UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Master's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science

The comparison of two static analysis tools for security testing - CogniCrypt$_{\text{SAST}}$ and Codyze

by
Shahrzad Asgharivaskasi

Thesis Supervisors:
Prof. Dr. Eric Bodden

Prof. Dr. Gregor Engels

Thesis Advisor:
Michael Schlichtig

Paderborn, January 31, 2022

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Abstract. There are several tools that detect misuses of Java cryptographic APIs within programs that either employ static or dynamic analysis or a combination of the two. These tools may use different approaches or improve upon an old one. Among those tools, CogniCrypt$_{\text{SAST}}$ and Codyze seem to follow a similar approach. Both tools use a white-listing approach and define rules to specify the correct usage of cryptographic APIs using DSLs, CrySL in CogniCrypt$_{\text{SAST}}$, and MARK in Codyze. This thesis aims to provide a fair comparison of Codyze and CogniCrypt$_{\text{SAST}}$ to discover the advantages and disadvantages of each approach to be able to improve them and create better analyzers in the future. Therefore, we conducted a theoretical and practical comparison of the two tools and their DSLs by reviewing their documentation, related papers, and source codes. Further, we translated the rules from one language to another to compare the DSLs' expressiveness. In addition, we evaluated the performance (precision and recall) of the tools by analyzing two benchmarks.
The rule translation revealed that not all elements of one language were translatable to the other. Moreover, we determined some analysis properties of each tool based on the results of the theoretical comparison and the performance evaluation. According to our results, Codyze and CogniCrypt$_{\text{SAST}}$ performed similarly; however, some aspects of both tools and their respective DSLs could be improved for tools to perform more effectively and efficiently.

Contents

1

Introduction

1.1 Introduction and Motivation

Protecting sensitive data is an essential part of software development. Nowadays, businesses, governments, hospitals, and many other organizations use software for transporting or storing their data, and they can experience huge complications from having sensitive information exposed. Financial loss, legal issues, or in some cases, loss of lives are some of the consequences of poor or lack of security in the software development process. Therefore, it is essential to have proper security in the application. Cyberattacks are inevitable problems; the costs of recovering from an attack can be high and impact the company or individual; however, it can be avoided by using cryptography in the software development process.

Cryptography is a method of information protection and secure communication so that only those who are supposed to read and process it can do so. Using cryptography correctly can reduce security vulnerabilities in code. There are some cryptographic algorithms that can help to add security to the application, and these algorithms have been implemented in Application Programming Interfaces (API) libraries. Cryptographic APIs offer cryptography in different usecases, like data encryption, secure communication, authentication, and other cryptographic use cases. The Java Platform provides Java Cryptographic Architecture (JCA) [jca] which is a set of APIs for implementing cryptography in Java. Listing 1.1 shows simple message encryption with the use of javax.crypto.Cipher [cip]. The method gets a message and a secret key as parameters, then encrypts the message using the key and returns the result.

Listing 1.1: Sample code of message encryption using Cipher object.

```java
public class EncryptionSample {
    public static byte[] main(byte[] message, SecretKey key) throws Exception{
      Cipher cipher = Cipher.getInstance("DES");
      cipher.init(Cipher.ENCRYPT_MODE, key);
      cipher.update(message);
      byte[] cipherText = cipher.doFinal();
      return cipherText;
    }
 }
```

The sample code does not raise a syntactical error. However, it contains misuses that make the program vulnerable. For example, this sample code uses the DES[1] algorithm to encrypt the data. DES was found to be vulnerable to attacks like brute-force. A brute force attack involves guessing login information, encryption keys, or locating a hidden web page through trial and error. Attackers try all possible combinations in the hopes of making the right guess. Therefore, DES is considered insecure, so it is better to use an alternative algorithm, such as AES[2]. AES is a more mathematically efficient and elegant algorithm for data encryption, since it supports block lengths of 128 bits while DES provides only 64 bits of block length. In addition, AES provides key lengths of 128 bits, 192 bits, or 256 bits while DES provides only 56 bits. When using cryptographic APIs, developers should know well how to integrate them into their codes. Otherwise, incorrect usages of these APIs would result in insecurity in the program. Lazar et al. [DLZ14] investigated 269 cryptography-related vulnerabilities. In only 17% of the cases, the problem was in cryptographic libraries themselves, where the other 83% were developers misusing cryptographic APIs. Another study on cryptography usage in Android applications suggests that approximately 90% of the applications contain at least one misuse [lCX16]. Developers often struggle with using cryptographic APIs. Studies show that developers find APIs too intricate [SNB16]. In addition to Oracle's documentation on cryptographic APIs [jca], developers also use the help of web pages like Stack Overflow[3]. The information provided on these websites is generated by regular developers who may not have crypto expertise [ASW$^+$17]. Furthermore, some classes have multiple methods with different possible parameters (some would be deprecated or insecure to use). When using a combination of them, establishing the correct implementation would be very difficult and confusing. Nadi et al. [SNB16] researched why developers find it challenging to use cryptographic APIs. The majority of participants had Java development experience (73%), and 86% had at least minimum knowledge of cryptography. The study showed that inadequate documentation with no examples and complicate API design that makes it hard for developers to apply them were the main reasons. When asked for suggestions, the participants wished for solutions to improve cryptography use in Java, like better documentation, new API styles, and more tool support.

Several tools are made to detect misuses of Java cryptographic APIs through static analysis, the method of analyzing the source code without executing the program. Various techniques exist in developing static analysis tools for finding cryptographic misuses [MEK13, MBB18, SR19, SKM19, coda]. For example, CryptoGuard [SR19] is a static analyzer that analyzes massive-sized Java projects for detecting cryptographic and SSL/TLS API misuses using a program slicing approach. Program slicing is a method for simplifying the programs by concentrating on particular elements of semantics and eliminating non-essential parts. The Related work chapter provides an overview of some examples of analyzers (cf. Chapter 2).

Among those tools, CogniCrypt$_{\text{SAST}}$ [SKM19] and Codyze [coda] stand out because they seem to have similar approaches. CogniCrypt is a platform with two components that aid in the proper use of cryptographic APIs in Java. CogniCrypt$_{\text{GEN}}$ generates code examples for cryptographic APIs, and CogniCrypt$_{\text{SAST}}$ is a static analyzer for finding cryptographic misuses in Java programs. Codyze is also a static analysis tool that analyzes Java or C++, or C programs to verify the correct implementation of cryptographic APIs. They both use a white-

[1]Data Encryption Standard

[2]Advanced Encryption Standard

[3]`https://www.stackoverflow.com/`

listing approach, where proper uses of cryptographic APIs are defined in specifications written in domain-specific languages (DSL), CrySL [SKM19], and MARK[4], respectively. Domain-specific languages are designed to perform specific tasks within a particular domain [MHS05], in this case, to define rules for cryptographic API usage.

As it seems on an abstract level, their approaches seem similar. However, if we look into more details and explore their implementations closely, we find several differences. As we mentioned earlier, both of them use white-listing approaches and define rules for usages of Java cryptographic APIs, which specify how each class should be used in order to make a secure program. However, they use different DSLs for that purpose. In addition, the way each tool examines the source code against those rules is different. Therefore, it is interesting to look further into them to discover where the similarities and differences lie and explore the advantages of these differences. This thesis compares these two tools in terms of DSLs, analysis techniques, performances, and their capabilities. Our results demonstrate the strengths and weaknesses of each tool. We can use this information to improve the tools in the future and create more reliable and efficient analyzers.

This thesis is structured as follows: Chapter 2 describes the related tools that can detect cryptography misuses in a program. Chapter 3 presents an overview of Codyze and CogniCrypt$_{\text{SAST}}$. In Chapter 4, we compare the two tools and their respective DSLs both theoretically and practically. Chapter 5 provides an evaluation of the tool's performance using two benchmarks and discusses the findings. Chapter 6 concludes this thesis, and finally, Chapter 7 describes future actions that can be taken to compare more aspects of the tools and to improve the quality of the tools.

[4]`https://www.codyze.io/docs/`

2

Related Works

Many tools have been developed to detect cryptography misuses in Java code in order to prevent insecurity within a software program. Such tools use static analysis or dynamic analysis or a combination of static and dynamic analysis. Static analysis is the method of analyzing a program's code without running it. On the other hand, dynamic analysis is the monitoring and examining of a program depending on its execution. It is typically used to identify subtle faults or bugs that manifest during runtime and whose origin is too complex to be observed through static analysis [lCX16].

Below tools deploy dynamic analysis for detecting cryptographic misuses.

AndroSSL is a framework to test mobile applications against connection security vulnerabilities [GFF+15]. It performs SSL man-in-the-middle attacks against android applications. A man-in-the-middle attack occurs when a third party intercepts a communication between two systems [MAST19]. Successful attacks confirm the application's vulnerability. The AndroSSL developers tested it with 90 popular Android applications. AndroSSL found 25 apps with a vulnerability.

Crypto keys are susceptible to being insecurely used at different stages of their usage, generation, derivation, operation, and cleaning. K-Hunt [LCZG18] uses binary dynamic analysis to identify such insecure keys in program executables. In order to identify the crypto keys used by the target program, the program is executed with coarse-grained binary code instrumentation [LCZG18]. In the next step, the program is executed by a heavyweight fine-grained instrumentation [LCZG18] to detect insecure keys. The K-Hunt developers analyzed 10 cryptographic libraries and 15 applications containing crypto operations via K-Hunt. Among the 25 evaluated programs, 22 contained insecure keys.

The following tools combine static and dynamic analysis, and each employs different approaches.

Crypto Misuse Analyser (CMA) [SGT+14] uses both static and dynamic analysis in Android applications. They created a cryptographic misuse model to identify cryptographic misuses. The static analysis detects apps where cryptographic APIs are used. CMA generates runtime logs by running them. Through comparing logs with models, CMA decides if the implementations have misuses. It only analyzes symmetric encryption, hash, and asymmetric encryption misuse vulnerabilities. The CMA developers tested it on 45 apps that involve storing or transporting sensitive data, like mobile bank clients. Just 5 of them were found to have no misuse.

Chatzikonstantinou et al. [lCX16] also conduct static and dynamic analysis to detect cryptographic misuses in Android applications. The static part is done manually on the source code by the developers, and dynamic analysis was done by Dalvik Debug Monitor Server[1] (DDMS). DDMS is a GUI-based application designed to help identify bugs in Android applications while the program is running. However, in this project, they have used the tool to examine cryptographic libraries invoked by Android programs during runtime. They evaluated 49 Android apps from the Google Play marketplace. According to their findings, 87.8% of the applications had cryptographic misuses, while in 12.2%, no cryptography was used.

Furthermore, there are other tools, which only use static analysis.

CryptoLint [MEK13] is a static analyzer based upon the Androguard [and] Android program analysis framework to discover JCA misuses in Android Dalvik bytecode. It uses the software slicing method to distinguish flows between cryptographic keys, initialization vectors, and other cryptographic data, as well as the cryptographic operations themselves, in a raw Android binary. It has a defined set of six rules. Violation of those rules is considered misuse. The result of their study on 11,748 Android apps shows that 10,327 systems (or 88%) use cryptography incorrectly.

BinSight [MBB18] is an automated static analysis system that identifies cryptographic misuses using the same rules as CryptoLint with program slicing. This tool is more effective in attributing Java class identifiers to its source, regardless of whether it is derived from an application source code or a third-party library, using a heuristic-based technique. It can handle large Android applications. They analyzed 132K Android applications, and the results suggested that 90% of the violating applications, which contain at least one call-site to Java cryptographic API, originate from libraries.

CryptoGuard [SR19] uses static analysis on massive-sized Java projects and performs forward and backward slicing methods based on specific criteria to detect 16 predefined vulnerabilities. After performing the slicing, each program slice is analyzed to find the presence of a vulnerability. They use refinement algorithms to reduce false positives. False positive is when the analysis identifies a misuse that does not exist, and false negative is when the analysis does not find an existing misuse. They also created a benchmark called CryptoAPI-Bench [ARY19] that covers all 16 cryptographic rules.

[1] `https://developer.android.com/studio/profile/monitor`

CogniCrypt$_{\text{SAST}}$ [SKM19] is a static analyzer that uses white-listing to define the correct usage of Java cryptographic APIs by writing rule specifications in the domain-specific language, CrySL, and statically analyzing the code against those rules. Static analysis is conducted on a graph derived from an intermediate representation of Java code in CogniCrypt$_{\text{SAST}}$. CogniCrypt$_{\text{SAST}}$ is accessible through Eclipse, or it can also be run independently as a command-line tool. CogniCrypt$_{\text{SAST}}$ developers scanned 10,000 Android applications and discovered at least one misuse in 95% of them.

Codyze [coda] is also a static analysis tool for Java, C, and C++ programs that uses white-listing. Codyze developers described API's and library's correct usage patterns using domain-specific language. The approach creates a graph of the source code and detects cryptographic misuses by performing static analysis on it. Besides integrating to common IDEs (e.g., Eclipse, IntelliJ, and Visual Studio), Codyze can also be used in command-line mode and provides an interactive console.

In contrast with other static analysis tools, CogniCrypt$_{\text{SAST}}$ and Codyze seem to follow a similar approach to detect misuses. Both defined rules to specify the correct usage of cryptographic APIs using DSLs and build a graph of the code to do static analysis to detect the misuses. We examine their approaches in more detail as well as the results of their analysis to discover the similarities and differences between these two tools, and we can then determine the advantages and disadvantages of their respective approaches. The purpose of this study is to determine the shortcomings and benefits of each tool in order to create more reliable and effective tools in the future. The following chapter will provide a brief overview of Codyze and CogniCrypt$_{\text{SAST}}$, allowing us to gain a better understanding of how they operate.

3

Introduction to CogniCrypt$_{\text{SAST}}$ and Codyze

In this chapter, we will discuss CogniCrypt$_{\text{SAST}}$ and Codyze in greater detail to gain a better understanding of the tools and their capabilities. In addition, this information can be utilized to compare the tools in Chapter 4 and to evaluate their performance in Chapter 5.

3.1 CogniCrypt$_{\text{SAST}}$

CogniCrypt$_{\text{SAST}}$ [Kr0] is an open-source static analyzer that analyses Java codes to find misuses of cryptographic APIs. CogniCrypt$_{\text{SAST}}$ is integrated into the CogniCrypt Eclipse plugin and can also be used as a standalone command-line tool to analyze Android and Java applications [cry]. CogniCrypt$_{\text{SAST}}$ consumes CrySL [SKM19] rules to configure a flow-sensitive and context-sensitive static data-flow analysis. Flow-sensitivity is the ability to determine the order in which statements are presented [SB15]. The context-sensitive analysis is an inter-procedural analysis that considers the calling context when analyzing a method [SB15]. An intra-procedural analysis analyzes a single procedure, while an inter-procedural analysis examines the interrelationships among procedures. Data flow analysis refers to the process of tracking data through a program in order to explain its behavior without actually executing the program [Spa19]. According to Krueger [Kr0], CogniCrypt$_{\text{SAST}}$ is also field-sensitive but not path-sensitive. Path-sensitive data flow analysis improves the precision of data flow analysis by analyzing the feasibility of paths [WZH^{+}13].

CogniCrypt$_{\text{SAST}}$ uses the Soot framework [PLH11] for static analysis. Soot enables Java developers to build their own static analyzer tool. It contains several intra-procedural and inter-procedural features to solve intra- and inter-procedural analysis problems. Direct analysis of Java bytecode can be hindered by the unclarity of data flow; hence CogniCrypt$_{\text{SAST}}$ employs Soot that offers several intermediate representations (IR) of Java code. Jimple is the fundamental Soot IR that simplifies analyzing and making a graph. CogniCrypt$_{\text{SAST}}$ builds a Call Graph (CG) from Jimple code. A call graph is a directed graph that represents the calling relation in a program function. Nodes represent methods, and edges represent a call relation between two methods [Ryd79]. In addition, analysis employs CFGs to identify the control flow of each method. For performing inter-procedural analysis of a complete program, the CogniCrypt$_{\text{SAST}}$'s analysis combines individual control flow graphs and the program's call graph in order to construct an inter-procedural control flow graph (ICFG).

CogniCrypt$_{\text{SAST}}$ utilizes CryptoAnalysis to build CGs and perform the analysis. CryptoAnalysis is a compiler that translates rules to static analysis. CryptoAnalysis warns the developer if any part of the specified CrySL rules is violated in the code. Three different static sub-analyses form CryptoAnalysis: typestate analysis, a Boomerang [SDAB16] instant and taint analysis for Android applications. Typestate analysis checks the valid sequence of operations based on the defined order on CrySL rules. For instance, the call sequence of the KeyGenerator object is `getInstance`, `init` (optional), and then `generatekey` (see Listing 4.1 Line 26). Any other combinations will be false. CryptoAnalysis uses Boomerang, a demand-driven pointer analysis for Java, to extract parameters on the fly to check them against the defined CrySL rules. For example, it checks if a valid `algorithm` is chosen for the `getInstance` method of KeyGenerator. Cryptoanalysis also includes taint analysis, which detects flaw injections or leaks of sensitive information. For analyzing Android applications, CryptoAnalysis employs Flowdroid [ARF$^+$14] which is a static taint analysis for Android applications.

It is possible to select different kinds of CG algorithms from CogniCrypt$_{\text{SAST}}$, CHA (Class Hierarchy Analysis), SPARK [LH03] and SPARK_LIBRARY that are available through command-line commands or Eclipse preference pages. The CHA is the default algorithm, which is not precise but is efficient. SPARK is a framework for call graph construction and points-to analysis, and it is a part of the Soot framework. SPARK's call graph construction algorithm considers any call that may occur at any point in the execution of the program [PLH11]. SPARK implements a context-insensitive points-to analysis. The points-to analysis associates variables with the allocation sites [SB15]. An analysis that is context-insensitive merges all call sites of a procedure. SPARK_LIBRARY is a SPARK mode specifically designed for the analysis of libraries. For all classes within the library, dummy allocation sites are instantiated, resulting in non-empty points-to sets for variables within the library [Spa19].

Misuses in CogniCrypt$_{\text{SAST}}$ are categorised into 7 categories, namely, ConstraintError, NeverTypeOfError, ForbiddenMethodError, ImpreciseValueExtractionError, TypestateError, RequiredPredicateError, IncompleteOperationError [cry]. The error messages that CogniCrypt$_{\text{SAST}}$ generates depend on the type of misuse and will be automatically generated at the end of the analysis.

3.2 Codyze

As of the date of this thesis, there have been no papers published on Codyze, and the documentation page is brief [coda]. Therefore, to understand Codyze, we investigated its source code on Github [codb], its documentation page and consulted with one of Codyze's developers (personal communications with one of the Codyze's developers via several emails and an online meeting on Sep 17, 2021).

Codyze [coda] is an open-source static analysis tool for Java, C and C++ programs developed by Fraunhofer AISEC[1]. It generates a code property graph (CPG) [YGAR14] from the source code and analyzes it against predefined MARK [coda] rules for finding cryptographic misuses. MARK is a domain-specific language (DSL) designed to write correct usages of Java, C, and C++ cryptographic APIs. Code property graphs allow Codyze to handle non-compiling or incomplete code. A CPG is a directed multigraph, which means that two nodes may be connected by multiple edges. Listing 3.1 illustrates an example code that shows a source method leaking (potentially) sensitive data and a sink method that may expose the data. Figure 3.1 depicts a CPG representation of Listing 3.1, in which there are multiple edges between the nodes PRED and DECL. The nodes represent program constructs. For example, in figure 3.1, PRED is a predicate node that indicates the condition of the if statement in Listing 3.1, Line 4 and DECL refers to the declaration statements of Line 6 in the Listing 3.1. This graph incorporates the properties of Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) in a single data structure [YGAR14]. An AST represents the abstract syntactic structure of the source code. It is an ordered tree with inner nodes representing operators and leaf nodes representing operands [YGAR14]. A CFG is a graph that represents all the possible paths that can be traversed during program execution. It is a directed graph that explicitly describes the order in which code statements are executed as well as conditions that need to be met for a particular path of execution to be taken [All70]. The program dependence graph indicates explicit dependencies between statements and predicates. Vertices in PDG represent the statements and predicates in the program. The edges representing the dependencies between components fall into two categories; data dependency edges reflect the influence of one variable upon another, and control dependency edges reflect the influence of predicates on the values of variables. After constructing each graph individually, the three graphs are merged to form the CPG. Graph nodes are the same as AST nodes, edges and labels are combinations of all three graphs, and sets of property keys and property values for nodes and edges are from AST and PDG graphs. Property keys and values as well as labels on AST edges are not shown in Figure 3.1. CFG and PDG edges are indicated by colors.

Listing 3.1: Code example for making it into a CPG [YGAR14].

```
1 void foo()
2  {
3     int x = source();
4     if (x < MAX)
5     {
6          int y = 2 * x;
7          sink(y);
8     }
9 }
```

[1]https://www.aisec.fraunhofer.de/

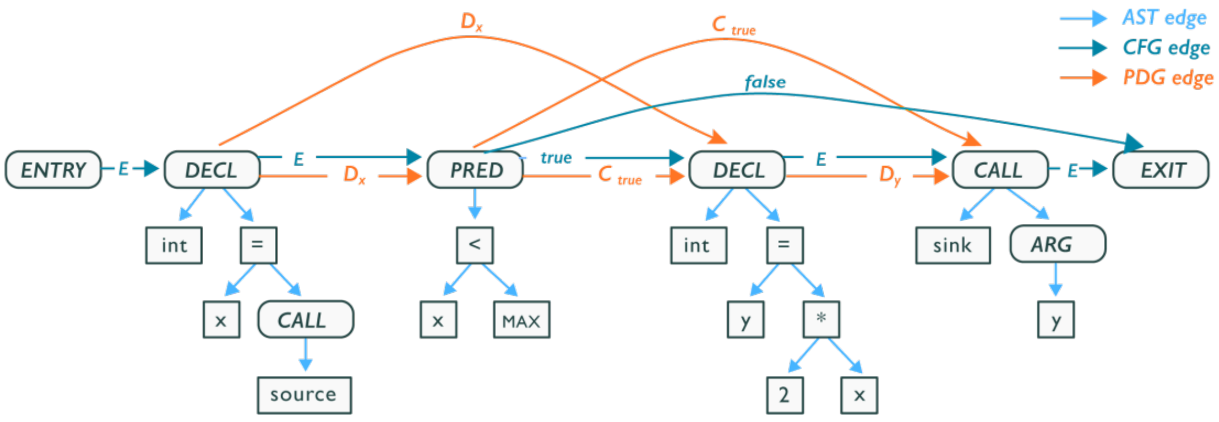


Figure 3.1: Code property graph for code sample in Listing 3.1 [YGAR14].

Observing CODYZE's source code, we discovered that CODYZE is a flow-sensitive static data-flow analysis. Through the use of CPGs and graph traversals, CODYZE is able to access the code flow and data dependencies associated with each node, thereby providing a data-flow analysis and can check the nodes in the CPG (e.g., methods, variables, etc.) against the defined MARK rules. In addition, there are MARK rules that specify the correct order of using an API's methods[2], which indicates that CODYZE is flow-sensitive. CPG and the query language are not enough to determine whether the program complies with the MARK order rules, so CODYZE developers implemented an intra-procedural typestate analysis. By using typestate analysis, CODYZE determines whether a sequence of operations is valid based on the MARK rules[3]. CPG does not cover inter-procedural analysis [YGAR14]. Further, it is stated in the Java documentation for CODYZE's implementation of typestate analysis that it is not inter-procedural; Thus, CODYZE is not considered to be inter-procedural. Nevertheless, when evaluating CODYZE's performance, we discovered that CODYZE is partially inter-procedural. At the time of this thesis, no published papers about CODYZE were available; therefore, we were unable to identify any other properties of it. However, we can better assess the analysis properties of CODYZE through evaluation of its analysis of example codes. We will discuss it further in the evaluation chapter (cf. Chapter 5).

CODYZE runs in three modes, interactive command line, non-interactive command line, and as a server for Language Server Protocol (LSP) [lsp]. CODYZE uses the LSP in order to provide greater flexibility in the use of different IDEs and supporting several languages. The LSP describes the communication protocol used between an editor or IDE and a language server, which enables features such as auto-complete, go to definition, etc. With a standard communication protocol, a single language server can be re-used in various code editors without too much effort [lsp]. CODYZE creates a language server that converts the source code from a client (IDE or command line) to a CPG. CODYZE developers have implemented a parser that parses MARK rules into a model that the analysis server can use. The analysis server analyzes the CPG against the MARK model with the help of Crymlin queries [gre]. Crymlin is an extension of the Apache

[2]`https://github.com/Fraunhofer-AISEC/codyze/blob/0468af19b90b16353402938db4e326b6dc63c4f7/src/dist/mark/bouncycastle/RulesBase_Cipher.mark`

[3]`https://github.com/Fraunhofer-AISEC/codyze/blob/e67d5fa0a1c956ce4be53fcdecf1990eb8dfb430/src/main/java/de/fraunhofer/aisec/codyze/analysis/wpds/TypestateWeight.java`

Gremlin graph traversal language that provides a variety of shortcuts for searching through code property graphs [gre].

Codyze also provides a JSON file (findingDescription.json) that contains all the possible errors based on the MARK rules in a human-readable format and is stored in the same location as the MARK rules (Listing 3.2). This file may later be used to create error messages in the client. It contains all possible errors, with the exception of errors related to the incorrect order of method calls and the use of insecure methods. The error messages for these two violations are generated automatically at runtime, since the error message for the first violation (the incorrect order) contains the missing methods[4] and the message for the second violation contains the forbidden call[5].

Listing 3.2 shows a sample error description in the findingDescription JSON file generated by Codyze. The error refers to the incorrect cryptographic algorithm or the absence of an algorithm during the creation of the Cipher object. The first line (Line 4) is the name of the error. HelpUri (Line 5) contains a URL from which MARK rules are derived (Bundesamt für Sicherheit in der Informationstechnik[6] or BSI [BSIa]). FullDescription (Line 7) provides a complete description of the error, while shortDescription (Line 10) is only a concise description. Line 13 describes the possible solutions to the problem. The fix, in this case, is to use the AES or RSA algorithm.

[4]`https://github.com/Fraunhofer-AISEC/codyze/blob/0468af19b90b16353402938db4e326b6dc63c4f7/src/main/java/de/fraunhofer/aisec/codyze/analysis/markevaluation/OrderNFAEvaluator.java`

[5]`https://github.com/Fraunhofer-AISEC/codyze/blob/04b5db6029a1bc6f8ad680cfecc796f408705031/src/main/java/de/fraunhofer/aisec/codyze/analysis/markevaluation/ForbiddenEvaluator.kt`

[6]Federal Office for Information Security

Listing 3.2: Part of findingDescription.json file generated by Codyze after analyzing Listing 1.1

```
...
{
    ...
    "Invalid_TR21021_Cipher": {
    "helpUri":
  ↪ "https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen
    /TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf",
    "fullDescription": {
      "text": "A cipher was detected that does not match one of the
  ↪ recommended ciphers by BSI TR-02102. Use of weak or unspecified
  ↪ ciphers may not guarantee sufficient security."
    },
    "shortDescription": {
      "text": "Use of an unspecified cipher"
    },
    "fixes": [
      {
        "description": {
          "text": "Use AES as symmetric-key (cf. 2.1) or RSA for
  ↪ asymmetric-key algorithm (cf. 3)"
        }}]
    }
    ...
 }
 ...
```

As well as spotting cryptographic misuses in a program, Codyze will also alert if a MARK rule is being used correctly. Codyze can be used as a console application or integrated into IDEs or CI pipelines. In the time of this thesis, Codyze is integrated into Eclipse, Visual Studio, and IntelliJ [coda]. Codyze offers a standard Python console for exploring the source code project using the Crymlin query interface.

Here we have described how CogniCrypt$_{\text{SAST}}$ and Codyze perform the analysis, the components they use to accomplish it, and the properties of their analysis. Having an understanding of Codyze and CogniCrypt$_{\text{SAST}}$, we can now begin to compare them. In the following chapter, we will provide a comparison of these tools.

4 Comparison

In the previous chapter (cf. Chapter 3), we learned how each analysis operates, so now we can begin comparing their approaches.Codyze and CogniCrypt$_{\text{SAST}}$ use rules that are written in different DSLs, MARK, and CrySL, respectively.Both tools employ graph structures to represent source code and facilitate the analysis of the code; however, CogniCrypt$_{\text{SAST}}$ generates call graphs (CG), whereas Codyze generates code property graphs (CPG). Codyze creates a CPG from the source code, then analyzes this graph based on the written rules, however, as stated in Section 3.1, CogniCrypt$_{\text{SAST}}$ creates a Soot IR from the source code and performs the analysis on the IR, not the source code itself. Therefore, it is important that we take an in-depth examination of the approaches of these tools in order to discover more similarities and differences, as well as the advantages and disadvantages associated with them.

Throughout this chapter, we compare the tools theoretically, then we compare them in practice, and finally, we compare their respective DSLs thoroughly.

4.1 Theoretical Comparison of CogniCrypt$_{\text{SAST}}$ and Codyze

To summarize the previous chapter, Codyze [coda] and CogniCrypt$_{\text{SAST}}$ [Kr0] are static analyzers for security testing that analyze the program to find cryptographic misuses. Codyze and CogniCrypt$_{\text{SAST}}$ consume MARK [coda] and CrySL [SKM19] rules respectively to perform the analysis. MARK and CrySL are both DSLs for specifying the correct usages of cryptographic APIs. We can divide the theoretical comparison of Codyze and CogniCrypt$_{\text{SAST}}$ into two parts, graph generation and how each tool analyzes the constructed graphs against the rules. Based on the information we provided on each tool on Chapter 3, we will compare them closely in analysis and graphs.

4.1.1 Comparison of the graphs

As we discussed in the previous chapter (cf. Chapter 3), both tools convert the source code into an intermediate representation in order to facilitate and speed up the analysis process. Codyze converts the source code into a graph representation (CPG), and CogniCrypt$_{\text{SAST}}$ transforms the source code into a Soot IR (Jimple), and then builds a call graph of the program or combines individual CFGs and the program's call graph into an ICFG.

The CPG is explicitly designed to identify vulnerabilities in large amounts of source code efficiently and effectively [YGAR14]. CPG can be generated of an erroneous code. Therefore, Codyze can analyze incomplete source codes and tolerate minor syntax errors [coda]. The CPG can either be used manually with a query language (e.g., NQL, SQL, or Gremlin) to identify the relevant patterns or automatically when integrated into CI/CD or IDEs [coda]. CPGs are independent of programming languages. When a program with a specific programming language is translated into a CPG, any query written for any other programming language can also apply to this CPG. Consequently, Codyze is capable of analyzing C and C++ programs as well as Java programs. The Code Property Graph (CPG) combines many representations of source code into one queryable graph database. This enables the CPG to understand the full flow of information across an application (AST) and represents the order in which statements are executed (CFG), and detects flaw injections or leaks of sensitive information (PDG). As a result, Codyze is a flow-sensitive data-flow analysis. CPG does not address inter-procedural analysis; however, it can be enhanced with call edges to provide inter-procedural analysis as shown in the study by Backes et al.[BRS+17].

CogniCrypt$_{\text{SAST}}$ first parses the programs into an IR (Jimple) using Soot [PLH11] and then constructs a call graph for it. IRs generally cover a limited number of constructs to analyze a program while still expressing all language elements. These limits simplify analysis by reducing the number of cases that analysis must cover. This leads to a simpler call graph. Call graphs show an abstraction of all method calls in a program that could be used for further analysis. In addition to call graphs, CogniCrypt$_{\text{SAST}}$ constructs CFGs of individual statements and sometimes creates an ICFG to represent the whole program. A similar approach applies to analyzing Android programs intra-procedurally. Moreover, the FlowDroid [ARF+14] extension allows the creation of Android-specific call graphs that can be used to construct the ICFG and perform inter-procedural analysis.

Currently, Soot only accepts Java code as an input to make IR, and therefore, CogniCrypt$_{\text{SAST}}$ cannot analyze other languages since it lacks the required IR. According to the readme file of the Soot Github repository [soo], Soot can process Java code up to Java 9. However, the Soot documentation page has not been updated and indicates that it only supports Java code up to Java 7 [PLH11]. CogniCrypt$_{\text{SAST}}$ have not been modified yet to use the most recent version of Soot and can only work with Java source code up to Java 8, and cannot analyze programs with Java versions higher than Java 8.

The graph construction is the most time-consuming part of the analysis in CogniCrypt$_{\text{SAST}}$. According to [SKM19], the majority of the total analysis time (83%) is spent on call graph construction in analyzing 8,422 apps by CogniCrypt$_{\text{SAST}}$. There are no publications or resources available to illustrate the time required to generate a CPG in Codyze, except for a presentation [pre] which included statistics of a Codyze analysis of 338 KB C++ source codes (12K lines of code and 486 methods). Based on their calculations, the CPG conversion time was 2.7 seconds or approximately 6 percent of the total analysis time. The results are unreliable to predict how much time will be required to construct a CPG on average. For this purpose, we would need to generate CPG for a significant number of programs to provide an accurate estimate. Upon further research on the CPG, we discovered that Backes et al. [BRS+17] generated CPGs with the call edges of 1,854 projects in their study. The results show that the AST generation of all projects took 40 minutes and 30 seconds, and the CFG, PDG, and call edge generations took 5 hours 10 minutes 33 seconds. However, this does not provide a reasonable estimate of how long it takes to generate a CPG since they did not create a pure CPG. We could further calculate the average time it takes to create a CPG using Codyze by analyzing several benchmarks and calculating the time required to create the CPG. This, however, was beyond the scope of our thesis as it involved a few modifications to Codyze's source code, which would require some time and, therefore, exceed our time limits. We will discuss it further in the further work chapter (cf. Chapter 7).

CPG is more complex and sophisticated than CG. Codyze extracts most of the analysis information directly from the CPG with the help of a query language to verify that it complies with the MARK rules, whereas CogniCrypt$_{\text{SAST}}$ performs three different analyses on the call graph to determine whether it complies with the CrySL rules (cf. chapter 3). Using only the query language is not sufficient for analyzing the order of operations in Codyze. Therefore, Codyze developers have implemented a typestate analysis for this purpose. We will discuss this in the next section (Section 4.1.2).

In order to determine which approach is more precise and effective, we will compare the results of the analysis performed by each tool with the same rules (same API specifications but with different DSLs) and the same source code in the Evaluation chapter (cf. Chapter 5).

4.1.2 Comparison of the analysis of the graphs

In the previous section, we discussed the graphs and the intermediate representations of the source code that each tool generates. In this section, we will examine the analyses that Codyze and CogniCrypt$_{\text{SAST}}$ perform on the IRs in order to identify cryptographic misuses in the program. As mentioned in the Chapter 3, Codyze and CogniCrypt$_{\text{SAST}}$ examine the code against the written MARK and CrySL rules, respectively. Both tools have parsers that convert the rules into models that the respective analysis can use.

According to Kreuger [Kr0], they have developed a parser for translating CrySL rules into the CrySL object model. CogniCrypt$_{\text{SAST}}$ transforms the rules in the CrySL object model into a static data-flow analysis that is both flow- and context-sensitive. CogniCrypt$_{\text{SAST}}$ contains a typestate analysis to validate the order in which the operations are called, a Boomerang instance to check the parameters against the CrySL rules, and a taint analysis for flaw injections and information leakage in Android applications. The analysis automatically examines Java or Android applications that have been converted into Jimple to determine whether they conform to the encoded CrySL rules. The error message that CogniCrypt$_{\text{SAST}}$ generates in the event of a misuse depends on the type of misuse and the factors that led to the misuse. There are several types of misuses that we named in Section 3.1.

Codyze converts the MARK rules into a MARK model that the analysis can use. Codyze utilizes CPG and query language to analyze a program's data flow and detect information leaks. In Codyze, the order of operations cannot be analyzed with only the query language; as a result, Codyze developers implemented a typestate analysis in that regard. Codyze error messages are derived from a JSON file created by Codyze maintainers (see Section 3.2), with the exception of order and forbidden call violations. Error messages related to those violations are generated automatically at Codyze's runtime.

Both tools utilize typestate analysis to provide flow sensitivity to ensure that the order of operations in the source code complies with the MARK or CrySL rules. Nevertheless, CogniCrypt$_{\text{SAST}}$ employs IDE$^{\text{al}}$ [SAB17], a framework for inter-procedural data-flow analysis, in order to perform a typestate analysis. The analysis defines a finite-state machine from the ORDER section of the CrySL rule to verify an object's trace in the program. It also defines the allocation sites from which to begin the analysis. An allocation site is a point at which an object is created in an object-oriented program. IDE$^{\text{al}}$ performs a flow-, field-, and context-sensitive typestate analysis from those allocation sites. Because C is not an object-oriented language, and C++ is not an entirely object-oriented language, this typestate analysis cannot be used to analyze C programs or all C++ programs.

Codyze, on the other hand, uses a Weighted Pushdown System (WPDS) [RLK07] for typestate analysis. The WPDS presents an abstraction of the data flows within a program. Currently, there is one WPDS per function in Codyze; therefore, it is not inter-procedural. Nonetheless, when evaluating the performance of Codyze, we realized that it is partially inter-procedural. We will discuss it in Chapter 5. The typestates are specified by a regular expression in the MARK `order` construct. The analysis converts the regular expression into a Non-deterministic Finite Automaton (NFA). The finite automata are called NFA when there exist many paths for specific input from the current state to the next state. Typestate NFA transitions are then represented as weights for a WPDS. This typestate can be used for non-object-oriented programs such as C.

In both tools, valid typestates are determined by regular expressions in the rules. In CrySL, regular expressions are defined in the ORDER section (see Listing 4.1 Line 26) and in MARK, the order construct (see Listing 4.7 Line 3).

Although we have an understanding of the differences in the implementation of analyzers of both tools, we will only be able to determine the effectiveness of each approach in an evaluation. We, therefore, will perform an evaluation in Chapter 5.

4.2 Comparison of CogniCrypt$_{\text{SAST}}$ and Codyze in practice

The next step will be to compare and discuss Codyze and CogniCrypt$_{\text{SAST}}$ in order to identify how they work in practice. This will include which editors and IDEs they support, what error messages they generate, and how they function in practice. As both tools are capable of analyzing Java programs, we have compared the tools for analyzing Java programs; Codyze's analysis of C++ and C programs is left to future research.

Setup

For this comparison, we used two virtual machines that had the same configuration, which means that they had the same hardware specifications (Intel(R) Xeon(R) CPU E5-2695, 2.30GHz machine with 2 virtual processors, 8 GB RAM, and 50 GB hard disc capacity) and the same operating system (Windows 10 Education). On the first machine we installed Java 8 (JDK version 1.8.0_301) and on the second Java 11 (JDK version 11.0.11) since Codyze requires Java 11 or above while CogniCrypt$_{\text{SAST}}$ requires Java 8. The Eclipse 2020-06 was installed on both machines since this is the latest version of Eclipse that supports the two Java versions. The versions of Codyze and CogniCrypt$_{\text{SAST}}$ that we used are not the published versions. We downloaded the source code of CogniCrypt$_{\text{SAST}}$ and Codyze up to the date of 31st of October 2021 and built them on the first and second virtual machines, respectively. We have downloaded the following source codes. CogniCrypt plugin source code on the develop branch to the commit ef64a0f[1] to use the CogniCrypt$_{\text{SAST}}$ as part of CogniCrypt Eclipse plugin. In addition, the CogniCrypt$_{\text{SAST}}$ standalone command-line tool from the develop branch to this commit 35d0916 [2]. Lastly, the source code of Codyze main branch to the commit 0468af1[3].

We will later use the same machines to compare the DSLs in the DSL comparison section (Section 4.3) and evaluate Codyze and CogniCrypt$_{\text{SAST}}$ performances in the Evaluation chapter (cf. Chapter 5).

Discussion

Currently, Codyze is available in Eclipse, IntelliJ, and Visual Studio with the Language Server Protocol (LSP) technology, while CogniCrypt$_{\text{SAST}}$ is only available within Eclipse. Codyze will automatically analyze Java, C, and C++ projects upon opening or saving, but there is no button in any of the IDEs for initiating or stopping the analysis. Both tools provide command-line support, Codyze also offers an interactive command-line that allows us to explore the CPG on our own.

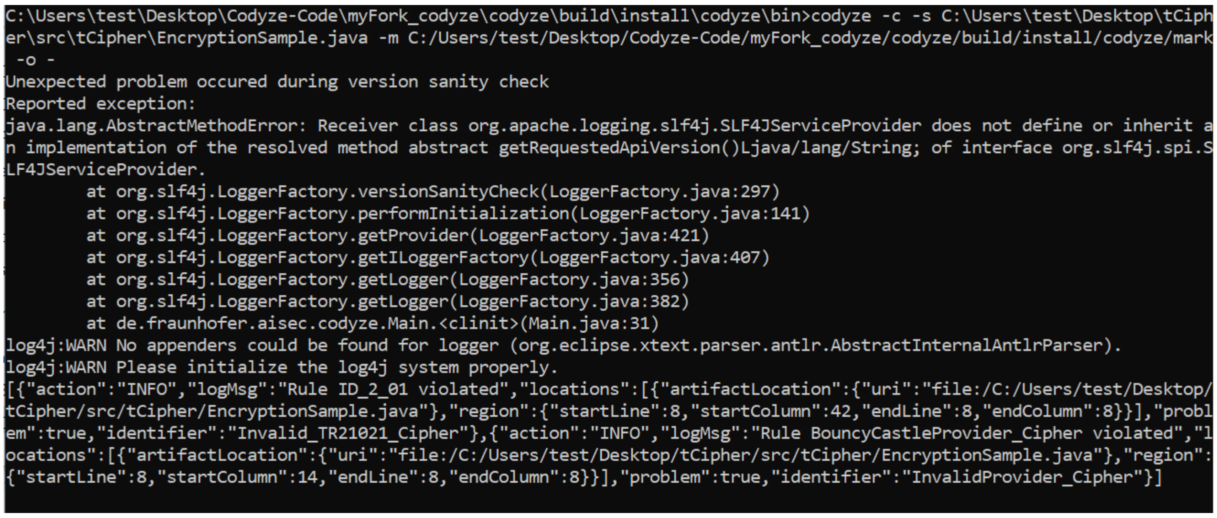
We were only able to use the Codyze plugin in Eclipse and view the results of the analysis. We successfully installed the Codyze plugin on IntelliJ and set it up according to the instructions provided on the Codyze documentation page [coda]. However, the analysis did not yield any results for any sample code with sufficient correct and incorrect usage of cryptographic APIs. We have also raised an issue on Codyze's Github page concerning this matter[4]. In the results of Codyze's analysis, in addition to showing violations against the rules, it also alerts when a rule is being used correctly.

We present the results of the Codyze and CogniCrypt$_{\text{SAST}}$ analyses on the code sample shown in Listing 1.1 in the command-line as a representative example of the analysis results of both tools. Codyze command-line tool gets the path to the source code file or a folder of the project, but CogniCrypt$_{\text{SAST}}$ command-line tool gets the path to the jar file; therefore, we made a jar file of the sample code.

Figure 4.1 shows the result of Codyze's analysis. The result indicates two violations, namely the unspecified provider name and the insecure Cipher algorithm. We will discuss these type of errors in Chapter 5. As illustrated in the figure, both errors have the action of INFO, causing Codyze to display the violations as information rather than as errors. Likewise, violations are displayed in the IDEs as information rather than errors which may cause confusion because

[1]`https://github.com/eclipse-cognicrypt/CogniCrypt/commit/ef64a0f2aa7bd54fdc70ae518691cfadf7f47f53`
[2]`https://github.com/CROSSINGTUD/CryptoAnalysis/commit/35d09163f97b6919a4359fcaa0e846af95c1fed1`
[3]`https://github.com/Fraunhofer-AISEC/codyze/commit/0468af19b90b16353402938db4e326b6dc63c4f7`
[4]https://github.com/Fraunhofer-AISEC/codyze/issues/374

verifications are also displayed as information. This problem has been reported on Codyze's Github page[5]. The name of the violated rule is indicated in the LogMsg. The Location points to the location of the analyzed project. The exact location in the code where the error occurred is given in the Region. The problem is true if this is an error, and the identifier is the onfail message of the rule that was violated. In the IDEs, the error message will be the error description from findingDescription.json that is described under Section 4.1. In Figure 4.1, the results are not well-formatted. This is an issue in the version of Codyze that we mentioned in the setup of this section. The results were more readable when using Codyze version 1.5.5, as shown in Figure A.1 in the appendices.

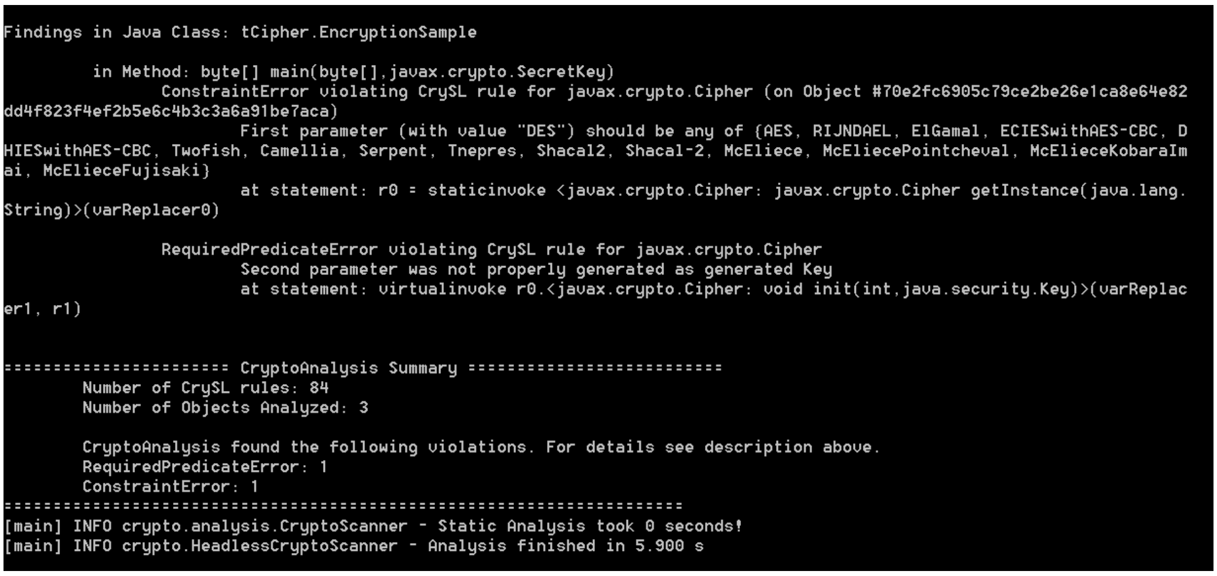


```
C:\Users\test\Desktop\Codyze-Code\myFork_codyze\codyze\build\install\codyze\bin>codyze -c -s C:\Users\test\Desktop\tCiph
er\src\tCipher\EncryptionSample.java -m C:/Users/test/Desktop/Codyze-Code/myFork_codyze/codyze/build/install/codyze/mark
 -o -
Unexpected problem occured during version sanity check
Reported exception:
java.lang.AbstractMethodError: Receiver class org.apache.logging.slf4j.SLF4JServiceProvider does not define or inherit a
n implementation of the resolved method abstract getRequestedApiVersion()Ljava/lang/String; of interface org.slf4j.spi.S
LF4JServiceProvider.
        at org.slf4j.LoggerFactory.versionSanityCheck(LoggerFactory.java:297)
        at org.slf4j.LoggerFactory.performInitialization(LoggerFactory.java:141)
        at org.slf4j.LoggerFactory.getProvider(LoggerFactory.java:421)
        at org.slf4j.LoggerFactory.getILoggerFactory(LoggerFactory.java:407)
        at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:356)
        at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:382)
        at de.fraunhofer.aisec.codyze.Main.<clinit>(Main.java:31)
log4j:WARN No appenders could be found for logger (org.eclipse.xtext.parser.antlr.AbstractInternalAntlrParser).
log4j:WARN Please initialize the log4j system properly.
[{"action":"INFO","logMsg":"Rule ID_2_01 violated","locations":[{"artifactLocation":{"uri":"file:/C:/Users/test/Desktop/
tCipher/src/tCipher/EncryptionSample.java"},"region":{"startLine":8,"startColumn":42,"endLine":8,"endColumn":8}}],"probl
em":true,"identifier":"Invalid_TR21021_Cipher"},{"action":"INFO","logMsg":"Rule BouncyCastleProvider_Cipher violated","l
ocations":[{"artifactLocation":{"uri":"file:/C:/Users/test/Desktop/tCipher/src/tCipher/EncryptionSample.java"},"region":
{"startLine":8,"startColumn":14,"endLine":8,"endColumn":8}}],"problem":true,"identifier":"InvalidProvider_Cipher"}]
```

Figure 4.1: Result of the analysis of Listing 1.1 by Codyze in command-line.

CogniCrypt's plugin provides toolbar and context menu buttons for triggering CogniCrypt$_{\text{SAST}}$ and CogniCrypt$_{\text{GEN}}$. Moreover, it provides an option to only analyze the dependencies of a project. Figure 4.2 shows the result of the analysis done by CogniCrypt$_{\text{SAST}}$ in command-line. This analysis identified a Constraint error, indicating that the Cipher algorithm was insecure and a RequiredPredicate error, indicating that the key used in the Cipher has not been generated with a proper key generator API. The Figure 4.2 displays the name of the class and the method in which the error occurs. Thereafter, the type of error and the violated rule are displayed. This is followed by the error message and the statement that contains the error. The description of an error generated by CogniCrypt$_{\text{SAST}}$ on the command-line and in the Eclipse IDE are similar.

[5]https://github.com/Fraunhofer-AISEC/codyze/issues/285



```
Findings in Java Class: tCipher.EncryptionSample

        in Method: byte[] main(byte[],javax.crypto.SecretKey)
                ConstraintError violating CrySL rule for javax.crypto.Cipher (on Object #70e2fc6905c79ce2be26e1ca8e64e82
dd4f823f4ef2b5e6c4b3c3a6a91be7aca)
                        First parameter (with value "DES") should be any of {AES, RIJNDAEL, ElGamal, ECIESwithAES-CBC, D
HIESwithAES-CBC, Twofish, Camellia, Serpent, Tnepres, Shacal2, Shacal-2, McEliece, McEliecePointcheval, McElieceKobaraIm
ai, McElieceFujisaki}
                        at statement: r0 = staticinvoke <javax.crypto.Cipher: javax.crypto.Cipher getInstance(java.lang.
String)>(varReplacer0)

                RequiredPredicateError violating CrySL rule for javax.crypto.Cipher
                        Second parameter was not properly generated as generated Key
                        at statement: virtualinvoke r0.<javax.crypto.Cipher: void init(int,java.security.Key)>(varReplac
er1, r1)


====================== CryptoAnalysis Summary ==========================
        Number of CrySL rules: 84
        Number of Objects Analyzed: 3

        CryptoAnalysis found the following violations. For details see description above.
        RequiredPredicateError: 1
        ConstraintError: 1
=====================================================================
[main] INFO crypto.analysis.CryptoScanner - Static Analysis took 0 seconds!
[main] INFO crypto.HeadlessCryptoScanner - Analysis finished in 5.900 s
```

Figure 4.2: Result of the analysis of Listing 1.1 by CogniCrypt$_{\text{SAST}}$ in command-line.

4.3 DSL Comparison

CrySL [SKM19] and MARK [coda] are the DSLs used by CogniCrypt$_{\text{SAST}}$ [SKM19] and Codyze [coda], respectively, to specify the correct usages of cryptographic APIs. This section aims to compare the DSLs on a theoretical and practical level fairly. Therefore, we performed a systematic review [Kit04] of ways to compare DSLs. We searched on different platforms like Google[6], GoogleScholar[7], IEEE explore digital library[8], and ResearchGate[9] for related articles and read the relevant ones, but there is no publication that compares DSLs in the same domain in terms of expressiveness. Rather, we found similar yet unrelated papers. For example, Cuadrado et al. compared two DSLs with the same syntax but different implementation techniques (internal and external) in their paper [ASCIM14]. But that is not suitable for our DSLs since MARK and CrySL are both external DSLs and this comparison is not useful. The other papers discussed comparing DSLs and GPLs (General Purpose Languages) to test program comprehension via user studies, which is not the focus of our study [KMC12] or measuring usage simplicity in DSLs and GPLs [KOM^{+}10], or comparisons of ways to define DSLs [PP08], which are all unrelated to our research. Several studies tested usability [BAG12] [BAGB11a] [BAGB11b] [HPvD09], which requires a user study and is beyond the scope of this thesis.

Thus, we propose translating CrySL rules to MARK rules, and vice versa, to explore their expressiveness. By translating one language to another and vice versa in a particular domain, such as defining correct usages of cryptographic APIs, we are expressing elements of one language in another while preserving the meaning, which demonstrates the expressiveness in that particular domain. There will be four possible scenarios in our case of translation. If we can translate MARK rules into CrySL and CrySL rules into MARK, then MARK and CrySL are equally expressive when defining rules for cryptographic APIs. However, if CrySL rules can

[6]https://www.google.com
[7]https://scholar.google.com
[8]https://ieeexplore.ieee.org
[9]https://www.researchgate.net

be expressed by MARK, but MARK rules cannot be translated into CrySL, then MARK is more expressive in specifying cryptographic API rules. Additionally, if MARK rules could be expressed by CrySL, and CrySL rules could not be expressed by MARK, then CrySL is more expressive than MARK in defining rules for cryptographic APIs. Finally, if CrySL rules and MARK rules cannot be translated into each other, then they express different things in defining rules for cryptographic APIs.

Prior to comparing the DSLs, we provide a short explanation of MARK and CrySL. We will then compare them practically and translate the rules, and finally, we will compare them theoretically.

4.3.1 CrySL

CrySL is a domain-specific language (DSL) that allows cryptography experts to describe secure usage of their crypto APIs in a lightweight special-purpose syntax [SKM19]. The compiler is built upon Xtext [xte], an open-source framework for developing domain-specific languages. Based on the CrySL grammar, Xtext provides parsing, type checking, and syntax highlighting capabilities. A CrySL rule is a simple text file and can be written in any text editor. Eclipse can be used to create CrySL rules with syntax correction by installing the CrySL editor.

Currently, CrySL rules have been developed by crypto experts of CogniCrypt developers for JCA, Bouncy Castle [bc], a Java library that extends Java Cryptographic Extension (JCE)[10], Bouncy Castle JCA [bc], Bouncy Castle provider for the JCA, and Tink [tk], an open-source easy to use and secure cryptographic library made by Google to reduce API misuses [api].

A CrySL rule is written for each class and consists of 6 mandatory and 3 optional parts. We introduce the semantics of CrySL rules by using the CrySL rule for javax.crypto.KeyGenerator in Listing 4.1 as an example. We further use part of the CrySL rule for javax.crypto.spec.PBEKeySpec (Listing 4.2) to elaborate some infrequently used terms.

[10]JCE is an API that provides a framework that allows Java developers to implement security features [jce].

Listing 4.1: KeyGenerator CrySL rule from JCA API [api].

```
1  SPEC javax.crypto.KeyGenerator
2
3  OBJECTS
4    int keysize;
5    java.security.spec.AlgorithmParameterSpec params;
6    javax.crypto.SecretKey key;
7    java.lang.String algorithm;
8    java.security.SecureRandom random;
9
10 EVENTS
11   g1: getInstance(algorithm);
12   g2: getInstance(algorithm, _);
13   Get := g1 | g2;
14
15   i1: init(keysize);
16   i2: init(keysize, random);
17   i3: init(params);
18   i4: init(params, random);
19   i5: init(random);
20   Init := i1 | i2 | i3 | i4 | i5;
21
22   gk1: key = generateKey();
23   GenKey := gk1;
24
25 ORDER
26   Get, Init?, GenKey
27
28 CONSTRAINTS
29   algorithm in {"AES", "HmacSHA256", "HmacSHA384", "HmacSHA512"};
30   algorithm in {"AES"} => keysize in {128, 192, 256};
31
32 REQUIRES
33   randomized[random];
34
35 ENSURES
36   generatedKey[key, algorithm];
```

4.3.1.1 Mandatory sections

On the first line (1), there is SPEC that indicates the class name. The OBJECTS (Line 4 to 4) are parameters or return values of methods in the EVENTS section. All methods that may contribute to the successful use of an object of the CrySL rule are described in the EVENTS section. Each method pattern is represented by a label in the EVENTS section (e.g., `Get` in Line 13). When different patterns of the same method are possible, only one of them based on the implementation will be chosen in the ORDER section. For example, in Listing 4.1 Lines 11 and 12 show two patterns of `getInstance` method: `g1` with only one parameter, and `g2` with two parameters. CrySL uses aggregates to represent the disjunction of labels (e.g., `Get` in line 13).

The ORDER section defines a usage pattern in the form of regular expression for methods in the EVENTS section. An example would be the `getInstance` call (Line 26) followed by the `init` call (which is optional), followed by a call to the `generateKey` method in Listing 4.1.

The CONSTRAINTS show constraints for objects. For instance, Line 29 maintains that `algorithm` must be one of the following: AES, HmacSHA256, HmacSHA384, HmacSHA512. The Line 30 indicates that if the `algorithm` is AES, then the `keySize` must be one of 128, 192, or 256.

Assuming that the object is used appropriately, meaning that all limitations in the CONSTRAINTS section and usage patterns in the ORDER section are considered, the ENSURES section defines what a class predicates. Line 36 of the keyGenerator CrySLrule demonstrates that using this class properly ensures the generation of a `key` with a specific `algorithm`.

CrySL allows us to define a method-event pattern using the *after* keyword (Line 20). For example, in Line 20 of Listing 4.2, `speccedKey` predicate is created if the appropriate method is called. Thus, the PBEKeySpec class ensures the generation of a key after a call to the appropriate constructor with a specified `keyLength`.

Listing 4.2: Part of CrySL rule for javax.crypto.spec.PBEKeySpec from JCA ruleset [api].

```
1  SPEC javax.crypto.spec.PBEKeySpec
2
3  OBJECTS
4      ...
5  FORBIDDEN
6    PBEKeySpec(char[]) => Con;
7    PBEKeySpec(char[],byte[],int) => Con;
8
9  EVENTS
10   Con: PBEKeySpec(password, salt, iterationCount, keyLength);
11
12   ClearPass: clearPassword();
13 ...
14 CONSTRAINTS
15   iterationCount >= 10000;
16   neverTypeOf[password, java.lang.String];
17   notHardCoded[password];
18 ...
19 ENSURES
20   speccedKey[this, keyLength] after Con;
21
22 NEGATES
23   speccedKey[this, _] after ClearPass;
```

4.3.1.2 Optional sections

Apart from mandatory sections that each CrySL rule must have, some CrySL rules may contain optional sections. The FORBIDDEN section involves calls to the methods that are insecure. The PBEkeySpec constructor call must consume a salt to generate a more secure key, whereas the signatures of the constructors in Lines 6 and 7 do not consume a salt; therefore, those calls are insecure and forbidden. After the `clearPassword` call, a PBEKeySpec object made by a constructor is no longer valid. CrySL allows invalidating a current predicate in the NEGATES field. As shown in Line 23, after `clearPassword` is called, the PBEkeySpec object is no longer effective. The predicates that one rule ENSURES can be used in the REQUIRES section of another rule. For example, Line 33 requires that the salt be generated from a random seed.

CrySL developers have added some simple built-in auxiliary functions to make it more expressive. For example, in Listing 4.2, the function `neverTypeOf` in Line 16 states that the password should never be of type `String` and Line 17 states that the password should not be hard-coded. [Kr0] fully described all of the built-ins in CrySL.

4.3.2 MARK

MARK is a domain-specific language for writing rules to specify the correct usage patterns of APIs or libraries [coda]. It stands for "Modellierungssprache fuer Anforderungen und Richtlinien der Kryptografie" (Modeling Language for Cryptography Requirements and Guidelines). MARK's grammar is written using Xtext. MARK's Eclipse plugin project [mar] provides several features for MARK language, including a language server for using MARK in IDEs with LSP support, an Xtext generator that creates Crymlin/Gremlin from MARK files, and an Eclipse plugin for syntax highlighting, auto-completion, and code correction [mar]. Unfortunately, there are no further details available regarding the use of MARK on other IDEs with LSP support or the creation of Crymlin/Gremlin out of MARK files. In order to fully grasp these concepts, it is necessary to conduct extensive research on the implementation of MARK, which is beyond the scope of this thesis.

Currently, there are MARK policies made by Codyze's developers for the Botan [bot], a C++ cryptography library, Bouncy Castle and Jackson [jac], a high-performance JSON processor for Java libraries for cryptography in Java. After reviewing the MARK ruleset on Codyze's Github page, we discovered that the Bouncy Castle MARK ruleset contains rules for the JCA API. However, we will continue to refer to it as the Bouncy Castle MARK ruleset.

Writing MARK rules for a library necessitates a thorough knowledge of the API and class structure of the library. MARK policies are separated into two parts that we will discuss in separate sections; Entities (see Section 4.3.2.1) and Rules (see Section 4.3.2.2). Rules determine the correct usage of the entities. Entities are an abstract grouping of API functions. Entity contains three parts:

- *Name*, the name of the rule.
- *Ops*, set of operations (op).
- *Var*, variables that refer to function arguments or return values.

Here we will provide a brief overview of the entity and rule parts of MARK policies.

4.3.2.1 Entity

To describe entities, we consider Listing 4.3 which shows the entity part of the MARK policy for javax.crypto.KeyGenerator and Listing 4.4 which is a part of MARK entity file for java.security.SecureRandom to describe a less frequently used optional keyword, `forbidden`, in MARK.

Listing 4.3: MARK entities for javax.crypto.KeyGenerator of Bouncy Castle ruleset [codb]

```
1 package java.jca
```

```
 2
 3 entity KeyGenerator {
 4
 5     var algorithm;
 6     var provider;
 7
 8     var keysize;
 9     var random;
10     var params;
11     var key;
12
13     op instantiate {
14         javax.crypto.KeyGenerator.getInstance(algorithm : java.lang.String);
15         javax.crypto.KeyGenerator.getInstance(
16             algorithm : java.lang.String,
17             provider : java.lang.String | java.security.Provider
18         );
19     }
20
21     op init {
22         javax.crypto.KeyGenerator.init(keysize : int);
23         javax.crypto.KeyGenerator.init(
24             keysize : int,
25             random : java.security.SecureRandom
26         );
27         javax.crypto.KeyGenerator.init(random : java.security.SecureRandom);
28         javax.crypto.KeyGenerator.init(params : java.security.spec.
    AlgorithmParameterSpecs);
29         javax.crypto.KeyGenerator.init(
30             params : java.security.spec.AlgorithmParameterSpec,
31             random : java.security.SecureRandom
32         );
33     }
34
35     op generate {
36         key = javax.crypto.KeyGenerator.generateKey();
37     }
38 }
```

The first step is to define model relevant classes as MARK entities. Only those classes which contain the relevant data or function must be identified as MARK entities. Even though many programming languages classes can be combined in an abstract MARK entity in several cases, it can at first be easier to map classes explicitly into entities. For instance, KeyGenerator class to entity KeyGenerator. Developers can freely choose the name of the entity [coda].

The second step is to define ops and variables. An op is a semantically equivalent or related group of functions, methods, or constructors, provided as completely qualified signatures. The most common ops in cryptographic libraries are, instantiate, initialize, update, finalize and reset. For example, in Listing 4.3, we have instantiate op, which shows the `getInstance` methods with different possible parameters (Lines 14, 15). The type of each op's parameters with a name is specified, but we can also have unnamed and untyped parameters (described with "_"). These parameters do not play any role in the definition of rules. Named parameters must also be declared as entity variables using the `var` keyword (Lines 5 to 11) [coda].

The third step, which is optional, is to blacklist the forbidden ops. In some cases, functions that are deprecated or established to be insecure should not be used in the program. The `forbidden` keyword in MARK indicates that the use of the following functions is insecure. For example, Lines 7 and 8 of Listing 4.4 are forbidden calls when using the SecureRandom class because they do not respect Bouncy Castle as a provider [coda].

Listing 4.4: Part of MARK entities for java.security.SecureRandom from Bouncy Castle ruleset [codb].

```
1      ...
2       op instantiate {
3          ...
4          java.security.SecureRandom.getInstanceStrong();
5
6          // forbidden calls because they don't respect BC as provider
7          forbidden java.security.SecureRandom();
8          forbidden java.security.SecureRandom(
9              seed : byte[]
10         );
11     }
12     ...
```

4.3.2.2 Rule

After defining entities, we can start writing rules. MARK rules apply to instances of entities and specify the conditions that must be met in these instances. Instances of MARK may correspond to actual objects but might also be abstract functions or variables in non-object-oriented languages and static methods. Listing 4.5 displays one of the MARK rules associated with the KeyGenerator class from the Bouncy Castle MARK files included in Codyze's Github repository.

Every rule has a unique name across all MARK files loaded into Codyze (Line 1). `Using` keyword (Line 2) initiates the declaration of instances of the MARK entities, here KeyGenerator and javax.crypto.Mac instances, and `ensure` shows the condition (Line 8). A violation of the condition will result in a finding with a message indicated by the `onfail` identifier (Line 11).

In some rules, some preconditions must be met before the main condition is evaluated. If they fail, the main condition will not be evaluated, and the rule will not return any results. Preconditions are declared by the `when` keyword (Line 5). Mark includes several built-in functions that can be used as predicates in conditions and preconditions (e.g., Line 7). They are called during MARK rule evaluation and operate over their input arguments (usually MARK objects or constants) and the evaluation context. By convention, built-ins should begin with "_". If a built-in fails, it will return an Error object which evaluates to not applicable, i.e., neither true nor false.

In the following rule example (Listing 4.5), once the preconditions are satisfied, i.e., if the Mac algorithm is AESCMAC (Line 6), and the variable Mac key equals the KeyGenerator key (Line 7), then the rule condition must be met, that is, the KeyGenerator's key must be greater than or equal to 128 (Line 10). Otherwise, the InsufficientCMACKeyLength error will be thrown.

Listing 4.5: A MARK rule associated with javax.crypto.KeyGenerator class from Bouncy Castle ruleset [codb].

```
rule ID_5_3_02_CMAC_Keygen {
    using
        Mac as m,
        KeyGenerator as kg
    when
        m.algorithm in ["AESCMAC"]
        && _is(m.key, kg.key)
    ensure
        _is(m.key, kg.key)
        && kg.keysize >= 128
    onfail
        InsufficientCMACKeyLength
}
```

In this example, the condition `_is(m.key, kg.key)` is mentioned in both the `when` and `ensure` sections. It might be a mistake since if it is true in the `when` section, then it is also true in the `ensure` section, so there is no need to verify it again. The developers of CODYZE did not elaborate on the matter further; therefore, we have opened an issue on CODYZE's Github repository[11].

4.3.3 Practical comparison of CrySL and MARK

This section aims to compare the DSLs on a practical level. In order to accomplish this, we proposed translating CRYSL rules to MARK and vice versa. Since there are MARK and CRYSL rules for the Bouncy Castle JCA API for Java, we first translate Bouncy Castle JCA rules from CRYSL to MARK and vice versa. As stated before, there are no MARK rules specified for the JCA library, which is the primary cryptography API for Java applications [SNB16]. In this regard, it is worthwhile to translate CRYSL JCA rules in MARK language and compare their functionality and result of their analyses on the same Java cryptography examples. So at the end we will have three translations, CRYSL Bouncy Castle JCA to MARK, MARK Bouncy Castle JCA to CRYSL and CRYSL JCA to MARK. Source codes for the translated rules are available in Appendices A.1.

We developed the following translation based on the information we obtained from reviewing existing MARK and CRYSL rules, CRYSL paper [SKM19] and CODYZE's documentation page [coda]. We will employ these translated rules in the evaluation of CODYZE and COGNICRYPT$_{SAST}$ (cf. Chapter 5).

4.3.3.1 Setup

We used the same virtual machines we explained in Section 4.2 to translate rules. On the first machine with Java 8, we installed the CRYSL Eclipse plugin and on the second machine with Java 11, we installed the MARK Eclipse plugin. We used the CRYSL rules from the Crypto-API-rules Github repository master branch [api] to the commit 1dbad34[12] and MARK rules from CODYZE Github repository main branch [codb] to the commit 8c74a13[13], to the date of October 31st, 2021.

[11]https://github.com/Fraunhofer-AISEC/codyze/issues/400
[12]https://github.com/CROSSINGTUD/Crypto-API-Rules/commit/1dbad342a46c47df62e891fc25f46944972d9e18
[13]https://github.com/Fraunhofer-AISEC/codyze/commit/8c74a13be2385b79875990c5e36ed67b39579662

4.3.3.2 Translation from CrySL to MARK

There are 47 Bouncy Castle JCA and 49 JCA CrySL rules on Github [api]. Each CrySL rule represents a class. While translating each rule to MARK, we put entities (ops and vars) in one file and the rules on another file. As an example, there are two MARK files for the KeyGenerator class, the KeyGenerator.mark file containing the entities of the class (see Listing 4.6), and the Rule_KeyGenerator.mark file containing the rules, such as order, constraints, etc (see Listing 4.7). Listings 4.6 and 4.7 show KeyGenerator MARK rule translated from its CrySL rule from JCA API in Listing 4.1 [api].

The objects (Lines 4 to 8) of the CrySL rules (Listing 4.1) are directly translated to vars (Lines 5 to 9). MARK variables are not typed. The type of variables will be specified in the ops. As an example, in CrySL the type of `keySize` is `int`. In translation to MARK, the type of `keysize` is specified in the operation where it is being used (Line 22). Return variables will not be typed in MARK (Line 34). Furthermore, the functions in the EVENTS section have exactly the same translation and grouping in the corresponding MARK rule (Lines 11 to 35) without any labels. We have three kinds of operations in this case, instantiate (Line 11) for `getInstance` methods, init (Line 19) for `init` methods and generate (Line 33) for `generateKey` method.

Listing 4.6: Entity part of translated KeyGenerator CrySL rule to MARK from JCA API.

```
package java.jca

entity KeyGenerator {

    var keySize;
    var params;
    var key;
    var algorithm;
    var random;

    op instantiate {
        javax.crypto.KeyGenerator.getInstance(algorithm : java.lang.String);
        javax.crypto.KeyGenerator.getInstance(
            algorithm : java.lang.String,
            _
        );
    }

    op init {
        javax.crypto.KeyGenerator.init(keySize : int);
        javax.crypto.KeyGenerator.init(
            keySize : int,
            random : java.security.SecureRandom
        );
        javax.crypto.KeyGenerator.init(random : java.security.SecureRandom);
        javax.crypto.KeyGenerator.init(params : java.security.spec.AlgorithmParameterSpecs);
        javax.crypto.KeyGenerator.init(
            params : java.security.spec.AlgorithmParameterSpec,
            random : java.security.SecureRandom
        );
    }

    op generate {
        key = javax.crypto.KeyGenerator.generateKey();
    }
}
```

Listing 4.7 shows the translated rules from ORDER, CONSTRAINTS, and REQUIRES sections of the KeyGenerator CrySL rule. The translation of CrySL ORDER section is the first rule in Listing 4.7 Line 3, which uses the same regular expression as the CrySL rule (Line 26). The CONSTRAINTS in CrySL rule are the constraints over the `algorithm` and the `keysize` (Lines 29 and 30) which direct translations are the second (Line 15) and third (Line 56) rules of Listing 4.7. MARK's analyzer checks all the rules, and if the ensure part is violated, an error is thrown even if the class has not been used in the analyzed project. We avoided this mistake by adding a when clause to all the MARK rules (except the order rules) to determine if a variable corresponding to that rule exists, using the built-in method `_has_value`. The rule should not be tested if the variable is not present. The built-in function `_has_value(a)` checks whether a given MARK variable, `a`, has a CPG-node which indicates that it exists as part of the program [codb]. In the Line 19, for instance, the analyzer checks whether the KeyGenerator's `algorithm` variable is present; otherwise, this rule is not executed.

Listing 4.7: Rule part of translated KeyGenerator CRYSL rule to MARK from JCA API.

```
package java.jca

rule JCAProvider_KeyGenerator_order{
  using
    KeyGenerator as kg
  ensure
    order
      kg.instantiate(),
      kg.init()?,
      kg.generate()
  onfail
    InvalidOrderOfKeyGenerator
}
//constraints
rule JCAProvider_KeyGenerator_algorithm{
  using
    KeyGenerator as kg
  when
    _has_value(kg.algorithm)
  ensure
    kg.algorithm in ["AES", "HmacSHA224", "HmacSHA256", "HmacSHA384", "HmacSHA512"]
  onfail
    InvalidSecretKeyAlgorithmOfKeyGenerator
}
rule JCAProvider_KeyGenerator_keysize{
  using
    KeyGenerator as kg
  when
    kg.algorithm in ["AES"] && _has_value(kg.keySize)
  ensure
    kg.keySize in [128, 192, 256]
  onfail
    InvalidKeySizeOfKeyGenerator
}
//requires
rule JCAProvider_KeyGenerator_randomized{
  using
    KeyGenerator as kg,
    SecureRandom as sr
  when
    _has_value(kg.random)
  ensure
    _is(kg.random, sr)
  onfail
    NotRandomranGenOfKeyGenerator
}
```

The ENSURES and REQUIRES sections of CRYSL bind two rules together, so when we translate them to MARK, we include all classes that contain the same predicate in the REQUIRES and ENSURES sections of their CRYSL rule as instances of the corresponding MARK rule. We have placed the translation for this binding in the MARK rule file of the class which used this predicate in its REQUIRES section. For instance, KeyGenerator CRYSL rule ENSURES `generatedKey[key, algorithm]` (Line 36) which means if the KeyGenerator is used properly, then it guarantees a key with the corresponding algorithm.

This predicate appears in the Mac[14] and Cipher[15] CrySL rule's REQUIRES section, so the MARK rules are placed within the Mac and Cipher MARK rule files.

Listing 4.8 illustrates the translation of the `generatedKey` predicate in the Mac MARK rule file. The Mac CrySL rule specifies this predicate as `generatedKey[key, _]` in its REUIRES section. This predicate indicates that the key variable used in creating the Mac instance must be generated appropriately using KeyGenerator or any other rule that has `generatedkey[javax.crypto.SecretKey,java.lang.String]` in their ENSURES section. The second parameter in the predicate, `generatedKey[key,_]`, is an underscore, which implies that the algorithm used to generate the key does not matter.

As well as KeyGenerator, there are other rules in CrySL that guarantee this predicate, such as SecretKeySpec, KeyStore and SecretKeyFactory. Therefore, we include all of them as instances in the Mac MARK rule (Line 4 to 7).

We then provide a precondition to guarantee the existence of the Mac instance by checking that a key variable for this instance exists in the program (Line 9).

According to the Mac CrySL rule, it does not matter what algorithm is being used to generate the key, therefore, we only need to confirm that the key is generated by one of the mentioned classes. To accomplish this, we use the built-in function `_is` (Line 11). The built-in `_is(a,b)` checks if two objects are equal [codb]. InvalidGeneratedKeyOfMac error message on Line 13 may appear if the program does not conform to this rule.

Listing 4.8: Part of translated Mac CrySL rule to MARK from JCA API.

```
 1 rule JCAProvider_Mac_generatedkey{
 2   using
 3     Mac as m,
 4     KeyStore as ks,
 5     SecretKeyFactory as skf,
 6     KeyGenerator as kg,
 7     SecretKeySpec as sks
 8   when
 9       _has_value(m.key)
10   ensure
11     _is(m.key, sks) || _is(m.key, skf.key)|| _is(m.key, kg.key) || _is(m.key, ks
       .key)
12   onfail
13     InvalidGeneratedKeyOfMac
14 }
```

In CrySL it is possible to specify a method-event pattern using the keyword *after*. For example, in Listing 4.2 of PBEKeySpec CrySL rule, we have the following predicate: `speccedKey[this, _] after ClearPass` (Line 23), which indicates that after using the `clearPassword` method, the speccedkey generated by PBEKeySpec is no longer valid. Since MARK does not provide this feature, it is not possible to translate the NEGATES section of the CrySL rules to MARK.

[14]https://github.com/CROSSINGTUD/Crypto-API-Rules/blob/master/BouncyCastle-JCA/src/Mac.crysl
[15]https://github.com/CROSSINGTUD/Crypto-API-Rules/blob/master/BouncyCastle-JCA/src/Cipher.crysl

We can define forbidden methods in MARK, which is equivalent to the FORBIDDEN section of the CrySL rules. The forbidden methods are located in the entity part of the MARK rule. Listing 4.9 shows a part of the translation of the PBEKeySpec CrySL rule to MARK entities. Lines 13 and 14 show the translation of the FORBIDDEN section of PBEKeySpec CrySL rule (Lines 6 and 7) to MARK. These lines forbid the use of the PBEKeySpec method with 1 or 3 parameters; in another way, the method PBEKeySpec should only be used with four parameters; otherwise, it is considered insecure.

Listing 4.9: Part of Entity part of translated PBEKeySpec CrySL rule to MARK from JCA API.

```
1  package java.jca
2
3  entity PBEKeySpec {
4      ...
5    op instantiate {
6
7      javax.crypto.spec.PBEKeySpec(
8        password : char[],
9        salt : byte[],
10       iterationCount : int,
11       keyLength : int
12     );
13     forbidden javax.crypto.spec.PBEKeySpec(_: char[]);
14     forbidden javax.crypto.spec.PBEKeySpec(_: char[],_: byte[],_: int);
15   }
16     ...
17 }
```

Discussion

CrySL's compiler provided auto-completion, syntax checking, and case sensitivity. During the translation of CrySL rules into MARK, some language features could not be translated. There is, for example, no such method in MARK where we can retrieve an element of an array. There is `elements` in CrySL, but no equivalent method in MARK. However, this is not a significant issue because this method can easily be added as a built-in method in MARK. The same holds true for other built-in functions as well, such as noCallTo, callTo. The three built-in functions, `mode`, `alg`, and padding are available via `_split` in MARK.

There is a keyword, `after` that has been used in the sections REQUIRES, ENSURES, and NEGATES. This keyword cannot be translated into MARK since there is no equivalent feature to `after` in MARK. Moreover, we cannot include method calls in conditions or preconditions. In such cases, we translated the rules without considering the `after` part. For instance, the Line 20 in Listing 4.2 ensures that a speccedKey with a certain keyLength is generated after a call to the constructor specified in the EVENTS section. This rule was translated without translating the `after` keyword, which is reasonable since the constructor call is the final and only call in the order section and will be called when generating a PBEKeySpec instance. When the method following the keyword `after` is not a final method, the rules are not equal, and the MARK translation is incomplete. Further, it is not possible to translate the NEGATES section of CrySL rules to MARK, as there is no equivalent feature to `after` in MARK and this section depends entirely on this keyword.

4.3.3.3 Translation from MARK to CrySL

There are MARK rules for 60 classes Bouncy Castle JCA API on Github[16]. However, we could only translate 57 of them cause three classes require Java 9 or higher versions, which are javax.crypto.spec.ChaCha20ParameterSpec, java.security.spec.XECPrivateKeySpec and java.-security.SecureRandomParameters. Our first step was to translate MARK entities to CrySL, then we added the written MARK rules from rule files to the CrySL rules. In MARK, interfaces, such as PrivateKey and PublicKey, should also be added, as a rule; otherwise, they are not recognized by MARK. The same is not true in CrySL, but we translated them too as a matter of fairness.

The vars (Lines 5 to 11) and ops (Lines 13 to 13) of KeyGenerator entity MARK file in Listing 4.3 translate to OBJECTS and EVENTS in CrySL. Listing 4.10 shows the KeyGenerator CrySL rule that is translated from the MARK rule. OBJECTS and EVENTS are shown on Lines 2 and 14 respectively.

[16]https://github.com/Fraunhofer-AISEC/codyze/tree/main/src/dist/mark

Listing 4.10: Translated KeyGenerator MARK rule[17] to CrySL from Bouncy Castle JCA API.

```
SPEC javax.crypto.KeyGenerator
OBJECTS
  java.lang.String algorithm;
  java.security.Provider provider;
  java.lang.String providerS;
  int keySize;
  java.security.SecureRandom random;
  java.security.spec.AlgorithmParameterSpec params;
  javax.crypto.SecretKey key;

//FORBIDDEN
//  getInstance(algorithm) => Instances;

EVENTS
  ins1 : getInstance(algorithm, provider);
  ins2 : getInstance(algorithm, providerS);
  //ins3 : getInstance(algorithm);
  Instances := ins1| ins2;

  i1 : init(keySize);
  i2 : init(keySize, random);
  i3 : init(random);
  i4 : init(params);
  i5 : init(params, random);
  Inits := i1| i2| i3| i4| i5;

  g : key = generateKey();
  GenK := g;

ORDER
  Instances?, Inits?, GenK?

CONSTRAINTS
  providerS in {"BC"};
  instanceOf[provider, org.bouncycastle.jce.provider.BouncyCastleProvider];
  keySize >= 128;

//REQUIRES
//*

ENSURES
  generatedKey[key];
  generatedKey[key, algorithm];
```

There is no single file containing all the MARK rules that are related to a single class in the Github. Among all the MARK rule files for Bouncy Castle JCA API, there were four rules containing the KeyGenerator class (Listing 4.11)

Listing 4.11: MARK rules for KeyGenerator of Bouncy Castle JCA API.

```
rule BouncyCastleProvider_KeyGenerator {
    using
        KeyGenerator as kg
    ensure
        _has_value(kg.provider)
        && (
```

[17]https://github.com/Fraunhofer-AISEC/codyze

```
            kg.provider == "BC"
            || _is_instance(kg.provider, "org.bouncycastle.jce.provider.BouncyCastleProvider")
        )
    onfail
        InvalidProvider_KeyGenerator
}
rule ID_5_3_02_GMAC {
    using
        Mac as m,
        KeyGenerator as kg,
        SecretKeySpec as sks,
        SecretKeyFactory as kf
    when
        m.algorithm in ["AES-GMAC"] && ( _is(m.key, kg.key)
        || ( _is(m.key, kf.outkey) && _is(kf.keyspec, sks) ) )
    ensure
        // find a keygenerator of sufficient size
        kg.keysize >= 128
        || (_has_value(sks.len) && sks.len >= 128)
        || (!(_has_value(sks.len)) && _has_value(sks.key) && _length(sks.key) >= 16)
    onfail
        InsufficientGMACKeyLength
}
rule ID_5_3_02_CMAC_Keygen {
    using
        Mac as m,
        KeyGenerator as kg
    when
        m.algorithm in ["AESCMAC"]
        && _is(m.key, kg.key)
    ensure
        // find a keygenerator of sufficient size
        _is(m.key, kg.key)
        && kg.keysize >= 128
    onfail
        InsufficientCMACKeyLength
}

rule ID_5_3_02_HMAC_Keygen {
    using
        Mac as m,
        KeyGenerator as kg
    when
        m.algorithm in ["HMACSHA256", "HMACSHA512/256", "HMACSHA384", "HMACSHA512", "HMACSHA3-256", "HMACSHA3-384", "HMACSHA3-512"]
        && _is(m.key, kg.key)
    ensure
        // find a keygenerator of sufficient size
        (
            _is(m.key, kg.key)
            && kg.keysize >= 128
        )
        || (
             _is(m.key, kg.key)
             && kg.algorithm == m.algorithm
        )
    onfail
        InsufficientHMACKeyLength
        }
```

In order to avoid complexity, we will only explain the parts of the rule relevant to KeyGenerator, although the explanations apply to other parts, as well.

According to the first rule (Line 1), when using the KeyGenerator class, the `getInstance` method must pass the `provider` parameter, which should either be the string "BC" or of type org.bouncycastle.jce.provider.BouncyCastleProvider. Lines 34 and 35 of Listing 4.10 present the translation of this rule, in which we define constraints on the variables, `providerS` and `provider`. Because the existence of a provider is required, we removed `getInstance` method patterns in the EVENTS section that did not include a provider as a parameter (Line 17) and created an aggregation of the rest (Line 18). Moreover, the first rule suggests that this pattern of `getInstance` method (Line 17) should not be used, which implies that the usage of this pattern is forbidden. We therefore added this method pattern to the FORBIDDEN section, but an error occurred as there is an issue with the CrySL grammar that currently allows us only to add constructor calls to the FORBIDDEN section. As a result, we commented the FORBIDDEN section. Unlike CrySL, a variable may have multiple types in MARK, which is why the variable `provider` in MARK is divided into two variables in CrySL, `provider` of type java.security.Provider, and `providerS` of type java.lang.String.

The second rule (Line 13) states that whenever a Mac object is used with the AES-GMAC algorithm and a key generated by the KeyGenerator, the `keySize` must be greater than or equal to 128. In CrySL we are not able to write such clauses with those premises; in other words, we cannot add constraints to a rule's OBJECTS from another rule. The solution would be to make a constraint on that variable, `keySize`, in the CONSTRAINTS section of the KeyGenerator CrySL rule (Line 36). This solution is logical since BSI (Bundesamt Für Sicherheit in der Informatikstechnik) has demonstrated that a `keySize` less than 128 bits is considered insecure [BSIa].

As indicated by the third rule (Line 30), if the Mac object is used with the AESCMAC algorithm and a key generated by KeyGenerator, `keySize` must be at least 128 bits. Translation of this rule is the same as for the second rule we previously translated.

If a Mac object uses the algorithms specified in the Line 50 of the fourth rule (Line 45), and the key was generated by KeyGenerator, then either the `keySize` should be greater than or equal to 128 or the algorithm used in Mac and KeyGenerator are the same. We have already implemented the first part in the second and third rules. However, for the second part of the rule, we added a predicate to the ENSURES section of the KeyGenerator rule that guarantees a key with a specific algorithm, and we used this predicate in the REQUIRES section of the Mac rule.

The CrySL language was designed to define a usage pattern for each rule in the ORDER section, even if the class only contains a constructor call. A rule will not function without a usage pattern. Therefore, we have included the order in a regular expression that includes all the aggregates from the EVENTS section in such a way that there is no correct order established (Line 31), except for the rules for which we had a defined order in the MARK version. Thus, the translation of MARK rules into CrySL may not be syntactically identical; however, they convey the same semantic information.

Discussion

The MARK compiler does not support some of the features that the documentation [coda] describes. The language lacks case-sensitivity and auto-completion, and it does not recognize non-defined variables but provides syntax checking. In MARK we have to write an entity file for every interface or class that we intend to use, even if there is only a package and entity name. For example, PrivateKey or PublicKey interfaces. The MARK grammar can detect if a class used in a rule is not defined in the folder in which all MARK rules are stored.

As mentioned previously, a variable of one class cannot be constrained by a rule of another class in CrySL. As a result, we had to add the constraint in the CONSTRAINTS section without considering the precondition, so the rule for CrySL was not exactly the same as the MARK rule. For instance, the translation of the MARK rule on Line 56 of Listing 4.11 is the Line 36 in the CrySL rule in Listing 4.10. In some cases, there were different constraints for the same variables under different preconditions. For example, the constraint for the `tLen` (the authentication tag length) variable of the GCMParameterSpec (gcm) class are `gcm.tLen >= 64` and `gcm.tLen >= 96` in 2 different rules with different preconditions. Our solution was to add the conjunction (`gcm.tLen >= 96`).

MARK and CrySL do not have fully equivalent built-in functions. The built-in functions `_is_instance`, and `_length` in MARK have the same functionality as `instanceOf`, and `length` in CrySL, respectively. There are MARK built-in functions that are not used in the current MARK rules but are defined in the Codyze Github repository [codb], namely: `_direct_eog_connection`, `_eog_connection`, `_inside_same_function`, `_get_code`, `_now`, `_receives_value_from`, `_split_disjoint`, `_split_match_unordered`, `_starts_with`, `_year` [codb].

4.3.4 Theoretical comparison of CrySL and MARK

MARK and CrySL DSLs are both designed to specify the correct use of cryptographic APIs. CrySL is intended to be used to write rules only for Java cryptographic APIs. Nevertheless, as we discussed in Section 4.3.2, MARK enables us to write rules for C and C++, as well as Java since the MARK parser is designed to parse MARK rules for C++ or C cryptographic APIs.

The DSLs MARK and CrySL are both external DSLs. Internal DSLs are those that are implemented within a general-purpose language as the host. An external DSL is a language that is parsed independently of the host [Fow10]. CrySL was designed specifically for cryptography experts and the existing CrySL rules were written by cryptography experts of the CrySL developers [Kr0]. The current MARK rules are based on the BSI [BSIa] TR-02102-1 version 2019-01 guideline for using cryptographic APIs.

CrySL and MARK have different syntaxes; however, their semantics are similar in some cases, for example, vars in MARK are equivalent to OBJECTS in CrySL, or ops are equivalent to EVENTS in CrySL. Since there is limited information about MARK, we cannot be certain of similarities and differences in semantics. Therefore, we were required to perform a practical comparison in which we converted MARK policies to CrySL and vice versa in Section 4.3.3.

This also applies to the built-in functions. In order to determine how close we could get to expressing one language in another, we are required to ensure that each DSL contains sufficient built-in functions to enable us to translate rules written in the other DSL into this one. MARK has several built-in functions, some of which are used by the current MARK rules for Java cryptographic APIs. However, some other MARK built-in functions are listed in the Codyze Github repository [codb] in the builtin folder but have not been used in the current MARK rules (in the path src\main \java\de\fraunhofer\aisec\codyze\crymlin\builtin). Further investigation of these functions was necessary to evaluate the expressiveness of the MARK DSL. In the end of translation, only the built-in functions of MARK, which are used by current MARK rules, proved helpful. Here is an example of a built-in function that was not helpful: the `_starts_with(String str, String start)` function that returns true if `str` starts with the `start` value.

The results of translations are discussed in Section 4.3.3. We determined that CrySL and MARK express different things, but there are also concepts that they both can express. For example, specifying the proper algorithm in KeyGenerator (Section 4.3.3.3). Furthermore, some language features of one language cannot be translated into another. As an example, MARK has no equivalent of the NEGATES section of CrySL, and in CrySL, it is not possible to restrict a parameter of one class from another class. Furthermore, we understood that the built-ins had some overlap. For example, `_is_instance` built-in in the MARK has its equivalence in CrySL, which is `instanceOf`. They were also built-in functions without counterparts in other DSLs. For instance, MARK does not provide a corresponding `callTo` function. Although CrySL did not include corresponding built-in functions for all the MARK built-in functions, all the useful functions in MARK had built-in equivalents in CrySL, apart from `_has_value`, which is not necessary because it is used to check if a parameter exists in a program and there is no use for it in CrySL as discussed in Section 4.3.3.3. We may add additional built-in functions to each DSL in the future to determine whether they will be equally expressive or not. We will discuss this in Chapter 7.

Additionally, we observed that MARK rules require more space than CrySL rules. Table 4.1 shows the sizes of all rulesets for different APIs. For example, the size of all CrySL rules for the JCA API (on the second column) and the size of translated JCA MARK rules (on the third column). In all cases, the size of the CrySL rules is less than the size of the MARK rules, as shown in the table. This indicates that MARK requires more space than CrySL when expressing the same rules. The primary reason is that every MARK rule must be written in a specific format with a `rule` name, a `when` clause, an `ensure` clause and an `onfail` error message. In contrast, CrySL rules are much more concise in describing the rules, and each rule need only be added to the corresponding section. For example, the file of KeyGenerator CrySL rule of JCA ruleset (Listing 4.1) has 36 lines and takes 717 bytes, while its translated version to MARK has combined (entity in Listing 4.6 and rule in Listing 4.7), 82 lines and takes 2003 Bytes. The entity (Listing 4.3) and rule (Listing 4.11) files of the MARK KeyGenerator rule of the Bouncy Castle ruleset together take up 104 lines and 2780 bytes, and its translation to CrySL (Listing 4.10) has 43 lines and takes 938 bytes.

	JCA CrySL	Translated JCA MARK	Bouncy Castle JCA CrySL	Translated Bouncy Castle JCA MARK	Translated Bouncy Castle CrySL	Bouncy Castle MARK
Size(KB)	36,6	83,30	39,7	95,20	30,8	55,6

Table 4.1: Sizes of the original and translated MARK and CrySL rule files.

5 Evaluation

In this chapter, we will evaluate and compare the performance (precision and recall) of Codyze and CogniCrypt$_{\text{SAST}}$ Eclipse plugins. To accomplish this, we will use Java code samples that contain the correct and incorrect usages of cryptographic APIs and compare the analysis results each tool generates after analyzing the code samples.

Consequently, we require a benchmark for comparing the efficiency of the tools in detecting cryptographic vulnerabilities. Unfortunately, Codyze and CogniCrypt$_{\text{SAST}}$ do not provide any generally accepted benchmark in this domain; therefore, we searched for alternative resources. One of the benchmarks we can use to compare analysis results that is more recent and relevant is the CryptoAPI-Bench [ARY19], which is the benchmark for CryptoGuard [SR19] and covers a wide range of misuse instances. We will discuss CryptoAPI-Bench and the results of analyzing it using both tools in section 5.1.2.

Since CryptoAPI-Bench might be biased towards CryptoGuard and therefore may not provide us with an accurate comparison, we further used CogniCrypt$_{\text{TestGen}}$[KR20]. CogniCrypt$_{\text{TestGen}}$ is a test generator for cryptographic APIs. It generates test suites that contain test cases with all correct and incorrect usages of an API. It consumes CrySL rules as an input, parses them, and produces test suites for each rule. We will explain more about the test generation and the result of analyses in the Section 5.2.2.2.

To avoid confusion, we refer to the translated rules as follows. The translated Bouncy Castle JCA CrySL rules to MARK, the translated Bouncy Castle JCA MARK rules. The translated JCA CrySL rules to MARK, the translated JCA MARK rules. The translated Bouncy Castle MARK rules to CrySL, the translate Bouncy Castle CrySL rules.

The following setup section describes the environment in which we conducted the evaluations. Next, we discuss the results of analyzing CryptoAPI-Bench using CogniCrypt$_{\text{SAST}}$ and Codyze, followed by discussing the results of analyzing CogniCrypt$_{\text{TestGen}}$ generated tests.

Setup

To analyze benchmarks with Codyze and CogniCrypt$_{SAST}$, we employed the same virtual machines with analyzers installed in them as described in the setup of Section 4.2. There is a total of 6 GB of memory allocated to Codyze's language server as Codyze developers recommend using 6 GB or more. Since our virtual machine has a maximum memory of 8 GB, we dedicated 6 GB to the language server. The type state analysis is set to `within functions only`. The other option, `whole program`, does not work, therefore the Codyze developer recommended setting it to `within functions only`. The call graph construction algorithm of CogniCrypt$_{SAST}$ is set to CHA, which is the default. In this chapter, we used the original CrySL and MARK rules as described in the Section 4.3.3.1 and the translated rules that are described in Section 4.3.3.

5.1 Evaluation with CryptoAPI-Bench

CryptoAPI-Bench is the first benchmark that we utilized to evaluate CogniCrypt$_{SAST}$and Codyze. In this section, we will first discuss the CryptoAPI-Bench and the types of misuses it covers. Following the discussion of experiment execution, we examine the results of the analyses of CryptoAPI-Bench using Codyze and CogniCrypt$_{SAST}$.

According to Afrose et al.[ARY19], CryptoAPI-Bench provides 171 cases for assessing the quality of various cryptographic vulnerability detection tools. CryptoAPI-Bench addresses 16 different types of cryptographic and SSL/TLS API misuse vulnerabilities. These include hardcoded secrets, improper certificate validations, improper hostname validations, insecure symmetric and asymmetric cryptographic primitives, and insecure hash functions [ARY19]. The CryptoAPI-Bench consists of 40 basic tests, 131 advanced tests, which includes 40 inter-procedural, 19 field-sensitive, 20 combined tests (combination of inter-procedural and field-sensitive), and 20 path-sensitive test cases [ARY19].

Each of these 16 categories of misuses of cryptographic APIs is regarded by the CryptoAPI-Bench as a cryptographic threat model, and each threat model has a corresponding cryptographic API [ARY19]. The CryptoAPI-Bench provides test cases for which neither Codyze nor CogniCrypt$_{SAST}$ provides rules for the corresponding APIs, namely, javax.net.ssl.HostnameVerifier, javax.net.ssl.X509TrustManager, javax.net.ssl.SSLSocket, and java.net.URL. Therefore, we do not consider the threat models related to those APIs in our evaluation because both tools did not identify any misuses in those models. We will only use the remaining 12 threat models that are shown in the Table 5.1. The following is a brief explanation of these 12 threat models presented by CryptoAPI-Bench in order to gain an understanding of the type of vulnerability they introduce.

Cryptographic Key. When using a key generated by the javax.crypto.spec.SecretKeySpec API for encryption, the Byte array that SecretKeySpec takes as input should be unpredictable and not constant or hard-coded. Otherwise, the attacker could easily read the key to obtain sensitive information.

Passwords in PBE (Password-based Encryption). It relates to the use of a constant or hardcoded password in the javax.crypto.spec.PBEKeySpec API to generate a SecretKey. A hardcoded or constant password may allow an attacker to acquire it and gain access to the key.

Passwords in KeyStore. The java.security.KeyStore API is used to store cryptographic keys or certificates. The KeyStore requires a password to access the stored keys, which should not be hardcoded or a constant. Otherwise, the keys and certificates stored in the KeyStore could be accessed by unauthorized parties.

Pseudorandom Number Generator (PRNG). The random number generator algorithm used by java.util.Random (Knuth's subtractive [HVO17]) is proven to be insecure, while java.security.SecureRandom generates a random number that is non-deterministic and unpredictable. Therefore, it is recommended to use SecureRandom instead of Random because the latter does not produce a completely random number.

Seeds in Pseudorandom Number Generator (PRNG). When using SecureRandom to generate random numbers, we may encounter the same random number in every run if we use a constant or not randomly generated seed as SecureRandom's parameter.

Salts in Password-based encryption (PBE). Javax.crypto.spec.PBEParameterSpec API takes the salt and iteration count when setting parameters for Password-based encryption. Salt should not be hardcoded, or a constant but rather should be randomly generated. Otherwise, it may result in producing an insecure key.

Mode of Operation. ECB (Electronic Codebook) is not considered a secure mode of operation in javax.crypto.Cipher since it does not conceal data patterns well and may disclose information about the plaintext. CBC or Galois/Counter (GCM) Mode should be used instead.

Initialization Vector (IV). To enhance cryptography security, initialization vectors (IVs) are used during encryption and decryption. A constant/static initialization vector may introduce vulnerabilities. Therefore, it is recommended to use a random initialization vector in the crypto.spec.IvParameterSpec API.

Iteration Count in Password-based Encryption (PBE). For Password-based Encryption (PBE) with the javax.crypto.spec.PBEParameterSpec API, it requires salt and iteration count. The Public Key Cryptography Standards (PKCS), which is a set of public-key cryptography standards published by RSA Security LLC (an American company that focuses on encryption and encryption standards [rsa]) 5 version 2.0 [MKR17], recommends that the number of iterations should be greater than 1000 to provide adequate security. Therefore, iteration counts less than 1000 are considered insecure.

Symmetric Ciphers. Symmetric cryptography employs the same key for both encryption and decryption. AES is the preferred symmetric cipher, as several other symmetric ciphers, including DES, Blowfish, RC4, RC2, IDEA, and RC4, are considered broken.

Asymmetric Ciphers. In asymmetric cryptography, a pair of keys, including a public key and a private key, is used to encrypt and decrypt data. It is, however, recommended to use 2048 bit key size for some asymmetric ciphers, such as RSA, because they are considered broken with

1024 bit key size.

Cryptographic Hash Functions. In cryptography, hash functions convert an arbitrary message to a fixed-size value known as a hash or message digest, which is utilized to verify message integrity, digital signature, and authentication. When two inputs of the same cryptographic hash function produce the same hash value, it is considered to be broken. SHA-256 is an alternative to insecure hash functions such as SHA1, MD4, MD5, and MD2.

5.1.1 Description of the execution of the experiment

To collect analysis results, we analyzed the CryptoAPI-Bench source code with CogniCrypt$_{\text{SAST}}$ Eclipse plugin by triggering the start button and analyzing the entire project, and with Codyze Eclipse plugin by opening all the class files in the Eclipse editor. As JCA is the most commonly used cryptographic library for Java [SNB16], we chose to use the JCA rulesets. However, the MARK developers did not provide rules for the JCA API, so we used CrySL rules for the JCA API [api] in and MARK rules for Bouncy Castle API [codb] in Codyze.

The Misuses that Codyze finds in a program in Eclipse are not displayed with a red flag except for the misuses related to the forbidden methods. They are instead displayed as information, such as when a MARK rule is verified. This made it difficult to distinguish between violations and verifications of Codyze's findings in Eclipse and increased the risk of mistakes occurring when collecting the analyses results. In our meeting with a Codyze developer, they mentioned that the problem lies within the file `CpgDocumentService.java`[1] . After Codyze has gathered all findings in a program, it categorizes their severity level into three categories: information, warning, and error. Nevertheless, it does not identify misuses within the error category; instead, misuses are considered information, which is the default severity level.

After performing the analysis, we obtained the true positives and false positives based on the results logs of each tool. Since our goal is to compare the cryptographic vulnerability detection of the two tools, we considered only cryptographic misuse alerts and disregard unrelated information. When a tool generates an alert based on the correct reason in a vulnerable test case, it is considered a true positive (TP). If in a correct test case of a threat model, a tool identifies a misuse related to that threat model, then it is considered as a false positive (FP).

5.1.2 Examining the results of analyses

In the following, we describe the results of evaluating the CryptoAPI-Bench with each detection tool and compare their performances. Table 5.1 presents the number of true positives (TP) and false positives (FP) detections of vulnerabilities by the Codyze and CogniCrypt$_{\text{SAST}}$ in the 12 cryptographic threat models in CryptoAPI-Bench that we previously discussed. In this context, true positive is when a tool detects misuse in a test case, and it is a vulnerability based on the CryptoAPI-Bench_details.xlsx file available on CryptoAPI-Bench's Github page [ben]. This Excel file provides an overview of secure and non-secure code as well as the vulnerabilities. Further, a false positive is when a tool identifies a misuse in a test case with no vulnerability based on the Excel file. TTP stands for total true positives, the number of

[1]`https://github.com/Fraunhofer-AISEC/codyze/blob/main/src/main/java/de/fraunhofer/aisec/codyze/crymlin/connectors/lsp/CpgDocumentService.java`

test cases with vulnerabilities provided by the CryptoAPI-Bench. TTN stands for total true negatives, the number of test cases provided by CryptoAPI-Bench with the correct usages of cryptographic APIs. TTN and TTP are calculated based on the Excel file. The last two columns in Table 5.1, indicate the recall and precision of Codyze and CogniCrypt$_{SAST}$ when analyzing each threat model. Pre and Rec are acronyms for precision and recall, respectively.

No.	Threat Models	TTP	TTN	Codyze				CogniCrypt$_{SAST}$			
				TP	FP	Pre(%)	Rec(%)	TP	FP	Pre(%)	Rec(%)
1	Cryptographic Key	8	2	0	0	0	0	8	2	80,00	100
2	Password in PBE	8	3	0	0	0	0	6	5	54,54	75,00
3	Password in KeyStore	7	3	0	0	0	0	6	2	75,00	85,71
4	PRNG	1	1	0	0	0	0	0	0	0	0
5	Seed in PRNG	14	3	0	0	0	0	0	0	0	0
6	Salt in PBE	7	2	0	0	0	0	7	2	77,77	100
7	Mode of Operation	6	2	1	1	50,00	16,66	5	1	83,33	83,33
8	Initialization Vector	8	2	8	2	80,00	100	8	0	100	100
9	Iteration Count in PBE	7	2	0	0	0	0	5	2	71,42	71,42
10	Symmetric Cipher	30	6	30	6	83,33	100	20	5	80,00	66,66
11	Asymmetric Ciphers	5	1	0	0	0	0	4	1	80,00	80,00
12	Cryptographic Hash	24	5	24	4	85,71	100	20	4	83,33	83,33
Total		125	32	63	13	**82,89**	**50,80**	89	24	**78,76**	**71,20**

Table 5.1: Comparison of Codyze and CogniCrypt$_{SAST}$ analysis on 12 threat models in CryptoAPI-Bench's 171 test cases.

Based on the Table 5.1, there are 10 models of cryptographic threads that are covered by CogniCrypt$_{SAST}$, while Codyze covers 4 models. From 125 vulnerable test cases, Codyze identified 63 and CogniCrypt$_{SAST}$ identified 89 true positives. Codyze and CogniCrypt$_{SAST}$ generate a total of 13 and 24 false alarms (false positives), respectively. In test cases related to the SecureRandom API (threat models 4 and 5 in the Table 5.1), CogniCrypt$_{SAST}$ does not detect any misuses, despite the fact that this type of misuse is explicitly specified by the SecureRandom CrySL rule of JCA APIs. This is clearly a false negative. Codyze does not detect any misuses for the models 1 to 6 and 9 and 11. It is because there are no MARK rules to specify those vulnerabilities.

As we mentioned before in Section 3.2, MARK developers wrote MARK rules based on BSI [BSIa] TR-02102-1 version 2019-01. The 2019-01 version of BSI TR-02102-1 is not publicly available; therefore, we contacted the BSI and asked for that version to verify that all the specifications of cryptographic APIs described therein are implemented as MARK rules. We received 2019-01 version [bsib] very late in the thesis timeline, and unfortunately, it was in German. We requested the English version but did not receive it when this thesis was written. However, we were able to check the rules with the help of translation tools and the latest version of BSI TR-02102-1 (version 2021-01) [BSI21]. In some cases, the rules in the BSI could not be described as MARK rules because MARK maintainers could not interpret them in MARK or Bouncy Castle (the API for which the MARK rules are written) did not provide the implementation for that rule.

For example, in section 2.1.2 (Betriebsbedingungen) of the BSI guideline, it is stated that the initialization vectors must not be repeated during the lifetime of a key, i.e., they should not be used for two different ciphers at the same time. Because it requires sufficient knowledge about the program's dynamic behavior, the MARK developers were unable to define it as a MARK rule; however, it is not clear whether this rule is included in the analysis. Another example is section 3.5 (RSA), which is related to the RSA Cipher and indicates that the size of the modulus in calculating the key length of the RSA Cipher must be at least 2000. It cannot be defined in MARK since we cannot adequately reason about the modulus of the RSA key.

The Table 5.1 indicates that Codyze has a higher precision than CogniCrypt$_{\text{SAST}}$, which means that it can identify misuses that CogniCrypt$_{\text{SAST}}$ cannot detect. Meanwhile, CogniCrypt$_{\text{SAST}}$ has a better recall than Codyze, which indicates that CogniCrypt$_{\text{SAST}}$ has identified more misuses than Codyze, likely because Codyze does not cover all threat models as it lacks the MARK rules that specify cryptographic misuses in the missing models.

The only case in which Codyze could not detect all misuses was in the mode of operation (number 7 in the Table 5.1) model. In only one of the test cases, Codyze was able to identify an incorrect mode of Cipher. In the other test cases involving such misuse, Codyze detected an invalid Cipher rather than an invalid mode of Cipher, which is a false positive.

According to the CrySL rule for the class PBEKeySpec of JCA API (see Listing 4.2), the iteration count should be more than 10000; whereas, based on CryptoAPI-Bench, iteration counts higher than 1000 are acceptable. One of the false positives in the CogniCrypt$_{\text{SAST}}$ results for the ninth threat model was caused by this difference.

It should also be noted that Codyze and CogniCrypt$_{\text{SAST}}$ produce false positives for misuses that are not listed in the threat model of CryptoAPI-Bench. For instance, CogniCrypt$_{\text{SAST}}$ identifies a misuse that indicates that the key used by Cipher is not generated correctly as a generatedKey. It occurred in every test case containing a Cipher object with a Key parameter. This is a true positive for the cases where the key is generated with an improper algorithm or key size, but for the cases where a key is properly generated based on the CrySL rules, this is a false positive. Nine of the test cases contain this false positive. This issue has already been reported to CogniCrypt$_{\text{SAST}}$'s Github page[2].

In addition to the misuses specified by CryptoAPI-Bench, Codyze and CogniCrypt$_{\text{SAST}}$ also detect other misuses in the benchmark. If the provider of an API is not specified, Codyze throws a provider error. According to the MARK rules, the provider should always be specified so that APIs used in the code and the API of MARK ruleset used to analyze, have the same provider such as the Bouncy Castle provider. For example, in the mode of operation threat model, Codyze detected that there was no provider specified when using the Cipher and KeyGenerator APIs. All the provider misuses discovered by Codyze were true positives. Additionally, there were errors detected by Codyze concerning calls to java.security.SecureRandom that were forbidden. The findings were also true positives according to the MARK rule for the SecureRandom from the Bouncy Castle API. The forbidden calls are shown in Lines 7 and 8 of Listing 4.4. Another finding by Codyze was that padding was not applied to the RSA cipher. Based on the MARK rules for Cipher, RSA Cipher must always have a padding (proper paddings are specified in the rule). Therefore Codyze found such misuses and they are true positives. According to Naccache et al.[NI04], it is necessary to apply encryption padding in

[2]https://github.com/eclipse-cognicrypt/CogniCrypt/issues/457

order to prevent dictionary attacks. A dictionary attack is a type of brute force attack in which a hacker uses a list of common words and phrases to attempt to crack a password-protected security system. Furthermore, there was an error related to the Cipher order, as the order in which the Cipher methods were called did not comply with the MARK order rule for Cipher (true positive). The same error was generated more than once within the same test case, which was redundant and unnecessary.

CogniCrypt$_{SAST}$ detects misuses that are not included in the CryptoAPI-Bench's benchmark, namely invalid ordering of PBEKeySpec, MessageDigest, and Cipher. Taking the CrySL rules for the JCA API into consideration, the order of Cipher error is a false positive, whereas the order of PBEKeySpec is the result of the absence of a call to the clearPassword method as the final call and is a true positive. The MessageDigest invalid order is also a true positive based on the CrySL rule.

Overall, Codyze's precision in detecting misuses is approximately 4 percent better than CogniCrypt$_{SAST}$. However, when we consider the four threat models for which both tools had rules (threat models 7, 8, 10, and 12), we can observe that CogniCrypt$_{SAST}$ has a better accuracy for two of the models and Codyze has a better accuracy for the other two. Further, the difference between precisions is greater when CogniCrypt$_{SAST}$'s precision is higher (7 and 8), since Codyze usually produces more false positives than CogniCrypt$_{SAST}$. Regarding the recall, Codyze does not provide sufficient MARK rules to define cryptographic vulnerabilities of all threat models, and as a result, its recall value is 20 percent lower than that of CogniCrypt$_{SAST}$.

The next table (Table 5.2) shows the result of analyzing CryptoAPI-Bench advanced test cases that both Codyze and CogniCrypt$_{SAST}$ have rules for the cryptographic APIs of their threat models. This includes the mode of operation (javax.crypto.Cipher), initialization vector (javax.crypto.spec.IvParameterSpec), symmetric cipher (javax.crypto.Cipher) and cryptographic Hash (java.security.MessageDigest). They are categorized into 7 categories that we shortly describe below.

In two-interprocedural cases, the source of vulnerability in one procedure is passed as an argument to the parameter of another procedure and then used in the crytographic API. In three-interprocedural cases, the source of vulnerability is passed as an argument to another procedure and then passed again to the third procedure. These inter-procedural test cases measure the ability to handle the data flow.

The field-sensitive cases check the ability of the tools in performing field-sensitive data flow analysis. Combined cases contain test cases with a combination of inter-proceural and field sensitivity. Path-sensitive test cases include conditional branches to examine the accuracy of determining the source of a vulnerability. Miscellaneous test cases are intended to assess the tool's ability to detect irrelevant constraints and other interfaces, such as Map. In multiple class test cases, the source of vulnerabilities that is originated in one class is passed to another class and used in a cryptographic API.

Advanced Test Cases	TTP	TTN	Codyze TP	Codyze FP	Codyze Pre(%)	Codyze Rec(%)	CogniCrypt$_{SAST}$ TP	CogniCrypt$_{SAST}$ FP	CogniCrypt$_{SAST}$ Pre(%)	CogniCrypt$_{SAST}$ Rec(%)
Two-Interprocedural	11	0	10	0	100	90,90	11	0	100	100
Three-Interprocedural	11	0	10	0	100	90,90	11	0	100	100
Field Sensitive	11	0	10	0	100	90,90	2	0	100	18,18
Combined Case	11	0	10	0	100	90,90	1	0	100	9,09
Path Sensitive	0	11	0	11	0	0	0	10	0	0
Miscellaneous Cases	2	0	2	0	100	100	2	0	100	100
Multiple Class methods	11	0	10	0	100	90,90	11	0	100	100
Total	57	11	52	11	**82,53**	**91,22**	38	10	**79,16**	**66,66**

Table 5.2: Comparison of Codyze and CogniCrypt$_{SAST}$ analyses of CryptoAPI-Bench's 68 advanced test cases (of all 12 threat models in Table 5.1) that both tools have common corresponding cryptographic rules for them.

Table 5.2 indicates that when we restrict the test cases to those for which both tools have rules that relate to them and are capable of analyzing them, then Codyze's recall increases to 91,22 percent. The only vulnerability that Codyze could not identify in the test cases is related to the mode of operation, as previously explained. If we examine the cases individually, CogniCrypt$_{SAST}$ provides better recall than Codyze, except in the field-sensitive and combined cases, in which CogniCrypt$_{SAST}$ detected only three misuses in 22 tests of field-sensitive and combined cases, whereas Codyze detected all of them. Therefore, we can conclude that Codyze is field-sensitive, and CogniCrypt$_{SAST}$ is partially field-sensitive. However, in Section 3.1, we learned that CogniCrypt$_{SAST}$ is field-sensitive. These results may be due to the fact that the CryptoAPI-Bench test cases are biased in favor of CryptoGuard and are designed only to examine whether CryptoGuard is field-sensitive or not and therefore do not provide completely field-sensitive cases or are insufficient to analyze field sensitivity. It may also be due to CogniCrypt$_{SAST}$ being only partly field-sensitive. In order to ensure that, it is necessary to test more field-sensitive test cases or to verify that the field-sensitive test cases in CryptoAPI-Bench address the field-sensitivity issue. Due to our limited time, we suggest performing this as a future work discussed in Chapter 7.

CogniCrypt$_{SAST}$ detected all of the inter-procedural misuses, which makes it inter-procedural. Codyze found 10 of the 11 misuses in the inter-procedural test cases, making it partially inter-procedural. The fact that Codyze is inter-procedural is unexpected. As stated in section 3.2, based on the Codyze Java docs and the fact that CPG is not inter-procedural, we concluded that Codyze is not inter-procedural. The contradiction may be because the Codyze Java documentation has not been updated yet, and Codyze is, in fact, inter-procedural, or it is the test cases problem. It is possible that the tests are not inter-procedural, or they are insufficient to determine if an analysis is inter-procedural. Our future work could include verifying that the inter-procedural tests of CryptoAPI-Bench are indeed inter-procedural, and providing enough inter-procedural test cases to test the tools to determine if they are inter-procedural.

Both tools have the same precision in each of the advanced test cases separately, but overall Codyze produces one more false positive than CogniCrypt$_{SAST}$ (in the path-sensitive case). The path-sensitive case consists of 11 test cases that do not contain any misuses, and if the analyzer detects misuses in those 11 tests, then it is not path-sensitive. Codyze detected that all the correct test cases have misuses which indicates that Codyze is not path-sensitive. CogniCrypt$_{\text{SAST}}$ found misuses in 10 out of 11 correct path-sensitive test cases, suggesting that CogniCrypt$_{\text{SAST}}$ is rarely path-sensitive. However, there is a possibility that CogniCrypt$_{\text{SAST}}$ did not find a misuse in this particular case by accident, since as mentioned in Section 3.1, CogniCrypt$_{\text{SAST}}$is not path-sensitive. CryptoAPI-Bench's test cases may not be sufficient in this case. As future work, we could use more path-sensitive test cases to verify whether the tools are path-sensitive or not.

For similar misuses that both Codyze and CogniCrypt$_{\text{SAST}}$ detect, they produce different error messages. Table 5.3 presents the error messages that Codyze and CogniCrypt$_{\text{SAST}}$ produce for the misuses they found in four mutual threat models. As we mentioned previously, there is a JSON file (findingdescription.json) produced by the Codyze developers, which contains all the possible error messages (cf. section 3.2), and CogniCrypt$_{\text{SAST}}$ generates the error messages based on the type of error (e.g., constraints, predicates, etc.), as we explained in the section 3.1. Therefore, Codyze always displays similar error messages when the same misuse occurs. As an example, it always generated the same error message for the incorrect Cipher as demonstrated in the Table 5.3 (symmetric cipher threat model). Codyze error messages are more abstract than CogniCrypt$_{\text{SAST}}$ error messages, as Codyze states the misuse of the API and the reason for its insecurity. CogniCrypt$_{\text{SAST}}$ error messages provide more details. For instance, in the mode of operation, symmetric cipher, and cryptographic hash, CogniCrypt$_{\text{SAST}}$ provides information for the appropriate algorithms, modes, and hashes to use, respectively. In the initialization vector, Codyze only mentions that the IV has an inadequate quality, whereas CogniCrypt$_{\text{SAST}}$ specifies that the IV needs to be a random number.

Threat Model \ Tool	CODYZE	COGNICRYPT$_{SAST}$
Mode of Operation	Use of an unspecified cipher mode for symmetric-key algorithms: A cipher mode for symmetric-key algorithms was detected that does not match one of the recommended Cipher modes by BSI TR-02102. Use of weak or unspecified cipher modes may not guarantee sufficient security.	First parameter (with value *"the parameter of cipher.getInstance(String)"*) should be any of the *"list of possible modes for that algorithm"*
Initialization Vector	Insufficient quality of IV for CBC cipher mode: The IV used with the cipher mode CBC is not sufficiently unpredictable. Predictable IVs, which an attacker could guess, compromises the security guarantees of CBC cipher mode.	First parameter was not properly generated as randomized
Symmetric Cipher	Use of an unspecified cipher: A cipher was detected that does not match one of the recommended ciphers by BSI TR-02102. Use of weak or unspecified ciphers may not guarantee sufficient security.	First parameter (with value *"the parameter of cipher.getInstance(String)"*) should be any of *"list of all valid parameter of cipher.getInstance(String)"*
Cryptographic Hash	Use of an unspecified hash function: An unspecified hash function is being used. Unspecified hash functions may be weak to attacks.	First parameter (with value *"the parameter of MessageDigest.getInstance(String)"*) should be any of *list of all valid parameters of MessageDigest.getInstance(String)*

Table 5.3: Comparison of the error messages produced by CODYZE and COGNICRYPT$_{SAST}$ for the similar misuses they found after analyzing CRYPTOAPI-BENCH.

In summary, overall CODYZE achieved a higher level of precision than COGNICRYPT$_{SAST}$ in analyzing CRYPTOAPI-BENCH, however COGNICRYPT$_{SAST}$ covered more cryptographic vulnerabilities than CODYZE. Therefore, when analyzing the test cases for which both tools have rules (Table 5.1), COGNICRYPT$_{SAST}$'s recall is noticeably higher than CODYZE. CODYZE analysis is context- and flow-sensitive, and partially field-sensitive and inter-procedural, but not path-sensitive. COGNICRYPT$_{SAST}$ analysis is inter-procedural and context-sensitive and partially field-sensitive, but seldom path-sensitive.

Furthermore, COGNICRYPT$_{SAST}$ produces more detailed error messages than CODYZE. There are also some issues with both tools. Codyze, for instance, produced duplicate errors when the Cipher order was violated. Codyze did not generate any false positives other than those identified in analyzing CryptoAPI-Bench in Tables 5.1 and 5.2. CogniCrypt$_{\text{SAST}}$ produces a false positive, that is related to the incorrect usage of the keyGenerator API, in several test cases. However, the misuses of the KeyGenerator API are not considered in the CryptoAPI-Bench threat models. Therefore, we did not count them as false positives in the evaluation.

CryptoAPI-Bench only covered a few cryptographic vulnerabilities, and the number of test cases they provided may not be sufficient for a fair comparison. Furthuremore, CryptoAPI-Bench was created to evaluate CryptoGuard's ability to detect misuses in usages of Java cryptographic APIs, and may therefore be biased toward CryptoGuard. Consequently, we will evaluate CogniCrypt$_{\text{SAST}}$ and Codyze in the next section with the test cases generated by the CogniCrypt$_{\text{TestGen}}$.

5.2 Evaluation with CogniCrypt$_{\text{TestGen}}$

Results of the analysis in the last section may not provide us with a fair comparison since the number of test cases may not be sufficient to provide a meaningful comparison. Additionally, they do not cover all of the vulnerabilities covered by Codyze and CogniCrypt$_{\text{SAST}}$, and the CryptoAPI-Bench was designed to evaluate CryptoGuard's cryptographic misuse detection capability. Therefore, we propose to analyze the test cases generated by CogniCrypt$_{\text{SAST}}$. Here, we discuss how we generated the test cases using CogniCrypt$_{\text{TestGen}}$, how we ran the experiment, calculated and recorded misuses, and finally, we discuss the results of the analyses.

With the help of a CogniCrypt$_{\text{TestGen}}$ developer, we set up the source code of the CogniCrypt$_{\text{TestGen}}$ Eclipse plugin and built it within Eclipse (Skype meeting with one of the CogniCrypt$_{\text{TestGen}}$ developers on Nov 15, 2021). We used the unpublished version of CogniCrypt$_{\text{TestGen}}$ on the evaluation branch[3] to the date of October 31$_{\text{st}}$ 2021 to the commit e834927[4].

In order to generate test cases, CogniCrypt$_{\text{TestGen}}$ utilizes CrySL rules. CogniCrypt$_{\text{TestGen}}$ covers most sections in the CrySL rules completely, partially covers the ENSURED and CONSTRAINTS sections, and does not cover the FORBIDDEN and NEGATES sections [KR20]. The generated tests could be used as unit or integration tests [KR20]. We used them as integration tests in our case because several test cases generated by CogniCrypt$_{\text{TestGen}}$ were incorrectly considered valid or invalid or included more than one misuse and therefore needed further examination.

We used CogniCrypt$_{\text{TestGen}}$ to generate test cases covering all the MARK and CrySL rules for the Bouncy Castle JCA and JCA APIs. Therefore, we used JCA and Bouncy Castle JCA CrySL rules and the translated Bouncy Castle CrySL rules that we created in Section 4.3.3.2. CogniCrypt$_{\text{TestGen}}$ generated test cases for 38 Java cryptographic APIs with JCA CrySL rules (see Table A.2.2), and it generated test cases for 38 Java cryptographic APIs with Bouncy Castle JCA CrySL rules (see Table 5.4). CogniCrypt$_{\text{TestGen}}$ also generated test cases for 32

[3]`https://github.com/CROSSINGTUD/CogniCrypt_TESTGEN/tree/evaluation`

[4]`https://github.com/CROSSINGTUD/CogniCrypt_TESTGEN/commit/e8349276c172bae24b6e1016c309fcd2b5b5d761`

Java cryptographic APIs with translated Bouncy Castle CrySL rules. There are several valid and invalid test cases for each of those APIs. CogniCrypt$_{\text{TestGen}}$ creates a valid test case for every complete sequence in the FSM (finite state machine) defined in the ORDER section with all the possible methods and parameters and creates an invalid test case for every incomplete sequence in the FSM. Therefore, in invalid test cases, the insecurity is the incorrect order of operators in the use of an API.

To simplify, we will refer to the test cases that CogniCrypt$_{\text{TestGen}}$ generated from CrySL Bouncy Castle JCA ruleset as the Bouncy Castle tests and the test cases generated from the JCA CrySL ruleset as the JCA tests. Moreover, we refer to the test cases generated from the translated Bouncy Castle CrySL rules as the MARK Bouncy Castle tests. In total, there are 228 valid and 233 invalid test cases in the JCA tests and 228 valid and 233 invalid test cases in the Bouncy Castle tests. Furthermore, there are 50 valid and 0 invalid test cases in the Bouncy Castle MARK tests.

We encountered some errors when generating test cases with Bouncy Castle JCA CrySL rules, and the CogniCrypt$_{\text{TestGen}}$ could not generate the test cases. We debugged the CogniCrypt$_{\text{TestGen}}$'s source code and fixed it with the assistance of one of the CogniCrypt$_{\text{TestGen}}$'s developers (Skype meeting with one of the CogniCrypt$_{\text{TestGen}}$ developers on Nov 26, 2021). We provided a pull request to solve this issue[5].

5.2.1 Description of the execution of the experiment

We analyzed the test cases using both tools on the virtual machines as we described in the previous section (see Section 5.1.2). Analyzing the test cases, we found that some of the valid test cases contained misuses, while some of the invalid test cases contained no misuses. The misuses in the valid test cases and the correct invalid test cases were detected by both tools. Some of them were detected by CogniCrypt$_{\text{SAST}}$, and others by Codyze. We will tackle all misuses further when discussing the results. We checked the remaining test cases in which none of the tools detected API misuse manually to ensure there was no API misuse. We considered all the misuses that each tool found in the test cases (both valid and invalid) in the results of the analyses. Moreover, the total number of misuses (for calculating precision and recall) was calculated by measuring the union of the set of misuses found by both tools. We will discuss the calculation of true and false positives in detail in Sections 5.2.2.1 and 5.2.2.2. We conducted eight analyses, four for each tool, as follows:

- Analyzing Bouncy Castle tests using Codyze with Bouncy Castle MARK rules and using CogniCrypt$_{\text{SAST}}$ with the translated Bouncy Castle CrySL rules.

- Analyzing Bouncy Castle tests using Codyze with translated Bouncy Castle JCA MARK rules and using CogniCrypt$_{\text{SAST}}$ with Bouncy Castle JCA CrySL rules.

- Analyzing JCA tests using Codyze with translated JCA MARK rules and using CogniCrypt$_{\text{SAST}}$ with JCA CrySL rules.

[5]`https://github.com/CROSSINGTUD/CogniCrypt_TESTGEN/pull/15`

- Analyzing MARK Bouncy Castle tests using Codyze with the Bouncy Castle MARK rules and using CogniCrypt$_{SAST}$ with translated Bouncy Castle CrySL rules.

The true positives and false positives are calculated the same as in the previous section (see Section 5.1.2).

5.2.2 Examining the results of analyses

Here, the results of all analyses are compared and discussed. The MARK Bouncy Castle tests consisted of 50 valid test cases and no invalid ones, with 9 of those 50 valid tests containing a compiler error (see Table A.1). Consequently, when analyzing these test cases using Codyze with the Bouncy Castle MARK rules and CogniCrypt$_{SAST}$ with the translated Bouncy Castle CrySL rules, the analysis results as shown in the Table A.1 in the Appendices indicate no misuses, except for the IvParameterSpec test cases. Two of the nine compiler errors occur in the two test cases for the IvParameterSpec API. Despite the compiler errors, Codyze detected two misuses in the IvParameterSpec tests, which were a result of the unspecified provider name of the SecureRandom API, which is a true positive (Listing 5.1 line 7). CogniCrypt$_{SAST}$ did not detect any misuses in all the cases. Nevertheless, when we removed the erroneous lines from the IvParameterSpec test cases, including the Line 9 from Listing 5.1, CogniCrypt$_{SAST}$ detected the provider misuses and produced two other errors that were related to this misuse. For example, on Line 7 of listing 5.1 the related error stated that the `secureRandom0` parameter was not appropriately generated as a random number, which is also a true positive. However, it is the same as provider misuse. We will discuss the dependent misuses more in Section 5.2.2.1. Even though both analyses detected the same problem after the fix, this finding illustrates the capability of Codyze to analyze erroneous test cases. Since this test case does not provide any additional useful results, we will not discuss it further.

Listing 5.1: A faulty test case from MARK Bouncy Castle test

```
1  public void ivParameterSpecValidTest2() throws NoSuchAlgorithmException ,
     NoSuchProviderException {
2
3      String algorithm = null;
4      byte[] seed = null;
5      int next = 0;
6      String provider = null;
7      SecureRandom secureRandom0 = SecureRandom.getInstance(algorithm , provider);
8      secureRandom0.setSeed(seed);
9      secureRandom0.next(next);
10     int offset = 0;
11     int len = 0;
12     IvParameterSpec ivParameterSpec0 = new IvParameterSpec(seed , offset , len);
13
14 }
```

In the following, we will examine the results of analyzing Bouncy Castle and JCA tests with Codyze and CogniCrypt$_{SAST}$. The results of analyzing Bouncy Castle tests with Bouncy Castle CrySL rules (original and the translation into MARK), and those of analyzing JCA tests with the JCA CrySL rules (original and the translation into MARK) are similar. Therefore, we focus our discussion on only one of them but provide results for both in tables. Since the Bounce Castle tests provide more information and cover the results of analyses on JCA tests, we will discuss the results of analyzing the Bouncy Castle tests in greater detail. We can therefore divide the analyses results into two sections: first, the results of analyzing Bouncy Castle and JCA tests using Codyze and translated Bouncy Castle JCA and JCA MARK rules, and using CogniCrypt$_{\text{SAST}}$ and Bouncy Castle JCA and JCA CrySL rules, and second, the results of analyzing Bouncy Castle tests using Codyze and Bouncy Castle MARK rules, and using CogniCrypt$_{\text{SAST}}$ and translated Bouncy Castle CrySL rules.

5.2.2.1 Discussing the results of analyzing Bouncy Castle and JCA tests using Codyze with translated Bouncy Castle JCA and JCA MARK rules, and using CogniCrypt$_{\text{SAST}}$ with Bouncy Castle JCA and JCA CrySL rules

In this section, we discuss only the results of analyzing the Bouncy Castle tests generated by CogniCrypt$_{\text{TestGen}}$. Because the results of analyzing the two test cases were similar, we chose Bouncy Castle tests since they covered more misuses and contained the misuses in the JCA tests' results.

Table 5.4 presents the results of analyzing Bouncy Castle tests with Codyze and CogniCrypt$_{\text{SAST}}$. Test name refers to the name of the API for which CogniCrypt$_{\text{TestGen}}$ generated test cases. TP refers to the number of true positives. An error is a true positive when it is caused by a violation of a CrySL rule or its translation to MARK. FP is the number of false positives. An error is a false positive when no violation of the corresponding CrySL rule or its translation into MARK has occurred.

Valid tests and invalid tests columns indicate the number of valid and invalid test cases in each of the 38 test cases, according to the CogniCrypt$_{\text{TestGen}}$ division of valid and invalid test cases. In total, there are 228 valid test cases and 233 invalid test cases in the Bouncy Castle tests. However, Codyze and CogniCrypt$_{\text{SAST}}$ discovered misuses in both valid and invalid test cases. The test cases generated by CogniCrypt$_{\text{TestGen}}$ contained several misuses that were not intended to exist. Some test cases contained more than one misuse, therefore in some cases in the table, the number of true positives is greater than the number of test cases. For example, for the AlgorithmParameters API, there are nine valid test cases, and Codyze and CogniCrypt$_{\text{SAST}}$ both detected 18 misuses in all nine valid test cases. Therefore, we analyzed misuses that Codyze and CogniCrypt$_{\text{SAST}}$ detected based on CrySL rules and calculated the number of false positives and true positives.

No.	Test Name	Valid tests	Invalid tests	Codyze				CogniCrypt$_{SAST}$			
				Valid		Invalid		Valid		Invalid	
				TP	FP	TP	FP	TP	FP	TP	FP
1	AlgorithmParameterGenerator	5	9	3	2	12	2	3	0	12	0
2	AlgorithmParameters	9	5	9	11	5	5	18	0	5	0
3	CertPathTrustManagerParameters	1	0	0	0	0	0	0	0	0	0
4	CipherInputStream	3	5	3	0	5	0	1	0	6	0
5	CipherOutputStream	3	5	3	0	5	0	1	0	6	0
6	Cipher	56	69	40	21	72	0	40	13	90	25
7	DHGenParameterSpec	1	0	1	0	0	0	1	0	0	0
8	DHParameterSpec	2	0	0	0	0	0	0	0	0	0
9	DigestInputStream	2	4	0	0	0	0	1	2	5	0
10	DigestOutputStream	2	4	0	0	0	0	1	2	5	0
11	DSAGenParameterSpec	2	0	0	0	0	0	0	0	0	0
12	DSAParameterSpec	1	0	0	0	0	0	0	0	0	0
13	ECGenParameterSpec	1	0	0	0	0	0	0	0	0	0
14	ECParameterSpec	1	0	0	0	0	0	0	0	0	0
15	GCMParameterSpec	2	0	0	2	0	0	1	0	0	0
16	HMACParameterSpec	1	0	0	0	0	0	0	0	0	0
17	IvParameterSpec	2	0	0	0	0	0	1	0	0	0
18	KeyAgreement	7	23	0	2	23	6	0	0	23	0
19	KeyFactory	6	0	0	4	0	0	0	0	0	0
20	KeyGenerator	8	8	0	3	8	3	0	0	8	0
21	KeyManagerFactory	6	4	0	8	4	0	0	0	4	0
22	KeyPairGenerator	6	10	4	0	14	0	4	0	12	0
23	KeyStoreBuilderParameters	1	0	0	0	0	0	0	0	0	0
24	KeyStore	13	21	0	0	17	0	0	0	17	0
25	Mac	12	21	0	7	21	0	1	0	22	0
26	MessageDigest	9	13	0	1	2	1	2	0	8	0
27	MGF1ParameterSpec	1	0	0	0	0	0	0	0	0	0
28	OAEPParameterSpec	1	0	0	0	0	0	0	0	0	0
29	PBEKeySpec	1	1	0	1	1	1	0	0	1	0
30	PBEParameterSpec	2	0	0	1	0	0	0	0	0	0
31	PKIXBuilderParameters	2	0	0	0	0	0	0	0	0	0
32	PKIXParameters	2	0	0	0	0	0	0	0	0	0
33	RSAKeyGenParameterSpec	1	0	0	0	0	0	0	0	0	0
34	SecretKeyFactory	3	2	0	4	2	0	0	0	2	0
35	SecureRandom	38	16	3	0	3	0	0	2	0	7
36	SSLContext	5	5	10	0	5	0	10	0	5	0
37	SSLParameters	4	4	0	1	4	0	0	0	3	0
38	TrustManagerFactory	6	4	0	13	4	0	0	0	4	0
Total		**228**	**233**	76	81	207	18	85	19	238	32
Results		**Precision(%)**		74,08				86,36			
		Recall(%)		85,75				97,87			

Table 5.4: Comparison of the results of Codyze and CogniCrypt$_{SAST}$ analyses of Bouncy Castle tests generated by CogniCrypt$_{TestGen}$.

CogniCrypt$_{SAST}$ generates misuses that depend upon other misuses. For instance, both tools have found misuses in the following code in listing 5.2. CogniCrypt$_{SAST}$ detects four misuses that are all true positives in this test case, and Codyze finds three misuses that are also true positives. The tools have two common misuses. First, on line 7, the parameter of Algorithm-ParameterGenerator's `init` method must be in any of 128, 192, 256 bits and not 1048 bits.

Second, the Cipher algorithm should not be RSA. In addition, CogniCrypt$_{SAST}$ identifies two more misuses on Line 11 that are caused by the first two misuses. The error indicates that the second parameter (`secretKey`) was not properly generated as a generated key, which is correct since the algorithms used in Cipher and KeyGenerator are different. The other error states that the third parameter (`algorithmParameters`) was not generated properly, which is also true due to the incorrect parameter used in the AlgorithmParameterGenerator. The dependent errors will be resolved by correcting the first two misuses. We have found more dependent errors in other cipher test cases but none in other test cases. Codyze detects another misuse that CogniCrypt$_{SAST}$ does not, which is the incorrect order of Cipher in Line 11. This misuse is caused by the fact that the Cipher has not been terminated properly, and a final call is required.

We counted the true positives for Cipher without the dependent errors in the Table 5.4. If we count the dependent misuses, then CogniCrypt$_{SAST}$ finds 71 true positives in the valid Cipher tests and 104 true positives in the invalid tests, which contain many duplicate errors. Therefore, we did not include the dependent errors when calculating precision and recall.

Suppose we only count the misuses caused by incorrect order of operations, as shown in the Table A.2.1 in the appendices. In that case, we can see that Codyze performed better in finding order misuses than CogniCrypt$_{SAST}$ in the Cipher test cases. If we calculate the precision and recall just for the Cipher test case for all valid and invalid cases, Codyze achieves 72,72% precision and 100% recall, and CogniCrypt$_{SAST}$ achieves 67,14 % precision and 83,92 % recall, which indicates that in this case, Codyze performs better than CogniCrypt$_{SAST}$. The overall precisions of CogniCrypt$_{SAST}$ and Codyze were very similar; however, Codyze's recall was about 3 percent higher, indicating that it was more effective in finding order misuses.

Listing 5.2: One of the valid test cases for the Cipher API that CogniCrypt$_{TestGen}$ generated from the original CrySL Bouncy Castle JCA ruleset.

```java
1  public void cipherValidTest7() throws NoSuchPaddingException,
     NoSuchAlgorithmException, InvalidKeyException,
     InvalidAlgorithmParameterException {
2
3     KeyGenerator keyGenerator0 = KeyGenerator.getInstance("AES");
4     SecretKey secretKey = keyGenerator0.generateKey();
5
6     AlgorithmParameterGenerator algorithmParameterGenerator0 =
       AlgorithmParameterGenerator.getInstance("AES");
7     algorithmParameterGenerator0.init(1048);
8     AlgorithmParameters algorithmParameters = algorithmParameterGenerator0.
     generateParameters();
9
10    Cipher cipher0 = Cipher.getInstance("RSA");
11    cipher0.init(1, secretKey, algorithmParameters);
12 }
```

CogniCrypt$_{SAST}$ detects some misuses that Codyze does not, and therefore the number of true positives for CogniCrypt$_{SAST}$ in some cases are more than Codyze. Those misuses are related to the constraints error. As an example, in the Cipher CrySL rule in the CONSTRAINTS section, it is stated that the input length of the Cipher object must be greater than 0. In the Listing 5.3 Line 4, the third parameter, which is the input length, is zero; therefore it is a misuse, which CogniCrypt$_{SAST}$ identified. However, Codyze does not detect this misuse, even though it has a relative MARK rule.

Listing 5.3: Part of a test case for the Cipher API from the Bouncy Castle tests

```
...
    Cipher cipher0 = Cipher.getInstance("RSA");
    cipher0.updateAAD(aadBytes);
    cipher0.doFinal(plainText, 0, 0, cipherText, 0);
...
```

In some cases, Codyze reports that a rule was violated and verified at the same time where there was no misuse, and we counted the violation as a false positive. For example, the listing 5.4 shows one of the invalid test cases of AlgorithmParameterGenerator from Bouncy Castle tests. Codyze identifies a violation of a rule on Line 3 that states that the parameter `secureRandom0`, that was used in AlgorithmParameterGenerator was not generated by the SecureRandom API. This error is a false positive since the SecureRandom API was correctly used to generate a random number (`secureRandom0`). On the same line (Line 3), Codyze identifies a rule verification that states that the `secureRandom0` was generated correctly with the SecureRandom API. This is in conflict with the previous misuse. This unusual behavior was also observed in several other test cases. There is also the same violation and verification at Line 6, which is redundant. On Line 6 of Listing 5.4, Codyze detected a violation against order because AlgorithmParameterGenerator was not properly terminated. It is a true positive, but in the Markers tab of Eclipse, this error had been displayed several times, which is unnecessary.

Listing 5.4: One of the invalid test cases for the AlgorithmParameterGenerator API that CogniCrypt$_{\text{TestGen}}$ generated from the original CrySL JCA ruleset.

```
1  public void algorithmParameterGeneratorInvalidTest5() throws
     NoSuchAlgorithmException {
2
3     SecureRandom secureRandom0 = SecureRandom.getInstance("DEFAULT");
4
5     AlgorithmParameterGenerator algorithmParameterGenerator0 =
     AlgorithmParameterGenerator.getInstance("AES");
6     algorithmParameterGenerator0.init(1048, secureRandom0);
7
8  }
```

Four MARK policies did not function, and as a result, they did not report any violations or verifications. They were MARK policies for the following APIs: DigestInputStream and DigestOutputStream, as well as CipherInputStream and CipherOutputStream. We checked for syntax issues in the relative files (entity and rule files), but there were no problems, and all of these policies were written the same as others and based on the instructions on Codyze's documentation page [coda]. Codyze, however, detected misuses in two of those API's related test cases (i.e., numbers 4 and 5 of Table 5.4) that were related to other APIs. In the CipherInputStream and CipherOutputStream tests, Codyze found violations in the order of Cipher in all of the test cases (valid and invalid), which are true positives. It was the same error that appeared in Listing 5.2 Line 11, which indicated that the Cipher was not correctly terminated.

The results of analyzing JCA tests using Codyze with translated JCA MARK rules and using with JCA CrySL rules are shown in the Table A.2.2 in the appendices. There were fewer incorrect test cases in the JCA tests than in the Bouncy Castle tests. Therefore, the number of misuses that each tool detected in total was less in the JCA tests than in the Bouncy Castle tests. As a result, there were fewer dependent errors in the Cipher test cases in the JCA tests than in the Bouncy Castle tests. The number of misuses in Table A.2.2 is calculated without considering the dependant errors. If we count the dependent errors, the number of misuses found in the valid Cipher cases will be 57 and in invalid 101. Codyze could not find some constraints errors in the JCA tests, as it did in the Bouncy Castle tests. Because the misuses found by each tool in the JCA tests are similar to those found in the Bouncy Castle tests, we will not discuss them again.

5.2.2.2 Discussing the results of analyzing Bouncy Castle tests using Codyze with Bouncy Castle MARK rules, and CogniCrypt$_{\text{SAST}}$ with the translated Bouncy Castle CrySL rules.

We further analyzed the Bouncy Castle tests using Codyze with the Bouncy Castle MARK rules and CogniCrypt$_{\text{SAST}}$ with translated Bouncy Castle MARK rules. The results of the analyses are presented in Table 5.5. In Table 5.5, all column titles are the same as those in Table 5.4 discussed in section 5.2.2.1. TP refers to the number of true positives. An error is a true positive when it is caused by a violation of a MARK rule or its translation to CrySL. FP is the number of false positives. An error is a false positive when no violation of the corresponding MARK rule or its translation into CrySL has occurred.

In total, Codyze identified three types of misuses in the test - invalid or non-specified provider name, non- or incorrect padding for RSA Cipher, using the forbidden calls of SecureRandom API, and one invalid size of the key in the RSAKeyGenParameterSpec test case, all of which were true positives. CogniCrypt$_{\text{SAST}}$ identified all of those types of misuses but in fewer cases. CogniCrypt$_{\text{SAST}}$ generally found fewer misuses than Codyze. For example, in the Cipher tests (number 6 in Table 5.5), it appears that CogniCrypt$_{\text{SAST}}$ did not identify as many misuses as Codyze. This is because CogniCrypt$_{\text{SAST}}$ was unable to detect all of the provider misuses. The reason is the partial translation of the MARK rules to CrySL. Not indicating a provider when using an API is considered a misuse, as stated in section 4.3.3.2. To translate this rule to CrySL, we must add the methods without a provider parameter to the FORBIDDEN section of the CrySL rule, which was not possible for all methods within an API. Therefore CogniCrypt$_{\text{SAST}}$ could not detect all the provider misuses. The same holds for SecureRandom forbidden calls. Since we could not add all the forbidden methods (see Listing 4.4 Lines 7 and 8) in the FORBIDDEN section of SecureRandom CrySL rule; therefore, CogniCrypt$_{\text{SAST}}$ could not detect all forbidden calls.

As indicated in the Table 5.5, Codyze does not generate any false positives, while CogniCrypt$_{\text{SAST}}$ generates some false positives. In some tests, CogniCrypt$_{\text{SAST}}$ detects order misuses, which are all false positives. The order specified in the rules includes only optional usage of the methods, so any order is acceptable. Therefore, any order misuse generated by CogniCrypt$_{\text{SAST}}$ is a false positive. As an example, the false positives in the Mac tests in the valid and invalid test cases generated by CogniCrypt$_{\text{SAST}}$ are all caused by order misuses. CogniCrypt$_{\text{SAST}}$generates order errors in Cipher, SecureRandom, Mac, and KeyAgreement test cases that are all false positives as described. An invalid order error can only be generated when using the Cipher API, with the Cipher mode being one of CCM, GCM, CBC, or CTR. This particular case requires specific method calls according to the MARK Bouncy Castle rules. However, the Cipher mode was not specified in any of the Bouncy Castle test cases.

CogniCrypt$_{\text{SAST}}$ did not detect all the non- or incorrect padding for RSA Cipher misuses. Based on the Bouncy Castle MARK rules, if an RSA Cipher is used without padding or with incorrect padding in the `getInstance` method of Cipher, this error should occur. This rule is defined as a constraint in the Cipher CrySL rule. CogniCrypt$_{\text{SAST}}$ can only find misuses of the parameters used within the EVENTS section. According to the MARK rules, the `getInstance` method without a provider was considered insecure. However, this method could not be added to the FORBIDDEN section of the Cipher CrySL rule, as explained in Section 4.3.3.2. Therefore, CogniCrypt$_{\text{SAST}}$ does not detect RSA padding misuse. Moreover, the other misuse of the RSAKeyGenParameterSpec API, regarding the incorrect size of the key, occurred in only one test case and was detected by both tools.

No.	Test Name	Valid tests	Invalid tests	Codyze				CogniCrypt$_{\text{SAST}}$			
				Valid		Invalid		Valid		Invalid	
				TP	FP	TP	FP	TP	FP	TP	FP
1	AlgorithmParameterGenerator	5	9	7	0	11	0	1	0	3	0
2	AlgorithmParameters	9	5	13	0	5	0	2	0	2	0
3	CertPathTrustManagerParameters	1	0	1	0	0	0	0	0	0	0
4	CipherInputStream	3	5	6	0	10	0	0	0	0	0
5	CipherOutputStream	3	5	6	0	10	0	0	0	0	0
6	Cipher	56	69	164	0	177	0	8	1	21	0
7	DHGenParameterSpec	1	0	0	0	0	0	0	0	0	0
8	DHParameterSpec	2	0	0	0	0	0	0	0	0	0
9	DigestInputStream	2	4	2	0	4	0	0	0	0	0
10	DigestOutputStream	2	4	2	0	4	0	0	0	0	0
11	DSAGenParameterSpec	2	0	0	0	0	0	0	0	0	0
12	DSAParameterSpec	1	0	0	0	0	0	0	0	0	0
13	ECGenParameterSpec	1	0	0	0	0	0	0	0	0	0
14	ECParameterSpec	1	0	0	0	0	0	0	0	0	0
15	GCMParameterSpec	2	0	2	0	0	0	0	0	0	0
16	HMACParameterSpec	1	0	0	0	0	0	0	0	0	0
17	IvParameterSpec	2	0	2	0	0	0	0	2	0	0
18	KeyAgreement	7	23	10	0	28	0	1	0	5	1
19	KeyFactory	6	0	10	0	0	0	3	0	0	0
20	KeyGenerator	8	8	11	0	11	0	2	0	2	0
21	KeyManagerFactory	6	4	6	0	0	0	0	0	0	0
22	KeyPairGenerator	6	10	6	0	10	0	1	0	3	0
23	KeyStoreBuilderParameters	1	0	0	0	0	0	0	0	0	0
24	KeyStore	13	21	13	0	21	0	4	0	6	0
25	Mac	12	21	27	0	35	0	0	1	0	2
26	MessageDigest	9	13	9	0	13	0	2	0	3	0
27	MGF1ParameterSpec	1	0	0	0	0	0	0	0	0	0
28	OAEPParameterSpec	1	0	0	0	0	0	0	0	0	0
29	PBEKeySpec	1	1	1	0	1	0	0	0	0	0
30	PBEParameterSpec	2	0	2	0	0	0	0	0	0	0
31	PKIXBuilderParameters	2	0	1	0	0	0	0	0	0	0
32	PKIXParameters	2	0	1	0	0	0	0	0	0	0
33	RSAKeyGenParameterSpec	1	0	1	0	0	0	1	0	0	0
34	SecretKeyFactory	3	2	5	0	2	0	1	1	1	0
35	SecureRandom	38	16	71	0	19	0	10	1	4	2
36	SSLContext	5	5	5	0	0	0	0	0	0	0
37	SSLParameters	4	4	0	0	0	0	0	0	0	0
38	TrustManagerFactory	6	4	6	0	0	0	0	0	0	0
Total		**228**	**233**	390	0	361	0	36	6	50	5
Results		**Precision(%)**		100				88,65			
		Recall(%)		100				11,45			

Table 5.5: Comparison of the results of Codyze and CogniCrypt$_{\text{SAST}}$ analyses with MARK rules of Bouncy Castle tests generated by CogniCrypt$_{\text{TestGen}}$.

To summarize, CogniCrypt$_{\text{SAST}}$ provided better precision and recall than Codyze in analyzing the Bouncy Castle and JCA tests; however, both tools performed well in general. Nevertheless, when comparing errors relating to order misuses, Codyze performs better than CogniCrypt$_{\text{SAST}}$.

The CogniCrypt$_{\text{SAST}}$ with translated CrySL rules did not perform well in identifying the misuses based on MARK rules. According to Table 5.5, CogniCrypt$_{\text{SAST}}$ only found 11 percent of the misuses. This could indicate that MARK is a more expressive DSL than CrySL. CogniCrypt$_{\text{TestGen}}$ generated tests, however, did not cover all sections of the CrySL rules. To determine whether all translated MARK rules deliver the same performance as the CrySL rules, we need a benchmark that covers all vulnerabilities related to all parts of the CrySL rules.

In addition, the rules defined by MARK in the Bouncy Castle ruleset did not address many of the cryptographic vulnerabilities we know today, such as the use of non-random keys in encryption. The available CrySL rules, in contrast, cover many known vulnerabilities. By analyzing the test cases generated by the CrySL rules using translated MARK rules in Codyze, we were able to understand how Codyze would perform with the new rules that define more vulnerabilities.

Although the test cases generated by CogniCrypt$_{\text{TestGen}}$ did not cover all sections of the CrySL rules, they still contain many vulnerabilities that detecting them could be very useful. Notably, the test cases included parts of the CrySL rules that could be translated to mark. Thus, we expected Codyze to be similar to CogniCrypt$_{\text{SAST}}$ in terms of rules. However, after the analysis, we discovered that some rules behave strangely, and some constraint rules do not work. Additionally, a few MARK policies did not function at all. The results also showed that CogniCrypt$_{\text{SAST}}$ could not detect all order misuses and, in some cases, find order misuses that are false positives.

6

Conclusion

In this thesis, we compared two static analysis tools, Codyze and CogniCrypt$_{SAST}$, that use white-listing approaches to detect misuse of cryptographic APIs in order to identify their similarities and differences. They are both capable of analyzing Java programs to detect misuses of Java cryptographic APIs, and Codyze can also analyze C and C++ programs. Both tools employ rules written in domain-specific languages that specify the proper usages of the cryptographic APIs. Codyze uses MARK rules, and CogniCrypt$_{SAST}$ uses CrySL rules. Moreover, both tools convert the source code into a graph representation for analysis.

The theoretical comparison (see Section 4.1) of the tools was based on examining published papers, documentation pages, and source code for each tool. Based on the theoretical comparison, we have concluded that both tools are flow-sensitive data-flow analyses. However, CogniCrypt$_{SAST}$ analysis is also context- and field-sensitive and inter-procedural, whereas Codyze analysis is intra-procedural. Nevertheless, when evaluating the performance of Codyze and CogniCrypt$_{SAST}$ with CryptoAPI-Bench (see Section 5.1.2), we discovered that Codyze detects misuses in 20 inter-procedural test cases out of 22 vulnerable test cases and is therefore partially inter-procedural. To check whether Codyze is entirely inter-procedural or not, we need more inter-procedural test cases. Furthermore, CryptoAPI-Bench does not provide context-sensitive test cases to check the context sensitivity of the tools. Generally, the number of test cases was limited and may not provide a reliable indication of the analysis properties of each tool; therefore, we require more test cases that cover different aspects of analysis to examine the tool's analysis properties in the future.

CogniCrypt$_{SAST}$ converts source code to a Soot IR, Jimple, and constructs call graphs of Jimple code. Therefore, CogniCrypt$_{SAST}$ can only analyze Java and Android programs since Soot does not support other programming languages than Java. In contrast, Codyze generates CPGs from the source code and writes queries to retrieve data from the CPG. Since CPGs are independent of the programming language, Codyze can analyze programs in several programming languages with the same queries. Currently, MARK rules are available for Java, C, and C++, and Codyze is capable of analyzing C, C++, Java, and Android programs.

Further, we performed a comparison of the MARK and CrySL DSLs in order to determine which DSL is more expressive in specifying correct usages of the Java cryptographic APIs (see Section 4.3). We translated the Bouncy Castle JCA and JCA CrySL rules to MARK, and since there were only MARK rules for the Bouncy Castle API, we translated the MARK Bouncy Castle ruleset to CrySL. We realized that CrySL and MARK express different things; however, they also have commonalities. Therefore, some parts of the rules of each DSL cannot be translated from one to another. For example, the ENSURES section of CrySL rules or the unspecified provider rules in MARK. Thus the translations do not entirely represent the original rules. Although, as part of our future work, it might be possible to include several built-in functions in both DSLs that will enable us to translate the entire contents of one language to another. Additionally, we discovered that CrySL sections are designed to be related, but the MARK rules are independent of each other, making it more flexible to write rules. This flexibility may make MARK more challenging to use, and this could be determined by conducting a user study for the use of the CrySL and MARK DSLs. As a user study is outside the scope of our thesis, it could be done as future work. Moreover, the results of rule translation indicated that when specifying the same rules for an API, MARK rules occupy more space than CrySL rules. Because, for every rule in MARK, we need to specify its structure, which consists of a name, a condition, a precondition, and a message on failure. While in CrySL, we only add the condition and precondition to the relative section.

We further evaluated the tools with two benchmarks, CryptoAPI-Bench and test cases generated by CogniCrypt$_{\text{TestGen}}$. Analyzing CryptoAPI-Bench showed that Codyze had better precision than CogniCrypt$_{\text{SAST}}$ in finding misuses, but CogniCrypt$_{\text{SAST}}$ covered more vulnerabilities. We also discovered that CogniCrypt$_{\text{SAST}}$ is partially field-sensitive and rarely path-sensitive. However according to the theoretical comparison, CogniCrypt$_{\text{SAST}}$ is field-sensitive but not path sensitive. Additionally, we noticed that Codyze is partially field-sensitive and, contrary to what we discovered in the theoretical comparison, is partially inter-procedural. CryptoAPI-Bench only includes a few test cases for each case, and also, the test cases could be incorrect or biased toward CryptoGuard. Therefore, to ensure the analysis properties of the tools, we need to analyze more test cases that we plan to do in the future. Lastly, the error messages that CogniCrypt$_{\text{SAST}}$ and Codyze generated for the same cryptographic vulnerability were different. By conducting a user study in the future, we may determine which one is preferred by users.

Moreover, we generated test cases with CogniCrypt$_{\text{TestGen}}$ using the Bouncy Castle JCA, JCA CrySL rules, and translated Bouncy Castle CrySL rules (see Section 5.2.2.2). Some test cases generated from the translated Bouncy Castle CrySL rules contained compiler errors. However, Codyze found two misuses in those faulty test cases, which were true positives, but CogniCrypt$_{\text{SAST}}$ could only find those misuses after we fixed the faulty test cases. This illustrates Codyze's capability to analyze uncompilable codes. In analyzing Bouncy Castle JCA and JCA tests, we used the translated MARK rules in Codyze. Results showed that Codyze performed better than CogniCrypt$_{\text{SAST}}$ in finding order misuse; however, in general, the performance of CogniCrypt$_{\text{SAST}}$ was better than Codyze in terms of precision and recall. Codyze was unable to detect some constraint misuses, and certain MARK policies did not work at all. In addition, we used translated Bouncy Castle CrySL rules and Bouncy Castle MARK rules to analyze the Bouncy Castle tests. Based on the results, CogniCrypt$_{\text{SAST}}$ could only detect 11% of the misuses defined by the Bouncy Castle MARK rules. This is due to the partial translation of MARK to CrySL. Currently, it is not possible to include all types of methods in the FORBIDDEN section of the CrySL rule. Further, the restrictions defined in the CONSTRAINTS section are only valid for parameters used in the EVENTS section. Thus, CogniCrypt$_{\text{SAST}}$ was unable to detect some misuses.

According to our results, Codyze and CogniCrypt$_{\text{SAST}}$ performed very similarly; however, some aspects of both tools and their respective DSLs could be improved. There should be more MARK and CrySL rules to address more cryptographic vulnerabilities. CrySL and MARK could have more or enhanced built-in functionality to specify rules more effectively and efficiently. The tools sometimes fail to detect misuse in a program that uses an API incorrectly, despite having rules for the relative API, which also requires correction. Also, we obtained information about the tools' analysis properties, as described above, which also needs further investigation that we will present in the future work chapter (cf. Chapter 7).

7 Future Work

In the future, we may conduct a user study to compare the tools and the DSLs and determine how user-friendly they are. Participants could use the tools to analyze sample codes, write rule specifications using DSLs, and record their observations. Consequently, we can determine which design users prefer and how each tool can be improved.

After translating the rules, we figured that they are not entirely translatable to each other. Adding more built-in functions to each DSL would enable them to express more contents of the other DSL. Afterward, we can translate the rules again and determine if MARK and CrySL DSLs are equally expressive. In addition, there was no appropriate estimation of the time required to produce a CPG. This could be accomplished by measuring the execution time of the CPG, and repeat the experiment for all benchmarks to determine an average time. In the same way, we could also obtain the run time for CogniCrypt$_{SAST}$ and Codyze and compare the tools' run times. CryptoAPI-Bench does not contain many test cases to examine the tools into finding out their actual properties and might not provide an accurate estimate of the analyzer's properties. In addition, it may favor CryptoGuard by only including test cases that demonstrate CryptoGuard's analysis properties. Moreover, it provided no context-sensitive test cases. Therefore, we require more benchmarks that include inter-procedural, field-, path-, and context-sensitive cases to examine the tools further. In this manner, we may obtain a more accurate estimate of each tool's properties.

The evaluation revealed that Codyze and CogniCrypt$_{SAST}$ contain some flaws in their analysis that resulted in false positives when analyzing the benchmarks. We thus raised issues on their Github pages; these issues must also be resolved by Codyze and CogniCrypt$_{SAST}$ developers to improve each tool's performance.

Bibliography

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[and] Androguard, a python tool to analyze android files. `https://www.github.com/androguard/androguard`. Accessed: 2021-12-20.

[api] Cognicrypt crypto api rules. `https://github.com/CROSSINGTUD/Crypto-API-Rules`. Accessed: 2021-07-23.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI 2014 - Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 259–269. Association for Computing Machinery, 2014.

[ARY19] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61. IEEE, 2019.

[ASCIM14] Information Association, Jesús Sánchez Cuadrado, Javier Izquierdo, and Jesús Molina. *Comparison between Internal and External DSLs via RubyTL and Gra2MoL*, pages 816–838. 01 2014.

[ASW+17] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, 2017.

[BAG12] Ankica Bariic, Vasco Amaral, and Miguel Goulão. Usability evaluation of domain-specific languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 342–347, 2012.

[BAGB11a] Ankica Barisic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. How to reach a usable dsl? moving toward a systematic evaluation. *presented at the 5th International Workshop on Multi-Paradigm Modeling (MPM'2011) at Models 2011*, Volume 50, 01 2011.

[BAGB11b] Ankica Barisic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. Quality in use of dsls: Current evaluation methods. 01 2011.

[bc] Bouncycastle cryptography api provider. `https://www.bouncycastle.org/`. Accessed: 2021-12-20.

[ben] Cryptoapi-bench github page. `https://github.com/CryptoGuardOSS/cryptoapi-bench`. Accessed: 2021-12-20.

[bot] Botan cryptography library for c++. `https://botan.randombit.net/`. Accessed: 2021-12-20.

[BRS+17] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 334–349, 2017.

[BSIa] Bsi, bundesamt feur sicherheit in der informationsteschnik, official webpage. `https://www.bsi.bund.de/DE/Home/home_node.html`. Accessed: 2021-12-20.

[bsib] Bsi tr-02102-1 version 2019-01. `https://github.com/CROSSINGTUD/Thesis-2021-Asgharivaskasi/blob/main/TR-02102-1%20Version%202019-01-deutsch.pdf`. Accessed: 2021-01-20.

[BSI21] BSI. Bsi tr-02102-1: "cryptographic mechanisms: Recommendations and key lengths" version: 2021-1, 2021.

[cip] Java cryptography architecture (jca)- cipher. `https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html`. Accessed: 2021-07-18.

[coda] Codyze documentation page. `https://www.codyze.io/docs/`. Accessed: 2021-07-24.

[codb] Codyze github repository. `https://github.com/Fraunhofer-AISEC/codyze`. Accessed: 2021-07-24.

[cry] Cryptoanalysis github page. `https://github.com/CROSSINGTUD/CryptoAnalysis`. Accessed: 2021-07-24.

[DLZ14] X. Wang D. Lazar, H. Chen and N. Zeldovich. Why does cryptographic software fail?: a case study and open problems. pages 1–7, 2014.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[GFF+15] François Gagnon, Marc-Antoine Ferland, Marc-Antoine Fortier, Simon Desloges, Jonathan Ouellet, and Catherine Boileau. Androssl: A platform to test android applications connection security. 10 2015.

[gre] Apache gremlin graph traversal language. `https://tinkerpop.apache.org/gremlin.html`. Accessed: 2021-07-24.

[HPvD09] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-specific languages in practice: A user study on the success factors. In *Model Driven Engineering Languages and Systems*, pages 423–437, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[HVO17] Cormac Herley and P.C. Van Oorschot. Sok: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 99–120, 2017.

[jac] Jackson, the java json library. `https://github.com/FasterXML/jackson`. Accessed: 2021-12-20.

[jca] Java cryptography architecture (jca) reference guide. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`. Accessed: 2021-07-18.

[jce] Jce oracle documentation page. `https://docs.oracle.com/cd/E19944-01/819-4480/IM_installation_appendixF_JCE.html`. Accessed: 2021-12-20.

[Kit04] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele Univ.*, 33, 08 2004.

[KMC12] Tomaž Kosar, Marjan Mernik, and Jeff Carver. Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empirical Software Engineering*, 17:276–304, 06 2012.

[KOM+10] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João, Maria Pereira, Matej Repinšek, Daniela Cruz, and Pedro Rangel Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 438, 05 2010.

[KR20] Rakshit Krishnappa Ravi. Cognicrypttestgen—generating tests for crypto apis, 2020.

[Kr0] Stefan Krüger. *CogniCrypt–The Secure Integration of Cryptographic Software*. PhD dissertation, Paderborn University, 2020.

[lCX16] Georgios Karopou-los lexia Chatzikonstantinou, Christoforos Ntantogian and Christos Xenakis. Evaluation of cryptography usage in android applications. In *9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerlyBIONETICS)*, pages 83–90, New York City, New York, USA, 2016. ICST.

[LCZG18] Juanru Li, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. pages 412–425, 10 2018.

[LH03] Ondrej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. volume 2622, pages 153–169, 04 2003.

[lsp] Microsoft language server protocol(lsp). `https://microsoft.github.io/language-server-protocol/`. Accessed: 2021-12-06.

[mar] Mark github repository. `https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin`. Accessed: 2021-07-24.

[MAST19] Avijit Mallik, Abid Ahsan, Mhia Shahadat, and Jia-Chi Tsou. Man-in-the-middle-attack: Understanding in simple words. 3:77–92, 01 2019.

[MBB18] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 133–146, New York, NY, USA, 2018. Association for Computing Machinery.

[MEK13] Yanick Fratantonio Manuel Egele, David Brumley and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS '13: Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*, New York, NY, USA, 2013. Association for Computing Machinery.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[MKR17] Kathleen M. Moriarty, Burton S. Kaliski, and Andreas Rusch. Pkcs #5: Password-based cryptography specification version 2.1. *RFC*, 8018:1–40, 2017.

[NI04] David Naccache and Gemplus International. Padding attacks on rsa. *Information Security Technical Report*, 4, 08 2004.

[PLH11] Ondrej Lhoták Patrick Lam, Eric Bodden and Laurie Hendren. The soot framework for java program analysis: a retrospective, 2011.

[PP08] Michael Pfeiffer and Josef Pichler. A comparison of tool support for textual domain-specific languages. 10 2008.

[pre] Codyze presentation by dennis titze. `https://omnisecure.berlin/wp-content/uploads/os20_Titze_Dennis.pdf`. Accessed: 2021-12-20.

[RLK07] Thomas Reps, Akash Lal, and Nick Kidd. Program analysis using weighted push-down systems. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07, page 23–51, Berlin, Heidelberg, 2007. Springer-Verlag.

[rsa] Rsa, computer and network security company. https://www.rsa.com/.

[Ryd79] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979.

[SAB17] Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. 2(1), 2015.

[SDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java (Artifact). *Dagstuhl Artifacts Series*, 2(1):12:1–12:2, 2016.

[SGT$^+$14] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.

[SKM19] K. Ali-E. Bodden S. Kreuger, J. Spath and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, Oct 2019.

[SNB16] M. Mezini S. Nadi, S. Krüger and E. Bodden. "jumping through hoops": Why do java developers struggle with cryptography apis? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[soo] Soot github page. `https://github.com/soot-oss/soot`. Accessed: 2021-12-20.

[Spa19] Johannes Spaeth. *Synchronized Pushdown Systems for Pointer and Data-Flow Analysis.* PhD dissertation, Paderborn University, 2019.

[SR19] Sharmin Afrose-Fahad Shaon Ke Tian Miles Frantz Danfeng (Daphne) Yao Murat Kantarcioglu Sazzadur Rahaman, Ya Xiao. 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*, pages 2455–2472, 2019.

[tk] Tink cryptography library. `https://opensource.google/projects/tink`. Accessed: 2021-12-20.

[WZH$^+$13] Kirsten Winter, Chenyi Zhang, Ian Hayes, Nathan Keynes, Cristina Cifuentes, and Lian Li. Path-sensitive data flow analysis simplified. volume 8144, pages 415–430, 10 2013.

[xte] Xtext website. `http://www.eclipse.org/Xtext/`. Accessed: 2021-12-20.

[YGAR14] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[redacted]

Appendix

A.1 Translations

All links related to the translated rules can be found below.

- Translation of JCA CrySL rules to MARK
  - Translated ruleset: `https://github.com/CROSSINGTUD/Thesis-2021-Asgharivaskasi/tree/main/TranslatedCrySLJCARulesToMARK`
  - Original ruleset: `https://github.com/CROSSINGTUD/Crypto-API-Rules/tree/master/JavaCryptographicArchitecture`
- Translation of Bouncy Castle JCA CrySL rules to MARK
  - Translated ruleset: `https://github.com/CROSSINGTUD/Thesis-2021-Asgharivaskasi/tree/main/TranslatedCrySLBCJCARulesToMARK`
  - Original ruleset: `https://github.com/CROSSINGTUD/Crypto-API-Rules/tree/master/BouncyCastle-JCA`
- Translation of Bouncy Castle MARK rules to CrySL
  - Translated ruleset: `https://github.com/CROSSINGTUD/Thesis-2021-Asgharivaskasi/tree/main/TranslatedMARKBCRulesToCrySL`

– Original ruleset: `https://github.com/Fraunhofer-AISEC/codyze/tree/main/src/dist/mark/bouncycastle`

A.2 Tables

A.2.1 Results of analyzing MARK Bouncy Castle tests

No.	Test Name	Valid tests	Invalid tests	Codyze				CogniCrypt$_{SAST}$			
				Valid		Invalid		Valid		Invalid	
				TP	FP	TP	FP	TP	FP	TP	FP
1	DHGenParameterSpec	1	0	0	0	0	0	0	0	0	0
2	DHParameterSpec	2	0	0	0	0	0	0	0	0	0
3	DHPrivateKeySpec	1	0	0	0	0	0	0	0	0	0
4	DHPublicKeySpec	1	0	0	0	0	0	0	0	0	0
5	DSAGenParameterSpec	2	0	0	0	0	0	0	0	0	0
6	DSAParameterSpec	1	0	0	0	0	0	0	0	0	0
7	DSAPrivateKeySpec	1	0	0	0	0	0	0	0	0	0
8	DSAPublicKeySpec	1	0	0	0	0	0	0	0	0	0
9	ECFieldF2m	3	0	0	0	0	0	0	0	0	0
10	ECFieldFp	1	0	0	0	0	0	0	0	0	0
11	ECGenParameterSpec	1	0	0	0	0	0	0	0	0	0
12	ECParameterSpec	1	0	0	0	0	0	0	0	0	0
13	ECPoint	1	0	0	0	0	0	0	0	0	0
14	ECPrivateKeySpec	1	0	0	0	0	0	0	0	0	0
15	ECPublicKeySpec	1	0	0	0	0	0	0	0	0	0
16	EncodedKeySpec	1	0	0	0	0	0	0	0	0	0
17	GCMParameterSpec	2	0	0	0	0	0	0	0	0	0
18	HMACParameterSpec	1	0	0	0	0	0	0	0	0	0
19	IvParameterSpec	2	0	2	0	0	0	0	0	0	0
20	MGF1ParameterSpec	1	0	0	0	0	0	0	0	0	0
21	OAEPParameterSpec	1	0	0	0	0	0	0	0	0	0
22	PBEKeySpec	3	0	0	0	0	0	0	0	0	0
23	PBEParameterSpec	2	0	0	0	0	0	0	0	0	0
24	PKCS8EncodedKeySpec	2	0	0	0	0	0	0	0	0	0
25	PSSParameterSpec	2	0	0	0	0	0	0	0	0	0
26	RSAKeyGenParameterSpec	2	0	0	0	0	0	0	0	0	0
27	RSAMultiPrimePrivateCrtKeySpec	2	0	0	0	0	0	0	0	0	0
28	RSAPrivateCrtKeySpec	2	0	0	0	0	0	0	0	0	0
29	RSAPrivateKeySpec	2	0	0	0	0	0	0	0	0	0
30	RSAPublicKeySpec	2	0	0	0	0	0	0	0	0	0
31	SecretKeySpec	2	0	0	0	0	0	0	0	0	0
32	X509EncodedKeySpec	2	0	0	0	0	0	0	0	0	0
Total		**50**	**0**	2	0	0	0	0	0	0	0

Table A.1: Comparison of the results of Codyze and CogniCrypt$_{SAST}$ analyses of MARK Bouncy Castle tests generated by CogniCrypt$_{TestGen}$.

A.2.2 Results of analyzing Bouncy Castle tests, only addressing the order misuses

No.	Test Name	Valid tests	Invalid tests	Codyze				CogniCrypt$_{\text{SAST}}$			
				Valid		Invalid		Valid		Invalid	
				TP	FP	TP	FP	TP	FP	TP	FP
1	AlgorithmParameterGenerator	5	9	0	0	9	0	0	0	9	0
2	AlgorithmParameters	9	5	0	0	5	0	0	0	5	0
3	CertPathTrustManagerParameters	1	0	0	0	0	0	0	0	0	0
4	CipherInputStream	3	5	3	0	5	0	0	0	5	0
5	CipherOutputStream	3	5	3	0	5	0	1	0	5	0
6	Cipher	56	69	9	21	47	0	0	0	47	23
7	DHGenParameterSpec	1	0	0	0	0	0	0	0	0	0
8	DHParameterSpec	2	0	0	0	0	0	0	0	0	0
9	DigestInputStream	2	4	0	0	0	0	0	1	4	0
10	DigestOutputStream	2	4	0	0	0	0	0	2	4	0
11	DSAGenParameterSpec	2	0	0	0	0	0	0	0	0	0
12	DSAParameterSpec	1	0	0	0	0	0	0	0	0	0
13	ECGenParameterSpec	1	0	0	0	0	0	0	0	0	0
14	ECParameterSpec	1	0	0	0	0	0	0	0	0	0
15	GCMParameterSpec	2	0	0	0	0	0	0	0	0	0
16	HMACParameterSpec	1	0	0	0	0	0	0	0	0	0
17	IvParameterSpec	2	0	0	0	0	0	0	0	0	0
18	KeyAgreement	7	23	0	0	23	0	0	0	23	0
19	KeyFactory	6	0	0	0	0	0	0	0	0	0
20	KeyGenerator	8	8	0	0	8	0	0	0	8	0
21	KeyManagerFactory	6	4	0	0	4	0	0	0	4	0
22	KeyPairGenerator	6	10	0	0	10	0	0	0	10	0
23	KeyStoreBuilderParameters	1	0	0	0	0	0	0	0	0	0
24	KeyStore	13	21	0	0	17	0	0	0	17	0
25	Mac	12	21	0	7	21	0	0	0	21	0
26	MessageDigest	9	13	0	1	2	1	0	0	5	0
27	MGF1ParameterSpec	1	0	0	0	0	0	0	0	0	0
28	OAEPParameterSpec	1	0	0	0	0	0	0	0	0	0
29	PBEKeySpec	1	1	0	0	1	0	0	0	1	0
30	PBEParameterSpec	2	0	0	0	0	0	0	0	0	0
31	PKIXBuilderParameters	2	0	0	0	0	0	0	0	0	0
32	PKIXParameters	2	0	0	0	0	0	0	0	0	0
33	RSAKeyGenParameterSpec	1	0	0	0	0	0	0	0	0	0
34	SecretKeyFactory	3	2	0	0	2	0	0	0	2	0
35	SecureRandom	38	16	3	0	0	0	0	2	0	7
36	SSLContext	5	5	0	0	5	0	0	0	5	0
37	SSLParameters	4	4	0	1	4	0	0	0	3	0
38	TrustManagerFactory	6	4	0	0	4	0	0	0	4	0
Total		**228**	**233**	18	30	172	1	1	5	182	30
Results		**Precision(%)**		85,97				83,94			
		Recall(%)		90,04				86,72			

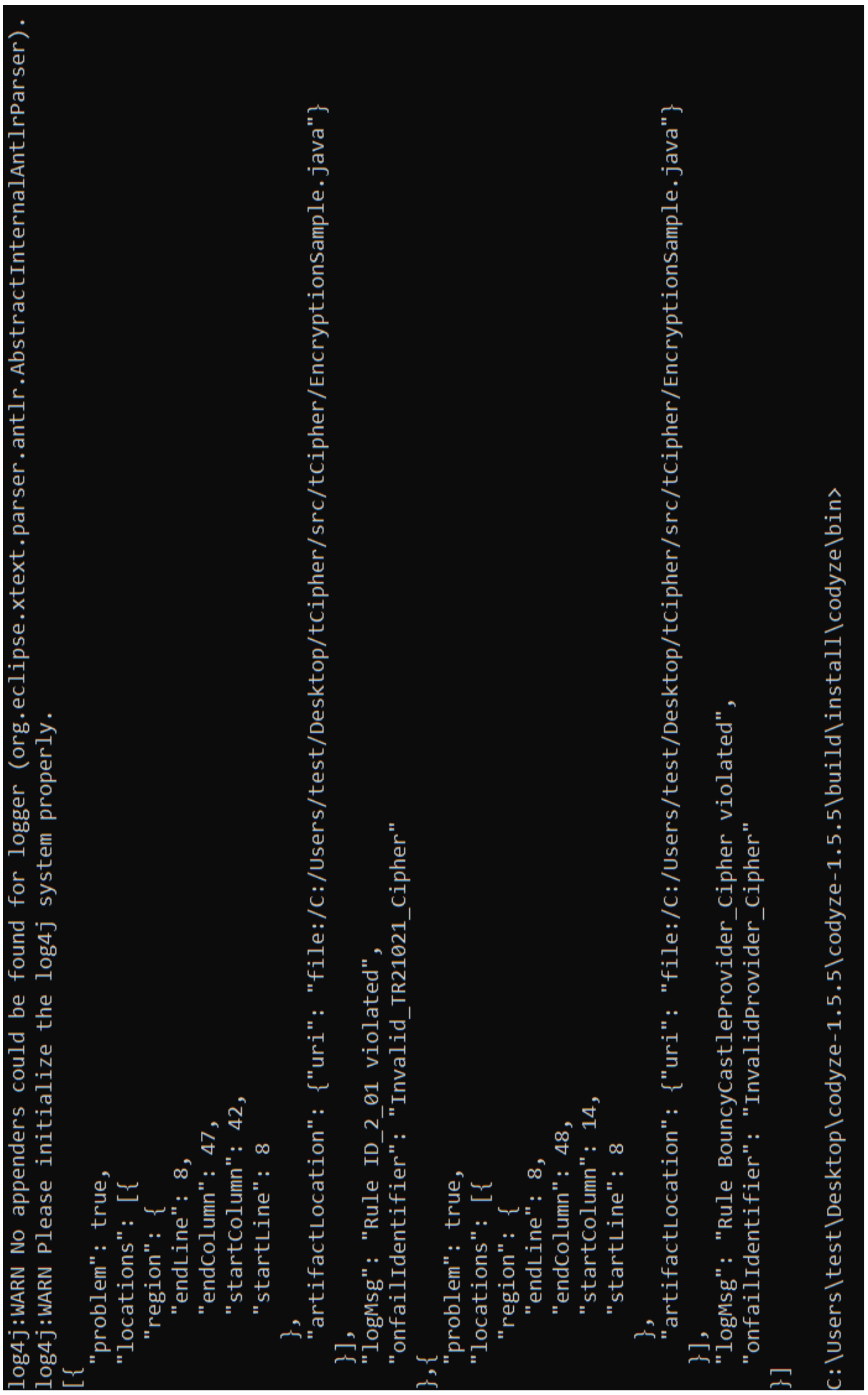
Table A.2: Comparison of the results of Codyze and CogniCrypt$_{\text{SAST}}$ analyses, only addressing the order misuses, of Bouncy Castle tests generated by CogniCrypt$_{\text{TestGen}}$.

A.2.3 Results of analyzing JCA tests

No.	Test Name	Valid tests	Invalid tests	Codyze				CogniCrypt$_{SAST}$			
				Valid		Invalid		Valid		Invalid	
				TP	FP	TP	FP	TP	FP	TP	FP
1	AlgorithmParameterGenerator	5	9	0	2	9	2	0	3	9	0
2	AlgorithmParameters	9	5	13	9	5	5	18	0	5	0
3	CertPathTrustManagerParameters	1	0	0	0	0	0	0	0	0	0
4	CipherInputStream	3	5	3	0	5	0	1	0	6	0
5	CipherOutputStream	3	5	3	0	5	0	1	0	6	0
6	Cipher	56	69	33	21	65	0	33	28	83	38
7	DHGenParameterSpec	1	0	1	0	0	0	1	0	0	0
8	DHParameterSpec	2	0	0	0	0	0	0	0	0	0
9	DigestInputStream	2	4	0	0	0	0	1	2	5	0
10	DigestOutputStream	2	4	0	0	0	0	1	2	5	0
11	DSAGenParameterSpec	2	0	0	0	0	0	0	0	0	0
12	DSAParameterSpec	1	0	0	0	0	0	0	0	0	0
13	ECGenParameterSpec	1	0	0	0	0	0	0	0	0	0
14	ECParameterSpec	1	0	0	0	0	0	0	0	0	0
15	GCMParameterSpec	2	0	0	2	0	0	1	0	0	0
16	HMACParameterSpec	1	0	0	0	0	0	0	0	0	0
17	IvParameterSpec	2	0	0	0	0	0	1	0	0	0
18	KeyAgreement	7	23	0	2	23	6	0	0	23	0
19	KeyFactory	6	0	0	4	0	0	0	0	0	0
20	KeyGenerator	8	8	0	3	8	3	0	0	8	0
21	KeyManagerFactory	6	4	0	8	4	0	0	0	4	0
22	KeyPairGenerator	6	10	4	0	14	0	4	0	12	0
23	KeyStoreBuilderParameters	1	0	0	0	0	0	0	0	0	0
24	KeyStore	13	21	0	0	17	0	0	0	17	0
25	Mac	12	21	0	7	21	0	1	0	22	0
26	MessageDigest	9	13	0	1	2	1	2	0	8	0
27	MGF1ParameterSpec	1	0	0	0	0	0	0	0	0	0
28	OAEPParameterSpec	1	0	0	0	0	0	0	0	0	0
29	PBEKeySpec	1	1	0	1	1	1	0	0	1	0
30	PBEParameterSpec	2	0	0	1	0	0	0	0	0	0
31	PKIXBuilderParameters	2	0	0	0	0	0	0	0	0	0
32	PKIXParameters	2	0	0	0	0	0	0	0	0	0
33	RSAKeyGenParameterSpec	1	0	0	0	0	0	0	0	0	0
34	SecretKeyFactory	3	2	0	4	2	0	0	0	2	0
35	SecureRandom	38	16	3	0	3	0	0	2	0	7
36	SSLContext	5	5	10	0	5	0	10	0	5	0
37	SSLParameters	4	4	0	1	4	0	0	0	3	0
38	TrustManagerFactory	6	4	0	13	4	0	0	0	4	0
Total		**228**	**233**	81	71	207	18	116	10	248	33
Results		**Precision(%)**		73,35				78,70			
		Recall(%)		81,15				92,09			

Table A.3: Comparison of the results of Codyze and CogniCrypt$_{SAST}$ analyses of JCA tests generated by CogniCrypt$_{TestGen}$.

A.3 Figures



```
log4j:WARN No appenders could be found for logger (org.eclipse.xtext.parser.antlr.AbstractInternalAntlrParser).
log4j:WARN Please initialize the log4j system properly.
[{
  "problem": true,
  "locations": [{
    "region": {
      "endLine": 8,
      "endColumn": 47,
      "startColumn": 42,
      "startLine": 8
    },
    "artifactLocation": {"uri": "file:/C:/Users/test/Desktop/tCipher/src/tCipher/EncryptionSample.java"}
  }],
  "logMsg": "Rule ID_2_01 violated",
  "onfailIdentifier": "Invalid_TR21021_Cipher"
},{
  "problem": true,
  "locations": [{
    "region": {
      "endLine": 8,
      "endColumn": 48,
      "startColumn": 14,
      "startLine": 8
    },
    "artifactLocation": {"uri": "file:/C:/Users/test/Desktop/tCipher/src/tCipher/EncryptionSample.java"}
  }],
  "logMsg": "Rule BouncyCastleProvider_Cipher violated",
  "onfailIdentifier": "InvalidProvider_Cipher"
}]

C:\Users\test\Desktop\codyze-1.5.5\codyze-1.5.5\build\install\codyze\bin>
```

Figure A.1: Result of the analysis of Listing 1.1 by Codyze version 1.5.5 in command-line.