Detecting Custom Cryptography in Android Applications

Detektion von selbst entwickelten kryptographischen Algorithmen und Obfuskierungsalgorithmen in Android Applikationen
Master-Thesis von Tim Ohlendorf
05/02/2018

1. Gutachten: Prof. Dr. Michael Waidner
Detecting Custom Cryptography in Android Applications
Detektion von selbst entwickelten kryptographischen Algorithmen und Obfuskierungsalgorithmen in Android Applikationen

Vorgelegte Master-Thesis von Tim Ohlendorf

1. Gutachten: Prof. Dr. Michael Waidner

Tag der Einreichung:
Thesis Statement pursuant to §22 paragraph 7 of APB TU Darmstadt

Hiermit versichere ich, Tim Ohlendorf, die vorliegende Master-Thesis gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. 
Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.
Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

I herewith formally declare that I, Tim Ohlendorf, have written the submitted thesis independently pursuant to §22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. 
I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. 
In the submitted thesis the written copies and the electronic version for archiving are pursuant to §23 paragraph 7 of APB identical in content.

Date: Signature:
# Contents

## List of Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>VII</td>
<td></td>
</tr>
</tbody>
</table>

## List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX</td>
<td></td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1. Motivation ................................................. 3  
1.2. Problem Statement and Research Question ................................. 3  
1.3. Methodology and Scope ........................................ 4  
1.3.1. Methodology ........................................... 4  
1.3.2. Scope .................................................. 4  
1.4. Outline .................................................. 4

## 2 Background

2.1. Sensitive Data .............................................. 5  
2.2. Android Ecosystem ......................................... 5  
2.2.1. System Architecture and Application Format ......................... 5  
2.3. Static Program Analysis .................................... 6  
2.3.1. Used Frameworks ........................................ 6  
2.3.1.1. Soot ................................................. 6  
2.3.1.2. FlowDroid ......................................... 7  
2.4. Machine Learning ........................................... 7  
2.4.1. Supervised Classification and Regression Algorithms ................. 8  
2.4.1.1. Naïve Bayes ....................................... 8  
2.4.1.2. Support Vector Machine (SVM) .......................... 9  
2.4.2. Feature Selection ...................................... 9  
2.4.3. Evaluation ............................................. 10  
2.4.4. Waikato Environment for Knowledge Analysis (Weka) .................. 12

## 3 Custom Cryptography

3.1. Introduction ................................................. 13  
3.2. Definitions and Common Characteristics ........................... 13  
3.3. Examples .................................................. 14  
3.3.1. Custom Cryptography in WhatsApp ............................ 14  
3.3.2. Ambiguous Usage of Encoding and Decoding Schemes ............... 15

## 4 Custom Cryptography Detection Framework

4.1. Architecture Overview ..................................... 17  
4.2. Preprocessing ............................................. 18
4.3. Feature Extraction .................................................. 18
  4.3.1. Method Signature ............................................... 18
    4.3.1.1. IsHasParam ............................................. 18
    4.3.1.2. ParamCount ............................................. 18
    4.3.1.3. KnownParamRatio ..................................... 18
    4.3.1.4. Known Parameter-return-value Type Combinations .... 19
    4.3.1.5. IsStaticMethod .......................................... 19
    4.3.1.6. IsHashCodeMethod ..................................... 19
    4.3.1.7. IsConstructor .......................................... 20
  4.3.2. Method Body .................................................... 20
    4.3.2.1. BinOpRatio ............................................. 20
    4.3.2.2. LoopRatio ............................................... 20
    4.3.2.3. APICallRatio .......................................... 20
    4.3.2.4. IsHasThisFieldAccess ................................ 20
    4.3.2.5. ArrayAccessRatio ..................................... 21
  4.3.3. Application Context .......................................... 21
    4.3.3.1. CallCountAbsolute and CallCountRatio .............. 21
    4.3.3.2. MaxCallChainLength ................................... 21
    4.3.3.3. DataFlowFromParamRatio .............................. 21
    4.3.3.4. IsInSensitiveDataFlow ................................ 22
  4.4. Training, Evaluation and Classification ........................ 22
    4.4.1. Training .................................................... 22
    4.4.2. Classification ............................................. 22
    4.4.3. Evaluation ................................................. 22

5 Implementation .......................................................... 23
  5.1. Overview .......................................................... 23
  5.2. Preprocessing & Feature Extraction .............................. 24
    5.2.1. Feature Extraction Example: CallCountRatio ............ 24
  5.3. Learning & Classification ...................................... 25
    5.3.1. Learning ................................................... 25
    5.3.2. Classification ............................................. 26
  5.4. Evaluation ........................................................ 26

6 Evaluation ............................................................. 27
  6.1. Test Set-up ....................................................... 27
  6.2. Classifier and Feature Evaluation ............................... 27
    6.2.1. Evaluation Dataset ........................................ 27
    6.2.2. Bias of Unbalanced Datasets ............................. 28
    6.2.3. 10-fold cross-validation .................................. 28
      6.2.3.1. Ranking of the features .............................. 28
      6.2.3.2. Validation Results .................................... 30
  6.3. Case Studies ........................................................ 32
    6.3.1. Google Play Store Applications .......................... 32
      6.3.1.1. True Positives ........................................ 33
      6.3.1.2. False Positives ........................................ 35
    6.3.2. Android Obfuscation Techniques .......................... 36
List of Figures

2.1 Android operating system architecture overview ................................................. 5
2.2 Deployment process of an Android application to a device ................................. 6
2.3 Supervised machine learning (ML) classification system ................................. 8
2.4 Classic segmentation model for the training and evaluation on the dataset .......... 10
2.5 10-fold cross-validation ................................................................................. 10
2.6 Explanatory figure for precision and recall ....................................................... 11
2.7 An information gain analysis of a randomly generated dataset in the Weka Explorer tool ................................................................. 12
3.1 Schematic of a custom cryptography algorithm .................................................. 14
4.1 Architectural overview of the custom cryptography detection framework components ................................................................. 17
5.1 Implementation overview of the custom cryptography detection framework .......... 23
5.2 Components involved in the feature extraction process of the custom cryptography detection framework ................................................................. 24
6.1 KnownParamRatio feature distribution of the resampled dataset ....................... 29
6.2 Accuracy of different classifier algorithms with increasing amount of features ..... 30
6.3 F1 score of different classifier algorithms with increasing amount of features ...... 31
6.4 Runtime of the 10-fold cross-validation and the case studies (feature extraction and classification/training) on the test system with 8 applications (apps) processed in parallel ................................................. 38
## List of Tables

4.1 Features of known parameter-return-value type combinations ........................................ 19

6.1 Total instances per class in the evaluation dataset ......................................................... 28
6.2 Ranking of the features according to their information gain ........................................... 29
6.3 Maximum F1 scores of the tested classifiers with all features enabled ............................. 32
6.4 Confusion matrix of the 10-fold cross-validation of the simple logistic classifier with all features enabled .......................................................... 32
6.5 Results of the Play Store app classification ................................................................. 33
6.6 Results of the classification of the timeouted Play Store apps without data-flow features enabled .......................................................... 33
6.7 Results of the classification of the obfuscation dataset with all features enabled .............. 37

A.1 Malware applications containing custom cryptography in the dataset ............................. 51
A.2 Benign applications containing custom cryptography in the dataset set .......................... 52
A.3 Applications which methods were used as samples for non custom cryptography ............. 53

B.1 Play Store application dataset ....................................................................................... 55
B.2 Play Store application dataset checksums ..................................................................... 55

C.1 Obfuscation experiment application dataset from the F-Droid store ............................. 57
C.2 Obfuscation experiment application dataset checksum .................................................. 57
app  application
ML   machine learning
APK  Android application package
CFG  control-flow graph
DFG  data-flow graph
SVM  support vector machine
SMO  sequential minimal optimization
LWL  locally weighted learning
Abstract

Being confronted with a large number of newly published applications every day, Android application stores, such as Google Play or the Amazon Appstore, have to ensure that every application complies with the store's development policies. Especially the users' security and privacy are primary concerns. Although there are several best-practices guidelines for secure development, developers still erroneously make use of self-implemented and sometimes also self-invented algorithms to secure user data.

To stop the usage of these potentially insecure approaches, we introduce the Custom Cryptography Detection Framework. The framework is the first fully automatic approach to detect and locate the self-invented and self-implemented algorithms, so-called custom cryptography, in Android binaries. It is based on supervised machine learning and uses features acquired by static code analysis. We therefore investigate various benign and malicious Android applications, to find common characteristics of custom cryptography implementations, which are used as possible features. The evaluation of the framework shows a decent detection rate, resulting in a F1 score of ~0.96. Nevertheless, further case studies reveal some challenges with respect to detection rate and performance which have to be solved to apply the framework on a big scale.
Introduction

1.1 Motivation

With its 2 billion monthly users [48], the Android OS is one of the most important operating systems on the worldwide smartphone market. Its application store, the Google PlayStore [19], currently serves over 3.3 million different applications (app) [46].

Often dealing with sensitive information (bank account credentials, personal files, ...), Android app developers have to ensure that this data is processed, stored and transmitted securely. To support them, Google has taken much effort to simplify the usage of security measurements, like file encryption, network traffic encryption, network traffic authentication and device authentication. Additionally, the Android Developers portal provides several best-practice guidelines for implementing security features in apps [14]. Despite these efforts, many developers still do not integrate technologies like TLS or Certificate Pinning into their apps. Instead, they transmit and store sensitive data in plain or make use of self-implemented, often also self-invented algorithms to secure the user data. Beside benign apps developers, also Android malware developers often make use of this so-called custom cryptography, when hiding malicious behavior by obfuscating application logic and data.

Developing new cryptographic algorithms requires specialized knowledge and review processes, which usually can not be provided by a software developer. As a result, custom cryptography do not fulfill common quality standards for secure transmission and storage of data [21].

A main reason for using custom, instead of established standardized cryptography, could be a lack of knowledge in secure software engineering. Focusing on malware development, custom cryptography might purport a higher effort for a malware analyst to understand the malware’s inner workings, which results in a longer period of not being detected by anti-virus software. Finally, performance issues with standardized cryptography algorithms could force both, benign and malware developers, to use custom solutions, particular on entry-level hardware.

1.2 Problem Statement and Research Question

As a consequence of the failure to comply with common security standards mentioned above, Android apps using custom cryptography solutions pose possible threats to the security and privacy of its users’ sensitive data. For an app store, it is essential to detect and remove apps using custom cryptography to preserve its reputation and market share. From a security analyst’s perspective, detecting custom cryptography in malware could drastically decrease the amount of work which is needed to understand the different aspects of the insights of the malware, like command & control communication, file encryption, etc.

A manual security audit of an Android app, to detect custom cryptography, is time consuming, error-prone and because of the high amount of apps in the common app stores not applicable in a real world scenario. Also the fact, that apps normally are deployed as a binary without the possibility to acquire the source code, would make manual reviews even more complex.

In this thesis we want to investigate these issues and try to find a solution for the problem of automatically detecting custom cryptography in Android application packages (APKs). Therefore the following research questions are raised:

"Is it possible to find a common definition for custom cryptography algorithms?"

"Is it possible to fully automatically detect previously unseen instances of custom cryptography in Android APKs with an appropriate amount of time and resources?"
1.3 Methodology and Scope

1.3.1 Methodology

To answer the research questions raised above, we will investigate different ML approaches to detect custom cryptography in Android binaries, so-called APKs. Therefore common characteristics and definitions of custom cryptography solutions will be identified, by manually reverse engineering benign and malware app samples. The collected characteristics will then be used to tailor the ideal feature set for the ML learners. The training of the learner and the evaluation of the resulting ML classifiers, a 10-fold cross-validation, will be performed with an also manually acquired dataset of benign and malware apps. Finally, further experiments will be carried out. The first experiment examines the performance of the final classifier in respect to accuracy and applicability in a real world scenario. The second experiment evaluates the performance when faced with different obfuscation techniques.

1.3.2 Scope

The scope of the used ML approaches is limited to supervised classifiers only. For feature selection, we only take features acquired by static code analysis into account. Neither native libraries nor apps built with a multi-platform framework, e.g. Apache Cordova [13] or Xamarin [20], resulting in JavaScript and HTML application code will be considered for investigation. As custom cryptography normally represents an independent and often reused application component, the final approach will be able to detect custom cryptography on method level only.

1.4 Outline

First, section 2 provides an introduction into this works’ terminology and used concepts, namely the Android ecosystem, ML approaches and static program analysis. Section 3 then presents the definition and characteristics of custom cryptography, followed by some real-world code examples. Afterwards, in section 4, the conceptual architecture of the custom cryptography detection framework is presented, followed by section 5, which focuses on the implementation details of the framework. Further, section 6 deals with the evaluation of the custom cryptography detection framework. A 10-fold cross-validation, as well as two case studies are presented. Then, 7 gives an overview of related work in the field of automatic (custom) cryptography detection in binaries. Section 8 reveals possible limitations of the presented approach. Finally, section 9 and section 10 provide the conclusion and future work.
2 Background

In the following, an introduction to the terminology of the concepts and technologies used in this work is given. This includes knowledge about static code analysis (tools), the Android ecosystem as well as machine learning.

2.1 Sensitive Data

Beside credentials like passwords, PINs and keys, our definition of the term sensitive data also includes sensitive information defined by Arzt et al. [36]. This could be location information, SMS messages, pictures, contact data or unique identifiers, like IMEI and MAC-address.

2.2 Android Ecosystem

The Android OS is a mobile operating system developed by Google and originally targeted smartphones and tablets. In the years after its initial release, Android has been adapted and ported to various other devices, like PCs, TVs, game consoles and other embedded hardware. As of 2017, Android’s share in sales to end users on the global mobile OS market lies around 86% [45]. This is about four times more than its direct competitor Apple could reach with its mobile OS iOS.

2.2.1 System Architecture and Application Format

Android is based on a modified version of the Linux kernel and is distributed as open source software [11]. Above the kernel, the Hardware Abstraction Layer (HAL) provides access to the hardware from higher-level components. Each application running on an Android device is executed inside its own process, called Android Runtime (ART) and also runs under its own Linux user. Resulting from that, applications are isolated from each other. Access to the Android feature-set is provided by the Java API Framework to the applications. Figure 2.1 shows a schematic overview of the above mentioned components.

![Android operating system architecture overview](image)

**Figure 2.1.:** Android operating system architecture overview

Applications can be written in Java or Kotlin. Native C/C++ components are supported via the Android NDK [9]. To deploy an Android app to a device (see figure 2.2), it is first compiled to Java byte code and then translated to Dalvik...
bytecode as a Dalvik Executable (dex). Finally, all application resources, native libraries as well as the dex file(s) are signed and packed as an APK.

![Deployment process of an Android application to a device](image)

**Figure 2.2.:** Deployment process of an Android application to a device

When installing an application on a device, the dex file inside the APK is compiled for the device’s CPU architecture using the on-device tool `dex2oat`. This, so-called, ahead-of-time (AOT) compilation [10] is applied to improve the apps’ performance at runtime and can be viewed in figure 2.2.

### 2.3 Static Program Analysis

In static analysis, a computer program is analyzed without execution [23] [34]. Its goal is to ensure software quality by identifying design flaws and implementation errors in source code or binaries [31]. In contrast to a code review, static analysis is performed by an automated process without the need of a human analyst. Its analysis level varies from statement analysis to analysis of the whole source code of a program. Thereby, different tools cover different aspects of the software quality and different representation formats are used for analysis.

For example, formal methods (e.g. formal specification or theorem prover), which can verify a program’s predefined specifications with respect to safety and security, solely obtain their results by applying mathematical methods on an abstract representation of the program.

Another code representation, called control-flow graph (CFG) [47], represents a program in form of a directed graph where each node corresponds to a statement or basic block and each edge maps the control flow of the program. The so-called data-flow analysis [42], which analyses between which components of a program data is passed and how it is altered, is one of various analyses which relies on the CFG’s control flow information.

In practice, the above analysis can perform various checks, like identifying memory errors, resource leaks or race conditions which, with further interpretation, can reveal possible security vulnerabilities or safety issues. Beside the already mentioned use cases, also compilers use static analysis for optimization purposes.

In contrast, in the field of dynamic program analysis, programs are explicitly executed to perform analyses and obtain results regarding software quality and performance aspects. As the approach in this thesis does not make use of dynamic program analysis, no further introduction to this topic is given.

### 2.3.1 Used Frameworks

#### 2.3.1.1 Soot

The Soot framework [41] [1], initially designed for Java bytecode optimization, nowadays provides powerful static code analysis capabilities. These capabilities can be applied to various fields of application in security, privacy and general software development [26].
Different supported input formats (e.g. jars, dex files) can be processed together and are transformed into Soot’s intermediate representation, called Jimple. This simple three-address language is typed and also provides typed local variables. Listing 2.1 shows a short example of Jimple’s syntax.

```java
public int foo(java.lang.String) { // locals
    r0 := @this; // IdentityStmt
    r1 := @parameter0;
    if r1 == null goto label0; // IfStmt
    $i0 = r1.length(); // AssignStmt
    r1.toUpperCase(); // InvokeStmt
    return $i0; // ReturnStmt
}
```

Listing 2.1: Jimple code example [12]

Furthermore, in the Android code analysis domain, plenty of other frameworks like Harvester [37], SuSi [36] and FlowDroid [2] use Soot as a base for their analyses.

### 2.3.1.2 FlowDroid

FlowDroid, introduced by Arzt et al. [2] [1], is a framework providing static data-flow analyses for Java and especially Android apps, based on the Soot framework [41] [1] and an IFDS solver [5] [40]. It is typically used to track data-flows of sensitive information (see section 2.1) inside a program, called sensitive flows. The goal is to identify possible unwanted leakage of this information to a third-party. In detail, a sensitive flow is defined as a data-flow between a predefined start and end method, called sources and sinks.

As a short example, we consider the "getter" method of a password input field as our source and the "write" method of a network socket as our sink. Using these two, FlowDroid now tries to find sensitive data-flows where passwords are send over the network and returns them to the user. To identify a privacy leak, a further analysis would have to check whether the password is encrypted or not before it is send.

Being platform independent, FlowDroid is able to abstract away from platform specific models. In Android, this model is called activity life cycle [15]. It defines the complete lifecycle for all components in an app and thereby results in multiple entry points when starting the app. Also it results in the inability to pre-determine its components execution order. To cope with that, FlowDroid introduces a dummy main which simulates the life cycle of the app and allows to model all possible transitions in it precisely. Furthermore, the concept of Android callbacks, which can be used to e.g. receive the result of an asynchronous task or to interact with the system’s UI, is also considered in the dummy main. Finally, FlowDroid also provides hand-crafted rules for the common native libraries in Android and Java, which allows to track an app's data-flow even when using native method calls.

### 2.4 Machine Learning

ML is a subdomain in the field of artificial intelligence (AI) research. It attempts to enable computer systems to learn from datasets without the need of being programmed by a human. In fact, an algorithm is used to automatically learn rules from patterns in the dataset which later can be applied to new and unseen data for classification or regression [7].
ML approaches are typically divided into two categories: supervised and unsupervised learning. In supervised learning, the algorithm gets input data and corresponding output data, so called labels. The goal is to find patterns in the relationships between input data and labels. During unsupervised learning, no labels for the input data are given. An algorithm has to find hidden structures in the input data by itself to cluster the output data or assign specific labels.

As the following work mainly focuses on supervised learning, the supervised ML system in figure 2.3 will be explained in detail. During the training phase, a specific set of features of each instance of the input data will be extracted. A feature is a quantity which describes the instance. The resulting feature vectors of each instance of the input data will be passed together with the corresponding labels to the chosen machine learning algorithm, called learner. The learning algorithm now tries to learn relationships between data and labels for further prediction. The resulting classifier can now be used to assign labels to unlabeled input data by extracting the data’s feature vector and applying the trained classifier algorithm.

![Supervised ML classification system](image)

**Figure 2.3:** Supervised ML classification system

### 2.4.1 Supervised Classification and Regression Algorithms

In the following, two supervised classification algorithms will be explained in detail to depict their inner workings.

#### 2.4.1.1 Naïve Bayes

The Naïve Bayes classifier belongs to the category of simple probabilistic classifiers. It is based on the Bayes rule and conditional probability. It can be trained very fast and handles binary, numeric and nominal features [38].

The conditional probability, showed in equation 2.1, predicts the probability that \( A \) will happen, given that \( B \) has already happened. This can be calculated by dividing the probability for the intersection of \( A \) and \( B \) by the probability of \( B \).

\[
P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{2.1}
\]
In the following, we will refer A to (known) Evidence E and B to Outcome O which results in equation 2.2.

\[
P(O \cap E) = P(O) \times P(E|O)
\]

\[
P(O \cap E) = P(E) \times P(O|E)
\]

(2.2)

Further, we can now transform equation 2.2 into Bayes rule as follows:

\[
P(B) \times P(A|B) = P(A) \times P(B|A)
\]

\[
P(O|E) = \frac{P(E|O) \times P(O)}{P(E)}
\]

(2.3)

The equation mentioned above can only predict one piece of evidence. To deal with multiple given evidence, naïve bayes can be used. This approach handles each piece of evidence as independent which leads to the following equation:

\[
P(O|E_1, ..., E_n) = \frac{P(E_1|O) \times P(E_2|O) \times ... \times P(E_n|O) \times P(O)}{P(E_1, E_2, ..., E_n)}
\]

(2.4)

The assumption of independence limits the usage of naïve bayes to ML problems where the features are independent.

2.4.1.2 Support Vector Machine (SVM)

A Support Vector Machine (SVM) is a binary classifier which can deal with numeric features only. Further, both classification and regression are supported. On a high level, each instance of a dataset is represented as a point in a n-dimensional space (support vector) [39]. n is defined by the amount of features in the feature vector. In case of classification, the goal is to find a hyper-plane that divides the support vectors of the two classes the best. To identify the hyper-plane, the following rules are applied [39]:

- **Rule of thumb**: Select a hyper-plane which segregates the two classes the best.

- **Maximize the margin**: The space between each class and the hyper-plane is called margin. Try to maximize the margin which results in a higher robustness when it comes to classification.

- **Accurate classifying prior maximizing margin**: It is more important to segregate both classes accurately than to maximize the margin.

- **Robustness**: Outliers, support vectors which could not be separated correctly, will to some degree be ignored by the algorithm.

- **Kernel trick**: If two classes could not be separated properly, the kernel trick can be applied. It transforms a low dimensional input space into a higher dimensional space which then enables a separation.

Because of different kernel functions and variation in number of support vectors, SVMs are very flexible. On the downside, this leads to a high computational complexity which, together with only supporting binary classification, limits the use cases.

2.4.2 Feature Selection

Feature selection is the process of finding values and attributes, so-called features, which describe the characteristics of an instance of the given input data. Further, the features must be evaluated and selected to avoid noise and overfitting, caused by redundant or irrelevant features. A sloppy feature selection can lead into poor classification results. Beside an empirical approach, common metrics like information gain and mutual information, can be used to select features. For more information see Manning et al. [32].
2.4.3 Evaluation

To evaluate the performance of a supervised classifier and analyze its classification errors, standard models and metrics are used. Having a relatively large amount of data available to train and evaluate the classifier, the dataset $set_{data}$ is normally divided into a development $(2/3 \times set_{data})$ and test set $(1/3 \times set_{data})$, whereas the development set is divided once again equally into the training and dev-test set. Figure 2.4 shows a dataset which fulfills the above requirements.

First, the training set is used to train the classifier. Then the dev-test set is used to improve the classifier’s results during further development by letting it classify instances from the dev-test set. As the right labels for each instance in the dataset is known, wrong predictions can be found and investigated. In the last step, the test set is used to evaluate the final classifier’s prediction output, by applying metrics, mentioned below.

![Figure 2.4: Classic segmentation model for the training and evaluation on the dataset](image)

Dealing with a small amount of data in the dataset, the above approach does not perform very well. Instead, a n-fold cross-validation [25] can be used. Figure 2.5 shows a n-fold cross-validation with $n = 10$. Here, the dataset is divided into 10 equal subsets (folds). During 10 runs of training and evaluation, one fold is used as the test set and the 9 remaining folds are used as the training set. The results of each run will then be evaluated independent and finally averaged.

![Figure 2.5: 10-fold cross-validation](image)

To process and interpret the prediction results from one of the described approaches, specific terminology and metrics are used. Figure 2.6 shows a generic prediction output of a classification process with test data. In the following, we define two prediction classes $class_A$ and $class_B$, respectively labels which can be assigned by a classifier to items from a test set. As of its definition, the test set consists of items and their corresponding labels, assigned by a human beforehand. Furthermore, at this evaluation we want to get the evaluation results solely for $class_A$, the relevant class:
• **True Positive (TP):** Item of class A which was assigned the label class A by the classifier.

• **False Positive (FP):** Item of class B which was assigned the label class A by the classifier.

• **True Negative (TN):** Item of class B which was assigned the label class B by the classifier.

• **False Negative (FN):** Item of class A which was assigned the label class B by the classifier.

![Diagram illustrating True Positives, False Positives, True Negatives, False Negatives, and selected elements.]

**Figure 2.6:** Explanatory figure for precision and recall [52]

With this terminology, we can now define the following metrics:

The formula in equation 2.5 represents the formal illustration of the precision, explained in figure 2.6. In detail, it indicates how many of the retrieved items (TP+FP) are relevant (TP).

\[
Precision = \frac{TP}{TP + FP} \tag{2.5}
\]

The formula in equation 2.6 represents the formal illustration of the recall, explained in figure 2.6. In detail, it indicates how many of the relevant items (TP+FN) are retrieved by the classifier (TP).

\[
Recall = \frac{TP}{TP + FN} \tag{2.6}
\]
The $F_1$ score in equation 2.7 is a metric to trade-off precision against recall.

\[
F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]  

(2.7)

The accuracy in equation 2.8 is the fraction of predictions that are correct.

\[
\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}
\]  

(2.8)

### 2.4.4 Waikato Environment for Knowledge Analysis (Weka)

Weka, developed by the University of Waikato, is an open source framework providing a collection of tools and Java libraries for machine learning and data mining [51]. One of the provided tools, the Weka Explorer allows easy feature and classifier evaluation within an graphical user interface. For example figure 2.7 shows the results of an information gain analysis on a random generated dataset. Another useful tool, called Auto-WEKA allows to randomly test different classifier-feature combinations, to automatically find the classifier-feature combination which performs the best.

![Figure 2.7.: An information gain analysis of a randomly generated dataset in the Weka Explorer tool](image)

The Weka Java library already provides various machine learning algorithms [49] and interfaces for feature types which can easily be integrated into own projects.
3 Custom Cryptography

3.1 Introduction

In general, the term cryptography describes the study of mechanisms which enable a secure transmission or storage of information in the presence of one or more adversaries [28]. In modern cryptography, so-called crypto systems based on cryptographic primitives (public-key, symmetric-key, elliptic-curve cryptography, hashing, etc.) provide those mechanisms and algorithms. Usually, they are accessible to the developer through a standardized library (e.g. in Android: JCA, Bouncy Castle, Spongy Castle).

Often mixed with or mistakenly assumed to be cryptography is the term obfuscation. Obfuscation describes algorithms or techniques which are applied to information with the intention to make it hardly or impossible to read for a third person [24]. However, in contrast to cryptography, obfuscation can easily be reversed, if the third person gets access to or knowledge about the used algorithm or technique.

In the following, we define a subset of the union of cryptography and obfuscation, to which we refer as "custom cryptography". Elements of this set share common significant characteristics which were extracted by an empirical investigation of various benign and malicious apps from the Android platform. Nevertheless, we assume that these characteristics and the resulting definitions of custom cryptography also hold for other platforms and programming languages, especially ones that base on or make use of the Java programming language. A list of the investigated apps can be found in appendix A.

3.2 Definitions and Common Characteristics

In this section, we develop a set of definitions, which formally describe custom cryptography.

**Definition 3.2.1.** *Subset definition:* Custom cryptography is a subset of the union of cryptography and obfuscation. Therefore, its purpose is to cryptographically secure or obfuscate information.

**Definition 3.2.2.** *Custom cryptography algorithm:* A custom cryptography algorithm is self-implemented and often also self-invented. The full algorithm is present in the application code.

Further, we define a custom cryptography algorithm as follows (see also figure 3.1):

**Definition 3.2.3.** *Arithmetic-logic transformation:* An input $i$ is transformed by an arithmetic-logic operation $op_{al}$ into an output $o$. Optionally, a key or secret $k$ can be required by $op_{al}$ for the process. The key can either explicitly be provided as a variable or can be present in the algorithm as a constant. The process of transforming $o$ back into $i$ by applying $op_{al}$ can be possible, but is not mandatory.
Definition 3.2.4. **Cryptographic Primitives:** Implementations of cryptographic primitives (e.g. AES, RSA, SHA1, ...) are excluded from the set of custom cryptography as they are standardized and security audited.

Definition 3.2.5. **Substitutability:** Replacing a custom cryptography algorithm inside a program with a corresponding cryptographic primitive does preserve the programs semantic, on condition that interacting remote components get replaced with the same cryptographic primitive.

Beside the formal definitions from above, custom cryptography implementations share common characteristics which we will mention shortly below:

**Occurrence in an Application**

Custom cryptography algorithms are normally not located in standardized (cryptographic) libraries.

**Custom Cryptography Primitives**

The following primitive operations are used, either alone or in combination, by developers, to design and program custom cryptography solutions:

- substitution (e.g. lookup table)
- logical operations (e.g. byte shifting, XOR)
- simple mathematical operations (e.g. addition, multiplication, substitution)
- sophisticated mathematical operations (e.g. prime factorization)

### 3.3 Examples

#### 3.3.1 Custom Cryptography in WhatsApp

The code example in listing 3.1 shows an implementation of a simple custom cryptography algorithm, used in the Whatsapp Messenger (see appendix B). Every character of the string parameter `param0` (line 3) is XORed with the static integer value 18 (line 4). Finally, the transformed string is returned (line 6).
private static String a(String param0) {
    StringBuilder strBuilder = new StringBuilder();
    for (int i = 0; i < param0.length(); ++i) {
        strBuilder.append((char)(param0.charAt(i) ^ 18));
    }
    return strBuilder.toString();
}

Listing 3.1: Example of custom cryptography in Whatsapp for Android (see appendix B)

In detail, the algorithm in listing 3.1 features the following characteristics:

- It is self-invented and self-implemented (the full algorithm is present in the code).
- It does not represent a cryptographic primitive, but could be substituted by one.
- It is used for obfuscating information (not shown here, but present in the surrounding application code).
- The input param0 is transformed by the arithmetic logic operation XOR into the return value.
- Its implementation is not located in a standardized library.
- A logical operation is used as the primitive.

As a result, listing B fulfills all definitions of custom cryptography.

3.3.2 Ambiguous Usage of Encoding and Decoding Schemes

Listing 3.2 represents a manual implementation of the Base64 encoding scheme. This and other so-called binary-to-text encoding scheme algorithms are used to encode binary data into human-readable strings, e.g. encoding of mail attachments. Beside the intended use case, our investigation on Android apps and malware shows that many developers misuse encoding schemes for the purpose of cryptographic operations or obfuscation. Without further context information, we can determine that the present algorithm in listing 3.2 complies with the common characteristics of a custom cryptography implementation. Furthermore, all definitions from section 3.2, except definition 3.2.1, are fulfilled as of the ambiguous use cases mentioned above.

public static String encode(byte[] data) {
    char[] tbl = {...}; // [A-Z], [a-z], [0-9], +, /
    StringBuilder buffer = new StringBuilder();
    int pad = 0;
    for (int i = 0; i < data.length; i += 3) {
        int b = ((data[i] & 0xFF) << 16) & 0xFFFFFF;
        if (i + 1 < data.length) {
            b |= (data[i+1] & 0xFF) << 8;
        } else {
            pad++;
        }
        if (i + 2 < data.length) {
            b |= (data[i+2] & 0xFF);
        }
    }
    return strBuilder.toString();
}
else {
    pad++;
}

for (int j = 0; j < 4 - pad; j++) {
    int c = (b & 0xFC0000) >> 18;
    buffer.append(tbl[c]);
    b <<= 6;
}

for (int j = 0; j < pad; j++) {
    buffer.append("=");
}

return buffer.toString();

Listing 3.2: Example of Base64 encoding [17]
4 Custom Cryptography Detection Framework

Based on the methodology and scope described in section 1.3, as well as the definitions and characteristics of custom cryptography in chapter 3, a framework is designed, with which the research question from section 1.2 can be investigated and evaluated. First, a high level overview of the framework’s architecture is given, followed by a more detailed presentation of each involved component.

4.1 Architecture Overview

Figure 4.1 shows the different components and phases of the custom cryptography detection framework. As defined in section 1.3, a machine learning algorithm is trained (5) with training apps (1) and labels (4), to be further used (6) to identify unknown custom cryptography methods in an app. To achieve both, an app binary (1 & 7) must first be preprocessed (2) by converting it into an intermediate representation and extracting possible meta data. In the next step, this representation and data is used to extract predefined features which are possible indicators for custom cryptography (3) and can then be used for training (5) or classification (6).

To evaluate various classifier-feature combinations, the framework has to be highly modular. This is achieved by providing interfaces for various machine learning algorithms and feature types. Also, as static analysis of large applications can be very computation-intensive, the feature extraction process (2 & 3) can be performed for multiple apps in parallel and independent from the later classification step (6).

Finally, the results of training, classification and evaluation are outputted in a human readable report (8) for further analysis.

Figure 4.1.: Architectural overview of the custom cryptography detection framework components
4.2 Preprocessing

In the first step of the framework, an Android app is preprocessed. This is necessary, as it is provided as a binary and therefore must be lifted into an appropriate intermediate representation which allows for further static analysis and metadata extraction, like the CFG and data-flow graph (DFG) generation. Also phantom, abstract and native methods, as well as methods from libraries on a predefined blacklist (cryptographic libraries, Android support libraries), are excluded. They are known to contain no custom cryptography and are therefore irrelevant to us. Finally, all collected data is stored in an object and is made accessible for the feature extraction. Additionally, all methods of an application are stored persistently in the file system or a database, as they are needed in the classification phase.

4.3 Feature Extraction

During the feature extraction phase, the feature extractor iterates over all methods of an app, provided by the preprocessing step and extracts feature values for each feature, resulting in a so-called feature vector. Then the vectors are stored persistently in the file system or a database. In case of training and evaluation, a label which indicates whether a feature vector represents a custom cryptography method (CC) or a non custom cryptography method (NC) is added. In case of classification, where the label is unknown and has still to be predicted by the classifier, a random label is chosen for compatibility purposes.

In the following, the features based on the definitions and characteristics of custom cryptography from chapter 3 will be explained in detail.

4.3.1 Method Signature

The following features solely rely on information gathered from a method's signature.

4.3.1.1 IsHasParam

The IsHasParam feature indicates whether a method has parameters or not. Ideally, this can be modeled as a boolean or numeric value, depending on the supported feature types of the used classifier. During our investigations, we discovered that no or a high amount of parameters is a strong indicator for a non custom cryptography method.

4.3.1.2 ParamCount

Beside the IsHasParam, the more precise ParamCount feature also counts the number of parameters of a method ([0..n]). This feature was chosen as we discovered that one to about three parameters are a possible indicator for a custom cryptography method.

4.3.1.3 KnownParamRatio

The KnownParamRatio feature indicates how many of all parameters' types and the return value's type of a method are listed in the so-called custom cryptography variables type list. This list contains the most common data types which are used as parameters and return values in custom cryptography methods:

- byte
- int
• char
• java.io.InputStream
• java.io.OutputStream
• java.lang.String

Equation 4.1 shows a more formal definition of the feature, where \( count_{\text{knownParams}} \) indicates how many of the parameters’ types are in the list above ([0 .. n]) and \( count_{\text{knownReturn}} \) indicates whether the return value’s type is in the list (0 or 1). If the method does not return a variable, the \( \text{KnownParamRatio} \) is 0.

\[
\text{KnownParamRatio}_i = \frac{\text{knownParams}_i + \text{knownReturn}_i}{\text{params}_i + 1}, \quad i \in \text{appmethods}
\] (4.1)

4.3.1.4 Known Parameter-return-value Type Combinations

Table 4.1 shows a set of features, indicating different parameter-return-value type combinations which are found to be likely for custom cryptography methods.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Type scheme combination ((param(_0),...,param(_n)) (-\rightarrow) return value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArrayStringParam</td>
<td>java.lang.String (-\rightarrow) byte([\ ]); byte([\ ]) (-\rightarrow) java.lang.String</td>
</tr>
<tr>
<td>CharArrayCharArrayParam</td>
<td>(char([\ ]), char([\ ])) (-\rightarrow) (java.lang.Void (\lor) char([\ ]) (\lor) java.lang.String)</td>
</tr>
<tr>
<td>StringStringStringParam</td>
<td>(java.lang.String, java.lang.String) (-\rightarrow) java.lang.String</td>
</tr>
<tr>
<td>ByteArrayByteArrayParam</td>
<td>(byte([\ ]), byte([\ ])) (-\rightarrow) (java.lang.Void (\lor) byte([\ ]) (\lor) int([\ ]) (\lor) java.lang.String)</td>
</tr>
<tr>
<td>StringStringParam</td>
<td>java.lang.String (-\rightarrow) java.lang.String</td>
</tr>
</tbody>
</table>

Table 4.1: Features of known parameter-return-value type combinations

Each of the features returns a boolean or numeric value (0 or 1), indicating whether the currently processed method fulfills its pattern. For example, a method with the signature public String foo(String bar) would only fulfill the pattern of the StringStringParam. All other features mentioned above would return false or the numeric value 0.

4.3.1.5 IsStaticMethod

The IsStaticMethod feature indicates whether a method is defined as static or not. During the investigation, we found out that custom cryptography methods are often static and, as a consequence, the feature is a possible indicator for custom cryptography. Ideally, this can be modeled as a boolean or numeric value, depending on the supported feature types of the used classifier.

4.3.1.6 IsHashCodeMethod

The IsHashCodeMethod feature indicates whether the current processed method is the hashCode() method of an object. In Java this kind of method is provided by every object, representing its unique ID as an instance of a class. By default, this is implemented as a hash function, applied to the data stored in the particular object, but can be overwritten for every class manually. As a hashCode() method serves a standardized purpose it is highly unlikely that it is a custom cryptography method.
4.3.1.7 IsConstructor

The IsConstructor feature indicates whether a method is the constructor of a class. During our investigation, we found no evidence that (custom) cryptographic operations were performed in a constructor.

4.3.2 Method Body

In the following, all features were tailored based on investigations of a method’s body.

4.3.2.1 BinOpRatio

Similar to Göbert et al. [16], during their investigation on how to detect standardized cryptography in app binaries, our investigation also revealed that custom cryptography methods manifest a high amount of binary operations. Therefore, the BinOpRatio feature, also shown in equation 4.2, returns the ratio of statements containing binary expressions $stmts_{WithBinOps_i}$ relative to all statements $stmts_i$ in a method $i$, which is a good indicator for a custom cryptography method.

$$BinOpRatio_i = \frac{stmts_{WithBinOps_i}}{stmts_i}, \quad i \in appmethods$$  \hspace{1cm} (4.2)

4.3.2.2 LoopRatio

Also similar to Göbert et al. [16], we found out, that the ratio of statements in loops relative to the rest of a method’s statements is a distinctive indicator for custom cryptography. For example, this can be seen in listing 3.1, where a string gets encrypted with custom cryptography by looping over each character of the string and performing a XOR operation. Equation 4.3 shows how the numeric value of the LoopRatio feature is calculated. $stmts_{InLoop_i}$ is the amount of statements in loops and $stmts_i$ represents the overall amount of statements in a method $i$.

$$LoopRatio_i = \frac{stmts_{InLoop_i}}{stmts_i}, \quad i \in appmethods$$  \hspace{1cm} (4.3)

4.3.2.3 APICallRatio

The APICallRatio feature measures the ratio of API calls from within a method to arbitrary libraries in an app, except a predefined blacklist ($java.lang.*, java.io.*, android.util.*$), to all statements in a method. This feature was chosen, as we observed that custom cryptography methods often do not use any external method calls, except from the blacklisted libraries. Equation 4.4 shows how the APICallRatio is computed, where $stmts_{WithAPICalls_i}$ represents the amount of statements containing an API call and $stmts_i$ represents the overall amount of statements in a method $i$.

$$APICallRatio_i = \frac{stmts_{WithAPICalls_i}}{stmts_i}, \quad i \in appmethods, \quad stmts_{WithAPICalls_i} \in APIs \setminus blacklist$$  \hspace{1cm} (4.4)

4.3.2.4 IsHasThisFieldAccess

The IsHasThisFieldAccess indicates whether a method accesses other member variables or objects of the class it is contained in. The investigations showed that custom cryptography methods often do not access members as they work as an independent component, containing all needed data for cryptographic operations.
4.3.2.5 ArrayAccessRatio

In practice, the usage of custom cryptographic primitives, described in section 3.2, often results in a high amount of array access operations. Beside substitution, where parts of an input value are substituted with fixed values from a predefined matrix or table, almost all custom cryptography algorithms transforming character or byte arrays. Thus, a high ArrayAccessRatio feature value can be an indicator for a possible custom cryptography method. Equation 4.5 shows the calculation of the feature. $stmts_{withArrayAccess}$ represents the amount of statements containing an array access and $stmts_i$ represents the overall amount of statements in a method $i$.

$$ArrayAccessRatio_i = \frac{stmts_{withArrayAccess}}{stmts_i}, \; i \in appmethods$$  (4.5)

4.3.3 Application Context

Beside method signature and body, the following features also take additional context features, like control or data-flow, into account.

4.3.3.1 CallCountAbsolute and CallCountRatio

With the help of the application's call graph, the CallCountAbsolute and the CallCountRatio feature inform about the amount of times, a method is called by other methods. Thereby CallCountAbsolute returns the absolute amount of calls and CallCountRatio returns a ratio of the call count $edgesInto_i$ of a method $i$ relative to the logarithmically smoothened overall count of each method’s call count $edgesInto_n$. As custom cryptography methods are used for encryption of sensitive data and (de-)obfuscation of strings in an application, it is very likely that they get called more frequent then other methods of the application. Because of that, high values in both features, especially in case of the smoothened ratio feature, are a good indicator for detecting custom cryptography. Equation 4.6 shows the computation of the CallCountRatio.

$$CallCountRatio_i = \frac{edgesInto_i}{\ln(\sum_{n=1}^{\text{methods in app}} edgesInto_n)}, \; i \in appmethods$$  (4.6)

4.3.3.2 MaxCallChainLength

As already indicated by the IsHasThisFieldAccess feature, together with the IsStaticMethod feature, custom cryptography methods, in most cases, tend to be an atomic, independent unit which contains all necessary data and logic for cryptographic operations in itself. The MaxCallChainLength feature represents another characteristic which indicates this assumption. For an independent method it is very unlikely to feature a deep call chain.

4.3.3.3 DataFlowFromParamRatio

As to definition 3.2.3, with high likelihood, custom cryptography methods feature a data-flow from the input value (parameter) to the return value. This also includes cases where an substitution is used as the cryptographic approach. The DataFlowFromParamRatio feature indicates from how many parameters of a method, a data-flow to the return value exists. More information about data-flows can be found in section 2.3. Equation 4.7 shows the formula of the ratio, where
\( \text{paramsWithReturnFlow}_i \) corresponds to the amount parameters which feature a data-flow and \( \text{params}_i \) represents the amount of parameters of a method \( i \).

\[
\text{DataFlowFromParamRatio}_i = \frac{\text{paramsWithReturnFlow}_i}{\text{params}_i}, \quad i \in \text{appmethods}
\] (4.7)

### 4.3.3.4 \( \text{IsInSensitiveDataFlow} \)

The \( \text{IsInSensitiveDataFlow} \) feature indicates whether a method lies on the control flow of a sensitive data-flow. As explained in definition 3.2.1, custom cryptography is used with the purpose to cryptographically secure or obfuscate information. In our experiments, we discovered that this especially applies when dealing with sensitive data, like credit card information, passwords, etc. (see section 2.1).

We therefore tailored a proper set which contains the common found sources and sinks for sensitive data being transformed by custom cryptography methods. For more information about data-flows, see section 2.3 and section 2.3.1.2.

### 4.4 Training, Evaluation and Classification

As already mentioned in the introduction, the custom cryptography detection framework provides an interface for different ML algorithms. Hence, the following steps can be performed with any ML algorithm that supports the above feature value types and is compatible to the interface.

#### 4.4.1 Training

To train a classifier, the feature vectors, computed in section 4.3, will be read in from the file system or database by the learner. Further, with the help of the corresponding labels, a classifier can be trained and saved for further classification tasks.

#### 4.4.2 Classification

To classify unlabeled methods of an app, a trained classifier is required. With this, the feature vectors from the feature extraction phase will be read in and classified towards the two labels \( \text{CC} \) and \( \text{NC} \). Finally, a classification report with the results will be generated. Additionally, the top \( n \) custom cryptography methods, sorted by the prediction probabilities of the classifier, can be saved for further analysis.

#### 4.4.3 Evaluation

The evaluation of the framework’s classifier is a combined process of training and classification with a labeled method dataset. First a classifier is trained with a part of the labeled dataset. Then, a classification is run with the rest of the labeled dataset. Finally, the labels, predicted by the classifier, can be compared with the given labels by the dataset. Now, various evaluation metrics can be applied. For more information about ML evaluation, see section 2.4.3.
5 Implementation

In the following, important aspects regarding the implementation of the custom cryptography detection framework from section 4 are explained in detail.

5.1 Overview

Figure 5.1 shows the implementation of the custom cryptography detection framework which is composed of two independent logical components. The FeatureExtractionLauncher (I.) is responsible for preprocessing and extracting the app method’s features vectors, whereas the CustomCrypto Classifier (II.) is solely responsible for the learning and classification task. Additionally, the FeatureExtractionLauncher can be parallelized, which means that multiple apps can be processed in parallel. As the feature extraction is highly computation-intensive, this leads to a performance gain with respect to the runtime, compared to a serial processing. In contrast to that, the learning and classification phase itself is not mend to be parallelized, as it appeared to be performant enough in our experiments and with the chosen classification algorithm.

Figure 5.1.: Implementation overview of the custom cryptography detection framework
The application logic of the framework is written in Java and uses several third-party libraries. Initially, the Soot framework is used to lift an app's binary into a suitable intermediate representation for feature extraction. Further, Soot provides several capabilities for static analysis, like a call graph generation and a loop detection which are used in different features. The FlowDroid framework which also bases on Soot, provides the data-flow detection capability for the DataFlowFromParamRatio and the IsInSensitiveDataFlow feature.

The Weka framework (see section 2.4.4), which is used in the CustomCryptoClassifier provides interfaces for various ML algorithms and supports easy evaluation of trained classifiers.

5.2 Preprocessing & Feature Extraction

The preprocessing and feature extraction process from figure 5.1 can be viewed in detail in figure 5.2. After providing an APK the app's binary (1) is processed by the two engines inside the AppData class (2). First, the SootEngine transforms the app's bytecode into its intermediate representation. Then, the FlowDroidEngine is initialized. However, its actual data-flow tracking is not started until requested, to avoid computational overhead when data-flow features are disabled, e.g. for evaluation purposes. Finally, the FeatureExtractor iterates over all methods of the app (3) (except abstract, phantom, native and blacklisted ones), provided by the SootEngine and computes their features vectors (5). The vectors are a list of various feature values, defined by the so-called FeatureSet (4). The FeatureSet is an array of feature objects whereas each object calculates a different feature type from the list defined in section 4.3.

![Figure 5.2: Components involved in the feature extraction process of the custom cryptography detection framework](image)

After processing all methods of an app, the feature vectors are stored in the local file system for further classification by the Weka framework. If the FeatureExtractor is run for training or evaluation (TRAIN EVAL mode) instead for simple classification (CLASSIFICATION mode), ground truth labels (6) for each method are required and will also be saved in the feature vectors.

5.2.1 Feature Extraction Example: CallCountRatio

The features from the mentioned FeatureSet in section are all subclasses of the abstract class Feature. Beside others, the abstract class defines the getFeature(AppData appData, Body b) method, which is used to generate a feature value.
for a method. Listing 5.1 demonstrates an implementation of this abstract method for the CallCountRatio feature. It returns the ratio of how many times a method is called by other methods relative to the amount of times all other methods in an app are called.

```java
@override
public DoubleFeatureValue getFeature(AppData appData, Body b) {
    int callMe = count(Scene.v().getCallGraph().edgesInto(b.getMethod()));
    if (callAll == -1) {
        for (SootClass clazzC : Scene.v().getApplicationClasses()) {
            for (SootMethod method : clazzC.getMethods()) {
                callAll += count(Scene.v().getCallGraph().edgesInto(method));
            }
        }
        double dCallAll = Math.log(callAll);
        double d = FeatureUtils.createRatioFeaturevalue(callMe, dCallAll);
        return BasicFeatureValues.getFeature(d);
    }
}
```

**Listing 5.1:** Feature generation logic of the CallCountRatio feature

In line 3, the call graph provided by the Soot framework (Scene.v()) is used to count the amount of times the current method is called by other methods in the processed app, named edgesInto. Then, in line 4, it is checked whether the total amount of calls of all methods was already computed and saved in the static member variable callAll by a prior instance. If it is the first time the method is called, indicated by the value $-1$ of callAll, it will be computed (line 5-9). After that, callAll is logarithmically smoothed (line 11). Finally, the feature ratio is calculated (line 12) and returned to the caller (line 13). For further information about the CallCountRatio feature, see section 4.3.3.1.

### 5.3 Learning & Classification

After successful feature extraction, the classification phase is started. The CustomCryptoClassifier therefore acts as the management component, containing the Weka classifier, as well as further evaluation and reporting tools. First, the feature vectors, written by the FeatureExtractor into the file system, are read in and converted back into Feature objects. Then, they are converted into Weka’s native feature type.

#### 5.3.1 Learning

If run in TRAIN_EVAL mode, a classifier of one of the supported Weka ML algorithms is trained and can be saved for further classification tasks by the framework. The following common algorithms provided by Weka [49] were tested and also applicable to our feature types:

- **naïve bayes**: A simple probabilistic algorithm. For more information see section 2.4.1 and John et al. [22].
- **simple logistic**: Algorithm based on linear regression models [27].
- **sequential minimal optimization (SMO)**: An algorithm to train SVM. For further information see section 2.4.1 and [35].
- **locally weighted learning (LWL)**: A local regression algorithm. See also Atkeson et al. [4].
- **random forest**: A classification and regression algorithm, based on decision trees. For more information see [6].
- **multilayer perceptron**: An artificial neuronal network. See [50].
5.3.2 Classification

In CLASSIFICATION mode, an already trained classifier is required. It is loaded by the Weka framework and used to classify provided unlabeled method feature vectors of an app. The classification result are saved in a report, consisting of the method's signature, the predicted label (CC/NC) and prediction probability. Also the top n custom cryptography methods of an app are saved for further analysis.

5.4 Evaluation

To compare the performance of classifiers and features for a sound evaluation, the TRAIN_EVAL mode provides a 10-fold-cross validation which can be performed in parallel with different classifier-feature combinations, mentioned above. For assessment of the evaluation results, a detailed report of all classifier-feature combinations is created.
6 Evaluation

The evaluation of classifiers and features is an important step in the process of developing a performant ML system with respect to detection rate, runtime and system load. In the following, we evaluate the custom cryptography detection framework, conceptually described in section 4 and implemented in section 5. First, we mention the chosen test setup and then carry out a 10-fold cross-validation, further described in section 2.4.3, followed by two case studies which show the performance of our system in a real world scenario.

6.1 Test Set-up

The evaluation and case studies are performed on a dedicated machine with the following specifications:

- Intel Xeon E5-4650 64 Cores @2.7GHz
- 1TB RAM
- Ubuntu 16.04.4 LTS x64
- 3.5.0-45-generic linux kernel
- Java SE Runtime Environment (build 1.8.0_112-b15)

It represents a common hardware setup used by researches and analysts in the code analysis field and can be seen as realistic setup for real world scenarios.

6.2 Classifier and Feature Evaluation

To choose the most accurate and performant classifier-feature combination, a classifier-feature trade-off represented as a 10-fold cross-validation is performed in this section.

6.2.1 Evaluation Dataset

As explained in section 2.4.3, a labeled dataset is required for evaluation. It must contain apps which implement custom cryptography (CC) algorithms, as well as apps that implement non custom cryptography (NC) algorithms on a method level. To achieve this, we used our results from chapter 3, where we already tailored a list with benign and malware apps, containing custom cryptography methods (see appendix section A.2 and section A.1). For the non custom cryptography method samples, we decided to use methods from apps which are highly unlikely to contain custom cryptography (secure messengers, apps provided by Google). In detail, the non custom cryptography methods were chosen randomly, to get a wide range of different method types. Additionally, this also reduces the probability that a possible custom cryptography method is in the NC set. A list with details about the non custom cryptography apps set can be found in appendix section A.3.
6.2.2 Bias of Unbalanced Datasets

The tailored dataset, mentioned above, contains an unequal amount of CC and NC labeled methods, showed in table 6.1 which leads to an unbalanced dataset. This is caused by the high effort which must be taken, to find and verify custom cryptography methods. Normally, this fact would have a high impact on the training and evaluation of the classifier and features, called data bias. Less relevant characteristics (features) from methods in the NC set would be rated more important, than highly relevant characteristics in the CC set, just because they appear statistically more often.

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>194</td>
<td>11185</td>
</tr>
</tbody>
</table>

Table 6.1.: Total instances per class in the evaluation dataset

To avoid this, the used Weka ML framework provides weightings. With this concept, the feature vectors of the methods are weighted accordingly to the overall amount of vectors in each subset (CC and NC), which results in a balanced dataset.

6.2.3 10-fold cross-validation

When evaluating a ML system, different evaluation approaches can be used. The proper approach is normally indicated by the size of the evaluation dataset. More information about this can be viewed in section 2.4.3. In our case, the dataset indicates the usage of a 10-fold cross-validation.

6.2.3.1 Ranking of the features

To find a classifier-feature trade-off, which results in high detection rate, low runtime and system load, we first rank the available features from section 4.3 according to their information gain. The information gain basically indicates the importance of a feature as a characteristic for custom cryptography. In detail, a feature type with a high information gain, is a more reliable indicator for custom cryptography than a type with a low information gain.

For the calculation, we use our dataset and Weka's built in Explorer Tool (InfoGainAttributeEval, Ranker). As the tool does not support weighted attributes for information gain calculations, we have to remove the weighting from the dataset and use another technique for balancing, called Resampling. Resampling simply clones or removes random feature values from each class (NC or CC) until both contain an equal amount of entries. After the successful calculation, the made modifications the dataset will be reverted.
The results of the ranking can be viewed in table 6.2. As to the information gain, the amount of known parameters, relative to all parameters of a method (KnownParamRatio), are one of the strongest indicators for custom cryptography methods. This can also be seen in figure 6.1, where the ratio of known parameters is more likely to be high for custom cryptography (blue) than for non custom cryptography methods (red). Our investigations support this assumption, but also showed that only relying on this feature leads to many false positives and therefore has to be combined with other evidence. For more information, see section 4.3.

The characteristics, binary operation ratio (BinOpRatio), loop ratio (LoopRatio) and data-flow from parameter to return value ratio (DataFlowFromParamRatio) are also significant for custom cryptography, having a high information gain from ~0.42 - 0.34. They were initially discovered by Gröbert et al. [16] during their investigation on detecting standard cryptography methods.
cryptographic primitives in app binaries, However, our investigations revealed that not all custom cryptography methods have a data-flow from parameters to return value, as they sometimes modify the parameters by reference and do not return anything.

Less significant as intended, the sensitive data-flow (IsInSensitiveDataFlow) only has an information gain value of $-0.047$. This is not backed by our investigations, as it represents a key aspect of custom cryptography. As a possible explanation, this low value could be caused by sensitive data-flows, which are not found by our data-flow engine. Spot tests reveal that in many cases the engine is unable to detect complex flows or flows which involve inter-component communication. Also some flows are not found because of missing sources and sinks, which were not properly defined by us or cannot be defined because of performance issues. For example, some malware tries to steal credentials by mimicking valid login interfaces. Trying to tag the fake login’s text input fields as sources, to detect its data-flow, would result in tagging all text input fields in a whole app as sources. The data-flow engine would now have to follow many more flows, which would result in an extensive analysis runtime.

The weakest information gain is provided by the IsHashCodeMethod. This is also plausible, as there are only 71 hashCode methods in the whole dataset.

Overall, beside the named outliers and their causes, the information gain results appear to be sound. Nevertheless, the following validation has to prove that this holds for all classifiers. For more information about feature metrics, see section 2.4.2.

### 6.2.3.2 Validation Results

With the ranked features, we then perform multiple 10-fold cross-validations for different classifiers, adding one more feature from the sorted feature list at each iteration. The performance results can be viewed in figure 6.2 and figure 6.3.

![Figure 6.2: Accuracy of different classifier algorithms with increasing amount of features](image-url)
As the accuracy does not represent a reliable metric for evaluating a classifier, the following statements will refer to the F1 score in figure 6.3 and table 6.3. As can be seen, all classifiers finally reach a F1 score of \( \geq 0.94 \), whereas the simple logistic algorithms appears to perform the best for our ML problem with an F1 score from about 0.96.

Using only the KnownParamRatio feature, already a F1 score above 0.90 is reached by all, except the smo classifier. On the one hand, this matches with our investigations and the information gain from section 6.2.3.1, as most of the custom cryptography methods use a fixed set of parameter types. On the other hand, using only this feature also result in a high amount of false positives, as many other non custom cryptography methods also feature this parameter types. A possible explanation for this observation could also be that the non custom cryptography methods from the dataset do not represent the full spectrum of methods available in the wild.

Further, the results also show that the first four features (KnownParamRatio, CallCountRatio, BinOpRatio, LoopRatio) have the most significance for identifying custom cryptography. This corresponds to the information gain from table 6.2 and also fits to the observations of Gröbert et al. [16]. Nevertheless, the weak performance of the IsInSensitiveDataFlow feature was already explained in section 6.2.3.1 and could be the result of an insufficient data-flow detection.

The weak performance of the smo classifier with the first features may be the result of the inability to distinctly separate custom cryptography from non custom cryptography with the used algorithm. Future work should investigate this in detail. For more information about the used algorithm, see section 5.3.1.

All other features, not mentioned above, are only able to enhance the F1 score by 0.01 to 0.02, which either means that they are not very characteristic for custom cryptography methods or that our dataset’s custom cryptography samples, as mentioned earlier, do not reflect the whole spectrum of characteristics. As to our investigations, both assumptions are possible.
As mentioned above, simple logistic is the most suitable machine learning algorithm for our classification task. Table 6.4 shows the confusion matrix of the 10-fold cross-validation of this classifier with all features enabled. A confusion matrix is a table which visualizes the performance results of a classifier. See section 2.4.3 for more information.

From a first look, the false positive rate seems to be very low as only 0.04% of non custom cryptography methods are categorized as custom cryptography. But taking into consideration that the amount of non custom cryptography methods in an app is normally much higher than the amount of custom cryptography methods, a false positive rate of 0.04% could be hundreds or thousand methods in practice.

For the following case studies, we use the simple logistic classifier.

### 6.3 Case Studies

To evaluate the performance of the custom cryptography detection framework in real-world scenarios, two further case studies are performed. Case Study "Google Play Store Applications" examines the performance in typical, daily-use applications. Study "Android Obfuscation Techniques" focuses on the framework classifier's detection rate, when applying different obfuscation approaches on apps.

#### 6.3.1 Google Play Store Applications

In this experiment, we evaluate the custom cryptography detection framework with real world apps from the Google Play Store. The apps are chosen from different categories, like Communication, Books & References, Lifestyle and more. As the Play Store is the biggest distribution platform for the Android OS and the apps were picked from various categories, the experiment results should be sound. More information about the apps' details can be found in appendix B.

As we do not know whether there actually is or is not custom cryptography in the examined apps, we can not provide further metrics, like precision, recall and F1 score. Instead, we acquire the top \( n \) methods, labeled by the framework as custom cryptography and sorted descending by the classification confidence. We then manually check for false or true positive in the results. This allows a subjective impression about the systems’ performance. For \( n \) we chose 10, as we think that this an appropriate amount of methods which in reality can be investigated by an analyst.

Table 6.5 shows the classification results of the framework with the simple logistic classifier and all features enabled. Beside the apps' names, also the processed methods, the methods classified as custom cryptography and the manually verified custom cryptography methods in the top 10 results are shown. Because different kinds of methods are filtered out during the preprocessing (see section 4.2), the methods processed can differ from the overall count of methods in the app. Also grayed out apps could not be processed in a predefined timeout (16h).
### Table 6.5: Results of the Play Store app classification

<table>
<thead>
<tr>
<th>Application name</th>
<th>Processed methods</th>
<th>Classified as CC</th>
<th>CC in top 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Easy LaCrypte</td>
<td>93</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>BMW Augmented</td>
<td>2770</td>
<td>133</td>
<td>1</td>
</tr>
<tr>
<td>Confide</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES File Explorer File Manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FreeMessage - free Messenger</td>
<td>28034</td>
<td>1438</td>
<td>0</td>
</tr>
<tr>
<td>KiK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINE: Gratis-Anrufe</td>
<td>24427</td>
<td>1424</td>
<td>0</td>
</tr>
<tr>
<td>Messenger Lite: Kostenlos Anrufe</td>
<td>14329</td>
<td>739</td>
<td>1</td>
</tr>
<tr>
<td>QQ</td>
<td>21242</td>
<td>1793</td>
<td>2</td>
</tr>
<tr>
<td>Skype - free IM &amp; video calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smart Plug</td>
<td>18200</td>
<td>1022</td>
<td>0</td>
</tr>
<tr>
<td>Snapchat</td>
<td>1975</td>
<td>137</td>
<td>0</td>
</tr>
<tr>
<td>TUCaN TU Darmstadt</td>
<td>6795</td>
<td>409</td>
<td>2</td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The performance does not significantly change, though the runtime and system load is drastically reduced. As the data-flow features do not feature much information gain and data-flow tracking is normally very time consuming, this outcome is expected. For more information on timings and system load, see section 6.4. Table 6.6 shows the results of the remaining apps, classified without data-flow features.

### Table 6.6: Results of the classification of the timeouted Play Store apps without data-flow features enabled

<table>
<thead>
<tr>
<th>Name</th>
<th>Overall methods</th>
<th>Classified as CC</th>
<th>CC in TOP10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confide</td>
<td>36856</td>
<td>3127</td>
<td>0</td>
</tr>
<tr>
<td>ES File Explorer File Manager</td>
<td>37772</td>
<td>2947</td>
<td>0</td>
</tr>
<tr>
<td>KiK</td>
<td>30793</td>
<td>1477</td>
<td>5</td>
</tr>
<tr>
<td>Skype - free IM &amp; video calls</td>
<td>37334</td>
<td>2219</td>
<td>1</td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td>39370</td>
<td>1973</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 6.3.1.1 True Positives

In half of all apps, custom cryptography methods were found and verified. In the following, we examine two findings in detail:

**TUCaN TU Darmstadt**

In the mobile client of the campus management system of TU Darmstadt, called TUCaN, we found two custom cryptography methods. One of them is partly responsible for encrypting the outgoing network traffic. In listing 6.1, one can see snippet of the method’s logic. A string is taken as an input. Then its characters are shifted and substituted in various ways (line 5 - 8 and line 13 - 16) in a loop until a certain condition is reached and the loop is terminated (line 19). Finally, the transformed string is returned (line 22). Nevertheless, it has to be mentioned that the custom encrypted traffic is additionally encrypted with TLS for transmission. It stands to reason that this additional layer of standard cryptography was added in advance and the custom cryptography algorithm is left for legacy purposes.

```java
1  public static String b(String $param0) {
2      // ...
3      do{
4          // binary operations, array access (substitution)
```
$arrchar_1[0] = ($arrchar[0] & 252) >> 2;
$arrchar_1[1] = (char)(((char)[0] & 3) << 4) + (($arrchar[1] & 240) >> 4));

// ...

for ($int_1 = 0; $int_1 < 4; ++$int_1) {
    //array access (substitution)
    StringBuilder stringBuilder = new StringBuilder().append($String)
        .append("ipkIBozSl8CVH3J7PvQfWYu" +
            "emx0C4rn5bgDqMaKXTy60h9wt-NZdjFAE12Gs."
        .charAt($arrchar_1[$int_1]));

    // ...
    break;
    // ...

} while(true);

return $String;

Listing 6.1: Custom cryptography method found in the TUCaN TU Darmstadt app

The above example features multiple characteristics of custom cryptography, from which we mention the most important ones:

- static method
- string parameter and string return type
- high known parameter ratio
- high loop ratio
- high binary operations ratio
- high array access ratio
- data-flow from parameter to return value
- high call count ratio

B Easy LaCrypte

B Easy LaCrypte claims to be an app for secure encryption of data on the SD card of a smartphone. In reality, B Easy LaCrypte uses custom cryptography algorithms for both, encryption/decryption of the password and the data itself. In the following, we investigate the custom cryptography method which encrypts a "file password" with a master password in listing 6.2.
private void cryptBlendPassword(byte[] $param0) {
    int $int = 0;
    int $int_1 = 0;
    while ($int_1 < $param0.length) {
        $param0[$int_1] = (byte)($param0[$int_1] + this.password[$int]);
        ++$int;
        $int %= this.password.length;
        ++$int_1;
    }
    return;
}

Listing 6.2: Custom cryptography method found in the B Easy LaCrypte app

In the listing above, the file password’s byte array is taken as an input. Each entry of its array is then summed up with each entry of a predefined master password (line 5). If the master password is shorter than the file’s password, it is repeated from the beginning (line 7).

The above example features the following custom cryptography characteristics:

- high known parameter ratio
- high loop ratio
- high binary operations ratio
- high array access ratio

6.3.1.2 False Positives

On the downside, as shown in table 6.5 and 6.6, also many false positives are present in the top 10. Following, we describe the most common ones.

Mathematical Operations

During the verification of the top 10 findings, it turned out that often mathematical operations are detected as custom cryptography. Listing 6.3 shows a typical false positive, caused by a high loop, binary operations and array access ratio. Additionally, also a high known parameter ratio is present. In fact, this method does perform an image transformation.

private void expand2(byte[] arrby, byte[] arrby2) {
    int n2 = arrby2.length;
    for (int i2 = 1; i2 < n2; i2 += 4) {
        int n3 = arrby[(i2 >> 2) + 1] & 255;
        switch (n2 - i2) {
            default: {
                arrby2[i2 + 3] = (byte)(n3 & 3);
            }
            case 3: {
                arrby2[i2 + 2] = (byte)(n3 >> 2 & 3);
            }
            case 2: {
                arrby2[i2 + 1] = (byte)(n3 >> 4 & 3);
            }
        }
    }
}
Standard Cryptography Algorithms
As mentioned in section 4.2, our framework excludes methods from known cryptographic libraries, to fulfill definition 3.2.4 of custom cryptography. The top 10 findings in the app "Confide" show that this solution leads to a high rate of false positives, if one of these libraries is not listed on our predefined blacklist. In the case of "Confide" the crypto library org.spongycastle is not filtered out and results in a high false positive rate in the top 10, as 9 methods were false positives from org.spongycastle.

Known Parameter Ratio Based Outliers
Another observation reveals that many false positives are solely rated as custom cryptography, because they feature a high known parameter ratio and a high call count ratio. Listing 6.4 shows one of these methods. Here, all parameters are string type which is a known parameter. Also, as this is a logging method which simply prints strings to the Android Log, it is called very often. Normally, methods from the Android libraries are filtered out during feature extraction, but in this case this was not possible, as Log is encapsulated inside another method.

```
public final void c(String string, String string2) {
    Log.i((String)string, (String)string2);
}
```

Listing 6.4: False positive method from the Facebook Messenger Lite app (see appendix B)

To avoid this kind of outliers, the API count ratio feature was introduced, but was rated as relatively unimportant during the learning phase of the classifier. Adding more of such methods to the train and test dataset could maybe eliminate this effect in the future.

Ambiguity of Encoding Schemes
As to the results, a high number of false positives are also caused by a misclassification of encoding schemes, mainly Base64, as custom cryptography. In all cases, the manual verification of these false positives proved benign, instead of custom cryptography usage. This confirms the ambiguous usage problem of encoding schemes, mentioned in section 3.3.2. With the current system's feature, there is no possible solution to avoid these kind of false positives at all.

6.3.2 Android Obfuscation Techniques
This case study investigates whether the performance of the trained classifier is influenced by code obfuscation techniques. These techniques are generally used and often deployed automatically to prevent or slow down reverse engineering attempts on apps.

For Android, there are various products on the market, which provide easy to use obfuscation. The techniques used for that vary from simple class and variable renaming to complex virtual machines, native libraries, Java reflection and much more.

In the following, we use ProGuard [44] for our experiments, as it is free to use and comes preinstalled in Android Studio. Once enabled in an app's build config, self-programmed classes and variables will be renamed with random values.
Additionally, further rules can be deployed, to obfuscate third-party libraries. Beside obfuscation, ProGuard also shrinks code and resources which can be seen in table 6.7. As this functionality is not relevant for the case study and also does not influence the results, no further information is given, but can be examined in the Android Studio documentation [8].

To evaluate ProGuard, we use two open source applications, available on the F-Droid store [30] and GitHub, called OpenPass and Pijaret. For further information on version and developer infos, see appendix C.

Both apps are build with and without ProGuard enabled. Then they are classified by the trained classifier from section 6.2. The results can be viewed in table 6.7 and show that in both apps the same amount of custom cryptography methods were found. In detail, even the ranking of the methods was identical in the obfuscated and unobfuscated versions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Processed methods</th>
<th>Classified as CC</th>
<th>CC in TOP10</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenPass</td>
<td>108</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>OpenPass - obfuscated</td>
<td>86</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>Pijaret</td>
<td>39</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Pijaret - obfuscated</td>
<td>22</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.7.: Results of the classification of the obfuscation dataset with all features enabled

These results can be explained by examining the features with which the classifier is trained. As they do not take any class or variable names into account, ProGuard’s class and variable name randomization obfuscation, does not change any feature value at all.

Nevertheless, we cannot rule out the possibility that other obfuscation tools, such as DexGuard [43], which for example alters the program logic or promon [3], which uses native libraries, weaken the classifiers performance. At the time of writing, there was no version available for testing. Further work should evaluate these tools.

### 6.4 Timings and System Load Evaluation

As the evaluations and case studies are executed on a shared machine, no exact statements regarding the time and computational performance can be made. However, the overall runtime and memory consumption of the 10-fold-cross validations and the experiments give a roughly idea about the order of magnitude in which our system is working. Figure 6.4 shows the collected informations about the overall runtime. Be aware that the y-axis has a logarithmic scale.
Figure 6.4.: Runtime of the 10-fold cross-validation and the case studies (feature extraction and classification/training) on the test system with 8 apps processed in parallel.

Additionally, a feature extraction process with 8 parallel extractions and enabled data-flow features take 10 times more RAM (~300GB) than an execution with disabled data-flow features (~30GB).

The details mentioned above show that the runtime of our custom cryptography detection framework strongly depends on whether data-flow features are acquired or not. Overall, a run with data-flow features enabled seems not to be performable in a suitable amount of time, keeping in mind that the framework operates on a high performance system. Also the runtime of a run without data-flow features enabled seems not to be practical for an analyst’s personal computer. For a large app store, which has to deal with a large amount of newly handed in apps by developers, could not used our system in daily routine. Despite a large computational infrastructure, apps or app updates would be published with a noticeable delay to the public.
7 Related Work

To the authors knowledge, the phenomenon of "custom cryptography" and its automatic detection, addressed in this work, was not investigated by other researchers until this point. Nevertheless, related work dealing with the detection of code patterns, especially cryptographic algorithms, will be mention in the following.

In 2011, Gröbert et al. [16] published their work regarding the detection of cryptographic primitives in program binaries. Their approach uses dynamic binary analysis based on the Intel PIN framework and self-defined heuristics based on characteristics of a set of investigated cryptographic algorithms. The final system performs fairly well, but its detection is limited to the investigated set of algorithms.

In contrast to Gröbert et al., Lestringant et al.'s [29] approach for detecting a predefined set of symmetric encryption algorithms bases on a static binary analysis. In detail, they use a concept called data-flow graph (DFG) isomorphism to build a distinct DFG signature for known algorithms which then can be matched with unknown samples. Because the DFG is normalized using code rewrite mechanisms, it is robust towards different compilers and options.

In their work, Matenaar et al. [33] evaluate different approaches for cryptographic algorithm detection available up to that time and combine them into the so-called Crypto Intelligence System (CIS). The CIS is able to detect various symmetric and asymmetric encryption, as well as hash algorithms in malware binaries.

Xu et al. [53] try to solve the problem of automatic cryptographic primitive detection by introducing bit-precise symbolic loop mapping. Possible cryptographic algorithms in binaries are found by executing them symbolically in a loop, followed by fuzzing, to match them with boolean formulas of known reference algorithms (TEA, AES, RC4, MD5, and RSA). Their approach is able to detect the algorithms even in the presence of various obfuscation techniques.

In contrast to the above metric based approaches, Hill et al. [18] introduced a deep learning based detection approach for cryptographic primitives in 2017. Using a dynamic convolutional neural network (DCNN), they are able to detect cryptographic algorithms, based on control flow diagnostic output from a dynamic trace of a program execution. This so-called Crypto Knight library archives an accuracy of 91%, while being able to detect various implementations of AES, RC4, Blowfish, MD5 and RSA.

Beside detection of cryptographic code patterns, Yamaguchi et al. present different approaches which allow to identify vulnerabilities, like buffer overflows or memory disclosures, by lifting source code into an abstract representation. To achieve this, the first approach from 2012 [54] uses abstract syntax trees and enables automatic detection of known vulnerabilities in source code. The second approach from 2014 [55] introduces a new combined representation, called code property graph, which merges the information from abstract syntax trees, control flow graphs and program dependence graphs into one data structure. With this, they were able to tailor templates for common vulnerabilities, resulting in the finding of 18 new Linux kernel vulnerabilities.
8 Limitations

In the following, we discuss the limitations of our approach for detecting custom cryptography which mainly base on the evaluation results from chapter 6.

As to definition 3.2.1, one main aspect which makes custom cryptography algorithms distinguishable from other similar algorithms is the purpose to obfuscate or securely transmit and store information. In our system, we try to model this characteristic in form of sensitive flows, described in detail in section 4.3.3.4. In short, we assume that algorithms which transform sensitive data on sensitive flows, like credit card numbers or passwords, are highly likely to be custom cryptography.

However, the evaluation results reveal that this assumption does not perform very well, indicated by a low information gain in section 6.2.3.1 and a high false positive rate in section 6.3.1.2. On the one hand, this can be attributed to a low detection rate of sensitive flows by the used data-flow framework, caused by complex data-flows or missing predefined sources and sinks, already discussed in section 6.2.3.1. On the other hand, the weak performance of this assumption could also be caused by the fact that custom cryptography algorithms are not the only algorithms which transform sensitive data. For example, various decoding and encoding schemes are falsely classified as custom cryptography during the experiments, as they shrink or format sensitive data before it is send over the network.

In result of both explanations, our system cannot distinguish very well whether a method is used as custom cryptography or for any other purpose, especially when dealing with dual use algorithms like encoding schemes.

Further, the case studies shows that our trained classifier cannot distinguish between custom cryptography and standard cryptographic primitives. In section 6.3.1.2, in the Confide messenger, 9 out of the TOP 10 methods classified as custom cryptography, are actually cryptographic primitives. Our chosen approach, to avoid this problem by removing known cryptographic libraries from feature extraction and classification, does not work in this example, as the used crypto library is not listed on the library blacklist. In result, there is an mandatory need for a complete cryptographic library blacklist, which seems not to be feasible in real world scenarios. Also, we must take into account that with the blacklist approach, a possible attacker could avoid detection by putting her custom cryptography methods into one of the blacklisted package. For the sake of completeness, the limitations introduced by our chosen scope in section 1.3.2, must also be mentioned in the following.

As of the nature of supervised machine learning, the performance of our system depends highly on the diversity of custom cryptography and non custom cryptography samples inside the training dataset. In practice, that means that our classifier may not be able to detect custom cryptography methods which do not feature any characteristics of custom cryptography methods from our dataset at all. Further work should keep this under observation.

Also, as we solely rely on static code analysis, our framework's analyses are very computation-intensive. This means that at the current state of development, the feature extraction and classification of an common app needs a high performance hardware and does not allow on-demand results as of the long runtime.

Caused by the chosen underlying static analysis framework, our detection of custom cryptography is currently limited to dex-bytecode. Native libraries and cross-platform frameworks are thereby excluded. Furthermore, we are able to detect custom cryptography on a method level only. Also we cannot explicitly detect interprocedural custom cryptography algorithms as we consider every method as independent.

Finally, it has to be stated that other obfuscation techniques than the mentioned ones in section 6.3.2 which alter a program's syntax could possibly lead to a drastic reduce in the detection rate. This especially includes Java reflection based approaches.
9 Conclusion

In this work, we investigated the phenomenon of custom cryptography and its detection in Android binaries, driven by the two research questions raised in section 1.2. Therefore, we successfully developed definitions and characteristics, describing custom cryptography algorithms and its implementations in chapter 3. In detail, it turned out that one of the main characteristics of custom cryptography is the purpose of using it to cryptographically secure or obfuscate information (see definition 3.2.4). Instead of standard cryptographic primitives, like AES or RSA, custom cryptography algorithms use self-implemented arithmetic-logic transformations (see definition 3.2.3), relying on substitution, logical operations and also further mathematical operations.

To automatically detect custom cryptography in Android apps on method level, we developed a ML framework in chapter 4 and 5 which relies on features derived from the definitions and characteristics above. To extract the features from the app binaries, the framework uses two static analysis tools. The first one, called Soot is used to lift the binaries into an intermediate representation and to extract method signature, as well as method body features, whereas the second one, called FlowDroid, is used for data-flow related features.

The evaluation of our framework in chapter 6 was performed with a manually tailored and labeled dataset (see appendix section A), containing various custom cryptography and non custom cryptography method samples inside apps. As of the relatively small size of the dataset, we performed a 10-fold cross-validation which resulted in a good F1 score of about 0.96 with a SimpleLogistic classifier. However, the following case studies, which were chosen to evaluate the framework's performance in real world scenarios, revealed some limitations. First, because of the highly uneven average ratio of custom cryptography methods and non custom cryptography methods in an app, the relatively small false positive rate of our classifier from about 0.04%, produces a huge amount of false positives in large apps. We fixed this circumstance by sorting the custom cryptography results accordingly to the classifier's confidence rating which allows to output the TOP n methods in an app, classified as custom cryptography. The results of this can be viewed in section 6.3. Nevertheless, still some problems remain. For example, our chosen approach to use data-flow features to detect the intended purpose of a method, explained in 4.3.3.4, does not work out very well. On the one hand, this can be attributed to problems with the data-flow framework, which results in missing data-flows, but can also be attributed to the ambiguity usage of encoding schemes, explained in section 3.3.2.

In conclusion, our work showed that custom cryptography algorithms share some common characteristics which make them distinguishable from other algorithms. Thereby, the introduced machine learning approach represents some fundamental techniques for automatic detection based on static analysis, on which further work can build on.
10 Future Work

Future work on the automatic detection of custom cryptography should address the limitations mentioned in chapter 8 in different ways.

First, when still relying on the machine learning approach for detection, a more diverse dataset for training should be tailored. Especially, the non custom cryptography method samples should be hand-picked to avoid a data bias during the learning process, caused by custom cryptography samples falsely labeled as non custom cryptography.

Second, to improve the detection of the purpose of a method, the sensitive flow engine has to be improved. This can be achieved by either identifying and adding new sources and sinks, which tend to be used in custom cryptography related sensitive flows, or by enhancing the flow detection itself, e.g. by improving inter-component communication tracking. However, the above mentioned improvements do not pose a solution for some of the ambiguously used methods, like encoding schemes mentioned in section 6.3.1.2. Future work addressing this, should search for other possible features to solve this ambiguity.

Third, the usage of the blacklist with known cryptographic libraries which are excluded from the classification process, should be avoided. It represents more of a work around than a solution to the problem that our classifier could not distinguish very well between cryptographic primitives and custom cryptography. As already mentioned in the limitations, a possible attacker could easily exploit this approach by renaming its custom cryptography package to one on the blacklist. Future work could use library detection as a feature which indicates that a method from a known library is present. Although our investigations on Android apps found out that common custom cryptography tend to be on method level only, further detection approaches could consider statement level classification as an option, to detect more sophisticated custom cryptography algorithms which reach over several methods.

Considering techniques from the field of dynamic analysis, as possible features, could also enhance the framework's detection rate and should therefore be investigated in detail.

Beside the usage as an automatic scanner for app stores, the initially intention was also that analysts could easily identify custom cryptography in malware during their daily work with the help of the framework. To meet this use case, future work has to develop a graphical user interface which visualizes the classification results.

Finally, also the overall runtime and system load has to be investigated in detail. The current approach could not be used for ad-hoc analyses, as the mean runtime is about 8 to 9 hours. Additionally, the computational resources needed are not feasible for standard computer. Therefore, future work should investigate ways to lower runtime and system load.
**Bibliography**


A Android Application Dataset

A.1 Malware Custom Crypto Samples

<table>
<thead>
<tr>
<th>Malware Family (Kaspersky)</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEUR: Trojan.AndroidOS.Triada.e</td>
<td>47b8588aa3fbaaa072ae4a7af3a0e6dc6d0cd2630e</td>
</tr>
<tr>
<td>HEUR: Trojan-Spy.AndroidOS.Geinimi.a</td>
<td>241311ca2752274fabdfc674a2cc55e825ebebf</td>
</tr>
<tr>
<td>SLocker</td>
<td>021975cfd9c3e2087d82887c3c77d5cd9c4b3</td>
</tr>
<tr>
<td>Lipizzan</td>
<td>19bed9b4ba8f7096c182d4d43c8965702a409e</td>
</tr>
<tr>
<td>HEUR: Trojan-Dropper.AndroidOS.Agent.n</td>
<td>cdbae6f027287335909e32d34e3f6b3bd007b</td>
</tr>
<tr>
<td>HEUR: Backdoor.AndroidOS.Maxit.a</td>
<td>04fc02b59efba40e6f02861d3537858aca6678</td>
</tr>
<tr>
<td>HEUR: Backdoor.AndroidOS.Maxit.a</td>
<td>25e40f90e5ff8306ba0432ec487206ae43604e2e</td>
</tr>
<tr>
<td>HEUR: Trojan-SMS.AndroidOS.Opfake.a</td>
<td>f36775ce5af42c2eb1710463a4e310e808b91230e</td>
</tr>
<tr>
<td>HEUR: Backdoor.AndroidOS.Maxit.a</td>
<td>27acb489789ea100a5a9eca243d5f471fb848b8</td>
</tr>
<tr>
<td>HEUR: Trojan-Banker.AndroidOS.Svpeng.a</td>
<td>2838613a36886a39345201f471fc037874f</td>
</tr>
<tr>
<td>not-a-virus: HEUR: RiskTool.AndroidOS.SMSreg.</td>
<td>c6a95c0f5c1e822b20eca88b60d0d9471570a0f8</td>
</tr>
<tr>
<td>HEUR: Trojan-Spy.AndroidOS.Lipizzan.a</td>
<td>f2c4544fbc402b7643666d9762a85646be47d7</td>
</tr>
<tr>
<td>not-a-virus: HEUR: Monitor.AndroidOS.Toreo.a</td>
<td>d1fc1ebf4c20d8a11949eb73157fc1da4f2e0</td>
</tr>
<tr>
<td>HEUR: Exploit.AndroidOS.Lotoor.bm</td>
<td>2a1b02ad04f3221ab7198b33af170675bae3285f</td>
</tr>
</tbody>
</table>

Table A.1.: Malware applications containing custom cryptography in the dataset
## A.2 Benign Custom Cryptography Samples

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Version</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.rebin.rebindictionary</td>
<td>1.6</td>
<td>64e3e5d4f3d99b0f14621536908f8a05a8879a1</td>
</tr>
<tr>
<td>surce.code.crypto</td>
<td>1.1</td>
<td>ea9b5c7a005e8482f4e9d51912fd49db2580</td>
</tr>
<tr>
<td>com.maitop.gravityBounds</td>
<td>1.2.5</td>
<td>72e87ca1fca9e5e39e325453250fd63ac3268</td>
</tr>
<tr>
<td>ezeelarning.acidbasesalt</td>
<td>1</td>
<td>94bd7f1883b2c45bba2be43f4f286b6a23e9f6</td>
</tr>
<tr>
<td>com.leagem.cheeslive</td>
<td>2.5</td>
<td>7a7898ae5cc758ce63ad2d93c64531c00b60a</td>
</tr>
<tr>
<td>com.goldbean.yoyo</td>
<td>1.8.2</td>
<td>67402dd0d0be0b51a93b01ff6eca359b9138e6</td>
</tr>
<tr>
<td>org.mmnx</td>
<td>4.44</td>
<td>ca896c7272bd333a2c6279daa5dd7b9c7b5f8de</td>
</tr>
<tr>
<td>com.hiro4476.onkeyboard</td>
<td>2.1.3</td>
<td>2411ba0d184ef152df9b334dfb00e65cc8a</td>
</tr>
<tr>
<td>com.cnlaunch.mycar</td>
<td>1.04.018</td>
<td>13cf1e1629f60f773c47257bdcfe5b58e84026e</td>
</tr>
<tr>
<td>com.maggamobile.game.MatchUpBuddy</td>
<td>1.0.1</td>
<td>dada5f75f168343d041d082e19c9d1e724d4b9</td>
</tr>
<tr>
<td>com.paythinksst500.multi</td>
<td>1.0.3.3</td>
<td>38b70c0ce7cb3ac2d12a134fc662c9a0bb4cd</td>
</tr>
<tr>
<td>com.hirschifilters.hifilter</td>
<td>1.0.12.0</td>
<td>6f474a444f919fe49176c6243912d9b92c</td>
</tr>
<tr>
<td>com.jang.QuizOXAll_an</td>
<td>6</td>
<td>fa4fe1e8536a454f00a9af6e4e7bad02a4eb</td>
</tr>
<tr>
<td>com.wendy</td>
<td>2.3.11</td>
<td>ebeec781b662fa17078a0a5963c0b1eaeceb89</td>
</tr>
<tr>
<td>com.asai.android.Planet</td>
<td>1.0.3.0</td>
<td>678134af8dd1f18df8f7a15ed71349f952bb86</td>
</tr>
<tr>
<td>com.ympark1227 shattered</td>
<td>40</td>
<td>88b16ba8ad6c33e35d9032b586fe082b0f</td>
</tr>
<tr>
<td>com.scard_members</td>
<td>8</td>
<td>ad26e0f5360099328778b41c7f5b8186ace5c4</td>
</tr>
<tr>
<td>ara.ben.dictionary</td>
<td>7.2</td>
<td>85ca88d8a8b543a5bfe51ea36420e25c9112d98</td>
</tr>
<tr>
<td>com.biosbio.yae_roo</td>
<td>1</td>
<td>43a38f361a61ba1eba1c161ba87d3aeabe5f143</td>
</tr>
<tr>
<td>jp.co.softbank.cr.ChefDorobou01</td>
<td>1.1.2</td>
<td>0359e9297287d696d19b2a70b25883cae53502</td>
</tr>
<tr>
<td>fr.bluemary.horoscope</td>
<td>1.1.2</td>
<td>4dc28e2458a143d29r38387311acaeebe0149f3</td>
</tr>
<tr>
<td>de.nichtlusch.android</td>
<td>2.0.1 (811)</td>
<td>ffe2721f1e18b90e1592d4d4ab7b12d2ec869f</td>
</tr>
<tr>
<td>com.nvdevelopment.ref.constaustralia.free</td>
<td>1</td>
<td>f481bd3d4e90e34db19add4d1792941790e0</td>
</tr>
<tr>
<td>com.stempelo.raceco</td>
<td>1.2</td>
<td>d479ze89f7ca0a0aed597dece5f179832af7</td>
</tr>
<tr>
<td>ru.vdevelopment.ref.kksrkk.free</td>
<td>1</td>
<td>ebc9c9315f5ca6279f60aef37e7c46dafa87</td>
</tr>
<tr>
<td>sites.mjwihita.cryptisms</td>
<td>3.1</td>
<td>bd5c5d9d268e0b9dccc1590999714d4ddc1c9e7</td>
</tr>
<tr>
<td>com.akdevelopment.ref.ekarstvarus.free</td>
<td>1</td>
<td>0e4d11ca37264f35c2b85f7e63baf03a55</td>
</tr>
<tr>
<td>com.eagles.learningkorean</td>
<td>1.2.9</td>
<td>d4f68fd14036185a16c6ca89a7a3e9049145cb</td>
</tr>
<tr>
<td>de.trucktrack.trucktrack</td>
<td>1</td>
<td>23649ca3f07e05037678b3f8b38d02722128d9</td>
</tr>
<tr>
<td>com.catvoiz.app</td>
<td>1.4.1</td>
<td>9b41a56ed8ea20624a49c9495c9a67e0f10c15</td>
</tr>
<tr>
<td>com.ultracilitation.limit</td>
<td>2.0.31</td>
<td>55517e3aaed57a2aa0963eb93d42227ab8efc4</td>
</tr>
<tr>
<td>nl.amrold.megadroid</td>
<td>1</td>
<td>70b5e6256868e6666e6c5e5e6d31744d44a41f3</td>
</tr>
<tr>
<td>com.webappclouds.domineque</td>
<td>1</td>
<td>e4da66f36f44b3a0ab5666eae7c9aadf2c7b083</td>
</tr>
<tr>
<td>com.webappclouds.metrobeauty</td>
<td>1</td>
<td>303d22de8e8e6ae0a0e8d57455b264486cbc327</td>
</tr>
<tr>
<td>com.dehun snapmeup</td>
<td>5.1.0</td>
<td>9862b83d3e932d6c75c1947ca50666ba12685a2f</td>
</tr>
<tr>
<td>com.smartcocoa.magicctrl</td>
<td>1.1.1.1</td>
<td>b1eaa42971a0a05734d11615afe472397a6ed8</td>
</tr>
<tr>
<td>com.coffeebeankorea.android</td>
<td>1.4.5</td>
<td>ec102f303d1b06e218e4533b694c454868c2</td>
</tr>
<tr>
<td>noisette.product.amegashtashiru</td>
<td>1.1.1</td>
<td>3bb9966d42dac1ac306b93963961c59dfc95c</td>
</tr>
<tr>
<td>com.bravolang.phrasebook</td>
<td>10.1.0</td>
<td>f2d8d8b8388bf3c979d2589068a35894527d2</td>
</tr>
<tr>
<td>com.vc.app.c8jda4113</td>
<td>1</td>
<td>fddce3a2967bc6a047a18e60ba4e0abe8d87</td>
</tr>
<tr>
<td>com.kdrama</td>
<td>2.0.0</td>
<td>1f63729922e00d66809d9dd6b159b550f4843</td>
</tr>
<tr>
<td>com.live2times.drknow</td>
<td>1.3</td>
<td>b20e834a552269f571d6e5d5d0f5ac052463b2d</td>
</tr>
<tr>
<td>com.irefrd.alivehospice</td>
<td>2.6.4</td>
<td>0381b334395f26a587b181766b2be9c95683</td>
</tr>
<tr>
<td>com.xiaomi.router</td>
<td>3.0.9</td>
<td>cee37214a2f114f6e46350718c7818a4d887cd3</td>
</tr>
</tbody>
</table>

Table A.2.: Benign applications containing custom cryptography in the dataset set
### A.3 Non Custom Cryptography Samples

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Version</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.google.android.apps.authenticator2</td>
<td>5.00</td>
<td>0740c4ea03fa20ed5eeec554966a1ceed35593b87</td>
</tr>
<tr>
<td>com.google.android.launcher</td>
<td>1.4.large</td>
<td>85baec5ecfd9cb3e6db21c7cc5cc281f1b093204</td>
</tr>
<tr>
<td>com.secuso.torchlight2</td>
<td>1.3</td>
<td>d0772de89ef2734b565ae879aab0b92d1e0a6f99</td>
</tr>
<tr>
<td>org.mozilla.klar</td>
<td>2.5</td>
<td>e9d3633ae125fd2d0884b6f0fdd05b0700d7d68f</td>
</tr>
<tr>
<td>org.secuso.privacyfriendlyactivitytracker</td>
<td>1.0.5</td>
<td>7e6a4447f1fcee2bd0c175e93ff03b6dd510f006d</td>
</tr>
<tr>
<td>org.secuso.privacyfriendlynotes</td>
<td>1.0.1</td>
<td>e25e82fa57bbe4b0a7ea10a2eac615e1b1008263</td>
</tr>
<tr>
<td>org.secuso.privacyfriendlysudoku</td>
<td>2.1.1</td>
<td>6f11389f8838968956488f4a38db7b248eb9f7</td>
</tr>
</tbody>
</table>

*Table A.3.: Applications which methods were used as samples for non custom cryptography*
B  Play Store Application Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>Developer</th>
<th>Version</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Easy LaCrypte</td>
<td>Bøbby</td>
<td>1.2</td>
<td>b0bby.easylacrypte</td>
</tr>
<tr>
<td>BMW Augmented</td>
<td>BMW GROUP</td>
<td>1.18</td>
<td>com.bmw.augmented.bmw7series</td>
</tr>
<tr>
<td>Confide</td>
<td>Confide</td>
<td>5.2.9</td>
<td>cm.confide.android</td>
</tr>
<tr>
<td>ES File Explorer File Manager</td>
<td>ES Global</td>
<td>4.1.6.2</td>
<td>com.estrongs.android.pop</td>
</tr>
<tr>
<td>FreeMessage - free Messenger</td>
<td>GMX</td>
<td>1.23.10</td>
<td>com.unitedinternet.portal.mobilemessenger</td>
</tr>
<tr>
<td>Kik</td>
<td>Kik Interactive</td>
<td>12.4.1.19850</td>
<td>kik.android</td>
</tr>
<tr>
<td>LINE: Gratis-Anrufe</td>
<td>LINE Corporation</td>
<td>8.2.4</td>
<td>jp.naver.line.android</td>
</tr>
<tr>
<td>Messenger Lite: Kostenlosen Anrufe […]</td>
<td>Facebook</td>
<td>26.0.0.7.188</td>
<td>com.facebook.mlite</td>
</tr>
<tr>
<td>QQ</td>
<td>Tencent Technology[…]</td>
<td>6.6.7</td>
<td>com.tencent.mobiledq</td>
</tr>
<tr>
<td>Skype - free IM &amp; video calls</td>
<td>Skype</td>
<td>8.8.0.61630</td>
<td>com.snapchat.android</td>
</tr>
<tr>
<td>Smart Plug</td>
<td>EDIMAX</td>
<td>1.9.3</td>
<td>com.edimax.smartplug</td>
</tr>
<tr>
<td>Snapchat</td>
<td>Snap Inc</td>
<td>10.26.5.0</td>
<td>com.snapchat.android</td>
</tr>
<tr>
<td>TUCan TU Darmstadt</td>
<td>Datenlotsen […]</td>
<td>1.7.2</td>
<td>de.datenlotsen.campusnet.tuda</td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td>WhatsApp Inc.</td>
<td>2.18.46</td>
<td>com.whatsapp</td>
</tr>
</tbody>
</table>

Table B.1.: Play Store application dataset

<table>
<thead>
<tr>
<th>Package</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b0bby.easylacrypte</td>
<td>6d30fc4844bdef78c30ca9b83797d6078e747f</td>
</tr>
<tr>
<td>com.bmw.augmented.bmw7series</td>
<td>ffc0abb3e336fcad31cd51462bc2849dc301e20</td>
</tr>
<tr>
<td>cm.confide.android</td>
<td>8af08845edebf925e2203469dc03598815470</td>
</tr>
<tr>
<td>com.estrongs.android.pop</td>
<td>dac0fd386742c89ba3d3f8bf87b920e6eb</td>
</tr>
<tr>
<td>com.unitedinternet.portal.mobilemessenger</td>
<td>14bc5a58e410ed39de43b494f334732db7ea5</td>
</tr>
<tr>
<td>kik.android</td>
<td>bb6b493c452eb842a5ad7a67f6468d4a010f6</td>
</tr>
<tr>
<td>jp.naver.line.android</td>
<td>26418decd7a74ca6137d06038562ec24ce48f24</td>
</tr>
<tr>
<td>com.facebook.mlite</td>
<td>850b16c249ca8a9b44274dc342a999edf0</td>
</tr>
<tr>
<td>com.tencent.mobiledq</td>
<td>6eef893a5fe900ce20619b504b5d2c8a4e3e3c66</td>
</tr>
<tr>
<td>com.snapchat.android</td>
<td>be5e434dac10edc615c9a7a454e1d24bd0d710</td>
</tr>
<tr>
<td>com.edimax.smartplug</td>
<td>a23f15e9229badad29c5b751289572b3b5ddbd2c</td>
</tr>
<tr>
<td>com.snapchat.android</td>
<td>a4616cf018804a71baaf05b4409eb89134ef85</td>
</tr>
<tr>
<td>de.datenlotsen.campusnet.tuda</td>
<td>12d19b5ae03e91c4f3ffcd3ed334ca6084fo7b763f</td>
</tr>
<tr>
<td>com.whatsapp</td>
<td>1fe780f42df5f73978e480d29371104669a03c8</td>
</tr>
</tbody>
</table>

Table B.2.: Play Store application dataset checksums
C Obfuscation Application Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>Developer</th>
<th>Version</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenPass</td>
<td>Arseniy Lartsev</td>
<td>1</td>
<td>com.example.openpass</td>
</tr>
<tr>
<td>OpenPass - obfuscated</td>
<td>Arseniy Lartsev</td>
<td>1-obf</td>
<td>com.example.openpass</td>
</tr>
<tr>
<td>Pijaret</td>
<td>Darius Musiolik</td>
<td>1.7</td>
<td>dmusiolik.pijaret</td>
</tr>
<tr>
<td>Pijaret - obfuscated</td>
<td>Darius Musiolik</td>
<td>1.7-obf</td>
<td>dmusiolik.pijaret</td>
</tr>
</tbody>
</table>

Table C.1.: Obfuscation experiment application dataset from the F-Droid store [30]

<table>
<thead>
<tr>
<th>Package</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.example.openpass</td>
<td>110996e33993a88301955ce4a2eba9efb0a997b0</td>
</tr>
<tr>
<td>com.example.openpass (obfuscated)</td>
<td>9af805e52d57b31f6c61f6c57b08303f417f6cb</td>
</tr>
<tr>
<td>dmusiolik.pijaret</td>
<td>683e1bb5299c9566f2638697e0739c3fd24dca4</td>
</tr>
<tr>
<td>dmusiolik.pijaret (obfuscated)</td>
<td>0ffca234a3b9d968b8d07fe90de0436937e38713</td>
</tr>
</tbody>
</table>

Table C.2.: Obfuscation experiment application dataset checksum