Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automatic Derivation of Rules for Static Security Analysis from Public Source Code Repositories

Jens Berberich

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Ana Christina Franco da Silva, Markus Haug, M.Sc.
Commenced:	January 11, 2022
Completed:	July 11, 2022

Kurzfassung

Unsichere Nutzungen von Crypto-APIs können auch in neuer Software häufig gefunden werden. Um unsichere Nutzungen zu verhindern können Entwickler Static Application Security Testing-Werkzeuge verwenden, die unsichere Nutzungen erkennen und melden können. Diese Static Application Security Testing-Werkzeuge benötigen jedoch eine große Anzahl an manuell erstellten Regeln, die die korrekte Nutzung von den APIs spezifizieren.

Diese Arbeit stellt CryptoRuleMiner als neuen Ansatz vor, der nutzbare Crypto API-Regeln erstellen kann. Hierfür nutzt CryptoRuleMiner die durch MuDetect generierten Nutzungsmuster, die es dann in CrySL-Regeln umwandelt, die von CogniCrypt zur Durchführung von statischen Analysen verwendet werden können.

Ein von CryptoRuleMiner aus 100 populären GitHub-Projekten erstelltes Regelset wird untersucht und bewertet. Zudem werden die Ergebnisse von statischen Analysen, die generierte Regeln verwenden, mit anderen Detektoren, die ebenfalls Fehlnutzungen ohne manuell erstellte Regeln erkennen, verglichen.

Abstract

Insecure crypto API usages are commonly found in modern software. To prevent insecure usages, developers can use Static Application Security Testing tools which provide them easy access to insights on whether any given API usage is secure. However, the tools typically rely on manually created rules that encode constraints on how the APIs should be used.

In this thesis, CryptoRuleMiner is presented as a novel approach that can generate usable crypto API rules by mining crypto API usages. CryptoRuleMiner uses the output of the MuDetect usage miner and transforms it into CrySL rules that can be used for conducting static analysis with CogniCrypt.

A ruleset generated by CryptoRuleMiner using 100 popular GitHub projects is evaluated. Additionally, the performance of the ruleset is compared to other usage mining based misuse detectors.

Contents

1 Introduction **15**

1.1 Motivation 15
1.2 Goals 15
1.3 Structure 16

2 Fundamentals **17**

2.1 Static Code Analysis 17
2.2 Usage Mining 18

3 Related Works **23**

4 Methodology **25**

4.1 Implementation 25
4.2 Evaluation 26

5 Implementation **29**

5.1 Mining 29
5.2 CryptoRule creation 31
5.3 Rule generation 35
5.4 Command Line Interface 41
5.5 MUBench-Runner 41

6 Results **43**

6.1 Correctness of generated rules (RQ1) 43
6.2 Reproduction of existing ruleset (RQ2) 44
6.3 Comparison to other detectors using MuBench (RQ3) 48

7 Discussion **53**

7.1 Results discussion 53
7.2 Threats to Validity 54

8 Conclusion and Outlook **55**

Bibliography **57**

List of Figures

2.1 AUG generated from the code in Listing 2.1 20

5.1 Activity diagram showing the typical rule generation process within CryptoRuleMiner 29
5.2 Example of how a CallOrder represents a graph 31
5.3 Rule exporting process example for two CryptoRules 35
5.4 Example of how an AUG gets split 36

List of Listings

2.1 Java Code using the `Cipher` API . . . 20
5.1 Construction of a `BranchingCallOrder` out of an AUG . . . 33
5.2 Example cryptography usage used for generating CrySL rules and AUGs . . . 38
5.3 `KeyGenerator` CrySL rule generated out of the code in Listing 5.2 . . . 38
5.4 Main method for the construction of the CrySL `ORDER` section . . . 40
6.1 `CertificateFactory` rule from the JCA CrySL ruleset . . . 46
6.2 `CertificateFactory` rule generated by CryptoRuleMiner . . . 46

Acronyms

API Application Programming Interface. 15

AST Abstract Syntax Tree. 20

AUG API Usage Graph. 20

JCA Java Cryptography Architecture. 15

SAST Static Application Security Testing. 15

1 Introduction

1.1 Motivation

Past studies have shown repeatedly that crypto Application Programming Interfaces (APIs) are often misused. Crypto API misuses lead to security flaws. For example, up to 96% of android apps contain at least one misuse of crypto APIs [EBFK13] [GKL+19].
In order to improve this situation, easy to use and reliable tooling that can detect and warn about misuses is essential. The existing tooling does however suffer from different problems. The rules that specify which usage is secure or insecure mostly need to be created manually by humans. They also need to be adapted by humans to consider new findings. This process is time and cost intensive. While there are some approaches that automatically generate typical usage patterns for APIs using only source code, no approach at this time generates rules that are usable by existing tooling. The necessity to install a separate tool for security analysis or to manually write rules for a tool does however prevent the easy usage of tools or drives up the cost. As shown in [OMGS18], these are however exactly the factors that prevent the adoption of Static Application Security Testing (SAST) tools.

1.2 Goals

The goal of this thesis was to create a tool that can semi-automatically generate rules for SAST tools. This should help to make the secure usage of crypto APIs easier. In order to do this, publicly accessible source code is used to generate API usage patterns. These usage patterns will then be filtered according to sensible criteria. Finally, functional rules that can be used by existing SAST tools should be created.

The tool focuses on generating rules for the Java Cryptography Architecture (JCA)[1] which contains a set of APIs for common cryptographic use cases such as encryption, hashing and digital signatures in Java.

However, it is important to note that this depends on the assumption that the frequency of an API usage pattern is correlated with the security of that pattern. As past studies have shown that many API usages are insecure [EBFK13] [GKL+19], it is essential to only use the generated rules after security experts have validated them. The chosen approach cannot validate or compare the actual security qualities of usages without human help and only depends on the above assumption.

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

1.3 Structure

The fundamentals of the work are described in Chapter 2. In Chapter 3, related works are discussed and the current state of the research is evaluated. Chapter 4 concerns the methodology of the implementation and evaluation of this work. Chapter 5 describes the implementation in detail. The results produced by the implementation are presented in Chapter 6. They are then discussed and evaluated in Chapter 7. Finally, Chapter 8 concludes the results of the work and shows some possible extension points.

2 Fundamentals

This chapter explains some fundamental technologies and principles used in this thesis. The Section 2.1 describes the approach of static code analysis and presents a specific static code analyzer. Section 2.2 describes different approaches of usage mining from source code and using the mined usages to analyze code.

2.1 Static Code Analysis

Static code analysis tries to find bugs and issues within programs. Unlike dynamic code analysis, the analysis uses either the source code or the bytecode output of the compiler. It does not need to execute the code and also does not rely on execution traces. The advantage of this approach is that the analysis can be performed much quicker. The results of a static code analyzer can therefore directly point out possible bugs while coding and prevent them from even being committed. Additionally, the analysis can usually consider all possible executions, while dynamic analysis considers only specific executions. The downside of this approach is a higher percentage of false positives. Static analysis tools typically rely on some approximation, as they cannot perfectly accurately guess how a program will be executed [Bar10].

SAST tools are static code analyzers that are specialized on finding security issues. There are several tools that provide SAST features such as SonarQube[1], CogniCrypt[2], Coverity[3] and Fortify[4].

For this work, CogniCrypt was chosen because of the rather simple CrySL syntax that makes the automatic generation of rules easier and because CogniCrypt integrates within the Eclipse IDE. Some of the other tools use more verbose and harder to generate languages for rule specifications, e.g., Java for SonarQube. CogniCrypt consists of the SAST detector $\text{CogniCrypt}_{\text{SAST}}$ and a code generator $\text{CogniCrypt}_{\text{GEN}}$ that can generate secure cryptography usages for common cryptography usage scenarios [Ecl]. The next chapter focuses solely on the SAST detector and its rule specification syntax CrySL as the code generation part of CogniCrypt is not used in this work.

[1] https://www.sonarqube.org/
[2] https://www.eclipse.org/cognicrypt/
[3] https://www.microfocus.com/portfolio/application-security
[4] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

2.1.1 CrySL

CrySL is a SAST rule specification language that was "specifically designed for (and with the help of) cryptography experts" [KSA+21, p. 2383]. It is therefore a perfect choice as an output of the proposed rule generator, as the rules should be checked by cryptography experts. CrySL-Rules can be used to perform static analysis using the $\text{CogniCrypt}_{\text{SAST}}$ tool.

CrySL follows a whitelist approach by default. If a rule exists for a class, any usage that does not follow the specification in the rule is considered a violation. With the notable exception of the optional `FORBIDDEN` section, CrySL rules don't specify which behavior is forbidden, but instead which behavior is allowed.

CrySL rules consist of several sections which each fulfill a distinct purpose. The sections are separated by keywords written in caps. There are six mandatory section.

The `SPEC` section contains the fully qualified name of the class that is covered by the rule. Each rule concerns only one class. The interaction between multiple classes can be described by predicates in the `ENSURES` and `REQUIRES` sections.

The `OBJECTS` section contains the type and name of objects and primitive types that are used in the rule. The objects can either be used as parameter or as target of a method call.

The `EVENTS` section contains all method and constructor calls of relevance to the rule. Each event gets assigned to a unique name. As each CrySL rule specifies the usage of one specific class, the method and constructor calls contained in this section are all performed on the class specified in the `SPEC` section. Additionally, it can be specified that the result of a method call is assigned to some variable introduced in the `OBJECTS` section and which events can be used alternatively.

The events are then used in the `ORDER` section to specify in which order they can be called. For this, the names of the events specified in the `EVENTS` section are used to form a string that specifies the allowed orders in which the events can appear. The syntax of the order string is similar to regular expressions. The supported operations are order, choice, repetition and grouping.

The `CONSTRAINTS` section contains all constraints that are not related to the order. It focuses on constraints placed on the value of variables used as parameters. The constraints can also be chained such that some constraints depend on other constraints being adhered to.

Finally, some guarantees can be formulated as predicates in the `ENSURES` section. These predicates can be used by other CrySL rules in the optional `REQUIRES` section such that a rule can depend on other rules and all requirements of one rule do not need to be written down in one rule file.

2.2 Usage Mining

Usage mining, sometimes also referred to as (usage) pattern mining or (usage) model mining, can be defined as the generation of typical usage patterns from source code. The usage patterns are typically represented as a graph that contains information on the interaction between API calls. This type of usage mining is classified as the inference of sequential usage patterns by Robillard et al. [RBK+13, p. 2]. In a second phase, the generated usage patterns can be used to spot faulty or non-conforming usages.

Amann describes how usage mining presents some additional challenges for crypto APIs compared to the general usage mining by looking at the JCA [Ama18, p. 95ff.]. It is especially important to note that the correct usage of crypto APIs is not equivalent to the error-free execution of the code [Ama18, p. 95ff.]. Insecure crypto API usages, where, e.g., the used encryption algorithm is weak or broken, can be executed without errors. Still, they should be recognized as a misuse. Additionally, the parameters of method calls are essential to determining the security of many crypto API usages. These parameters are used to specify security-relevant properties such as the used encryption algorithm in the JCA function `Cipher`. Which parameters are considered to be secure can also change over time [Ama18, p. 95ff.].

According to Liu et al., the state-of-the-art approaches to mine API usages can be classified into the two categories snapshot-based and history-based [LCP+21, p. 15ff.]. The snapshot-based approach uses a single source code revision to generate a usage pattern from the code. The history-based approach additionally uses information from the version control system to look at the difference of patterns before and after a change of an API usage. The history-based approach could be preferable if the hypothesis that misuses are eventually discovered and fixed is true.

There also exist multiple approaches to detecting faulty usages of APIs using usage patterns. The two most common approaches are the implementation of a custom misuse detector that uses the usage patterns as a substitute for traditional rules, which is used among other projects by MuDetect [ANN+19b] and Tikanga [WZ11], and the manual transformation of the usages into either rules for an existing code analysis tool or a custom rule-based code analysis tool, which is for example used by DiffCode [PTRV18] and Cpam [LCP+21].

A custom misuse detector typically works as follows: A graph is generated for the code that should be inspected, equivalent to the graph generation in the first phase. This graph is then compared to the graphs of typical usage patterns. A scoring function tries to determine whether the graph is most probably a misuse or not.

For the manual rule creation approach, the mined usage patterns are usually first grouped by similarity. This is done because similar usage patterns should describe similar usages that will probably also end up in the same rule. The grouping can for example be done by clustering. The generated groups of usage patterns can then be observed. Rules or a custom analyzer which implements the derived rules can be written for the groups that describe reasonable usages. These rules can also be written in a syntax of an existing SAST tool, such that the goal of easy integration can be achieved.

Listing 2.1 Java Code using the `Cipher` API

```
1 public static byte[] encrypt(String data, Key key) throws GeneralSecurityException {
2     Cipher c = Cipher.getInstance("RSA/ECB/PKCS1Padding");
3     c.init(Cipher.ENCRYPT_MODE, key);
4     byte[] enc = c.doFinal(data.getBytes());
5     return enc;
6 }
```

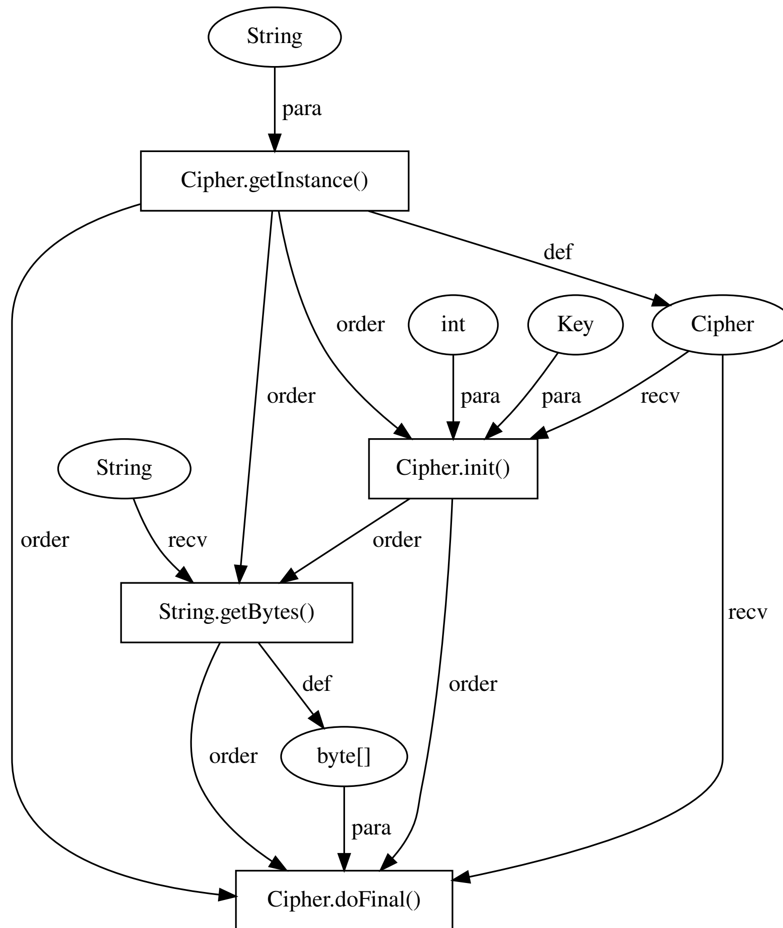


Figure 2.1: AUG generated from the code in Listing 2.1

2.2.1 API Usage Graphs

API Usage Graphs (AUGs) are a specific usage pattern representation which was defined by Amann et al. [ANN+19b] for the use in their API misuse detector MuDetect. While they are used together with a custom detector within MuDetect, they can also be exported and individually used. They are used as an intermediate step for generating crypto API rules in the proposed tool.

AUGs are directed acyclic graphs focused specifically on showing the interaction with an API. They are derived from an Abstract Syntax Tree (AST). The current AUG implementation in MuDetect can generate AUGs from Java source code. For example, the AUG shown in Figure 2.1 was

generated from the code shown in Listing 2.1. Additionally, MuDetect handles the merging of functionally identical graphs that are mined from different code snippets. MuDetect can also wrap AUGs within `APIUsagePatterns` which also contain the frequency in which the pattern occurs and each location where the pattern was mined. MuDetect is therefore a snapshot-based usage mining approach as described in the previous section 2.2.

The nodes in an AUG can be divided in `ActionNodes`, which represent an action such as a constructor call, a method call or a control flow structure, and in `DataNodes`, which represent variables, constants or literals and their contained primitives or objects. `DataNodes` are represented as square boxes within visualized AUGs. The AUG in Figure 2.1 contains four action nodes which each correspond to a method call within the mined code and a total of six data nodes that represent either an object that was used as a parameter, like `int` or `byte[]`, or an object such as `Cipher`. Each node can also contain additional information such as identifiers or the source code location which lead to the generation of a specific node. This additional information is not displayed within the graph visualizations.

The nodes are connected by directed edges which can also be divided into several subtypes. `ControlFlowEdges` usually connect `ActionNodes` and show the control flow through the program. The most frequently used type of `ControlFlowEdge` is the `OrderEdge`. `OrderEdges` connect each `ActionNode` with all `ActionNodes` that can be reached from the code location corresponding to the `ActionNode`, with the exception of loops, gotos and similar constructs, as the graph should be acyclic. In the example shown in Figure 2.1, the `Cipher.getInstance()` node is connected to the `Cipher.init()`, `String.getBytes()` and `Cipher.doFinal()` nodes as the corresponding method calls for those nodes are all executed after `Cipher.getInstance()`.

Repetition constructs such as a `while` loop can be handled by `RepetitionEdges` which connects the Node containing the repetition condition to each node that is being repeated. Other `ControlFlowEdge` types can handle additional control flow such as selections by conditions and exception handling.

`DataFlowEdges` contain a `DataNode` as source and/or target. They show the uses of data or objects such as definitions with `DefinitionEdges`, parameter usages with `ParameterEdges` and the target object of method calls with `ReceiverEdges`. For example, there are receiver edges between the `Cipher` `DataNode` and both the `Cipher.init()` and `Cipher.doFinal()` `ActionNodes`, as both of these method calls target the same `Cipher` object that is named `c` in the source code shown in Listing 2.1.

3 Related Works

This chapter describes several works that are either used in this thesis or are related to it. It presents reasearch on the reasons for mining API usages, different approaches and the evaluation of crypto API misuse detectors. Some of the approaches are also explained in detail in Chapter 2.

A detailed overview over different approaches to derive API properties is given by Robillard et al. [RBK+13]. In total, over 60 techniques are surveyed. They are then categorized according to the properties of the mining result. Some approaches are based solely on mining often used API calls and combinations of often used API calls. More sophisticated methods also mine sequential usage patterns that can not only show which API calls often occur together but also the ordering of these co-occurring API calls. A third category of approaches is the "behavioral specification" which is mainly concerned with the behavior of the API and how it is affected by the current state.

Gao et al. [GKL+19] evaluated crypto API misuses within android apps. They analyzed roughly 600'000 versions of 39'000 apps using the static analysis tool CogniCrypt$_{SAST}$. As over 90% of the analyzed apps contained at least one cryptographic API misuse, they concluded that deriving crypto API rules from android apps is not advisable. They also found out that updates in general do not fix the found API misuses. Instead, they introduce new misuses at a similar rate to fixing old misuses.

A study by Egele et al. [EBFK13] came to similar conclusions. Over 88% of the 11'748 android apps that were examined by Egele et al. violated at least one of the six security rules.

Nevertheless, many approaches to mine API usages exist. Amann et al. propose MuBench [ANN+19a], a benchmark which compares the precision and recall of four different Java API usage mining approaches. All techniques are trained using the same code samples and need to perform an analysis on the same code that contains some misuses. GROUMiner features the highest recall in an ideal scenario which is designed to evaluate the conceptional limitations of the different approaches. Therefore, the authors denote that GROUMiner is the most promising approach.

GROUMiner introduces `GROUM` graphs which represent the interaction between multiple objects by connecting action or control nodes with usage order and/or data dependency edges [NNP+09]. To detect violations, GROUMiner fist mines usages to create a set of usage patterns. GROUMiner then also creates `GROUM`s for the code it examines. If a subgraph exists that is also contained as a true subgraph in a pattern but that is not extensible in the mined `GROUM`, the `GROUM` is considered to be a violation of the pattern. A violation is then only considered to be a misuse if it is too rare, i.e., if there do not exist many other occurrences of the same inextensible pattern in the patterns.

MuDetect [ANN+19b] can be viewed as a follow-up to MuBench. It tries to implement a new API usage mining approach which is conceptually based on the `GROUM` graphs of GROUMiner. The AUGs that are used instead of `GROUM`s in MuDetect are further explained within Section 2.2.1. MuDetect reaches a higher precision and recall compared to GROUMiner when tested using MuBench. The measured precision is up to 33% while the recall is up to 42.2%. Especially the

variation MuDetectXP, which uses different projects to generate usage patterns rather than using only the project where misuses should be detected to generate the usage patterns, achieves good results. MuDetect uses a custom detector, which constructs usage graphs for the examined project and compares them to the previously generated usage graphs. A usage is considered a violation if there exist mined patterns with strict subgraphs that are equivalent to the usage graph, but there exists no pattern that has the same graph as the usage.

Paletov et al. [PTRV18] create crypto API rules by mining API usage changes. They construct direct acyclic graphs representing each object and the method calls on this object both before and after a usage change and mine the differences between those graphs. Unlike the AUGs generated by MuDetect, the graphs of Paletov et al. do not contain detailed information on the order of method calls. Instead, each graph focuses on an object with its method calls and the parameters of each method call. Each parameter is again extended to a graph if neccessary to show the initialization of the parameter. In the end, thirteen rules have been manually created by looking at clusters of all the mined usage changes.

Liu et al. [LCP+21] propose Cpam, which also uses usage changes as a base for mining patterns. It does not focus on crypto APIs and instead targets any API changes. Cpam first tries to filter out commits that likely fix API misuses by looking at properties such as commit messages and commit sizes. It then constructs ASTs both before and after the filtered commits were made and tries to group the changes made to the syntax tree into five change types out of which only the argument modification and API call replacement change types get further processed. Lastly, the pattern changes are ranked by a weighted frequency sum both within- and cross-project. Cpam was able to detect many bug patterns which are not contained in the rulesets of state-of-the-art static bug analysis tools. However, the bug patterns need to be manually derived from the mined usage change patterns.

Other than the previously mentioned MuBench, several attempts have been made to create a benchmark for SAST tools. However, as described by Schlichtig et al., no standard benchmark exists in the domain of crypto API misuse detection [SWK+22]. Instead, each benchmark focuses on different aspects and specific use cases. Because no standard benchmark exists, each new detector is evaluated using different methods and on different source code. This makes the comparison between detectors hard. Schlichtig et al. propose the creation of a new benchmark called CamBench which aims to be developed in the open and to become a standard benchmark. However, it currently is not ready for use.

One of the more complete benchmarks is CryptoAPIBench [ARY19]. It compares several conventional API misuse detectors, including CogniCrypt. In a follow-up paper, Rahaman et al. come to the conclusion that the newly developed detector CryptoGuard achieves both a higher precision and a higher recall compared to the other detectors [RXA+19]. However, as Schlichtig et al. noted, the CryptoAPIBench benchmark contains misuses found by CryptoGuard [SWK+22, p. 2]. It also covers the same 16 misuse categories that CryptoGuard should detect [RXA+19, p. 2464]. It is therefore unclear if these results include some bias.

4 Methodology

This chapter describes the methodological foundations of this thesis. As the created software is a main part of the thesis, Section 4.1 describes the methodology used for the software implementation process. Section 4.2 presents the research questions and explains how they will be evaluated.

A rapid review was used to gather information on currently used approaches in the space of automatic usage mining [CPS20]. The rapid review was bounded to the problem of automatic usage mining and conducted within the context of reusing results of the found approaches. The intention of the rapid review was to reduce the time needed to gain an overview over the existing approaches and to begin with the implementation of a new tool. The last aspect of communicating the results of the rapid review in an appealing medium however wasn't fulfilled.

The search was mainly limited to papers published after 2010 that were either found on Google Scholar or through citations. A quality appraisal was not performed. The results were synthesized narratively and were presented both in a presentation and in the Chapter 2 and Chapter 3.

4.1 Implementation

Throughout the implementation process, rapid prototyping was used as development method. As the goal of this thesis is the exploration of the viability of generating SAST rules out of Java source code, development speed was the primary concern. Rapid prototyping was chosen as the development approach as it enables a high initial development speed. Some time-consuming software engineering activities such as the specification of detailed requirements and the creation of system model were not performed.

4.1.1 Technology

CryptoRuleMiner uses MuDetect [ANN+19b] to perform the extraction of AUGs from source code. MuDetect is available under the MPL license. The functionality of the graphs generated by MuDetect is described in more detail in Section 2.2.1. Cpam [LCP+21] was also considered as a potential alternative. It uses a more advanced change-based usage mining approach that could lead to better results compared to the more primitive snapshot-based approach used by MuDetect. Cpam was however not used as it didn't work when testing due to database connection problems and as no licensing information was available.

As MuDetect is written in Java and the author is familiar with Java, Java was chosen as the programming language for CryptoRuleMiner. Additionally, Java has the advantage that the code is portable and can run on any architecture and operating system where Java is supported. The chosen build tool is Maven. To ensure the easy executability of the code, a command line interface providing access to the most needed configuration options was implemented.

4.2 Evaluation

The evaluation should in general answer the question whether the chosen approach could be used to semi-automatically generate a ruleset for a given SAST tool. The created ruleset is therefore compared with existing rulesets that are known to be good. Additionally, benchmarks can measure the precision and recall that can be achieved using the chosen approach. Both precision and recall are especially important to judge the viability of the approach. The recall needs to be high enough such that most vulnerabilities will be detected, as this is the primary goal. The precision is however also very important, as a imprecise detector will produce many false warnings and will therefore not be very usable. Oyetoyan et al. studied the factors hindering the adoption of SAST tools and reported that “developers were unanimous about their concerns with false positives” [OMGS18]. The following research questions should be answered by the evaluation:

RQ1: Are the rules encoded in valid CrySL syntax? Can they be used to perform static analysis without manual modifications?

RQ2: What parts of the existing CrySL ruleset can be recreated when mining a sufficiently large dataset of code that uses the Java Cryptography Architecture? How do the generated rules compare to the existing rules?

RQ3: How do recall and precision of the CogniCrypt static analyzer using the created rules compare to other solutions, especially the existing CrySL ruleset and MuDetect?

In order to answer the research questions 1 and 2, a large dataset containing code that uses the Java Cryptography Architecture is needed. In order to get this large dataset, several open source projects using the Java Cryptography Architecture APIs were selected. The selected projects have a GitHub repository that was created between the 1st of January 2018 and the 1st of June 2022 and that has at least 100 stars and 100 commits. These criteria should prevent the CryptoRuleMiner from mining old or amateur code. As the number of stars on GitHub might not strongly correlate with the security awareness of developers, it is still expected that insecure usages will be encoded in the generated rules.

The research question 1 will be evaluated by using the generated CrySL ruleset as an input for the CogniCrypt$_{SAST}$ static analyzer. Performing any static analysis using CogniCrypt$_{SAST}$ only works if the rules conform to the CrySL specifications. Additionally, the CrySL plugin for the Eclipse IDE which provides syntax highlighting and error detection for CrySL rules will be used to find errors in the generated ruleset.

In order to reduce the scope of the evaluation, only usages of the Java Cryptography Architecture API will be mined and evaluated. The JCA was picked as it is widely used and included in the datasets of benchmarks such as MuBench [ANN+16]. This enables the comparison between the proposed CryptoRuleMiner and other API misuse detectors.

4.2.1 Rule Evaluation

To evaluate the research question 2, a classification of the rules into 5 categories is proposed. For the evaluation, the generated rules are compared with the existing CrySL ruleset. This assumes that the CrySL rules are complete and do not encode unwanted behaviours. The proposed classification should be able to measure whether the rules are useful in the real world or if they could even be harmful. The rules do not necessarily fit into only one category, as a rule could, e.g., encode both new usages and miss some usages that are contained in the existing CrySL ruleset and would therefore be classified as both new usage and incomplete.

1. **Superfluous rules**
   Compared to the CrySL rule, redundant or non-security related aspects are added to the rule. As the generated rules only make use of the subset of CrySL that defines specific usages as allowed, adding aspects to the rule that are not necessary to increase the security of usages could mean that some usages do not fulfill the added, unnecessary constraints and will therefore be falsely marked as insecure. This could increase the false positive rate.

2. **New usage rules**
   An API usage that isn't allowed in the CrySL ruleset is encoded. This new usage could either be overlooked by the CrySL ruleset or could be intentionally not included in it and therefore malicious. These rules are therefore further classified into two subcategories: Likely malicious new usage rules and likely correct new usage rules.

3. **Incomplete rules**
   The generated rule matches significant, cryptography-related parts of an existing CrySL rule. However, some constraints, predicates or method calls are missing or the ordering is not equivalent.

4. **Correct rules**
   The generated rule matches an existing CrySL rule. Rules are considered to be matching if both rules express the same behavior and allow or forbid the same usages.

4.2.2 Rule Usage Evaluation

As the generated rules can not be viewed in isolation, this evaluation uses the generated rules as input for CogniCrypt$_{SAST}$ in order to perform a static analysis and to evaluate the results of the applied ruleset. In order to answer research question 3, the tool can then be evaluated using existing benchmarks. A subset of MuBench [ANN+16] that includes usages of the JCA API is used as a benchmark. The datasets `JCA-Usages`, `JCA-Changes-GH` and `JCA-Changes-SF` are used. Additionally, the misuse dataset `MSR19-FindBugs-Exp2-and-Exp3` that was added to MuBench by Wickert et al. was also used [WRE+19].

In order to use CogniCrypt and CryptoRuleMiner within the benchmark, a MuBench Runner that executes CryptoRuleMiner and CogniCrypt$_{SAST}$ needed to be created. The JCA ruleset that is provided by CogniCrypt is also evaluated to provide another comparison.

Using an established benchmark as part of the evaluation has several advantages. MuBench consists of three experiments that provide several metrics that can be used to compare different detectors:

1. The recall upper bound (RUB) experiment provides correct usages for the usage mining part. The detector then has to find the labeled, known misuses. This experiment provides a recall upper bound value for the evaluated detector.
2. The precision (P) experiment lets the detector mine usages using each complete project. The detector then also runs the detection on the same project. A precision value is calculated using the 10 top ranked misuses reported by the detector.
3. The recall (R) experiment also provides the detector whole projects for usage mining. However, it is only evaluated whether any misuse matches the known, labeled misuses, equivalent to the RUB experiment. This experiment provides a recall value that should better reflect actual usage mining compared to the RUB experiment.

It also already has integrations that can be used to benchmark several other API misuse detectors, including MuDetect, Jadet, Tikanga, DMMC and FindBugs. The three values can be easily compared to see if the proposed method produces good results.

When using the rules generated by CryptoRuleMiner as input for CogniCrypt$_{SAST}$, it is expected that the performance will be equal or worse than MuDetect, as a only slightly modified version of MuDetect was used for the actual usage mining. However, the approach focuses mainly on the CrySL rule generation part. If the deviation between CryptoRuleMiner and MuDetect is minimal, the result is considered to be good.

5 Implementation

This chapter describes the implementation of the CryptoRuleMiner proof of concept rule generation program. Figure 5.1 shows an overview over the typical process of generating rules using the final system. First, MuDetect is used in the mining phase to generate a usage model out of some given Java source code. This usage model is then used to generate `CryptoRules`, which are a wrapper around the API Usage Graphs contained in the model. `CryptoRules` also contain a call order structure that greatly simplifies the generation of crypto API rules. These `CryptoRules` are then used to generate CrySL rules which can be exported as text files. The following sections each explain one of these three steps in more detail.

5.1 Mining

Within the CryptoRuleMiner code, MuDetect is first called to extract a model out of the given projects. This model consists of `CryptoAPIPatterns`, which provide both a usage graph and the locations and methods which lead to the generation of the pattern.

To perform the mining, MuDetect needs to be given two configurations. They are both changed in comparison to the default configuration that is used by the standalone MuDetect. Additionally, the code of MuDetect was modified in order to change some behaviors of MuDetect.

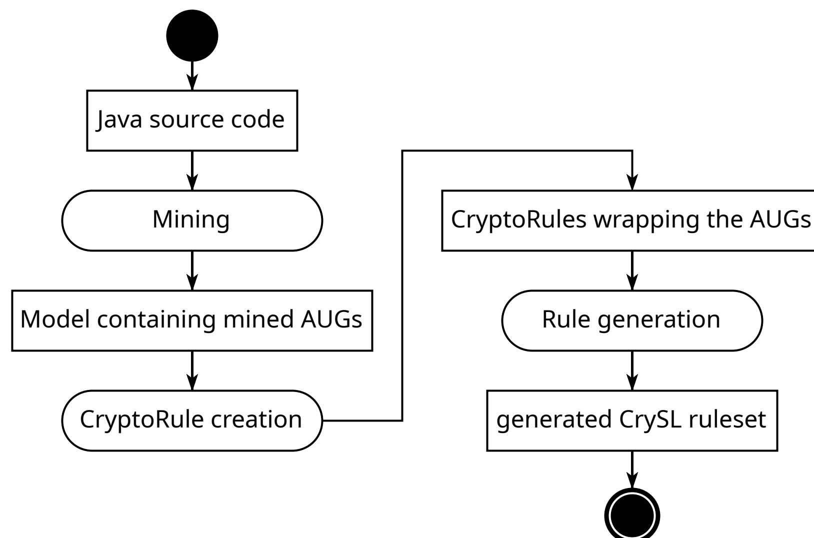


Figure 5.1: Activity diagram showing the typical rule generation process within CryptoRuleMiner

MuDetect often focuses on generating graphs that are less accurate in order to increase the chance that any two graphs describing a similar usage can be recognized as similar graphs. CryptoRuleMiner however needs to produce CrySL rules, which require an accurate representation of the source code, as some constructs used by MuDetect such as the combination of parameters of the same type into one parameter can not be expressed with CrySL. The following sections will describe the most important configuration or code changes and provide reasons for them.

One of the most important configuration changes is that repeated calls are not disallowed. This means that two calls to the same method which occur directly after one another are both represented as a node in the graph instead of being collapsed into one node. If they were disallowed, the resulting CrySL rule could classify some usages that use repeated method calls as misuse even if a matching usage was found in the rule mining dataset.

Additionally, `RepetitionEdge` and `SelectionEdge` are added to the `extensionEdgeTypes`. This ensures that the graph is not split along these edges. If this isn't set, MuDetect can split the graph into smaller parts than wanted and thereby remove information on repetitions or selections.

Conditionals such as `if` or `else` could also lead to the splitting of a graph. To prevent this, the removal of independent control edges needs to be disallowed.

There is also a first filtering step done within MuDetect. Graphs are only generated for classes which contain an import statement matching the Regex `import ((javax\.crypto)|(javax\.security))\..*\n"`. This is done as the CryptoRuleMiner should only generate rules for JCA usages. Most of the non-security-related code should be filtered out, such that the computationally expensive parts of MuDetect and CryptoRuleMiner are performed only for relevant code.

In addition to the configuration changes, the source code of MuDetect was also modified. The original MuDetect code only considered method calls that didn't start with `get` to be core actions. Like this, simple getters that are most often not relevant for API usage patterns will be ignored. The concept of core actions is used, among other things, to decide whether a usage graph gets created for a code fragment. The JCA however has several classes that use the singleton pattern and thus have methods such as `Cipher.getInstance()` that start with `get` and are used for essential functionality instead of being simple getters. If they are not considered as core actions, some crucial interactions with crypto APIs would get lost. To prevent this, all method calls are considered to be core nodes, regardless of whether the method signature starts with `get` or not.

Another change made to MuDetect was the removal of literal collapsing. If literal collapsing is enabled, multiple `DataNodes` that are used as parameters for a single `MethodCallNode` are combined into a single `AggregateDataNode` which contains the information of all the combined `DataNodes`. This is undesirable because `AggregateDataNodes` are also created by other means, such as for merging equivalent nodes, regardless of whether literal collapsing is enabled. Literal collapsing therefore removes the ability of counting the number of method arguments and their distinct values. Additionally, the separation of parameters required by CrySL syntax would be prevented by the literal collapsing.

To separate the mining part from the rule generation part, additional methods that enable the import and export of models were implemented.

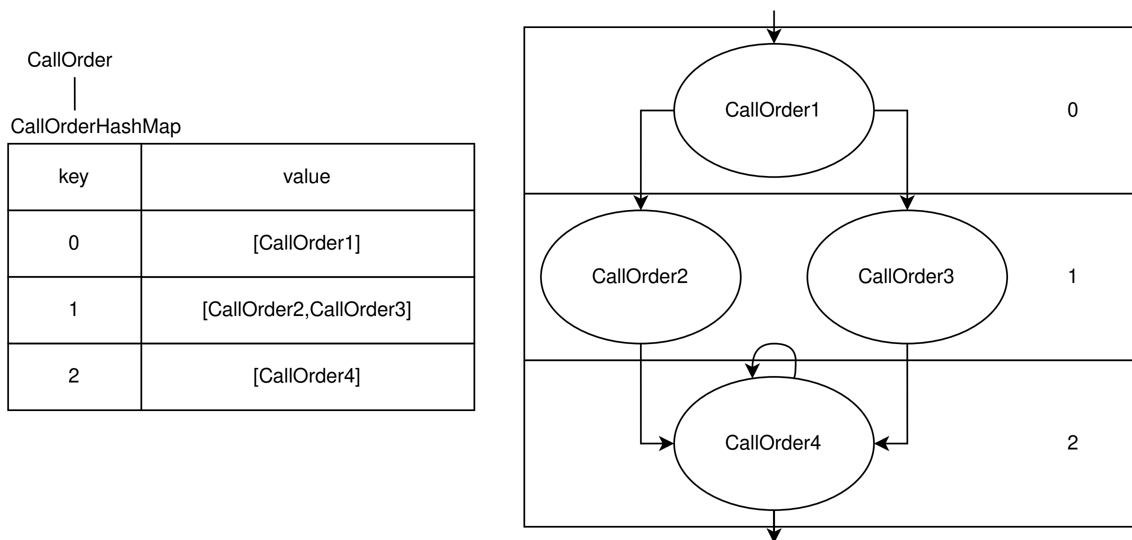


Figure 5.2: Example of how a CallOrder represents a graph

5.2 CryptoRule creation

The model created by MuDetect consists of `APIUsagePatterns`. These `APIUsagePatterns` get wrapped inside `CryptoRules` by CryptoRuleMiner. `CryptoRules` provide helper methods that make it easier to query information that would otherwise require multiple method calls on AUGs. An example for a helper method is `calculatePreviousCallsToObject`. It is given a node representing an object and then searches for all previous calls to objects which have at least one method calls which use the given node as a parameter. While this method is quite complex, it is used several times for calculating dependencies.

Additionally, a simplified call order is created when a `CryptoRule` is initialized. This `CallOrder` is created because AUGs such as the one seen in Figure 2.1 contain many nodes and edges that aren't relevant for simply getting the order of the contained method calls. Getting the order of the contained method calls is however a important operation for constructing crypto API rules. The AUG also doesn't provide easy access to some frequently used nodes such as the starting nodes. Traversing an AUG is made harder by the fact that for every method call node, a `OrderEdge` to every other method call node exists. The AUG also contains edges such as `DataFlowEdges` which are not needed for the call ordering.

5.2.1 CallOrder fundamentals

`CallOrders` use the composite design pattern. This means that the whole `CallOrder` itself and all the contained individual `CallOrders` are of the same class. The root call order is stored in a `BranchingCallOrder` object. It stores the call order in a map that contains a number signifying the order as key and a List containing `CallOrders` as value. If the `CallOrder` list for a number in the map contains multiple `CallOrders`, they are viewed as alternatives. Like this, branching can be

represented in the call order. A simple example of this is shown in Figure 5.2. After the contents of CallOrder1, either the contents of CallOrder2 or the contents of CallOrder3 can be executed. After CallOrder2 or CallOrder3 have been executed, CallOrder4 finally gets executed.

Compared to the source AUG, some information such as which function or variables determines which branch is picked is lost. This lost information is however mostly not important for crypto rules. It can also still be retrieved from the AUG, which is stored in the `CryptoRule`.

The `CallOrder` recursion is resolved by leaf objects of the type `SimpleCallOrder`. These `CallOrder` objects contain only a reference to one node or no node. SimpleCallOrders that contain no node can be used in a non-empty list to signify that the contents of the other call orders contained in the list are optional. If, for example, CallOrder4 in Figure 5.2 is an empty SimpleCallOrder while CallOrder3 contains some nodes, then the nodes contained in CallOrder3 can all be regarded as optional. A control flow going through CallOrder1 and then CallOrder4, skipping the contents of the list in key 1, would therefore be permitted.

Each `CallOrder` object also contains a boolean specifying whether the contents of the call order can be repeated arbitrarily often or whether they are each called only once. In the example in Figure 5.2, only CallOrder4 in the graph can get executed arbitrarily often - therefore the CallOrder4 object has the boolean set to true, while it is set to false for all other CallOrders in this example. As the CallOrder structure is recursive, the repetition of an arbitrary sequence of method calls can be represented by putting the method calls into one CallOrder where the repetition boolean is set to true.

5.2.2 CallOrder generation

To generate a `CallOrder`, the starting nodes are first determined. Starting nodes contain method calls that are called first out of all method calls contained in an AUG. As an AUG can contain multiple independent sequences, multiple starting nodes can exist. For AUGs with multiple connected method call nodes, all nodes that contain outgoing but no incoming `OrderEdges` are starting nodes. For AUGs which contain only one method call node or a method call node that isn't connected to other method call nodes, each of these non-connected method call nodes is also considered to be a starting node.

Each of the previously determined starting nodes is then used to iteratively generate a call order. The code that is responsible for the call order generation can be seen in Listing 5.1. If no starting nodes exist, a call order containing only one empty `SimpleCallOrder` is returned in line 5. Otherwise, the normal call order generation loop begins.

Special case: multiple starting node groups

This case applies if multiple starting nodes exist that are not all controlled by the same selection node. It is handled in the lines following line 11. The `groupControlledNodes` method returns lists each containing all the nodes contained in `nextNodes` that are connected to the same `SelectionNode`. Each node that is connected to no `SelectionNode` is returned in a separate list. A new call order is

Listing 5.1 Construction of a `BranchingCallOrder` out of an AUG

```java
private HashMap<Integer, List<? extends CallOrder>> constructCallOrder(APIUsageGraph usageGraph,
↪   Collection<Node> startingNodes) throws NodeNotFoundException {
    HashMap<Integer, List<? extends CallOrder>> callOrder = new HashMap<>();
    if (startingNodes.isEmpty()) {
        callOrder.put(0, Collections.singletonList(new SimpleCallOrder()));
        return callOrder;
    }
    List<Node> nextNodes = new LinkedList<>(startingNodes);
    int iterationCount = 0;
    do {
        LinkedList<CallOrder> nestedCallOrders = new LinkedList<>();
        if (nextNodes.size() > 1 && (!containsSelectedNode(nextNodes, usageGraph) ||
↪   calculateControllingNode(nextNodes, usageGraph).isEmpty())) {
            for (List<Node> nextNodeGroup : groupControlledNodes(nextNodes, usageGraph)) {
                nestedCallOrders.add(new BranchingCallOrder(nextNodeGroup, usageGraph));
            }
            nextNodes.clear();
        } else if (containsSelectedNode(nextNodes, usageGraph)) {
            Node controllingNode = calculateControllingNode(nextNodes, usageGraph).get(0);
            for (Node nextNode : nextNodes) {
                nestedCallOrders.add(new BranchingCallOrder(nextNode, controllingNode, usageGraph));
            }
            if (!isNodeControlledByRepetition(nextNodes.get(0), controllingNode, usageGraph) &&
↪   nextNodes.size() == 1) {
                nestedCallOrders.add(new SimpleCallOrder());
                nextNodes = calculateDirectSuccessorNodes(nestedCallOrders.get(0).getLastNodes().get(0),
↪   usageGraph);
            } else {
                List<Node> lastNestedNodes = new LinkedList<>();
                for (CallOrder nestedCallOrder : nestedCallOrders)
                    for (Node lastNode : nestedCallOrder.getLastNodes())
                        if (lastNode != null) lastNestedNodes.add(lastNode);
                nextNodes = calculateCommonSuccessorNodes(lastNestedNodes, usageGraph);
            }
        } else {
            nestedCallOrders.add(new SimpleCallOrder(nextNodes.get(0)));
            nextNodes = calculateDirectSuccessorNodes(nextNodes.get(0), usageGraph);
        }
        callOrder.put(iterationCount, nestedCallOrders);
        iterationCount++;
        nextNodes = nextNodes.stream().filter(node -> !(node instanceof
↪   CatchNode)).collect(Collectors.toList());
    } while (!nextNodes.isEmpty());
    if (callOrder.size() == 1 && callOrder.get(0).size() == 1 && callOrder.get(0).get(0) instanceof
↪   BranchingCallOrder && callOrder.get(0).get(0).isRepeating()) {
        repeating = true;
        return ((BranchingCallOrder) callOrder.get(0).get(0)).callOrderMap;
    }
    return callOrder;
}
```

generated for each group and then added to the nestedCallOrders list. As the call order generation is then handled by these nested call orders, the nextNodes List is emptied such that no next iteration will be performed on the outer call order.

Default case

The simplest case of call order generation can be found in line 31. This case occurs if there is a simple sequence of statements without any branching. A SimpleCallOrder containing the single node that is contained in nextNodes is created and added to the nestedCallOrders list. The nodes that are handled in the next iteration are calculated by calculateDirectSuccessorNodes. The calculateDirectSuccessorNodes returns all nodes that are connected by an incoming order edge to the current node and have exactly one more incoming order edge than the current node.

Special case: selection or repetition

The next special case applies if the nodes contained in nextNodes are subject to a selection or repetition, e.g., are within a for loop or if statement body. It can be found in line 16. The method containsSelectedNode checks whether there are any incoming or selection edges that have any node contained in the nextNodes list as target. If this is the case, BranchingCallOrders starting from each node contained in nextNodes are generated and added to the nestedCallOrders list. Each nested BranchingCallOrder is also given the controlling node, i.e., the node that is the origin of the selection or repetition edge that has the starting node as target, calculated by calculateControllingNode. These nested BranchingCallOrders each only contain nodes that are connected by repetition or selection edges to the given controlling node.

If nextNodes contains only a single node and that node is not connected to the controlling node by a repetition edge, i.e. it is connected by a selection edge, an empty SimpleCallOrder gets added to the nestedCallOrders list. This situation occurs if some code is wrapped by an if-Statement. The code can therefore be regarded as optional, which is shown by the empty SimpleCallOrder. In this case, the next nodes can also be calculated by the calculateDirectSuccessorNodes method described in the previous section.

Otherwise, the last nodes from each of the nested BranchingCallOrder are used as input for calculateCommonSuccessorNodes in order to calculate the next nodes. The method calculateCommonSuccessorNodes first collects all nodes that are reachable by selection edges from all nodes given as input. It then returns only those nodes that have the minimal number of incoming order edges, as those are the direct successors.

The nestedCallOrders list that was filled by one of the above cases is then added to the callOrder HashMap using the iteration count i as a key. This process is repeated as long as some successor nodes exist.

Line 39 handles setting the repeating boolean. If the call order contains exactly one BranchingCallOrder that also contains one call order which is marked as repeating, the current call order will be marked as repeating and will contain the call order map from the nested BranchingCallOrder. This simplifies the final call order by removing excessive nesting.

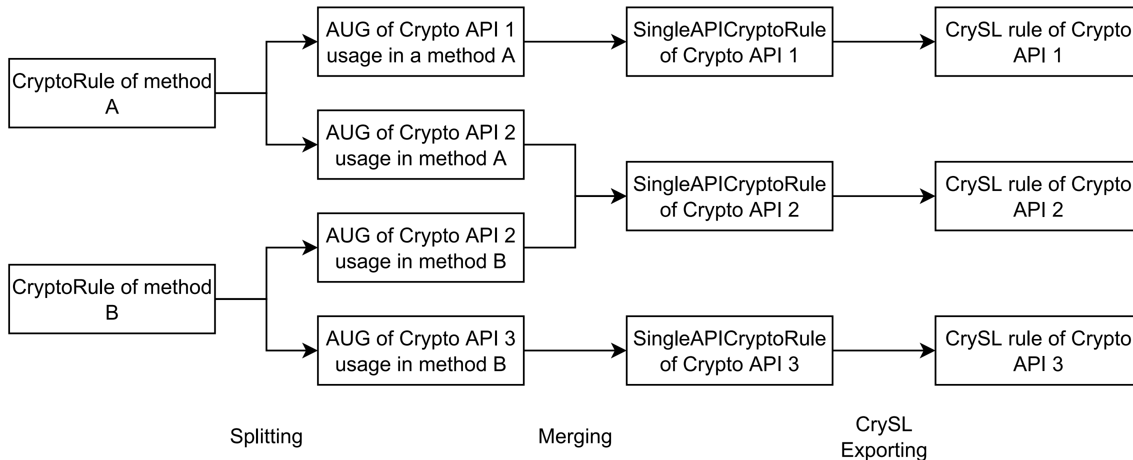


Figure 5.3: Rule exporting process example for two CryptoRules

5.3 Rule generation

The rule generation is the core part of the CryptoRuleMiner. An overview of the process can be seen in Figure 5.3. In the first step, each `CryptoRule` is split into AUGs each containing only the nodes and edges that are relevant for a single API. As interactions for an API can be contained in multiple CryptoRules, there can be multiple AUGs representing the same API. The AUGs for the same API are therefore merged and wrapped in a `SingleAPICryptoRule`. As the `SingleAPICryptoRules` contain only method calls on one class, they can be easily transformed into CrySL-Rules.

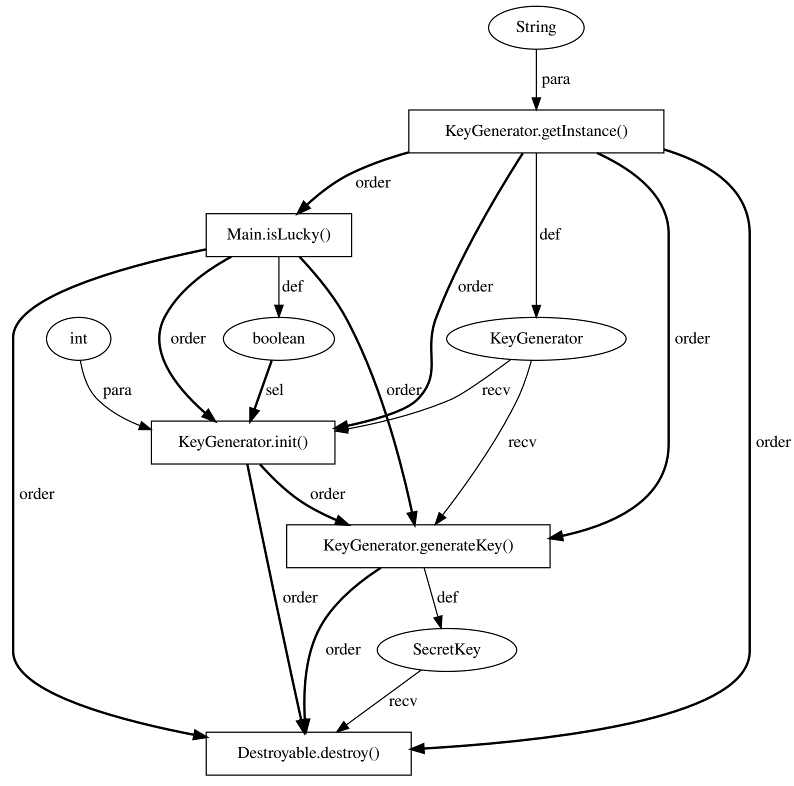
The creation of `CryptoRules` and `SingleAPICryptoRules` are both preprocessing steps and not included in the CrySL rule generation. This has the advantage that the existing code could be extended with additional exporters that output crypto API rules in different syntax. These exporters could use the `CryptoRules` and would therefore not have to process the AUGs again.

5.3.1 CryptoRule splitting and merging

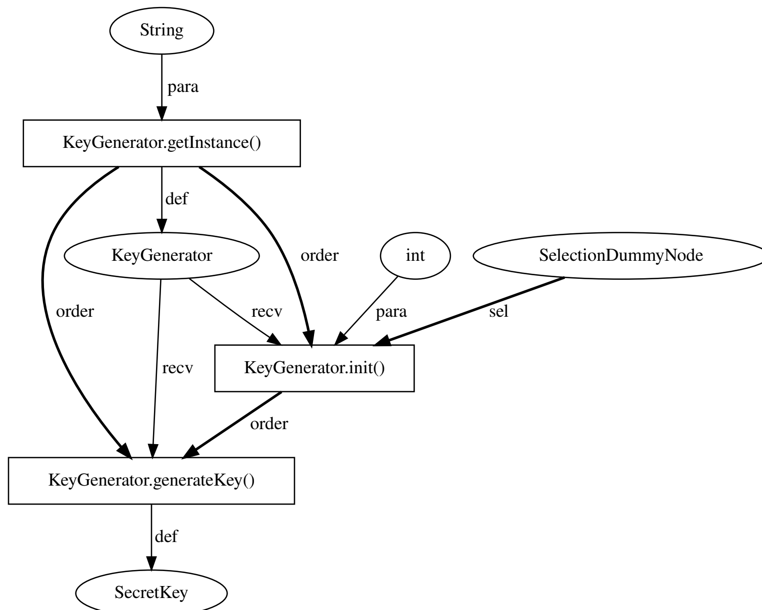
As each CrySL rule concerns only one class while the AUGs can contain the interaction between multiple classes, the `CryptoRule` objects containing the AUGs are first split such that they only contain method calls on a single class. An example for the splitting of a crypto rule can be seen in Figure 5.4. In the next step, all split rules that concern the same class are merged again. Only then the actual rule generation can begin.

The splitting process takes the usage graph contained in a `CryptoRule` and splits it into one subgraph per used crypto API. For the example in Figure 5.4a, a split graph is generated for the `KeyGenerator` and the `SecretKey` API. Methods called on classes that are not contained in the JCA such as `Main.isLucky()` are not contained in the split graphs.

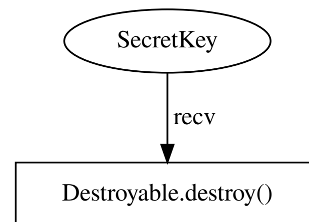
To split a graph, all nodes in the original graph are first copied and assigned to a new ID. A mapping between the original nodes and the copied nodes in the cloned graph and vice versa is produced. All edges are also transferred such that they now have the copied nodes with their new IDs as target and source node.



(a) The full graph before splitting



(b) The split subgraph for the `KeyGenerator` API



(c) The split subgraph for the `SecretKey` API

Figure 5.4: Example of how an AUG gets split

Then, all the nodes that are not method calls on the intended crypto API or used by method calls on the intended crypto API are removed from the cloned graph. However, there are some additional special cases.

All nodes that are necessary to represent a selection or repetition such as a `while` loop or `if` statement are not completely removed, but only replaced by `SelectionDummyNodes`. These `SelectionDummyNodes` inherit only the outgoing and incoming `SelectionEdges` or `RepetitionEdges` from the selection or repetition nodes and the nodes that define those selection or repetition nodes that they replace. Like this, the information about the selection or repetition is preserved while the `SelectionDummyNodes` themselves are not appearing in the call order, as they have no incoming or outgoing `OrderEdges`. An example for this can be seen in Figures 5.4a and 5.4b, where a `SelectionDummyNode` has replaced the `boolean` Node and is connected with a `SelectionEdge` to the `KeyGenerator.init()` node.

If `Nodes` that are connected by a `DefinitionEdge` to at least one object of the intended crypto API exist, they are handled seperately. If these nodes are method calls that output an object of the intended crypto API, the object node that is defined by them is added to the list of precondition nodes and the node itself is removed from the graph. If they do not define any object of the intended crypto API, they are themselves added to the precondition nodes. Additionally, any incoming and outgoing `OrderEdges` are removed such that these precondition nodes do not appear in the call order. The call order is also modifed such that precondition nodes are never picked as starting nodes. The constructed list of precondition nodes can then be used later on to construct dependencies between different classes. An example for this case can be seen in Figures 5.4a and 5.4c where the `SecretKey` node would be marked as a precondition node as there is a `DefinitionEdge` connecting the `KeyGenerator.generateKey()` node to it.

`DataNodes` that are connected by a `DataFlowEdge` to at least one method call to the intended crypto API, i.e., are used as parameter, are added to the list of precondition nodes before being removed.

Additionally, `DataNodes` that have an incoming `DefinitionEdge` from a `MethodCallNode` to the intended crypto API, i.e., are defined by a method call to the intended crypto API, are not removed. They are used for assignments in CrySL rules.

After the `CryptoRules` have been split, all AUGs that concern the same API are merged again. The merging is done by adding all nodes and edges into one graph and merging the dependency lists. A complex logic to merge similar or identical subgraphs could however lead to less redundant call orders and therefore more compact CrySL rules. However, the simple merge still leads to more compact and easier to understand rules compared to using separate graphs and rules to generate separate CrySL rules. When constructing a CrySL rule out of a graph, the exporter will merge equivalent objects. The primitive merge will therefore only make the `ORDER` section unnecessarily large. The primitive merge will in the end lead to `CallOrders` being generated that contain `CallOrders` built from the merged graphs in the list contained with key 0 in the call order map of the outer call order. New `CryptoRules` are then generated for the merged AUGs.

The split and re-merged `CryptoRules` are then evaluated in order to store the dependencies more efficiently. To do this, the dependency list of each graph that was generated in the splitting step is used. For each node contained in this dependency list, a `CryptoRuleDependeny` object is created. It contains the node and the `CryptoRule` for both the dependendent node and the class containing the original node.

Listing 5.2 Example cryptography usage used for generating CrySL rules and AUGs

```
1 public static void main(String[] args) throws GeneralSecurityException, DestroyFailedException {
2     KeyGenerator gen = KeyGenerator.getInstance("AES");
3     if (isLucky()) {
4         gen.init(128);
5     }
6     SecretKey secret = gen.generateKey();
7     secret.destroy();
8 }
```

Listing 5.3 `KeyGenerator` CrySL rule generated out of the code in Listing 5.2

```
1 SPEC javax.crypto.KeyGenerator
2
3 OBJECTS
4 javax.crypto.SecretKey secret;
5 int object18;
6 java.lang.String object20;
7
8 EVENTS
9 GetInstance: getInstance(object20);
10 Init: init(object18);
11 GenerateKey: secret = generateKey();
12
13 ORDER
14 GetInstance, Init?, GenerateKey
15
16 CONSTRAINTS
17 object20 in {"AES"}  => object18 in {128};
18 object20 in {"AES"};
19
20 ENSURES
21 predicate0[secret];
```

5.3.2 CrySL rule exporting

The actual CrySL exporting is handled by the `CryptoRuleToCrySL` class, which builds a rule conforming to the CrySL syntax for a given `CryptoRule`. CrySL rule exporting poses the challenge that various parts of a CrySL rule such as the rule seen in Listing 5.3 that was generated out of the code seen in Listing 5.2 depend on each other. The `OBJECTS` part for example defines the used objects. These objects are however also important for the `EVENTS` part which describes the interaction between those objects, such as the parameters needed for a method call. The events are then used to construct valid call orders for the `ORDER` part. Lastly, the `CONSTRAINTS` part of the rules defines restrictions on parameters of method calls. The `ENSURES` and `REQUIRES` parts containing predicates that specify interactions between different rules are also exported. It is also dependent on the previously defined events and objects. The first parts that it builds are the `SPEC` and `OBJECTS` sections.

OBJECTS section

Each object entry in CrySL consists of the type of the object and an identifier. In the example contained in Listing 5.3, there is for example an object with the type `java.lang.String` and the identifier `object20`. However, the type that is saved in `CryptoRules` generated from AUGs is not the needed full class name. For example, the class name `SecretKey` is saved instead of the complete `javax.crypto.SecretKey`. Therefore, the `CryptoRuleToCrySL` exporter tries to expand the class name by looking at the import statements in the files related to the given `CryptoRule` and matching the shortened type with an import statement. If no such matching import statement exists, e.g., because wildcard import statements such as `import javax.crypto.*` have been used, a hardcoded expansion list will be used.

For each object in the `OBJECTS` section, all other objects are checked for approximate equivalence in order to prevent equivalent objects from being exported. Two objects are considered equivalent if they both have the same type and are used by `MethodCallNodes` that are not identical but roughly equivalent, i.e. `MethodCallNodes` that have the same number of parameters and the same method signature. All equivalent objects will be saved into a map such that they can be retrieved later on. For each group of equivalent objects, only one entry in the `OBJECTS` section gets created.

For the identifier, the user is optionally queried for input. Some information such as the location of the object and how it is used is given to the user. They can then decide whether they want to name the object themselves. Otherwise, the object will be named according to the identifier in the code. If no naming in the code exists, it is named `objectID` where `ID` is the id of the node otherwise.

EVENTS and ORDER sections

The `EVENTS` and `OBJECTS` sections of the CrySL rule are created simultaneously. A helper method is given a `CallOrder` which it traverses. The nodes found in the call order are added to the `EVENTS` and `OBJECTS` strings. The main logic can be seen in Listing 5.4. If multiple non-optional `subCallOrders` exist within the current `callOrder`, they are enclosed with brackets in lines 4 and 15. The nodes within each `subCallOrder` are added to the `EVENTS` and `ORDER` strings using `addNodesFromCallOrder` in line 7 and separated with vertical bars in 9. For optional call orders, a question mark is added if necessary in line 13. An example for this can be seen in Listing 5.3, where the `Init` event is optional. The events in a sequential call order are separated by commas that are added in line 17.

Another core part of the `EVENTS` and `OBJECTS` section generation is the method `addNodesFromCallOrder`. If it is given a `BranchingCallOrder`, it uses `addNodes` to add all nodes from this call order to the `EVENTS` and `OBJECTS` strings and encloses the generated events string with brackets if multiple events are contained. If it is given a `SimpleCallOrder` which contains only one node, it adds the contained node to both the `EVENTS` and `OBJECTS` strings. Events that represent method calls with outgoing definition edges, i.e. that assign some value to a variable, are also properly exported. An example for this is the `GenerateKey` event in the example CrySL rule shown in Listing 5.3. It represents the method call node `KeyGenerator.generateKey()` which is connected by a definition edge to a `SecretKey` data node in Figure 5.4b. `addNodesFromCallOrder` also handles repetition by appending a `*` if the given call order is repeating.

Listing 5.4 Main method for the construction of the CrySL ORDER section

```
1  void addNodes(BranchingCallOrder callOrder) {
2          for (List<? extends CallOrder> subCallOrders : callOrder.getCallOrderList()) {
3                  if (subCallOrders.size() != 1 && !subCallOrdersOptional(subCallOrders)) {
4                          orderStringBuilder.append("(");
5                  }
6                  for (int i = 0; i < subCallOrders.size(); i++) {
7                          addNodesFromCallOrder(subCallOrders.get(i));
8                          if (i != subCallOrders.size() - 1 && !(subCallOrders.get(i + 1) instanceof
   ↪   SimpleCallOrder && ((SimpleCallOrder) subCallOrders.get(i + 1)).isEmpty())) {
9                                  orderStringBuilder.append(" | ");
10                         }
11                 }
12                 if (subCallOrdersOptional(subCallOrders) && !subCallOrders.get(0).isRepeating() &&
   ↪   !(subCallOrders.get(0) instanceof BranchingCallOrder && subCallOrdersOptional(((BranchingCallOrder)
   ↪   subCallOrders.get(0)).getCallOrderList().get(0)) && subCallOrders.size() == 2)) {
13                         orderStringBuilder.append("?");
14                 } else if (subCallOrders.size() != 1 && !subCallOrdersOptional(subCallOrders)) {
15                         orderStringBuilder.append(")");
16                 }
17                 orderStringBuilder.append(", ");
18         }
19         if (!callOrder.getCallOrderList().isEmpty()) {
20                 orderStringBuilder.delete(orderStringBuilder.length() - 2, orderStringBuilder.length());
21         }
22 }
```

CONSTRAINTS section

The `CONSTRAINTS` section of the CrySL rule is constructed by adding constraints for all `DataNodes` that have the type string or integer. In the example given in Listing 5.3, there are constraints for the key size represented by `object18` and the algorithm represented by `object20`.

The `CONSTRAINTS` section can also contain dependent constraints. For each object that has a method call which uses the `DataNode` as a parameter, all previous method or constructor calls that are preformed on the object or that define the object and that have at least one parameter are calculated. If any previous method call has any incoming parameters, it is assumed that they could have influenced the state of the object and that a dependent constraint should be constructed. In the provided example for which the AUG can be seen in Figure 5.4b, `KeyGenerator.getInstance()` is calculated as a previous method call when using the `int` parameter of `KeyGenerator.init()` as input. As the `KeyGenerator.getInstance()` method call has a `String` parameter, a dependent constraint is constructed. The constructed dependent constraint in this example is `object20 in {“AES”} => object18 in {128};`. The dependent constraint signifies that a constraint like `object18 in {128}` only needs to be fulfilled if the dependencies, in this example `object20 in {“AES”}`, were also fulfilled. If no previous method calls with parameters were found, a simple constraint such as `object20 in {“AES”}` is constructed.

REQUIRES and ENSURES section

The REQUIRES and ENSURES sections are constructed by using the information contained in the CryptoRuleDependency objects that were generated during the AUG splitting and merging step. For each CryptoRuleDependency where another CryptoRule acts as dependent node, a predicate is added to the ENSURES section. For CryptoRuleDependencies where the current CryptoRule is dependent on a different CryptoRule, a predicate is added to the REQUIRES section.

A predicate consists of a predicate name and the name of the object that is facilitating the dependency. The predicate name can optionally be specified by the user. For predicates in the REQUIRES section, conditions are also supported. They can express that a predicate only needs to be fulfilled if the conditions are also fulfilled.

For the example given in Listing 5.3, there is a dependency named predicate0 between the generated KeyGenerator and Cipher rules. As shown in Figure 5.4, the KeyGenerator is used to create a SecretKey object named secret that is then used by the Cipher. The predicate is therefore added to the ENSURES section of the KeyGenerator rule.

FORBIDDEN section

The optional CrySL section FORBIDDEN is not generated by CryptoRuleMiner. The chosen mining approach can only show which patterns are frequently used and then assume that those patterns are secure. It cannot however be used to get faulty or insecure usages. Known faulty usages would however be needed to generate the FORBIDDEN section, which specifies patterns which are insecure and therefore should not be used. To mine those patterns, additional assumptions would be needed. The change-based usage mining approach described in Section 2.2 could for example be used to generate this section. For this, it could be assumed that patterns that have often been replaced and do not exist in newer version of the observed projects are likely to be insecure or faulty. These patterns could then be used to generate the FORBIDDEN section.

5.4 Command Line Interface

A command line interface is provided to enable the easy execution of the CryptoRuleMiner. It supports the most common interactions such as the generation of CrySL-rules for any number of projects. The generated rules can be printed to the terminal and also exported into a directory. The generated AUGs can also be exported for debugging or visualization purposes.

5.5 MUBench-Runner

In order to evaluate the rules generated by the CryptoRuleMiner, they are used as input for the CogniCrypt$_{SAST}$ static analyzer. The CogniCrypt$_{SAST}$ static analyzer then tries to find misuses in the MuBench dataset. Both the generation of rules by CryptoRuleMiner and the static analysis by CogniCrypt$_{SAST}$ using the previously generated rules are automated. For this, a so-called “runner” that performs the required steps was implemented.

For the precision and recall benchmarks, the used MuDetect configuration differs from the configuration used by the CLI. While the default configuration used by CryptoRuleMiner exports each mined usage into a rule, configuration variations that only export usages that have been encountered in a specified amount of methods are provided.

A runner that performs a static analysis using the existing CrySL ruleset as input for $\text{CogniCrypt}_{\text{SAST}}$ was also implemented. Like this, the effectiveness of the generated ruleset and the existing CrySL JCA ruleset can be compared to each other.

6 Results

In this chapter, the results from the evaluation are shown. The chapter is structured according to the research questions described in Section 4.2.

6.1 Correctness of generated rules (RQ1)

A modified version of the GitHub querying script by Wickert et al. [WRE+19] was used to mine open source projects conforming to the criteria mentioned in Section 4.2. In particular, only the projects that have at least 100 stars and 100 commits as of the 21st June 2022 were considered. In total, 303 projects fulfilled the criteria. As the usage extraction part provided by MuDetect failed both for some projects individually and when mining too many projects at once, only the 100 most-starred projects on GitHub that fulfill all criteria were used to mine the required usage patterns.

The evaluated ruleset was generated using a modified version of CryptoRuleMiner that only considers usages that occur in at least two different projects. This filtering approach should prevent some misuses from being mined, as any usage considered for the ruleset would have to be included in two separate popular GitHub repositories.

Out of the 44 generated rules, six rules are not valid CrySL rules. The three rules for `java.security.Principal`, `java.security.PrivilegedAction` and `java.security.PrivilegedExceptionAction` all contain only a single event corresponding to the instantiation of an anonymous inner class. However, as all these classes are abstract, this is not a supported operation within CrySL.

The generated rule for `java.security.Security` contains an object of type `org.bouncycastle.jce.provider.BouncyCastleProvider` which is not supported as the ruleset does not depend on BouncyCastle. The rule for `java.security.KeyStore` is not valid as it contains two objects of the type `null`.

The last invalid rule is for the `java.security.Signature` API. It contains the method call `getAlgorithmName()` within its events, even though there is no such method in the `java.security.Signature` class. This error is caused by the hardcoded expansion of class name as described in Section 5.3.2. Here, the `Signature` class name that is saved within the AUG was expanded into `java.security.Signature` instead of the actually used `app.coronawarn.server.services.federation.upload.config.UploadServiceConfig.Signature`. This error can occur for any other anonymous inner classes that have the same name as a class that is specified in the hardcoded expansion list, as they do not occur within the import statements and the expansion therefore always falls back on the hardcoded list.

The remaining 38 rules conform to the CrySL syntax. However, even if the faulty rules are removed, the remaining rules can not be used to perform static analysis without requiring further manual modifications. To use the ruleset, any predicates expressing dependencies that are related to the six invalid rules need to be manually deleted from the remaining rules. The $\text{CogniCrypt}_{\text{SAST}}$ detector then works using the smaller and modified ruleset.

It can therefore be concluded that most but not all of the generated rules conform to the CrySL syntax. Therefore, a ruleset generated by mining sufficiently many large projects can not be used to perform static analysis without some small manual modifications.

6.2 Reproduction of existing ruleset (RQ2)

The generated ruleset that was described in Section 6.1 was also used to evaluate how the generated rules compare to the original CrySL ruleset. The number of rules contained in both rulesets is comparable. While the original CrySL JCA ruleset[1] consists of 49 rules while the generated ruleset has 44 rules in total. However, the classes affected by the rules differ substantially. Table 6.1 shows the contained classes for each ruleset. Only 21 classes are represented in both rulesets.

The generated ruleset does not contain any SSL- or web-related rules for classes such as `Cookie` or `SSLEngine`. This is caused by the filtering of GitHub projects, where only projects with usages of classes within the `java.security` and `javax.crypto` package were considered. The CryptoRuleMiner itself would also need to be modified in order to generate rules for classes in the `java.net.ssl` package that is part of the Java Secure Socket Extension[2].

The JCA ruleset includes rules for many `ParameterSpec` classes such as `DHGenParameterSpec`, `DSAGenParameterSpec` and `ECGenParameterSpec` while the generated ruleset only includes a rule for the `PSSParameterSpec` class that was contained within the original ruleset. A possible reason for this could be that not enough usages of the algorithms for which parameter specifications only exist in the original JCA ruleset were contained within the dataset used to generate the rules.

Additionally, the original JCA ruleset contains only one rule for the `Key` interface while the generated ruleset contains separate rules for many public or private keys implementing the `Key` interface. The added information within these three rules is however not strictly relevant for secure cryptography as the three rules `PublicKey`, `ECPublicKey` and `RSAPublicKey` only contain non-cryptography related method calls without any relationship to other rules or parameter constraints. They are therefore classified as superfluous.

There are several pairs of rules for which one rule depends on the other, but both rules do not specify any security-related behaviour. These pairs are `AccessController` and `PrivilegedExceptionAction` and `CodeSource` and `ProtectionDomain`. As they do not specify security-related behaviour, the rules contained in these pairs are classified as superfluous.

[1]Newest CrySL ruleset, as of July 4th: https://github.com/CROSSINGTUD/Crypto-API-Rules/tree/82d575190ec0f11291a381eb9e1766336a3c3d69/JavaCryptographicArchitecture/src

[2]https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

In both datasets	Only in generated dataset	Only in original JCA dataset
CertificateFactory	AccessController	AlgorithmParameterGenerator
Cipher	Certificate	AlgorithmParameters
CipherInputStream	CodeSource	CertPathTrustManagerParameters
CipherOutputStream	DESKeySpec	Cookie
DigestInputStream	DigestException	DHGenParamterSpec
ECParameterSpec	ECFieldFp	DHParameterSpec
GCMParameterSpec	ECPoint	DSAGenParamterSpec
IvParameterSpec	ECPublicKey	DSAParameterSpec
KeyFactory	EllipticCurve	DigestOutputStream
KeyGenerator	NoSuchAlgorithmException	ECGenParameterSpec
KeyPair	PKCS8EncodedKeySpec	HMACParameterSpec
KeyPairGenerator	PSSParameterSpec	Key
KeyStore	Principal	KeyAgreement
Mac	PrivilegedAction	KeyManagerFactory
MessageDigest	PrivilegedExceptionAction	KeyStoreBuilderParameters
SecretKey	ProtectionDomain	MGF1ParameterSpec
SecretKeyFactory	Provider	OAEPParameterSpec
SecretKeySpec	PublicKey	PBEKeySpec
SecureRandom	RSAPublicKey	PBEParameterSpec
Signature	RSAPublicKeySpec	PKIXBuilderParameters
X509EncodedKeySpec	Security	PKIXParameters
	SignatureException	PasswordAuthentication
	X509Certificate	RSAKeyGenParameterSpec
		SSLContext
		SSLEngine
		SSLParameters
		TrustAnchor
		TrustManagerFactory

Table 6.1: Comparison of classes contained in the generated and the original dataset

There are also isolated superfluous rules that either only contain only method calls that are not directly security related or do not contain any important parameter constraints, predicates or ordering information. These include `DESKeySpec`, `Certificate`, `EllipticCurve`, `PSSParameterSpec` and `ECParameterSpec`.

6.2.1 Rule comparisons

For some selected rules that exist within both the generated and the original JCA ruleset, detailed comparisons were done to highlight some of the achievements and shortcomings of the rule generation.

Listing 6.1 `CertificateFactory` rule from the JCA CrySL ruleset

```
1  SPEC java.security.cert.CertificateFactory
2
3  OBJECTS
4      java.io.InputStream inStream;
5      java.lang.String encoding;
6      java.lang.String type;
7
8  EVENTS
9      g1: getInstance(type);
10     g2: getInstance(type, _);
11     Get := g1 | g2;
12
13     gc1: generateCertificate(inStream);
14     GenCert := gc1;
15
16     gen1: generateCertPath(_);
17     gen2: generateCertPath(inStream, encoding);
18     GenCertPath := gen1 | gen2;
19
20     gcrl1: generateCRL(inStream);
21     GenCRL := gcrl1;
22
23 ORDER
24     Get, (GenCert | GenCertPath | GenCRL)+
25
26 CONSTRAINTS
27     type in {"X509", "X.509"};
28     encoding in {"PKCS7", "PkiPath"};
29
30 ENSURES
31     generatedCert[type];
```

Listing 6.2 `CertificateFactory` rule generated by CryptoRuleMiner

```
1  SPEC java.security.cert.CertificateFactory
2
3  OBJECTS
4  java.lang.String object103164;
5  java.io.InputStream inStream;
6  java.io.InputStream object104355;
7
8  EVENTS
9  GetInstance: getInstance(object103164);
10 GenerateCertificate: generateCertificate(inStream);
11 GenerateCertificates: generateCertificates(object104355);
12 GenerateCertificate0: generateCertificate();
13
14 ORDER
15 (GetInstance | (GetInstance, GenerateCertificate) | (GetInstance, GenerateCertificates) | (GetInstance,
   ↪  GenerateCertificate) | (GetInstance, GenerateCertificate0))
16
17 CONSTRAINTS
18 object103164 in {"X.509","X509"};
```

The generated and original rules for `CertificateFactory` are shown in Listing 6.1 and Listing 6.2. Some clear similarities between the generated and the original rule can be seen. They both allow the same certificate types as parameters for the `getInstance` method. The `generateCertificate(InputStream)` method also exists within both rules. However, there are also some shortcomings of the generated rule. Unlike in the original rule,the execution of only a `getInstance` method call is allowed but does the arbitrary repetition of the `generateCertPath`, `generateCertificate` and `generateCRL` functions is not allowed. Additionally, the order section contains some duplicates. The `generateCertificate` and `generateCertPath` methods are missing altogether. This shows that the mining would need an even bigger amount of projects to succesfully extract all methods that could be securely used. The incomplete rule could lead to false positives if it is used as-is. Calling `generateCertificate` without any parameters and `generateCertificates` is only allowed in the generated rule. The rule for `CertificateFactory` is therefore classified both as new usage and as incomplete.

The generated `CipherInputStream` rule only contains a single method call within its events. The method call is incorrect, as it only contains one of the two needed parameters. The rule is therefore considered as superfluous.

The `CipherOutputStream` also only contains a single constructor call that also has a missing parameter within the events section. However, it does correctly contain a predicate specifying that the cipher needs to be properly generated. The rule is considered to be incomplete as the faulty method call will still be matched by CogniCrypt and the rule therefore at least provides some security.

The `GCMParameterSpec` does not contain all the needed constraints and contains an incorrect amount of parameters in the specified method call. It is therefore classified as incomplete.

The generated `IvParameterSpec` rule only contains one of the method calls contained in the original rule and does therefore also not contain any of the constraints of the original rule, as they are all focused on the method call that is not contained. The generated rule does however correctly contain an ensured predicate that is required by the `Cipher` rule. It is therefore classified as incomplete.

Some of the generated rules have relaxed constraints compared to the original JCA ruleset. For example, the generated `KeyGenerator` rule allows the “HmacSHA256” and “AES” algorithms also allowed by the original CrySL rule. However, it additionally allows the usage of the “HmacSHA1” algorithm which uses the `SHA1` hash function that is considered to be broken and insecure [SBK+17]. “SHA1” itself is also allowed as an algorithm in the generated `MessageDigest` rule, along with “MD5”. Both hashing algorithms are not contained in the original JCA ruleset and therefore not considered to be secure.

The final classification for all the generated rules can be seen in Table 6.2. Out of the 44 generated rules, 24 are classified as only superfluous and therefore could be removed without any negative consequences. 17 rules have some overlap with existing CrySL rules and are classified as incomplete. 13 of these rules are however also classified as new or superfluous as they contain new, sometime unnecessary information. There are three rules which only contain new information that was not contained in the CrySL ruleset but is also not superfluous. Out of these rules, `PKCS8EncodedKeySpec` and `DESKeySpec` contain the construction of the respecitve key spec and predicates that enable their use within key factory classes. The `Provider` rule contains constraints on which provider can be used. No CrySL rule could be reprodued entirely, as such, no rules are classified as correct.

Table 6.2: Classification of the generated rules

Rule	Classification	Rule	Classification
AccessController	1	Mac	2, 3
Certificate	1	MessageDigest	2, 3
CertificateFactory	2, 3	NoSuchAlgorithmException	1
Cipher	2, 3	PKCS8EncodedKeySpec	2
CipherInputStream	1	PSSParameterSpec	1
CipherOutputStream	3	Principal	1
CodeSource	1	PrivilegedAction	1
DESKeySpec	2	PrivilegedExceptionAction	1
DigestException	1	ProtectionDomain	1
DigestInputStream	1	Provider	2
ECFieldFp	1	PublicKey	1
ECParameterSpec	1	RSAPublicKey	1
ECPoint	1	RSAPublicKeySpec	1
ECPublicKey	1	SecretKey	3
EllipticCurve	1	SecretKeyFactory	2, 3
GCMParameterSpec	3	SecretKeySpec	2, 3
IvParameterSpec	3	SecureRandom	2, 3
KeyFactory	1, 2, 3	Security	1
KeyGenerator	2, 3	Signature	2, 3
KeyPair	2, 3	SignatureException	1
KeyPairGenerator	2, 3	X509Certificate	1
KeyStore	2, 3	X509EncodedKeySpec	1

Classification according to Section 4.2.1:
1 - superfluous, 2 - new usage, 3 - incomplete, 4 - correct

Only 17 out of 49 rules could be partly reproduced by the proposed CryptoRuleMiner. As no rule could be reproduced entirely and most of the rules contained in the original ruleset were not reproduced at all, it can be concluded that the proposed CryptoRuleMiner is not able to reproduce large parts of the original ruleset when using 100 large and popular projects as input. The generated rules additionally allow some new usages that would enable insecure behaviour.

6.3 Comparison to other detectors using MuBench (RQ3)

In order to evaluate and compare the performance of CryptoRuleMiner, MuBench was used as a benchmark as described in Section 4.2. Out of the chosen datasets `MSR19-FindBugs-Exp2-and-Exp3`, `JCA-Usages`, `JCA-Changes-GH` and `JCA-Changes-SF`, the projects `jeesuite-libs`, `spring-boot-student`, `whatsmars`, `adempiere`, `jmrtd` and `progin5` were skipped as they either did not compile or were not cor-

Detector	Experiment RUB		Experiment P		Experiment R	
	Hits	Recall Upper Bound	Confirmed Misuses	Precision	Hits	Recall
CogniCrypt	29	70.7%	35	37.6%	26	35.6%
CryptoRuleMiner	20	48.8%	0	0%	0	0%
CryptoRuleMiner2	-	-	1	5.6%	0	0%
CryptoRuleMiner3	-	-	1	4.5%	0	0%
DMMC	0	0%	1	0.7%	0	0%
FindBugs	5	12.2%	5	3.7%	5	6.8%
Jadet	2	4.9%	0	0%	0	0%
MuDetect	11	26.8%				
Tikanga	1	2.4%	0	0%	0	0%

Table 6.3: Excerpt of the results of the MuBench benchmark

rectly downloaded by MuBench. As a result, a total of 17 projects were used in the benchmark. For the recall upper bound benchmark, only 13 projects were used as the `MSR19-FindBugs-Exp2-and-Exp3` dataset did not include the needed correct usages for four projects.

MuDetect was only evaluated in the recall upper bound benchmark as the online evaluation interface of MuBench did not work for the results provided by MuDetect and evaluation therefore needed to be done manually on the raw detector output.

As described in Section 5.5, variations of the CryptoRuleMiner that used only usages that occured in at least two, three, four, five, six or ten methods were evaluated. They are referred to as CryptoRuleMiner2 to CryptoRuleMiner10 where the number determines in how many methods a usage needs to be contained in order to be mined. The variations CryptoRuleMiner4 to CryptoRuleMiner10 are however not shown in the Table 6.3 as they did not find any misuses. The variations were only tested in the precision and recall benchmarks, as the recall upper bound benchmark provides only one correct example to mine and therefore should not be used with any additional mining constraints.

For the recall benchmarks, the top ten misuses of each detector were manually classified as actual misuses or false alerts. Bugs and misuses that are not directly related to crypto APIs were classified as false alerts in order to simplify the classification task. This could however lower the recall of any detector that does not only detect crypto API misuses. Therefore, the calculated recall values do not reflect the ability to detect API misuses but instead the ability to detect and prioritize crypto API misuses.

6.3.1 CryptoRuleMiner and CogniCrypt results

The results for different detectors in the MuBench benchmark with datasets of JCA misuses can be seen in Table 6.3. While the CryptoRuleMiner fails in the precision and recall benchmark where it does not recognize any actual misuses, it performs better in the recall upper bound experiment where it recognizes almost 50% of the misuses.

The 70.7% recall upper bound score of the CogniCrypt detector using the JCA ruleset is close to the perfect reachable recall upper bound percentage that can be achieved using CogniCrypt as a detector. The remaining 29.3% of misuses that were not found by CogniCrypt using the JCA ruleset were either not identified due to some fault or deficiency in the CogniCrypt detector or not considered by the default JCA ruleset. Some of the misuses that were not identified by CogniCrypt using either the default or the generated ruleset could however possibly be found by modifying CryptoRuleMiner to also consider the usages of objects that use objects output by a cryptographic method but that are not cryptographic methods by themselves and by also considering exception handling. An example for this is the first `battleforge` misuse, where the result of a cipher operation is used as a parameter for a string constructor without specifying the encoding. The ruleset that was generated by CryptoRuleMiner also does not consider those cases as it was created by using the existing JCA CrySL ruleset as a reference for which aspects should be contained in crypto API rules.

The hits found using the CryptoRuleMiner ruleset are a strict subset of the hits found using the default ruleset, i.e. the ruleset created by CryptoRuleMiner did not lead to the discovery of any additional misuses. Using the ruleset generated by CryptoRuleMiner, CogniCrypt is able to identify 20 out of 29 or 69% of the misuses that it could identify with the default JCA ruleset.

Out of the nine misuses that were identified by CogniCrypt in the recall upper bound benchmark with the default JCA ruleset but not with the ruleset created by CryptoRuleMiner, five misuses concerned a parameter that was either not properly randomized or statically defined. Two additional misuses concerned cases where a key was generated without proper randomization and then used as a parameter.

These seven misuses concerning missing parameter randomization are not identified using the CryptoRuleMiner ruleset because the the mined known-correct usage and the misuse that should be identified do not use the same exact methods. An example for this is the fourth misuse of the `smart` project. The misuse uses `Cipher.init(int opmode, Key key, AlgorithmParameterSpec params)` where the second parameter `key` of the type `SecretKey` is not a properly generated key. The correct usage however proposes a fix using `Cipher.init(int opmode, Key key, SecureRandom random)` where `key` is of the type `SecretKey` and generated by calling `KeyGenerator.generateKey()`. While the generated rule does correctly recognize that a correct usage of `Cipher.init(int opmode, Key key, SecureRandom random)` where key is of the type `SecretKey` needs a correctly generated key, it does not generalize this fact for similar `Cipher.init` calls. CogniCrypt therefore does not apply this key generation constraint to the different `Cipher.init` call in the misuse that uses a `Key` instead of a `SecretKey`. This shows a limitation of the approach: If no correct usages using the same methods and the same parameter types exist, a misuse cannot be detected, even if a correct, functionally equivalent usage of slightly different method calls was mined. Other misuses caused by incorrect randomization that are also contained in the MuBench datasets were recognized if the provided correct usage was more similar to the misuse.

The last two misuses are both caused by the usage of a “forbidden”, insecure method. As explained in Section 5.3.2, these misuses cannot be recognized by CryptoRuleMiner due to the used usage mining approach.

The precision benchmark shows a high rate of false positives for CryptoRuleMiner. Most of the false positives are related to false call orders. This shows that the detector is not able to encode all correct call orders into the created rules even when encoding every usage into a rule, i.e. when using the default CryptoRuleMiner configuration. This could be due to faulty rules or due to bugs within CogniCrypt, as the default ruleset also sometimes falsely recognized faulty orderings.

The recall benchmark shows that CryptoRuleMiner is not able to distinguish correct and incorrect misuses when creating rulesets out of source code that also includes misuses, as CogniCrypt was not able to find any misuses using the generated ruleset. CryptoRuleMiner encodes the misuses it encountered when mining usage patterns into crypto rules and therefore legitimized these misuses.

The CryptoRuleMiner variations mostly also fail to find any misuses in the precision and recall benchmarks. However, the variations that are constrained to usages that are contained in at least two or three methods both find one actual misuse in the misuse benchmark. The correctly identified misuse is the usage of MD5 as hash function, which is considered to be insecure. The other false positive misuses found in experiment 2 by CryptoRuleMiner and its variations mostly concern falsely identified faulty orderings, such as missing or unexpected method calls.

6.3.2 Results of other misuse detectors

MuDetect is the detector with the third highest recall upper bound score of 26.8%. While it is also used within CryptoRuleMiner, there is no overlap between the misuses that were detected by CryptoRuleMiner and MuDetect. Together with the nearly doubled recall upper bound of CryptoRuleMiner, this shows that the changes between MuDetect and CryptoRuleMiner which uses MuDetect for the usage mining are very significant. The modification of the MuDetect usage mining code and configurations made for CryptoRuleMiner and/or the replacement of the custom MuDetect misuse detection logic with the CogniCrypt detector based on the generated rules had a big impact on the performance of the detectors.

Out of the other detectors, FindBugs performed best with a recall upper bound of 12.2% and an actual recall of 6.8 %, which is the second highest measured recall value after CogniCrypt. FindBugs does however not rely on usage mining and uses predefined bug patterns [APH+08]. The next highest recall upper bound of 4.9% was achieved by Jadet, while Tikanga managed to find one misuse and therefore achieve a recall upper bound of 2.4%. Both Tikanga and Jadet failed to find any misuses in the precision and recall experiments. DMMC found one misuse in the precision experiment and achieved a precision of 0.7%, but did not find any misuses in the recall and recall upper bound experiments.

The comparatively bad results of most of the general purpose fault detection tools such as Tikanga, Jadet, FindBugs and DMMC show the need for specially optimizing tooling for crypto API misuses. While they performed well in the original MuBench benchmark by Amann et al. which contained different bug types, they seem to perform much worse for crypto API misuses. The recall benchmark in particular highlights another problem. While the general-purpose bug detection tools might be able to detect bugs, the security-critical crypto API misuses could be missed as many non-security related bugs are rated higher than them. This issue is magnified by the fact that the

ranking of misuses may depend on the frequency that a pattern was used. There are usually however less interactions with crypto APIs, especially when compared to other core language APIs such as Strings or UI Frameworks where some classes are often used repeatedly in a project.

The MuBench benchmark clearly shows the current limitations of the chosen approach. While CryptoRuleMiner is able to use correct crypto API usages such as the ones that are provided in the recall upper bound benchmark to create CrySL rules that have a comparable performance to the default JCA ruleset, the benchmarks recall and precision show that the current approach is not able to reliably filter out only correct usages when using a dataset that contains both correct usages and misuses. However, CryptoRuleMiner is clearly able to beat all other examined misuse detectors that do not rely on hardcoded rules in the recall upper bound benchmark. The CryptoRuleMiner variations that filter out some usage patterns in the mining step are also the best-performing usage mining-based detectors in the precision experiment.

7 Discussion

In this chapter, the results presented in Chapter 6 are examined. The achieved results of the chosen approach are discussed. Additionally, the threats to validity are described.

7.1 Results discussion

The MuBench recall upper bound benchmarks results clearly show that CryptoRuleMiner is able to produce correct CrySL rules which can be used to detect cryptography misuses when it is provided a high-quality dataset which contains corresponding correct usages for each misuse. CryptoRuleMiner even beats the recall upper bound of MuDetect which it is based on.

However, when using CryptoRuleMiner without any constructed correct usages, the created rules frequently include incorrect usages and miss some valid usages. This leads to the situation that the CryptoRuleMiner is unable to find a significant amount of misuses in the precision and recall experiments. When using a usage mining configuration that filters out some usages that only rarely occur, the ruleset can be slightly improved. This can be seen in the higher precision results of the CryptoRuleMiner2 and CryptoRuleMiner3 variations in MuBench. Even with a large dataset of 100 popular GitHub projects containing JCA usages, the ruleset generated by CryptoRuleMiner does not contain all method calls that are contained in the default JCA CrySL ruleset for any cryptography rule. Only 17 out of 49 rules can be partly recreated using CryptoRuleMiner.

The basic problem of both too many encountered misuses and too few encountered actual correct usages also can be seen in the comparison between the generated and the original JCA ruleset. Even though the ruleset discussed in Section 6.2 was generated using only usages that occur in at least two different projects, it still contains insecure usages. Additionally, only 29% of the original CrySL ruleset could be partly reproduced. This clearly shows the need for either a bigger and better dataset for usage mining or better filtering and mining approaches.

However, distinguishing between insecure and secure cryptography misuses remains a hard problem. Past studies have shown that most projects include cryptography misuses and that the misuses are usually not fixed [GKL+19] [EBFK13]. It is therefore uncertain whether an approach that is based on mining usages can at all succeed in creating secure and complete crypto API rules.

7.2 Threats to Validity

There are several threats to the validity of the reported results. The first is the chosen benchmark for the evaluation of the third research question. There exist several different benchmarks that use different approaches and different datasets and therefore could produce differing results. MuBench was chosen because of the availability, ease of use and the number of detectors which can be evaluated using it.

The second treat is the used datasets within MuBench. All datasets that contain JCA usages and are generally available were used. However, this includes the `MSR19-FindBugs-Exp2-and-Exp3` dataset. This dataset mostly consists of misuses that were identified by CryptoRuleMiner [WRE+19]. It is therefore reasonable to expect that CogniCrypt performs better than usually on this dataset, especially in the recall and recall upper bound tasks. As the proposed CryptoRuleMiner also uses CogniCrypt as a detector, it could also benefit from this dataset, as it includes only misuses that could be identified by a good CogniCrypt ruleset.

Another threat to validity of the detector comparisons is the possibility that there exist bugs within other detectors when mining the projects contained in the MuBench dataset. While any identified bugs were fixed within MuBench, some of the other detectors were implemented by third parties that did not write the detectors themselves. It is therefore plausible to assume that some bugs might exist within the other detectors and that they might be able to perform slightly better, were those bugs to be fixed. As plausible results are produced for each detector and it is assumed that the detectors were thoroughly tested using other approaches, this advantage is considered to be negligible.

8 Conclusion and Outlook

A new approach for generating crypto API usage rules that are usable by SAST tools out of public source code repositories called CryptoRuleMiner was created for this work. It is based on MuBench, an existing usage mining approach.

First, basic concepts and methods used in this thesis such as the CrySL specification that is used by the CogniCrypt SAST tool and AUGs were described. A short overview over different usage mining approaches was given. Out of the evaluated approaches, MuDetect was chosen for CryptoRuleMiner as it is open source and has an easily understandable usage representation. By using the output of a modified MuDetect mining approach, CryptoRuleMiner can use AUGs generated by MuDetect, which are focused on usage mining and already provide some helpful abstractions and an easy structure focused on control- and dataflow. This also enables CryptoRuleMiner to filter which usage approaches are mined. The implementation chapter shows some of the challenges of converting an AUG to a CrySL rule. Some of the biggest challenges include the creation of `CryptoRules` containing `CallOrders` which further simplify the controlflow aspects of AUGs and the intricacies of the CrySL format which includes interactions between the different sections of a rule and between multiple rules.

The results and discussion chapters shows the performance of the CryptoRuleMiner, including an evaluation of the misuses that were found when using the generated ruleset. The proposed CryptoRuleMiner is able to generate correct and working CrySL rules when given correct usages. CryptoRuleMiner and its variations perform better than all other surveyed mining-based misuse detection approaches within MuBench when considering only crypto API misuses. The performance of CryptoRuleMiner on actual public code repositories is nevertheless found to be lacking. The generated rules suffer from both false usages and missing correct usages in the mined repositories. This leads to a ruleset that is able to partly encode the same restrictions as the original CrySL JCA ruleset while also including some misuses as valid and missing some permitted usages. While the missing permitted usages could possibly be generated by using a bigger dataset for usage mining, this would also increase the amount of misuses that are encoded in the ruleset. The core problem of the approach thus seems to be the prevalence of misuses in the mined repositories and the primitive filtration that could not prevent the misuses from being encoded into the ruleset.

Outlook

There are some possible improvements and similar approaches that could lead to better results. Using a change-based usage mining approach could alleviate parts of the problem of mining misuses, if the hypothesis that changes eventually fix crypto API misuses proves to be true. Usages that are often removed by a commit and replaced with different usages could be removed from the generated ruleset, thus preventing replaced misuses from being encoded into the ruleset.

The approach could also be improved by curating a dataset containing only projects of particularly good quality that respect all good practices for using crypto APIs. This dataset could then be used as an input for the CryptoRuleMiner to generate better rules. Alternatively, advanced novel filtering approaches could also be implemented within CryptoRuleMiner in order to improve the quality of the generated rules.

Another possible future work could look into integrating the mined crypto API rules within an existing ruleset. Some logic for parsing existing CrySL rules into `CryptoRule` objects already exists within the project. Like this, the `CryptoRule` merge operation that is already needed for the rule generation could also be used for the generation of a merged ruleset.

Bibliography

[Ama18] S. Amann. “A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses”. PhD thesis. Darmstadt: Technische Universität, June 2018 (cit. on p. 19).

[ANN+16] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, M. Mezini. “MUBench: A Benchmark for API-misuse Detectors”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR ’16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 464–467. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903506 (cit. on pp. 26, 27, 51).

[ANN+19a] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, M. Mezini. “A Systematic Evaluation of Static API-Misuse Detectors”. In: *IEEE Transactions on Software Engineering* 45.12 (Dec. 2019), pp. 1170–1188. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2018.2827384 (cit. on p. 23).

[ANN+19b] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, M. Mezini. “Investigating Next Steps in Static API-Misuse Detection”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. May 2019, pp. 265–275. DOI: 10.1109/MSR.2019.00053 (cit. on pp. 19, 20, 23, 25).

[APH+08] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (Sept. 2008), pp. 22–29. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2008.130 (cit. on p. 51).

[ARY19] S. Afrose, S. Rahaman, D. Yao. “CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses”. In: *2019 IEEE Cybersecurity Development (SecDev)*. Sept. 2019, pp. 49–61. DOI: 10.1109/SecDev.2019.00017 (cit. on p. 24).

[Bar10] A. G. Bardas. “Static Code Analysis”. In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107 (cit. on p. 17).

[CPS20] B. Cartaxo, G. Pinto, S. Soares. “Rapid Reviews in Software Engineering”. In: *Contemporary Empirical Methods in Software Engineering*. Ed. by M. Felderer, G. H. Travassos. Cham: Springer International Publishing, 2020, pp. 357–384. ISBN: 978-3-030-32489-6. DOI: 10.1007/978-3-030-32489-6_13 (cit. on p. 25).

[EBFK13] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel. “An Empirical Study of Cryptographic Misuse in Android Applications”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 73–84. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516693 (cit. on pp. 15, 23, 53).

[Ecl] Eclipse Foundation. *CogniCrypt - Secure Integration of Cryptographic Software | CogniCrypt*. https://www.eclipse.org/cognicrypt/ (cit. on p. 17).

[GKL+19] J. Gao, P. Kong, L. Li, T. F. Bissyandé, J. Klein. "Negative Results on Mining Crypto-API Usage Rules in Android Apps". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. May 2019, pp. 388–398. DOI: 10.1109/MSR.2019.00065 (cit. on pp. 15, 23, 53).

[KSA+21] S. Krüger, J. Späth, K. Ali, E. Bodden, M. Mezini. "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs". In: *IEEE Transactions on Software Engineering* 47.11 (Nov. 2021), pp. 2382–2400. ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2948910 (cit. on p. 18).

[LCP+21] W. Liu, B. Chen, X. Peng, Q. Sun, W. Zhao. "Identifying Change Patterns of API Misuses from Code Changes". In: *Science China Information Sciences* 64.3 (Mar. 2021), p. 132101. ISSN: 1674-733X, 1869-1919. DOI: 10.1007/s11432-019-2745-5 (cit. on pp. 19, 24, 25).

[NNP+09] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, T. N. Nguyen. "Graph-Based Mining of Multiple Object Usage Patterns". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering on European Software Engineering Conference and Foundations of Software Engineering Symposium - ESEC/FSE '09*. Amsterdam, The Netherlands: ACM Press, 2009, p. 383. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595767 (cit. on p. 23).

[OMGS18] T. D. Oyetoyan, B. Milosheska, M. Grini, D. Soares Cruzes. "Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital". In: *Agile Processes in Software Engineering and Extreme Programming*. Springer, Cham, May 2018, pp. 86–103. DOI: 10.1007/978-3-319-91602-6_6 (cit. on pp. 15, 26).

[PTRV18] R. Paletov, P. Tsankov, V. Raychev, M. Vechev. "Inferring Crypto API Rules from Code Changes". In: *ACM SIGPLAN Notices* 53.4 (June 2018), pp. 450–464. ISSN: 0362-1340. DOI: 10.1145/3296979.3192403 (cit. on pp. 19, 24).

[RBK+13] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, T. Ratchford. "Automated API Property Inference Techniques". In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 613–637. ISSN: 1939-3520. DOI: 10.1109/TSE.2012.63 (cit. on pp. 18, 23).

[RXA+19] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, D. (Yao. "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2455–2472. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3345659 (cit. on p. 24).

[SBK+17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov. "The First Collision for Full SHA-1". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by J. Katz, H. Shacham. Vol. 10401. Cham: Springer International Publishing, 2017, pp. 570–596. ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_19 (cit. on p. 47).

[SWK+22] M. Schlichtig, A.-K. Wickert, S. Krüger, E. Bodden, M. Mezini. "CamBench – Cryptographic API Misuse Detection Tool Benchmark Suite". In: (Apr. 2022). DOI: 10.48550/arXiv.2204.06447 (cit. on p. 24).

[WRE+19] A.-K. Wickert, M. Reif, M. Eichberg, A. Dodhy, M. Mezini. "A Dataset of Parametric Cryptographic Misuses". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. May 2019, pp. 96–100. DOI: 10.1109/MSR.2019.00023 (cit. on pp. 27, 43, 54).

[WZ11] A. Wasylkowski, A. Zeller. "Mining Temporal Specifications from Object Usage". In: *Automated Software Engineering* 18.3 (Dec. 2011), pp. 263–292. ISSN: 1573-7535. DOI: 10.1007/s10515-011-0084-1 (cit. on p. 19).

All links were last followed on July 7, 2022.

Declaration

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

---

place, date, signature