure 2.2: Example Call Graph

2.7 Taint Analysis

Taint vulnerabilities, are vulnerability types that presuppose that any variable containing a user-defined value poses a potential security risk. In other words, that variable is tainted unless sanitized, meaning checked for insecure values, or overwritten by a secure value. Additionally if a tainted variable or it's value is assigned to another variable, that second variable is now also tainted.

If a variable that can be tainted is present, a user can exploit it, by saving malicious executable code segments to the tainted variable. The parts of a program where a user can provide possibly insecure input are called sources. The then tainted variable will be assigned that input and propagate throughout the program until it reaches a sink. Sinks are functions or methods that enable the execution of the code contained within a tainted variable. A sink is therefore defined by a method and the tainted input parameter or parameters that are assigned the value of the tainted variable and pose the security risk.

So the presence of a tainted variable in itself is not enough to pose a real security risk. There also needs to be a feasible path from a source to a sink present in the code for it to be classified as a taint vulnerability. Analysis tools that want to detect such vulnerabilities, need therefore be path-sensitive.

Listing 2.5 Example of a taint vulnerability. The variable $name poses a security risk because it passes from the source (line 8) to the sink (line 5) without sanitization.

```
01 <?php
02 function checkuser($user){
03 if(strlen($user)<6) exit("too short");
04 $welcome="Welcome ".$user;
05 print($welcome);
06 }
07 $from=$_GET['from'];
08 $name=$_GET['name'];
09 if($from!='user') exit("error !");
10 checkuser($name);
11 ?>
```

An example of a taint vulnerability is shown in Listing 2.5 which is taken directly from [YWM17]. It is a short code segment that takes two arguments from the user and prints out a short message depending on these inputs. The sink in this example is the function `int print(string $arg)`. If `$arg` contains executable PHP code, the print function will execute that code. Here the variable `$name` in line 8 saves the user input, then gets appended to another string and passed unsanitized, to the print function in line 5.

3 Related Work

This chapter provides an overview of theses, research papers, books and tools related to some of the topics discussed in this thesis.

When looking at the topic of SAST tools in general, one finds an abundance of theses exploring a vast amount of different angles. For starters, many papers or books give an introduction to SAST tools in general, such as [CM04] or [CW07].
Another category of theses concerned with SAST tools, are conducted studies on the usage or behaviour of these tools. For example, [SJM+15] is interested in the type of questions software developers have, when it comes to the usage of SAST tools. [ANG+19] presents a study of the warnings created by SAST tools, in an attempt to show a connection between the types of warning and the rate of FPs produced.
A different path of research, is concerned with evaluating and benchmarking SAST tools. Examples with a focus on this type of topic are [PDM17], [Pas17] or [HLH+19].

When the field of SAST tools is narrowed down to static analysis tools specialized in detecting incorrect use of cryptographic libraries, the number of existing research becomes significantly smaller. [KSA+21] introduces CrySL, a specification language to define the secure usage of classes in cryptographic libraries. CrySL is used in [KNR+17] to implement CogniCrypt, a SAST tool specialized in detecting cryptographic misuses and the tool whose output we will analyze using our implementation. [RXA+19] and [Fra20] introduce CryptoGuard, which uses def-use analysis and forward and backward program slicing to detect cryptographic misuses.Another cryptographic SAST tool is Find Security Bugs [FSB] which is a plugin for SpotBugs [SB] which is the spiritual successor of FindBugs [FB] first introduced in [APH+08].

Another major topic we discuss is path-sensitive algorithms of static analysis. In [BSI+08] a sequence of path-insensitive runs through code, by an abstract interpreter is used to identify infeasible paths in the CFG. [ZZ13] uses path- and context-sensitive analysis to detect Remote Code Execution (RCE) attacks on web applications. Finally, [YWM17] uses a new path-sensitive static analysis method to find taint-style vulnerabilities in PHP programs. This method is implemented in a tool called POSE and evaluated for precision and performance.

We will use the algorithm presented in [YWM17] as a basis, adapt it to detect certain misuses of the JCA and implement and evaluate said algorithm.

4 Study Design and outline of our approach

This chapter focuses on defining the problems that we want to solve with this thesis and detailing the steps we took to get there. We also introduce an existing algorithm for detecting taint vulnerabilities in PHP code, which we will use as a basis for our approach. We go into detail how we will modify this algorithm to meet our goals. We also introduce the third-party tools that we will be using.

4.1 Study Design

Our goal for this thesis is to reduce FP warnings in reports from SAST tools that specialize in detecting misuses of cryptographic libraries. It is left entirely to ourselves how and from which angle we want to tackle this endeavour.

To achieve our goal, we did a library search with the following questions in mind:

1. What SAST tools are available specialized in detecting misuses of cryptographic libraries and how do they work?
2. What types of cryptographic misuses exist?
3. What are existing methods in static analysis, to detect cryptographic vulnerabilities and vulnerabilities in general?

We then determined that path insensitivity seems to be a major proponent of a high rate of FPs and did another library search on existing approaches to implement path-sensitive methods in static analysis. One of the algorithms we found is discussed in detail in his chapter. We have decided to use this algorithm for our purpose, altering it to detect FPs in static analysis reports of cryptographic misuses and implementing this approach as a working prototype.

A final library search for methods to evaluate SAST tools has helped us to decide how we will evaluate our prototype. We will use an existing benchmark for SAST tools specialized in cryptographic misuses to evaluate the precision of our prototype and look at the data gathered from the execution of the benchmark and 2 other randomly chosen open source executable java projects of the internet, to try to assess the performance of our prototype.

4.2 Path Searching Algorithm

A path-sensitive algorithm that can detect taint vulnerabilities in PHP code is introduced in [YWM17]. This algorithm requires the CG of the program and all the CFGs of any method on the path from a source to a potential sink. Furthermore, it requires a list of sink definitions, that is, a list of method signatures of methods that, depending on their parameters, pose a real security risk.

The algorithm iterates through all sinks within the program and works backward through the code until it reaches a source or the beginning of the program (which means it has not found a source on that particular execution path). Upon reaching the end, it will output the path information for that particular sink and path, which includes all code lines traversed in order from sink to source and the conditions picked up along the path. The algorithm is split into 4 distinct parts, path searching within a basic block, path searching between basic blocks, path searching across function calls and the final taint analysis.

4.2.1 Path Searching during a Basic Block

This part of the algorithm defines the behavior within a basic block and is shown in its entirety in Algorithm 4.1. This part also serves as the entry point for the entire algorithm with the $stack variable containing the context of a possible sink at the start. The context of a sink is a construct that contains the method and the tainted parameter, the block in the CFG and the line number that contains this sink and the path information that will be calculated by the algorithm. The variables $cfg and $cg contain the relevant CFG and CG.

The algorithm iterates through the $stack grabbing the top context (line 3), which would be the context of the sink we are investigating at the start. It extracts a lot of data from the stack, saving it in variables (lines 4-7). Since the algorithm is only interested in the code lines from the sink to the start of the block, it extracts those specific lines and reverses them so we can move backwards through the block (lines 8-13). Then the path information of the sink context is updated to the start of the block (lines 14-15). Finally, the algorithm checks whether it needs to enter the "Path searching between blocks or the "Path searching across function callsroutine, depending on whether the algorithm has reached the start of the CFG and needs to jump across functions using the CG or not, and enters the relevant routine (lines 17-19).

4.2.2 Path Searching between Blocks

This part of the algorithm comes into play when finding the predecessors of a basic block within the same CFG. It is depicted in it's entirety in Algorithm 4.2.

The input parameters for this part of the algorithm are the $stack variable, that contains the sink context so far and is used to save the updated context at the end of this routine, the current CFG and the sink context so far. The algorithm begins retrieving all predecessors of the current code block (lines 2, 3). Then iterates through these blocks, updates the path information, and constructs the updated sink context (lines 4-6). Finally, for each predecessor, the respective context gets saved to the stack, so the other parts of the algorithm can pick it up again (line 7).

4.2.3 Path Searching across Function Calls

This part of the algorithm is used when the current block does not have any predecessors in the associated CFG and the relevant CG is used to find a valid predecessor for the block. This part is depicted in Algorithm 4.3.

Algorithm 4.1 Path searching algorithm during a basic block from [YWM17]

```
01    function path_search($stack,$cfg, $cg){
02        while(!empty($stack)){
03            $context=pop_stack($stack);
04            $block=$context->block;
05            $line=$context->line;
06            $path=$context->path;
07            $codes=$block->codes;
08            $tmp_codes=array();
09            foreach($codes as $s){
10                if($s!=$line) $tmp_codes[]=$s;
11            }
12            if(!empty($tmp_codes)){
13                $tmp_codes=reverse($tmp_codes);
14                $path[]=$tmp_codes;
15                $context->path=$path;
16            }
17            $ret=path_search_blocks($stack, $cfg, $context);
18            if($ret==false){
19                $ret=path_search_call($stack, $cg, $cfg, $context);
20            }
21        }
22    }
```

Algorithm 4.2 Path searching algorithm between blocks from [YWM17]

```
01    function path_search_blocks($stack,$cfg,$context){
02        $cur_block=$context->block;
03        $new_blocks=get_front($cur_block,$cfg);
04        foreach($new_blocks as $front){
05            $path_con=get_path_condition($cur_block,$front);
06            $new_context=construct_context($front,$context,$p ath_con);
07            push_stack($stack,$new_context);
08        }
09    }
```

The input parameters for this part are almost identical to Algorithm 4.2, but additionally contain the CG used to travel across function calls. The algorithm begins by getting all callees of the current function (lines 2,3). It then iterates through each callee, retrieving all code including the line at which the current function gets called, and then creating the updated sink context for the new function (lines 5-8). The context then gets saved to the stack for the other parts of the algorithm to retrieve (line 9).

Algorithm 4.3 Path searching algorithm across function calls from [YWM17]

```
01    function path_search_call($stack,$cg,$cfg,$context){
02        $cur_func=$context->cur_func;
03        $callees=get_callee($cur_func,$cg);
04        foreach($callees as $callee){
05            $s=get_point($cur_func,$callee,$cg);
06            $line=get_line($s);
07            $front=get_block($s);
08            $new_context=create_context($context,$callee,$cur_func,$line,$front);
09            push_stack($stack,$new_context);
10        }
11    }
```

4.2.4 Final Taint Analysis

The final part of the algorithm, takes the sink context, after it has been through the previous 3 parts of the algorithm, and traces the tainted input parameters of the sink method to their origin, outputting a vulnerability if a source is reached. The algorithm is depicted in Algorithm 4.4.

The single input for this part of the algorithm is the updated sink context. The algorithm extracts the path information and the tainted input parameters of the sink (lines 2, 3). It then iterates through the path line by line and input parameter by input parameter (lines 4-6). The inner loop first checks whether or not the current line sanitizes the tainted input. If that is the case, it means that the current input parameter is not a cause of a taint vulnerability and can be ignored from this point forward (line 7). It then checks whether the current line is an assignment to the tainted input parameter of some kind. If that is the case, the traced variable for the input parameter needs to be updated for the next round of the outer loop, according to a specified rule set (lines 8, 9). Furthermore, if the traced variable is identified as a source, that is as user input, a vulnerability will be reported for this sink, with the current input parameter and source in the report (lines 10, 11). All traced input parameters that have not been sanitized or part of a vulnerability report are saved into the array that is being iterated through on the next round of the outer loop (lines 12 - 16). At the end, the context of the sink gets updated to contain only the tainted input parameters, that do not cause a real vulnerability in the code (line 18).

4.3 Changes and Additions to the Path Searching Algorithm

As mentioned before, we will use the algorithm introduced in [YWM17] and that we have discussed in the last section, as a basis for a path-sensitive static analysis, focused on identifying certain types of vulnerability related to the use of cryptographic libraries. To do so, we must modify and update the algorithm with additional functionality.

Since we will be using this approach to identify crypto-vulnerabilities of the previously established categories 11, 14 and 16, we will need to modify this algorithm to detect these kinds of vulnerability. Our strategy is to liken these crypto-vulnerabilities to taint vulnerabilities. Let us take a look at the *Cryptographic Hash Functions* vulnerability to better illustrate this.

Algorithm 4.4 The final taint analysis from [YWM17]

```
01 function taint_analysis($context){
02     $vars=$context->vars;
03     $cur_path=$context->path[];
04     foreach($cur_path as $s){
05         $new_vars=array();
06         foreach($vars as $var){
07             if(is_sanitation($s,$var)) continue;
08             else if($rule=select_rule($s,$var)){
09                 $new_var=update_var($var,$rule);
10                 if(is_source($new_var))
11                     report_vul($new_var,$s,$context);
12                 else put_array($new_var,$new_vars);
13             }
14             else put_array($var,$new_vars);
15         }
16         $vars=$new_vars;
17     }
18     $context->vars=$vars;
19 }
```

A taint vulnerability consists of a sink, with it's function definition and the tainted parameters, a source, a code line that maps user input to a variable, and a path from the source to the sink. The sink in the example of the a *Cryptographic Hash Functions* vulnerability would be the method `java.security.MessageDigest.getInstance(String algorithm)` and it's overloads, alongside the tainted parameter `String algorithm`. Because the methods in our vulnerability definitions do not have a problem with malicious user input, since these methods only accept very specific strings and do not run any code contained inside these strings, we define our sources as the entry points of the respective programs. So in order for us to accept a vulnerability there would need to be a path from the beginning of the program to a sink, as defined above, that doesn't filter out insecure parameters.

Another thing to keep in mind, is the fact that the problem with our vulnerabilities is that the associated sink methods return insecure objects. But these objects only pose a real risk if they are actually used after creation. If they are not used throughout the rest of the program or the variables used to refer to them are being overwritten before these objects can be used, then they can not be classified as real vulnerabilities. We will therefore need to check for those cases as well.

Additionally, we will need to define our own functionality for checking for sanitization of the tainted variables and for updating the variables. Both of these are used in the taint analysis part of the algorithm and since we will be aiming to detect different kinds of vulnerabilities and using a different language, we will need to update them to our needs. Since we will be using Java instead of PHP, we may generally need to make some minor changes.

An important programming structure that does not seem to be handled within the present algorithm are loops. When the algorithm, which tries to find a path from a source to the sink, through backward traversal of the CG and the CFGs, encounters a loop, it continues traveling the loop endlessly, since

it does not have any information about whether the break condition has been met at this point. In our approach, we will skip over loops during the path searching, while also acknowledging the uncertain state of all variables who are assigned new values within the loop, when evaluating the path.

Additionally, to loops, recursion is not handled either. We will need to address that as well.

Finally, we need CGs and CFGs of the relevant code sections and a list of potential sinks. We will use third-party tools to help us create these. For the generation of the CGs and CFGs we will use the Soot framework [VCG+10] and for help of computing the possible sinks in a piece of code we will use *CogniCrypt* [KNR+17].

4.4 CogniCrypt

CogniCrypt [KNR+17] is a plugin for the Eclipse IDE [ECL01]. It is a tool designed to help developers in the secure usage of cryptographic libraries. Its two main functions are code generation for the most common tasks a cryptographic API provides, i.e. encryption, secure communication, and secure storage, and the static analysis of code that makes use of cryptographic APIs. To detect vulnerabilities, *CogniCrypt* makes use of *CrySL* [KSA+21], a language for defining the secure usage of classes within cryptographic libraries. These definitions are then parsed by *CogniCrypt* and checked against when analyzing a given code sample. Like most SAST tools, *CogniCrypt* is path-insensitive.

We will use *CogniCrypt* as a SAST tool, which will generate a list of vulnerabilities for the code segments to be analyzed. We will use this list of vulnerabilities as input in our program and output a filtered version of that list, one that has been rid of the kinds of FPs we seek to find.

4.5 Soot

Soot [VCG+10] [SOOT] is a framework for optimizing and transforming Java code. It can be used as a standalone tool as well as a library whose functionality one can integrate into one's own code.

For the purpose of code optimization, *Soot* provides different types of analyses, like null pointer, array bounds or liveness analysis [EN08]. These check the code for parts that can potentially cause null pointer exceptions or array bound violations and provide information about live variables at each point in the code, respectively.

To help with the analysis, *Soot* makes use of four different intermediate byte code representations, all with their own set of advantages and disadvantages: *Jimple*, a typed 3-address, statement based representation, that heavily reduces the amount of possible instructions to 15 different types, *Shimple*, the Static Single Assignment-form that guarantees exactly one single assignment point for each local variable, *Grimp*, a code representation similar to *Jimple*, that strikes a balance between being easier to read than *Jimple* but better for analysis than regular Java code, and lastly *Baf*, a stack based code representation.

Soot divides the various steps of it's code optimization and transformation routine into packs. The name of each pack is an abbreviation that specifies the intermediate representation used and the operation performed. For instance the name of the pack *sop* stands for Shimple-Optimization-Pack.

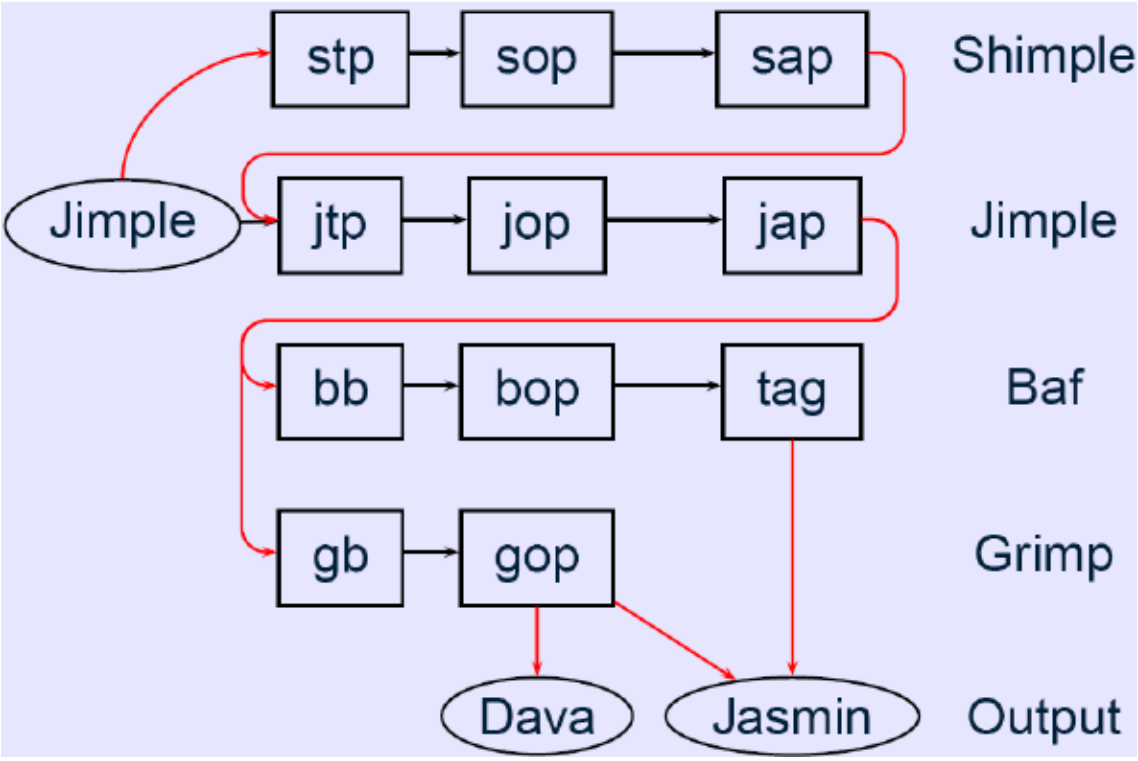

Figure 4.1: Execution flow of Soot from [HLL+03]

Additionally the packs *jtp* and *stp* perform user-defined transformation, meaning these are the points of the analysis that can be customized by the user. Figure 4.1 from [HLL+03] shows the execution flow of *Soot*.

For our purposes we will be using *Soot* as a tool to generate both the necessary CGs and CFGs. Additionally the CFG will use *Jimple* as the representation for it's basic blocks. The CFG we will use will also benefit from *Soot's* analysis, by already having dead code and never used variable values marked.

5 Implementation

This chapter goes into the details of the implementation, specifically with regard to the altercations made to the original algorithm. We start by giving an overview of the program and its components and their relationship. The rest of the chapter will be an in-depth look at each of these components, with code excerpts from the most relevant parts.

5.1 Overview

From here on out we will be referring to the implementation of our approach as the *Crypto-FP-Reducer*, wherein *Crypto* stands for *cryptographic library* and FP stands for *false positive*.

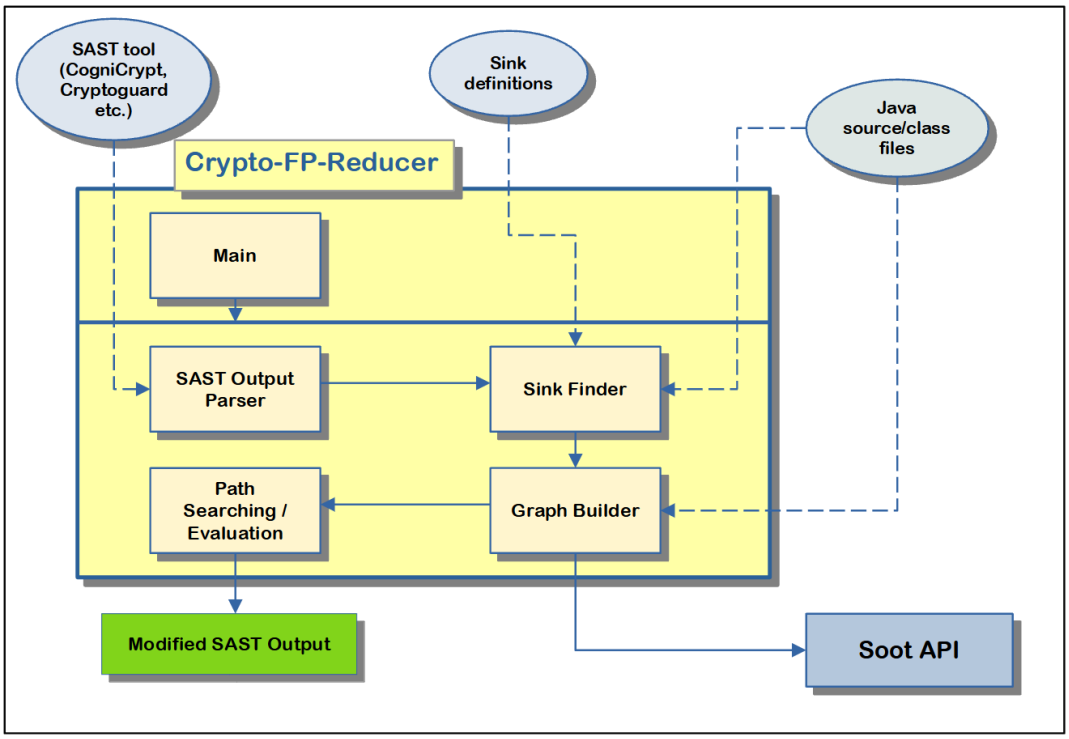

Figure 5.1: An overview of the components of the Crypto-FP-Reducer and their relationships.

Figure 5.1 shows an overview of the *Crypto-FP-Reducer*, it's various components and their relationship to each other. The application is split into two layers. The upper layer can be seen as the front-end of the application; it serves as an entry point, accepts all user input, and passes those inputs to the components of the lower layer. The lower layer is the back end of the application; it provides all of the logic and functionality for the front end.

The solid lines in the graphic represent the sequence flow of the program. The dotted lines represent the input that needs to be provided to specific components of the back-end. For the sake of readability, the fact that all these inputs are channeled through the *Main* component is omitted.

The execution is started by the *Main* component of the front-end layer. It reads all user input, calls all necessary functionality of the back-end in order, and outputs the results at the end of the execution.

The first component of the back-end whose functionality is needed is the *SAST Output Parser*. This component accepts the output of *CogniCrypt*, which has run an analysis of the program code that is being tested for vulnerabilities. This output provides a list of vulnerabilities found, some of which may be FPs. The *SAST Output Parser* parses this list and saves it for later use.

The next stage of the execution is the *Sink Finder*. This component accepts a txt-file, containing a list of sink definitions, and the path to the Java source and class files of the program code to be analyzed as input. It then searches this code for all possible sinks and saves them for further processing.

The next part of the program uses the functionality of the *Graph Builder* component to construct the CGs and the CFGs needed for the analysis. The *Graph Builder* makes use of the *Soot* library for this purpose.

Finally, the *Path Searching and Evaluation* component features a modified version of the path searching algorithm discussed in the last chapter. It analyzes the sinks found by the *Sink Finder* and computes which of those sinks don't pose an actual security risk by traversing the relevant CGs and CFGs and evaluating the resulting paths, therefore identifying the relevant FPs. It then cross checks the list of found FPs with the list of vulnerabilities found by *CogniCrypt* and parsed by the *SAST Output Parser* and outputs those vulnerabilities found by *CogniCrypt* that it can link to a FP.

5.2 Main

This component functions as a starting point for execution. It constitutes the entirety of the front-end of the *Crypto-FP-Reducer*. It parses all the inputs from the user and outputs the results of the algorithm. It has access to all the functionality the back-end exposes and constructs the sequence of routines that provide the desired results.

Since, algorithmically speaking, the interesting things are happening in the back-end, we will now focus on the various inputs that need to be provided by the user. The inputs are currently passed along to the *Main* component as command line arguments. They are in the order they need to be input: The absolute path to the source-files-directory of the code that needs to be analysed, the absolute path to the class-files-directory of said code, the path to a txt-file containing the output of a

Listing 5.1 Example of a correctly formatted line in the txt-input-file for the SAST Output Parser component.

```
Error description    SourceFileContainingVulnerability.java  my/custom/package/  line 21
```

CogniCrypt analysis of said code and the absolute path to a txt-file containing the sink definitions the user aims to focus on. These values are then passed on to the components of the back-end that require them.

The *Main* component will run the back-end components in the order established in the previous section. When this process is done and the *Crypto-FP-Reducer* has finished determining which of the provided vulnerabilities, detected by *CogniCrypt*, are FPs, it outputs a list of vulnerabilities from the original report by *CogniCrypt* that can be linked to a found FP.

5.3 SAST Output Parser

This component is responsible for parsing the output of the *CogniCrypt* analysis and saving all detected vulnerabilities in a list for future use. Currently the *SAST Output Parser* is set up to only detect the output of the SAST tool *CogniCrypt* or more specifically, a list of errors pasted from Eclipse, that have been detected by *CogniCrypt*. If the user wishes to rely on a different SAST tool as the basis for static analysis, the *SAST Output Parser* would have to be modified and an additional input parameter, that picks between *CogniCrypt* and other SAST tools, needs to be introduced. This modification should be easy enough to do and could be part of future updates to this application. Alternatively, the user can parse the output of his SAST tool of choice to the input format supported by our application, by using a third party tool. To make this possible, we will now specify the correct format needed for our application.

The detected vulnerabilities need to be available as a txt-file, and the absolute path to that file needs to be provided to our program. Each line of the txt-file represents a vulnerability found by the SAST tool and contains a set of properties that define that vulnerability. These properties are separated by tab stops. Our application only uses the first four of these properties. They are in order: A short text containing a description of the type of vulnerability, the name of the source file containing the vulnerability, the relative path from the root folder of the code's project to the source file mentioned before, and finally the keyword *line* followed by a space and the actual line number of the vulnerability. An example of a correctly formatted line is shown in Listing 5.1.

5.4 Sink Finder

The job of this component is to find all potential sinks within the code that is being analyzed. For this purpose the component is divided into two parts. First, it parses the sink definitions that are available via the input parameters, and second, it searches the code that is being analyzed for sinks that meet those definitions.

Listing 5.2 Example of a correctly formatted line in the txt-file containing the sink definitions.

```
java.security.MessageDigest:getInstance(java.lang.String):1:SHA-256,SHA-384,SHA-512
```

A txt-file, containing the definitions of all types of sinks that need to be searched for, needs to be provided to the *Sink Finder* via the file's absolute path. The file contains one sink definition per line. A sink definition consist of the method signature, containing the class the method belongs to and the method header surrounded by angled brackets, the index of the vulnerable method parameter (all the vulnerabilities that we focus on in our analysis, are tied to exactly one vulnerable input parameter) and a list of valid values for that parameter, separated by commas. These properties are separated by colons. An example of a correctly formatted line is shown in Listing 5.2.

After obtaining a list of all possible sink definitions, the second part of the *Sink Finder* searches all source files for sinks that match those definitions. In addition to the properties of the sink definitions that the found sinks inherit, the found sinks also contain the line number of the source file they are in, the name of the object, if there exists such an object, calling the sink method and the signature of the surrounding method that contains the sink.

The sinks found are saved and subsequently assigned to those CGs from which the surrounding method can be reached. Finally, the sinks will be used in the *Path Searching* component, where it will be determined which sinks correspond to FPs and which to TPs warnings.

5.5 Graph Builder

This component is responsible for creating all CGs and CFGs and mapping the found sinks to CGs. To create the necessary graphs, this component relies heavily on the *Soot* API.

Listing 5.3 shows the class responsible for generating the CGs. We consciously omit showing some of the code that is self-explanatory or not interesting, like comments or repetitions.

The *generateGraph* method (line 36) accepts the paths to the classes and the main class, the class containing the main method that will serve as the entry point to the CG, as inputs and outputs the CG. The call *G.reset()* (line 37) right at the start resets the state of the Soot variables. *Soot* saves some global settings across objects, so when we generate a new CG, we need to make sure to start fresh. Next, we need to tell Soot the paths to our code and the Java JRE that we are using (lines 39-44). We then set some options for the analysis Soot will execute. These are in order: Telling Soot to analyse all classes as a collective rather than separate from each other (line 49), telling *Soot* to analyze all classes on the path rather than just the main class (line 50) and telling *Soot* to save information about line numbers of the original source code (line 51). Before creating the graph we must tell *Soot* which class to use as the main class (lines 53-55). We use the *CHATransformer* to generate the CG (line 57) and finally return the result (line 61).

Listing 5.4 shows the class responsible for generating the CFGs. This class is fairly similar to the class generating the CG, with a few notable exceptions. As with the code for generating CGs we omit showing some of the code for the reasons mentioned above. As with the class for creating the

Listing 5.3 The class responsible for generating the call graphs.

```
01  package graph_builder;
    ...
26  public class CallGraphGenerator{
    ...
36    public static CallGraph generateGraph(String classPath, String mainClass){
37        G.reset();
38
39        String javaPath = System.getProperty("java.class.path");
40        javaPath = classPath.substring(0, classPath.indexOf("\\bin\\") + 4)
          + javaPath.substring(javaPath.indexOf(";"));
41        String jrePath = System.getProperty("java.home")+"/lib/rt.jar";
42        String finalPath = javaPath+File.pathSeparator+jrePath+File.pathSeparator+classPath;
43
44        Scene.v().setSootClassPath(finalPath);
          ...
49        Options.v().set_whole_program(true);
50        Options.v().set_app(true);
51        Options.v().set_keep_line_number(true);
52
53        SootClass sootMainClass = Scene.v().loadClassAndSupport(mainClass);
54        Scene.v().setMainClass(sootMainClass);
55        Scene.v().loadNecessaryClasses();
56
57        CHATransformer.v().transform();
          ...
61        return Scene.v().getCallGraph();
62    }
    ...
93  }
```

CG, the *generateGraph* method is responsible for creating the graph. The CFG we are building is a *Soot* class by the name of *ExceptionalUnitGraph* (line 52). Contrary to the builder for the CG we need to define our own transformation step (lines 61, 62). We want the analyzed code to be in the *Jimple* format when we create our graph, as this format will be easier to process by the *Path Searching and Evaluation* component. Since at the end of a standard *Soot* analysis, the code will be in the *Baf* format, we extract our CFG during the analysis. That is why we need our custom transformation step (lines 111-116). Next we set our analysis options (lines 67-69). The option in line 69 lets us keep our original variable names throughout the analysis. Finally, we run the actual *Soot* analysis by running the packs (line 75) and return the CFG created during our custom phase (line 77).

Additionally to the classes for generating the graphs, this component also consists of a wrapper class for the CG. The wrapper contains the generated CG and a list of sinks associated with that graph. It therefore also features a method that maps sinks to CGs. This method iterates through all edges of the CG and checks whether the end of an edge points to any sink within the list of potential sinks found by the *Sink Finder*. All sinks that are identified as a target of a CG edge will be added to that CG's list of associated sinks.

Listing 5.4 The class responsible for generating the control flow graphs.

```
001 package graph_builder;
    ...
024 public class ControlFlowGraphGenerator extends BodyTransformer{
025   private String pathToClasses;
026   private String pathToSingleClass;
027   private String mainClass;
028   private String methodName;
029   private static ExceptionalUnitGraph cfg;
      ...
052   public ExceptionalUnitGraph generateGraph(){
          ...
061       ControlFlowGraphGenerator analysis = new ControlFlowGraphGenerator(this.
          pathToClasses, this.pathToSingleClass, this.mainClass, this.methodName);
062       PackManager.v().getPack("jtp").add(new Transform("jtp.ControlFlowGraphGenerator",
          analysis));
          ...
067       Options.v().set_app(true);
068       Options.v().set_keep_line_number(true);
069       Options.v().setPhaseOption("jb", "use-original-names:true");
          ...
075       PackManager.v().runPacks();
076
077       return ControlFlowGraphGenerator.cfg;
078   }
      ...
111   @Override
112   protected void internalTransform(Body b, String phaseName,
113   Map<String, String> options){
114       if(b.getMethod().getSignature().equals(this.methodName))
115           ControlFlowGraphGenerator.cfg = new ExceptionalUnitGraph(b);
116   }
117 }
```

5.6 Path Searching and Evaluation

The *Path Searching and Evaluation* component consists, as the name suggests, of two main parts: The Path Searching algorithm, which finds all feasible paths from the start of a program to a sink, by doing a backward search through the relevant CGs and CFGs, and the evaluation of these paths that iterates through the found paths in order, propagating variable values and evaluating conditional statements.

5.6.1 Path Searching

The path searching mainly consists of the 3 methods *pathSearch*, *pathSearchBlocks* and *pathSearchCall*, which are based on the first 3 routines of the path searching algorithm discussed in Chapter 4.

Listing 5.5 The signature of the method responsible for path searching inside a block.

```
01  public static void pathSearch(List<Context> contexts, Context context,
    ExceptionalUnitGraph cfg, CallGraph cg, String pathToClasses, String
    pathToSingleClass, String mainClass) {
        ...
31  }
```

Recall that our goal in this section is to find all feasible paths from a sink to the start of the program. We use a list to save the feasible paths found. The *pathSearch* method is our entry point into our routine, and we travel through the 3 methods above recursively. First, we travel through a single node of a CFG, then we call the *pathSearchBlocks* method that uses the *pathSearch* method on all predecessors of the current node. In this way, we search backward through the entire CFG. Once we arrive at the start of the CFG, we call the *pathSearchCall* method, that provides us all possible CFGs, that are connected to our current CFG via method call, and the particular nodes inside those CFGs from which that method call originates. We iterate through those nodes and call the *pathSearch* method, starting the cycle anew. This routine is executed until we reach the start of the program. Then we add the path to our list of feasible paths.

Listing 5.5 shows the signature of the *pathSearch* method. The input parameters are the list of contexts of type *Context*, which is used to save the results of the path searching routine, the current context of type *Context* which is a n object that contains and is used to save the information about the current path, the current CFG, the associated CG, the paths to all relevant classes, the current class and the main class of the CG. The path information is necessary when creating a new CFG when tracing a method call back to it's caller during the method *pathSearchCall*.

The *pathSearch* method starts by saving all path information of the current node to the current context. The method then decides whether to call the *pathSearchBlocks* method, the *pathSearchCall* method, save the current context to the list of contexts and finish or just finish without saving the current path information. This decision is based on whether there are any predecessor nodes in the current CFG and CG, and whether or not we have reached the start of the program or the path has reached a dead end making it infeasible.

The *pathSearchBlocks* method has the same input parameters as the *pathSearch* method. This method iterates through all predecessor nodes of the current CFG node and calls the *pathSearch* method for each valid predecessor. Additionally, this part of the routine also handles loops. We detect whether or not a predecessor of the current node represents the end of a loop. If that is the case, we travel backward through the loop once, save the path information of the loop to our context, and then skip the loop, or else we face infinite recursion through the *pathSearch* and *pathSearchBlocks* methods. The reason why we save the loop information to our context is so that we can mark the values of all variables used inside the loop as uncertain when we evaluate the context. Since we don't know how many times the loop will be executed, the state of the variables that get values assigned to them within the loop, is also uncertain.

As with the previous method, the *pathSearchCall* method has basically the same input parameters as the *pathSearch* method. The method starts by iterating through all possible callers of the current method, represented by it's CFG. We first compare the signature of the caller to the current method's signature and skip the caller if the signatures are identical, because this means that we are dealing

with a recursive call. The next step is to construct the new CFG for each caller. This involves adjusting the path information in our input parameters. After the creation of the new CFG we also have to find the caller node within that graph. This involves iterating through all CFG nodes and checking if it contains the method being called and the line number of the call matches the information we have. When we finally have obtained the correct node, we call the *pathSearch* method and continue our search with the new node and the new CFG.

5.6.2 Evaluation

After obtaining all possible paths for a particular sink, we need to evaluate these, to determine whether or not any of these paths is actually feasible given the values of the conditionals encountered on the path and whether or not the sink argument can possibly carry an insecure value.

The method responsible for the evaluation is *isValidButInsecurePath* and its signature is given in Listing 5.6. The inputs are the path that needs to be evaluated and the sink to which that path belongs.

Evaluation of the path occurs from the source to the sink, as opposed to the path searching, which finds paths through backward traversal from the sink to the source.This way, we can trace the value of each variable in execution order. We keep two lists throughout the evaluation: one that contains all known variables and the other that contains all their values. Unknown values are marked by a '?'. In the following, we will list all the checks that occur during an evaluation for each line of the path.

The evaluation first checks if the line consists of an assignment. If that is the case and the variable that has a new value assigned to it is part of our variables list, we update the value of that variable. If the variable is newly introduced, we add it to our variables list and the assigned value is added to the values list.

The evaluation also checks if the new line signifies a jump to a different method. If that is the case, we have to save the input values of the new method and assign these values to the new local variables used within that method that represent the method's parameters.

Another thing we do is check whether the current line is the start of a loop. If that is the case, we fast forward to the end of the loop. New variables introduced within that loop will not get stored, and variables that are already known to the evaluation and get assigned a new value within the loop, are marked with an '?' since we cannot be sure whether or not that assignment takes place during an actual execution.

We also evaluate conditional statements, since *Jimple* does a good job of reducing the syntactical complexity of conditionals. First, we check whether the values and variables involved in the conditional are literals, numbers, or booleans. Then we decide whether the conditional operator is a '==', '!=', '>=' etc. Using the lists of variables and values, we then try to evaluate the conditional. If we are unsure about the values involved in the conditional, we skip the conditional and continue evaluation of the path. If the conditional evaluates to true or false, we check the jump target of the conditional. If the conditional is true and the jump target is equal to the next line on our path, or if the conditional is false and the jump target does not equal the next line on our path, we continue the evaluation of the path. In all other cases, we abort the evaluation.

Listing 5.6 The signature of the method responsible for evaluating the feasible paths.

```
195 public static boolean isValidButInsecurePath(List<String> path, Sink sink) {
    ...
465 }
```

If we reach the end of the evaluation, i.e. the line containing the sink method, we check if the values the sink parameter can attain are permissible. If all values in the possible value range are valid or secure, the sink itself is secure and this particular path is secure. In cases where the evaluation is aborted preemptively, the path is also considered secure, since it doesn't actually reach the sink. Only in the case where there is a real possibility that the sink parameter has an insecure value, the evaluation method returns true.

Finally, we also check whether any conditionals are used to sanitize the sink input parameter. We execute these checks in a separate run-through at the end of our evaluation, when we have information about the name of the variable used as sink input and its aliases, that is variables that share the same value and who can be used to sanitize the original variable. We go through the entire code again, checking whether any conditionals, involving the mentioned variable or one of its aliases, sanitize the sink input, by making sure it contains only valid values. If the sink input has been sanitized we output the boolean value false.

In order for a sink to be considered a FP, all paths of that sink, that have been found during the path searching phase, must evaluate to false. If only one path evaluates to true, we have a TP on our hands.

5.7 Limitations

We want to use this section of the thesis to point out some of the limitations of our approach to identify FPs in the output of SAST tools specialized in cryptographic libraries.

As we have already stated in Chapter 2 we limit our approach to path-sensitive warnings of the categories described. Furthermore, our implementation is focused on warnings generated by misuse of the JCA, using the SAST tool *CogniCrypt* as a basis. Although modifying that aspect of the implementation shouldn't be hard at all.

Since we are also using *Soot* and *Soot* requires a main class for generating a CG, our analysis can only be performed on executable projects or libraries that have a "gateway"to it's functionality present, that is a path from an executable class, to the methods and functions that we want to test.

When traveling backward through the execution paths of the program, we ignore any side paths, as this would severely increase the complexity of the algorithm and reduce the performance, possibly making it practically not viable. Any variables that are assigned new values as a result of these side paths, will be assigned values that reflect that they cannot be resolved conclusively.

Our analysis also circumvents some of the more complex code structures like loops and recursion. This speeds up the evaluation but possibly reduces the FPs found. Future enhancements to the code could solve this issue.

Finally, our approach has very limited capabilities in detecting when an object is never used again for the rest of a program, after it's sink method has been executed, therefore making the sink a FP. This is because we only analyze the path from sink to source and ignore the rest of the program. For a more conclusive path sensitive analysis, we would also need to track the next usage of that object.

6 Evaluation

This chapter contains the evaluation of the implementation of our approach. We test our implementation for accuracy and performance; hence, this chapter is split into two parts. For each case we describe the individual set up and give an overview of the results.

6.1 Accuracy

The first part of our evaluation is concerned with determining the accuracy or precision of the program. By accuracy we mean whether or not the program performs as intended. In order to measure it's semantical correctness we use a benchmark. We expand the benchmark to include additional cases not yet covered by said benchmark.

6.1.1 Crypto-API-Benchmark

Crypto-API-Benchmark is a library specifically designed to provide test cases for 16 different categories of cryptographic library misuse. We have already gone over these categories in Chapter 2. In fact, our approach is specifically designed to correctly identify FP vulnerabilities in the categories *Mode of Operation*, *Symmetric Ciphers* and *Cryptographic Hash Functions*. Test cases for these three categories are located in the packages `org.cryptoapi.bench.ecbcrypto`, `org.cryptoapi.bench.brokencrypto` and `org.cryptoapi.bench.brokenhash`. Within these packages, the acronym *PS* is used to mark classes containing path-sensitive test cases. We focus our program testing on these classes.

Path-sensitive test cases consist entirely of cases in which an insecure object is created but never used again, because it is being immediately overwritten. As we have mentioned in the previous chapter, our implementation only recognizes some of these cases as FP, namely when the FP is locally contained within a method.

Listing 6.1 shows an example, directly taken from the Crypto-API-Benchmark, where our program falls short of recognizing the FP. The FP we do not recognize is located on line 14. Here the `cipher` objects gets initialized by a sink method using an insecure parameter. This however is not a vulnerability, because the object is overwritten in line 16 never to be used again. The if-statement in line 15 always resolves to true because `choice` has the value 2. Our program fails to recognize this, because the value of `choice` is initialized in another method. To elaborate a bit further, cases where an insecure object is overwritten before being used, are recognized and eliminated during the CFG graph creation phase and at that point the application has only access to the local values of that graphs method.

Listing 6.1 An example of a false positive not recognized by our program. Class taken directly from Crypto-API-Benchmark.

```
01  package org.cryptoapi.bench.brokencrypto;
    ...
10  public class BrokenCryptoABPSCase1 {
11      public void go(int choice) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
            ...
14          Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
15          if (choice > 1)
16              cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
17
18          cipher.init(Cipher.ENCRYPT_MODE, key);
19      }
20
21      public static void main (String [] args) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
22          BrokenCryptoABPSCase1 bc = new BrokenCryptoABPSCase1();
23          int choice=2;
24          bc.go(choice);
25      }
26  }
```

The rest of the path-sensitive test cases in the three packages mentioned before, also involve an insecure object that is overwritten before being used. These cases, however, only involve local values, and so our implementation recognizes these FPs without fail.

6.1.2 Modifications of the benchmark

We want to test for two additional path-sensitive cases: Cases in which the insecure object is created on a path that is never visited during execution and cases where the sink input parameter is being sanitized beforehand. To do so, we create additional classes within the library.

Listing 6.2 shows an example of the type of test case where the path on which the sink occurs is never visited because of the result of a conditional statement. Here the integer variable `choice` has the value 2. Therefore, the if-statement in line 15 is always true, and as a consequence, the insecure initialization of the variable `cipher` in line 18 is never executed. As a result, the sink on line 18 is a FP that *CogniCrypt* will falsely report.

The other type of test case that we introduce to the benchmark is depicted in Listing 6.3. In this example, the variable `cipher` is initialized using the value stored in `cryptoValue`. Our implementation recognizes that the variable `cryptoValue` is sanitized in line 18, meaning it is ensured that the value of this variable can only be the secure value *AES/CBC/PKCS5Padding* in order to even reach the sink method. Line 19 will consequentially be marked as a FP by our implementation. The thing to note here is that *CogniCrypt* will also not mark the sink in line 19 as a vulnerability. This however

Listing 6.2 An example of a false positive test case. Line 18 contains an insecure initialization of the Cipher object. Since choice has the value 2 line 18 is never visited.

```
01  package org.cryptoapi.bench.brokencrypto;
    ...
10  public class BrokenCryptoABPSCase1Visited {
11      public void go(int choice) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
            ...
14          Cipher cipher;
15          if (choice > 1)
16              cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
17          else
18            cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
19
20          cipher.init(Cipher.ENCRYPT_MODE, key);
21      }
22
23      public static void main (String [] args) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
24          BrokenCryptoABPSCase1Visited bc = new
            BrokenCryptoABPSCase1Visited();
25          int choice=2;
26          bc.go(choice);
27      }
28  }
```

is not because *CogniCrypt* recognizes the FP. In fact, it does not matter what value you compare the variable `cryptoValue` in line 18 to, *CogniCrypt* will not mark line 19, even if it happens to be a very real vulnerability.

6.1.3 Results

Regarding the case of overwritten insecure objects, *CogniCrypt* fails to recognize the unused object as a FP. Our program pinpoints the FP correctly, when it can be decided locally that the insecure object will not be used in the future.

Regarding the cases of vulnerabilities on invalid paths and sanitized variables, used as input for sink methods, *CogniCrypt* fails to recognize the FPs in the case of invalid paths and will not deal with cases where values are possibly sanitized, at all. Both of these cases are marked correctly as a FP by our program. However, since we cross check with the output of *CogniCrypt* and *CogniCrypt* has no report about the cases with sanitized values, we only output the reported vulnerabilities in the invalid path cases, as FPs.

Listing 6.3 An example of a false positive test case. The variable cryptoValue used to create the Cipher object is sanitized beforehand.

```
01  package org.cryptoapi.bench.brokencrypto;
    ...
10  public class BrokenCryptoABPSCase1Sanitize {
        ...
14      public void go() throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
15          KeyGenerator keyGen =
            KeyGenerator.getInstance(String.valueOf(crypto));
16          SecretKey key = keyGen.generateKey();
17          String cryptoValue = "IDEA";
18          if(cryptoValue.equals("AES/CBC/PKCS5Padding")) {
19            Cipher cipher = Cipher.getInstance(cryptoValue);
20            cipher.init(Cipher.ENCRYPT_MODE, key);
21          }
22      }
        ...
37  }
```

6.2 Performance

Aside from evaluating the accuracy of our program, we also want to give a rough estimate of its performance. Performance evaluation proves to be difficult for several reasons. First, there are no benchmarks to evaluate the performance of cryptographic SAST tools. Second, our approach is specialized in detecting FPs in specific cases, making it difficult to find similar tools with which we can compare our program. We decided to analyze the Crypto-API-Benchmark and two randomly chosen executable Java programs from *GitHub* using *CogniCrypt* and then analyze those programs with our implementation. We will record the results in a table.

6.2.1 Set up

The three projects we analyze for performance are the Crypto-API-Benchmark used for the accuracy evaluation, a two-player poker game called *BigTwoGame* [BTG] and a management system for medical drugs [MMS].

We alter the source code of both the poker game and the medical management system to include some true and false cryptographic vulnerabilities. We use additional executable test classes to really expose all the functionality of these systems and make all included vulnerabilities reachable for our program.

We then run an *CogniCrypt* analysis on the three projects and keep track of the time it takes *CogniCrypt* to finish its analysis. After that we perform an analysis using our program and record the execution time.

We save all results in a table that includes execution time, total lines of code, and the number of main classes of the project.

6.2.2 Results

Table 6.1 lists the results of the *CogniCrypt* analysis, and Table 6.2 lists the results of the analysis of our program.

When it comes to execution speed our program is roughly as fast as *CogniCrypt* with the big exception being the Crypto-API-Benchmark. Our program has to generate a new CG and execute the routine that maps sinks to a CG for every main class, and since the number of main classes in the Crypto-API-Benchmark is exceedingly high, we have to execute this procedure a lot. This seems to be the bottleneck in our implementation.

Another thing to note, in regards to the number of vulnerabilities contained within a project; the higher the number of vulnerabilities, the more time the computation takes. Looking specifically at the *BigTwoGame* project, we have tested it both with only 10 and a 100 vulnerabilities inserted into the code and execution time has grown. Given the tests we made it seems that our prototype seems to scale slightly worse than *CogniCrypt* with the amount of vulnerabilities.

Table 6.1: A table containing the results of the CogniCrypt analysis.

Project	Lines of Code	Number of Main Classes	Execution Time [s]
Crypto-API-Benchmark	4139	163	22
BigTwoGame (10 vulnerabilities)	3346	1	15
BigTwoGame (100 vulnerabilities)	3602	1	32
Medical Management System	2142	1	27

Table 6.2: A table containing the results of the analysis of our program.

Project	Lines of Code	Number of Main Classes	Execution Time [s]
Crypto-API-Benchmark	4139	163	124
BigTwoGame (10 vulnerabilities)	3346	1	18
BigTwoGame (100 vulnerabilities)	3602	1	55
Medical Management System	2142	1	13

7 Conclusion & Outlook

The goal of this thesis was to develop and implement an approach for reducing FPs warnings in SAST tools reporting, specifically for tools specialized in cryptographic libraries. To achieve this goal, we have conducted research on existing approaches and settled for an approach that eliminates path sensitive FPs of the three categories *Mode of Operation*, *Symmetric Ciphers* and *Cryptographic Hash Functions*. We used an existing approach that uses a backward search through CGs and CFGs, and adapted it to our needs. We implemented a prototype and evaluated it for accuracy and performance, showing off its capabilities and limitations.

Our implementation is a first attempt at providing path-sensitive analysis of cryptographic vulnerabilities in specific cases. There is still a lot of work to do to make it a viable solution for static analysis. Some future work on the prototype might include: Making it independent of *CogniCrypt* or any other SAST tool, making it independent of the *Soot API* and, therefore, giving it more freedom and flexibility in the creation of the graphs, extending the range of the categories of vulnerabilities covered, and getting rid of some or all of the limitations related to the path sensitive search, such as properly dealing with loops, recursion, and side branches. We also estimate that by introducing parallel processing for the different CGs, we could speed up performance noticeably.

Generally speaking, we think there is a lot of potential in improving accuracy and performance in SAST tooling. One possible path for achieving progress in that respect, that we have not touched on in this thesis, might be the inclusion of deep learning techniques in static analysis. Using deep learning might prove more viable in tackling the complexity that comes with path sensitivity. This is of course all just speculation.

Bibliography

[ANG+19] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, Y. Higo. "An empirical study of security warnings from static application security testing tools". In: *Journal of Systems and Software* 158 (2019), p. 110427. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110427. URL: https://www.sciencedirect.com/science/article/pii/S0164121219302018 (cit. on p. 29).

[APH+08] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130 (cit. on p. 29).

[ARY19] S. Afrose, S. Rahaman, D. Yao. "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses". In: *2019 IEEE Cybersecurity Development (SecDev)*. 2019, pp. 49–61. DOI: 10.1109/SecDev.2019.00017 (cit. on p. 20).

[Bar+10] A. G. Bardas et al. "Static code analysis". In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107 (cit. on p. 19).

[BSI+08] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, A. Gupta. "SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement". In: *Static Analysis*. Ed. by M. Alpuente, G. Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–254. ISBN: 978-3-540-69166-2 (cit. on p. 29).

[BTG] Q. Haoran. *BigTwoGame*. URL: https://github.com/James-QiuHaoran/BigTwoGame-with-Network-Facilited (cit. on p. 52).

[CM04] B. Chess, G. McGraw. "Static analysis for security". In: *IEEE Security and Privacy* 2.6 (2004), pp. 76–79. DOI: 10.1109/MSP.2004.111 (cit. on p. 29).

[COV22] Synopsys, Inc. *Coverity*. 2022. URL: https://scan.coverity.com/ (cit. on p. 20).

[CW07] B. Chess, J. West. *Secure programming with static analysis*. Pearson Education, 2007 (cit. on p. 29).

[ECL01] Eclipse Foundation. *Eclipse*. 2001. URL: https://www.eclipse.org/ (cit. on p. 36).

[EN08] A. Einarsson, J. D. Nielsen. "A survivor's guide to Java program analysis with soot". In: *BRICS, Department of Computer Science, University of Aarhus, Denmark* 17 (2008) (cit. on p. 36).

[FB] W. Pugh, D. Hovemeyer. *FindBugs*. URL: http://findbugs.sourceforge.net/ (cit. on p. 29).

[Fra20] M. Frantz. "Enhancing CryptoGuard's Deployability for Continuous Software Security Scanning". PhD thesis. Virginia Tech, 2020 (cit. on pp. 20, 29).

[FSB] P. Arteau, D. Formánek, T. Polešovský. *Find Security Bugs*. URL: https://find-sec-bugs.github.io/ (cit. on p. 29).

[HLH+19] G. Hao, F. Li, W. Huo, Q. Sun, W. Wang, X. Li, W. Zou. "Constructing Benchmarks for Supporting Explainable Evaluations of Static Application Security Testing Tools". In: *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2019, pp. 65–72. DOI: 10.1109/TASE.2019.00-18 (cit. on p. 29).

[HLL+03] L. Hendren, P. Lam, J. Lhotak, O. Lhotak, F. Qian. "Soot, a tool for analyzing and transforming java bytecode". In: (2003). URL: http://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.ps (cit. on p. 37).

[KNR+17] S. Krüger, S. Nadi, M. Reif, K. A. 0001, M. Mezini, E. Bodden, F. Göpfert, F. G. 0001, C. Weinert, D. Demmler, R. Kamath. "CogniCrypt: supporting developers in using cryptography". In: *Proceedings of the 32nd International Conference on Automated Software Engineering*. IEEE Computer Society, 2017, pp. 931–936. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115707 (cit. on pp. 20, 29, 36).

[KSA+21] S. Krüger, J. Späth, K. Ali, E. Bodden, M. Mezini. "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs". In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2382–2400. DOI: 10.1109/TSE.2019.2948910 (cit. on pp. 29, 36).

[MMS] chetan7020. *Medical-Management-System*. URL: https://github.com/chetan7020/Medical-management-system (cit. on p. 52).

[Pas17] I. Pashchenko. "FOSS Version Differentiation as a Benchmark for Static Analysis Security Testing Tools". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 1056–1058. ISBN: 9781450351058. DOI: 10.1145/3106237.3121276. URL: https://doi.org/10.1145/3106237.3121276 (cit. on p. 29).

[PDM17] I. Pashchenko, S. Dashevskyi, F. Massacci. "Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017, pp. 163–168. DOI: 10.1109/ESEM.2017.24 (cit. on p. 29).

[RXA+19] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, D. (Yao. "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-Sized Java Projects". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 2455–2472. ISBN: 9781450367479. DOI: 10.1145/3319535.3345659 (cit. on pp. 20, 29).

[SB] *SpotBugs*. URL: https://spotbugs.github.io/ (cit. on p. 29).

[SJM+15] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, H. R. Lipford. "Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 248–259. ISBN: 9781450336758. DOI: 10.1145/2786805.2786812. URL: https://doi.org/10.1145/2786805.2786812 (cit. on p. 29).

[SOOT] Sable Research Group. *Soot*. URL: http://soot-oss.github.io/soot/ (cit. on p. 36).

[SQ22] SonarSource S.A. *SonarQube*. 2022. URL: https://www.sonarqube.org (cit. on p. 20).

[VCG+10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan. “Soot: A Java Bytecode Optimization Framework”. In: *CASCON First Decade High Impact Papers*. CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224. DOI: 10.1145/1925805.1925818. URL: https://doi.org/10.1145/1925805.1925818 (cit. on p. 36).

[YWM17] X.-X. Yan, Q.-X. Wang, H.-T. Ma. “Path sensitive static analysis of taint-style vulnerabilities in PHP code”. In: *2017 IEEE 17th International Conference on Communication Technology (ICCT)*. 2017, pp. 1382–1386. DOI: 10.1109/ICCT.2017.8359859 (cit. on pp. 27, 29, 31, 33–35).

[ZZ13] Y. Zheng, X. Zhang. “Path sensitive static analysis of web applications for remote code execution vulnerability detection”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 652–661. DOI: 10.1109/ICSE.2013.6606611 (cit. on p. 29).

All links were last followed on July 30, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature