Identifying Cryptographic Functionality in Android Applications

Alexander Oprisnik, Daniel Hein and Peter Teufl
Institute for Applied Information Processing and Communications Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria

Keywords: Mobile Application Security, Machine Learning, Detection of Cryptographic Code, Container Applications, Password Managers, Data Encryption on Mobile Devices, Semantic Pattern Transformation, Correct Deployment of Symmetric and Asymmetric Cryptography.

Abstract: Mobile devices in corporate IT infrastructures are frequently used to process security-critical data. Over the past few years powerful security features have been added to mobile platforms. However, for legal and organisational reasons it is difficult to pervasively enforce using these features in consumer applications or Bring-Your-Own-Device (BYOD) scenarios. Thus application developers need to integrate custom implementations of security features such as encryption in security-critical applications. Our manual analysis of container applications and password managers has shown that custom implementations of cryptographic functionality often suffer from critical mistakes. During manual analysis, finding the custom cryptographic code was especially time consuming. Therefore, we present the Semdroid framework for simplifying application analysis of Android applications. Here, we use Semdroid to apply machine-learning techniques for detecting non-standard symmetric and asymmetric cryptography implementations. The identified code fragments can be used as starting points for subsequent manual analysis. Thus manual analysis time is greatly reduced. The capabilities of Semdroid have been evaluated on 98 password-safe applications downloaded from Google Play. Our evaluation shows the applicability of Semdroid and its potential to significantly improve future application analysis processes.

1 INTRODUCTION

Mobile devices[1] are becoming an integral part of all our lives. Mobile devices manage our personal data, but are also useful for managing work-related data. Using private devices (Bring Your Own Device (BYOD)) for working with corporate data poses a security challenge. The security challenge is to protect the confidentiality and integrity of corporate and personal data in the face of potentially adverse applications on the same mobile device, or attackers that gain physical access to a device. Today's mobile devices provide a number of security functions helping to overcome this security challenge. These security functions include application data encryption, platform encryption, hardware secured key management, etc. Unfortunately, using these security functions in a corporate BYOD scenario, or private use cases, poses two problems. **First**, the heterogeneity of employee (privately) owned devices and their varying platform security functions makes it both hard and expensive to achieve a specific security level. For instance, if an app requires secure storage of cryptographic keys, than this can be implemented differently on platforms that provide hardware support through secure elements, than on platforms that lack this feature. **Second**, many high quality, hardware supported security functions can only be used if the user has activated the support on her, or his device. For example, automatic application data encryption might only work if the user has set a password for the device. In a BYOD device scenario the employer – mainly due to legal reasons – has no handle to enforce policies on a privately owned device. Therefore, so-called container applications were created that portably implement the required security functions on different, heterogeneous mobile devices, while simultaneously enforcing corporate security policies on the data. The same problems can also be found in many other application types that are used for private and corporate use cases. A good example for security-relevant applications are password-managers that face similar problems as container applications. Due to the uncertain existence of platform security features, such applica-

[1] Referring to current smartphones, phablets and tablets.

Oprisnik A., Hein D. and Teufl P..
Identifying Cryptographic Functionality in Android Applications.
DOI: 10.5220/0005056301510162
In *Proceedings of the 11th International Conference on Security and Cryptography* (SECRYPT-2014), pages 151-162
ISBN: 978-989-758-045-1
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

tions need to implement their own encryption mechanisms that are vital for the protection of the password data stored on the device. Application developers need to solve two major issues: **First**, the security level of platform security functions cannot be achieved when porting these functions to the applications. Typically, platform security features have the advantages of being tightly integrated into the operating system and have access to platform specific hardware security features like secure elements for storing cryptographic keys. **Second**, the requirement to reimplement these security functions is a further source for implementation errors that can significantly impact the security of the whole application.

Recently, several research teams (Egele et al., 2013; Georgiev et al., 2012; Fahl et al., 2012) have pointed out that security critical code is often incorrectly implemented. We manually inspected a number of security critical apps and concur that cryptographic functions are often either incorrectly implemented, or incorrectly used, or both. Automatically analyzing the correct use of cryptographic functions is viable, as demonstrated by (Egele et al., 2013). However, such methods can only be applied to a priori known (standard) implementations of cryptographic functions. We have implemented Semdroid to facilitate the analysis of applications using non-standard implementations of cryptography. Semdroid has been designed to provide a generic, static analysis tool for Android applications that can be executed on the device or on external systems. In the scope of this work, Semdroid was configured to use various machine learning techniques for identifying security critical functions within mobile applications. The main emphasis is placed on the identification of code that implements cryptographic functionality, such as symmetric/asymmetric key cryptography and hash functions. Apart from the obvious operations related to data encryption, signature and hash creation, these operations are also highly relevant in key derivation functions required by many container applications or password managers to derive cryptographic keys from passwords. Further use cases, which have already been superficially tested but not evaluated, include the identification of SMS communication channels, root check functionality and directly identifying custom key derivation functions, instead of finding cryptographic component functions.

The next section surveys related work and is followed by a detailed description of the Semdroid framework and the techniques it employs. Finally, the evaluation section evaluates the machine learning based analysis by an empirical analysis conducted on popular password-safe applications downloaded from the Google Play Store.

2 RELATED WORK

Recently, the incorrect use of cryptographic protocols and functionality by application programs has garnered much attention by researchers. Georgiev et al. (Georgiev et al., 2012) have demonstrated that correctly using a well-established end-to-end security protocol such as Transport Layer Security (TLS) through equally well-established implementations thereof is a challenging task. Specifically, incorrect certificate validation leaves many widespread applications vulnerable to Man-in-the-Middle (MITM) attacks. In addition to pointing out these problems, they propose several remediation techniques to mitigate them. While Georgiev et al. have analyzed a number of different systems including PC and mobile devices, Fahl et al. (Fahl et al., 2012) have performed a similar analysis with a focus on Android applications. They have used their tool MalloDroid to analyze more than 13,500 applications from the Android market, where 1,074 of these applications were found to use TLS with insufficient certificate validation. Of those, Fahl et al. manually analyzed 100 applications and were able to launch successful MITM attacks on 41 of them. Whereas MalloDroid is a static code analysis tool build on the Androguard[2] reverse engineering framework, Semdroid uses machine learning methods to identify cryptographic code. Furthermore, MalloDroid's goal is to detect use of TLS, whereas Semdroid aims at detecting code implementing cryptographic functionality.

Egele et al. (Egele et al., 2013) have investigated Android applications for programming mistakes, when using cryptographic functionality. To that end they have developed CryptoLint, a tool that uses static code analysis to detect applications that use cryptographic functionality and determines the parameters with which the app invokes this cryptographic functionality. CryptoLint is able to check these parameters against a set of rules defining common programming mistakes. Their analysis shows that of the 145,095 Google Play Store applications they examined, 15,134 use cryptographic functionality, of which CryptoLint was able to successfully analyze 11,748. Only 1,421 of these applications did not violate any rules. Similar to Semdroid CryptoLint uses static code analysis on compiled Android applications. As opposed to Semdroid's machine learning based cryptographic functionality detection, CryptoLint's code analysis is based on type analysis,

[2] http://code.google.com/p/androguard/

super control flow graph extraction, and static slicing to determine the parameters with which the cryptographic functionality is invoked. By its very nature CryptoLint's analysis needs to work from well known starting points such as certain functions in Java's Cryptography Extension. Semdroid on the other hand is able to find non-standard implementations of cryptographic functionality, for further analysis.

The publications by Georgiev et al. (Georgiev et al., 2012), Fahl et al. (Fahl et al., 2012), and Egele et al. indicate that cryptographic functionality and protocols are often used incorrectly. This fact warrants extra attention to the correct use of cryptographic functionality in security-critical applications. Semdroid facilitates analysis of such applications, by identifying portions of code pertaining to cryptographic functionality.

Semdroid employs machine-learning techniques to identify relevant code segments. The use of machine-learning techniques for mobile-application analyses is a proven technique. Various authors used static approaches in combination with machine-learning techniques to categorize mobile applications (Shabtai et al., 2010), (Ghorbanzadeh et al., 2013) and to detect suspicious mobile applications (Wu et al., 2012). Beside static approaches, also dynamic analysis solutions have been combined with machine-learning techniques for application classification and malware detection purposes by various researchers. Representative examples are proposed and discussed in related work by Shabtai, Kanonov et al. (Shabtai et al., 2011) and Bruguera et al. (Burguera et al., 2011).

3 Semdroid

Semdroid is a static Android application analysis framework capable of detecting certain functionality within Android applications. Semdroid is implemented in Java, and has a flexible architecture that can be used for various classification- and analysis tasks. Multiple analysis plugins can be employed that analyze Android application packages (`.apk` files) and generate application analysis reports. These plugins can be based on any static analysis approach. First, we are going to present the Semdroid framework and then we will discuss the analysis plugins used to detect cryptographic code.

3.1 The Framework

The Semdroid framework handles several pre- and post-processing operations. The basic analysis workflow performed by the Semdroid framework is depicted in Figure 1 and consists of the following steps: **(1)** The Android application package under analysis is parsed, and the application structure is reconstructed. **(2)** The resulting *AppObject* is then handed to all analysis plugins. Each plugin then analyzes the contents of this object and creates an application analysis report. **(3)** The Semdroid framework collects all of these reports and bundles them to a single analysis report.

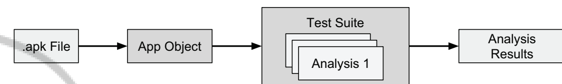

Figure 1: Semdroid architecture.

The *AppParser* implemented in the Semdroid framework represents one of the core components that extracts and prepares the data contained in an Android application for the subsequent analyses. It has two main tasks: **First**, it parses the contents of the *AndroidManifext.xml* file which contains metadata, such as used permissions, defined activities, or services. **Second**, the *classes.dex* file is parsed, which contains the Dalvik bytecode of the respective Android application. The class structure is reconstructed and for each method of each class, the corresponding opcodes, method calls, and local variables are extracted. To parse the Dalvik executable, we adapted the *dex2jar* library[3] to suit our needs. We implemented custom visitors for all application components (classes, methods, fields, and opcodes), and also added several filters in order to filter out irrelevant data. Such filters can be used, for example, to discard all code with the exceptions of methods with a minimum amount of opcodes, methods that require certain permissions, or methods that contain certain API calls. This filtering process is performed globally and applies to all subsequent analysis plugins. Later, each analysis plugin can perform a second filtering process to select analysis-relevant data only. The output of the *AppParser* component is a data structure called *AppObject*, which is then used as input for the subsequent analyses.

The Semdroid framework can be used on a personal computer or directly on an Android device. On a personal computer, a command line interface can be used to perform the analysis process. The analysis results are stored in an XML and an HTML file. For on-device analysis, the Semdroid Android application has to be installed on the Android device. The user can then select an application to be analyzed from a list of all installed applications. Then, the analysis

[3]https://code.google.com/p/dex2jar/

will be performed and the user will be notified once this analysis process has been completed. The analysis results can then be viewed.

3.2 Analysis Plugins

Semdroid analysis plugins can use any static analysis approach. Each plugin receives the *AppObject* and creates an analysis report containing all findings. This report contains a number of labels, where each label corresponds to a property or a functionality, like "cryptographic" or "normal" code. All application components, for example all methods or classes of the application, can be analyzed separately and labeled according to their functionality. These results are then included in the analysis report.

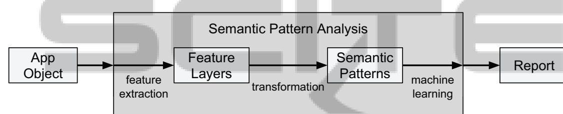

Figure 2: Semantic Pattern Analysis workflow.

In the scope of this work, we have implemented machine learning based analysis plugins. For each of these plugins, the *AppObject* is subject to an instance generation and feature extraction process that prepares the information required for the specific analysis. Figure 2 shows the workflow for the *Semantic Pattern Analysis* we use in this work. The input for the analysis plugin is the *AppObject*. **First**, we generate feature layers that contain characteristic features of the application. **Second**, we apply the Semantic Pattern Transformation on these feature layers to get Semantic Patterns, which are simple double vectors. **Finally**, we supply the Semantic Patterns to machine learning algorithms, which output a list of labels, one for each pattern. These labels are then added to the final analysis report.

(1) Feature Layer Generation: Semdroid allows to separately analyze different components of an application, called *instances*. These instances can either be generated for methods, classes, the package, the complete *AppObject*, or any other selection criterion. Instance filters are used to select analysis-relevant components. For example, it is possible to analyze only classes that extend a certain class or to analyze only methods that perform at least a certain number of operations. For each instance, features are extracted from the corresponding application component according to the analysis configuration. The current analysis procedure implements various feature types, such as Dalvik opcodes, method calls, local variables, fields, super classes, required permissions, and intent filter parameters. For each feature type, another filtering and grouping mechanism is employed to retrieve relevant data and to improve the analysis performance. The following features are required for the analysis plugins used here:

- **Dalvik Opcodes:** The Dalvik opcodes[4] are used for almost all analyses. They are either used directly or can be grouped according to pre-defined rules (e.g., creating a group for different logical operations).
- **Method Calls:** Method calls invoked by the component under analysis are commonly used as features. A distinction is made between external API calls and internal calls to methods implemented by the application. In general, API calls provide more information for the analyses than very application-specific internal method calls.
- **Local Variables:** This refers to the types of local variables, which are used in a method. They can be separated into two categories: The first category is comprised of basic data types (*integer*, *float*, *double*, *byte*, etc.). The second group are composite data types (Java class objects), which can be separated again into internal objects implemented by the application itself, and external objects provided by the Android platform (e.g., `java.math.BigInteger`, or `android.telephony.SmsMessage`).

The resulting feature layer contains a list of instances. For each instance the extracted instances contain the extracted feature values and are then converted into Semantic Patterns. The Semantic Patterns are then used as input for the machine learning based training- and classification process.

(2) Semantic Pattern Transformation: The plugins utilized in this work are based on standard machine learning algorithms. However, prior to their application, the Semantic Pattern Transformation (Teufl et al., 2013) is used to transform the standard feature vectors commonly used in machine learning into another representation that models the semantic relations between the feature values. These vectors and their relations can either be analyzed directly by using simple vector-based operations (e.g. achieving semantic-aware search), or used as input for standard supervised or unsupervised machine learning algorithms. By doing so, many of the machine learning specific pre-processing steps, such as normalization, choosing a specific algorithm or finding a representation for numerical and symbolic features values, can

[4]http://developer.android.com/reference/dalvik/bytecode/Opcodes.html

be avoided. This allows for deploying knowledge discovery procedures in heterogeneous domains by applying a minimal number of domain-specific adaptations only. Due to the various features, such as opcodes, opcode histograms, method names, local variables, etc. within the Dalvik code, and the different classifiers based on different combinations of those features, we have chosen to transform the raw feature vectors into Semantic Patterns prior to the application of supervised and unsupervised machine learning algorithms.

(3) Machine Learning: Once all instances have been transformed to Semantic Patterns, they are used as input for the machine learning framework. Here, the common procedures for training and applying classifiers are applied. We use only supervised algorithms, but there are also scenarios that would mandate the use for unsupervised algorithms, e.g., when anomaly detection procedures are required, or a better understanding of unknown data and relations should be gained.

SCIENCE AND TECHNOLOGY PUBLICATIONS

4 EVALUATION

The evaluation process of Semdroid and specifically its cryptographic code detection facilities were designed with the following goals in mind: **First**, for the deployment in real application analysis scenarios, we need to know the accuracy of the classifiers in terms of false positive/negative rates. **Second**, experience on the capabilities of Semdroid and the machine-learning based analysis needs to be gained by the application of the trained classifiers to real application data. **Third**, possible improvements and research required for the analysis of real applications should be revealed by analyzing real application data. We aim to achieve these goals with the following evaluation scenarios E1 to E4:

- **E1 and E2 – Training of classifiers and application using verified test sets:** By extracting and manually analyzing and labeling methods from well-known cryptographic libraries and standard apps that do not contain cryptographic functionality, the respective classifiers can be trained and a performance analysis can be conducted (E1). In E2, the performance of the classifier on obfuscated code is evaluated. Due to the manually verified test sets, the basic performance characteristics of the symmetric and asymmetric classifiers in terms of false positive and false negative rates are revealed by E1 and E2.
- **E3 (Symmetric Cryptography) and E4 (Asymmetric Cryptography) – Application on Real Apps:** In these scenarios the previously trained classifiers for symmetric and asymmetric cryptography are applied to 98 password safe applications downloaded from the Google Play Store. The methods, which are tagged as cryptographic code by the trained classifiers are subject to an extensive manual analysis. The aim of this analysis is to understand the nature of the code that can be detected by the classifiers and thereby deduce its capabilities. Since, knowing these capabilities is crucial for real application analyses, a major part of this work is dedicated to E3 and E4. While both scenarios enable us to learn further details of the false positive rates of the classifiers, additional conclusions on the false negative rate cannot be drawn. The reason is the substantial number of analyzed methods, which makes a manual inspection of the non-cryptographic methods infeasible.

4.1 Analysis Setup

The evaluation scenarios E1 to E4 have been implemented by using the Semdroid analysis framework. The following extraction and analysis steps are applied to the analyzed applications by utilizing the Semdroid framework.

(1) Instance Generation – Method Filters: The instance extraction process has been configured to generate an instance for each analyzed method. Two different method filters have been applied. For the dataset related to symmetric key cryptography only those methods that contained equal to, or more than 30 opcodes are considered for analysis. For the asymmetric evaluation only methods with 4 or less opcodes are filtered. This choice is explained by the fact that functionality from the standard Java SDK API can be used for the mathematical operations required for asymmetric cryptography (e.g. calculations based on the *BigInteger* type), which decreases the amount of required basic opcodes. In contrast, many operations essential for symmetric key cryptography require a number of logical and mathematical operations to mangle the input data with the cryptographic key.

(2) Instance Generation – Feature Extraction: For the asymmetric classifier the method call inclusion depth is set to 0. While depth 0 means that only the features from the analyzed method are extracted, depth 1 causes the analyzer to extract the features of the methods called in the analyzed methods, and depth 2 also includes the features from the methods called at depth 1. Since, methods for symmetric ciphers require more opcodes that are typically arranged in multiple methods, the inclusion depth was

set to 2 for this classifier. For the evaluation scenarios we have used the following features:

- **Opcode Histograms:** The opcodes extracted from the analyzed methods have been filtered, grouped and rearranged in histogram vectors. These vectors reflect the number of occurrences of specific opcodes or opcode groups. Opcodes groups have been used to reduce the dimensionality of the feature vectors and noise. For the symmetric classifiers only the mathematical (e.g., *add*, *sub*, *div*) and logical opcodes (e.g. *xor*, *shl*, *shr*) were used for the histogram vector generation. In total, 26 opcode and opcode groups have been used for the symmetric histogram vectors.

  For the asymmetric classifier, additional opcodes used by typical implementations have been added: opcodes related to array specific operations (e.g., *array_length*), comparing variables (e.g., *cmp*, *cmpl*, *cmpg*), and branch operations (e.g., *if_eq*, *if_ne*, *if_lt*). In total, 29 opcodes and opcode groups have been used for the asymmetric histogram vectors.

- **Local Method Calls:** Method calls from classes within the packages `java.math.*` have been used as features for the asymmetric classifier. The rationale is that many asymmetric implementations rely on Java SDK functionality to implement asymmetric ciphers.

- **Local Variables:** For the asymmetric classifier the local variable types implemented in the `java.math.*` packages have been used as features. Especially the *BigInteger* type is heavily used for implementations of asymmetric algorithms.

(3) Semantic Pattern Generation: The method instances are comprised of the previously selected features for symmetric and asymmetric classifiers. The gained feature vectors are then transformed into Semantic Patterns.

(4) Supervised Learning: The cryptographic code classifiers were trained on the Semantic Patterns contained in the carefully chosen training data sets using the Weka (Witten et al., 2011) implementation of the Support Vector Machine (Cortes and Vapnik, 1995) algorithm.

4.2 Datasets

The following data sets have been used for the evaluation scenarios:

Training Set – Symmetric Key Cryptography: The training set consists of 6 methods from 3 different AES implementations extracted from the package `org.bouncycastle.crypto.engines` of the Bouncy Castle library[5]: namely the *encryptBlock* and *decryptBlock* methods of the *AESEngine*, *AESLiteEngine* and *AESFastEngine* implementations. We emphasize that methods belonging to hash functions were not included in the training set, but only in the test set, to highlight their similarity with symmetric cipher implementations. In addition, 100 "normal" methods that do not included cryptographic code have been extracted from one of the author's applications. This application is used for network traffic monitoring and analysis and does not contain any cryptographic functions. There was no special selection process for these methods – the first 100 methods that fulfilled the filter criteria have been extracted. There are two reasons for choosing a small training set, especially for the methods containing cryptographic code: **First**, only cryptographic libraries could be used to automatically extract methods that definitely contain cryptographic functionality. Since there is only a limited number of such libraries and cipher implementations we tried to keep the training set as small as possible in order to have an adequate test set for the evaluation procedure. **Second**, previous evaluation procedures and the literature already indicated that cryptographic code could be identified with high accuracy due to its distinct features.

Verified Test Set – Symmetric Key Cryptography and Hash Functions: All of the methods included in this test set have been manually verified for their usage of cryptographic functions. The following 105 methods have been extracted:

- **Bouncy/Spongy Castle Providers:** 67 methods used by symmetric ciphers implementations have been extracted from `org.bouncycastle.crypto.engines`[6] and methods related to hash functions from `org.bouncycastle.crypto.digests`[7].
- **Cryptix Provider:** 24 methods have been extracted from `cryptix.provider.cipher` of the cryptix provider[8].
- **Other Sources:** 14 methods have been extracted from applications where an analysis revealed that

[5] http://www.bouncycastle.org

[6] e.g., AES, IDEA, DES, Serpent, TEA: http://www.cs.berkeley.edu/∼ jonah/bc/org/bouncycastle/crypto/engines/package-summary.html

[7] e.g., MD5, RIPEMD, SHA-family, Whirlpool: http://www.cs.berkeley.edu/ jonah/bc/org/bouncycastle/crypto/digests/package-summary.html

[8] e.g., Blowfish, CAST5, RC4: http://ds0.cc.yamaguchi-u.ac.jp/ joji/doc/cryptix3.2.0/cryptix/provider/cipher/package-summary.html

SCIENCE AND TECHNOLOGY PUBLICATIONS

cryptographic algorithms (including AES, SHA1, Bcrypt) have been implemented.

As “normal” methods, we have selected 900 methods that have been extracted from different Android components, such as services or activities. 20 out of these 900 methods were extracted from asymmetric cipher implementations (taken from the verified asymmetric test set). The selection of these methods and the classification results highlight the capability of the classifier to differentiate symmetric and asymmetric cryptography.

Training Set – Asymmetric Key Cryptography: The Bouncy Castle implementations of RSA (`org.bouncycastle.crypto.engines.RSACoreEngine`) and ElGamal (`org.bouncycastle.crypto.engines.ElGamalEngine`), have been used as asymmetric training data. In addition, 200 “normal” methods have been randomly selected that do not contain asymmetric cryptography.

Verified Test Set – Asymmetric Key Cryptography: Signature algorithms based on asymmetric cryptography, like DSA, have been considered as asymmetric code as well. Since we did not find that many asymmetric cryptography implementations, the evaluation test set is smaller than for the symmetric case. In total, we used 20 methods implementing asymmetric cryptography and 980 randomly selected “normal” methods. These “normal” methods also include the 105 methods of the symmetric cipher implementations and hash functions that were used in the test set for the evaluation of the symmetric classifier.

Verified Test Set – Obfuscated Code: To show the impact of obfuscated code on our classifiers, we have also created obfuscated versions of the Bouncy Castle library. For obfuscation[9] the default optimized Android ProGuard configuration has been used. Only shrinking, which removes dead code, and method name obfuscation has been disabled in order to be able to better compare the results.

Empirical Test Set – Password Safes: The test set for the empirical analysis of the classifiers for symmetric and asymmetric classifiers consists of 98 password safes applications downloaded from the Google Play Store. Due to the functionality offered by password safes it is expected that such applications either use existing crypto libraries or implemented custom cryptographic functions.

[9] http://developer.android.com/tools/help/proguard.html

4.3 E1 – Classifier Training and Evaluation, E2 – Obfuscated Code

The classifiers for symmetric and asymmetric cryptography have been evaluated as follows: **First**, the classifiers were trained on the training data sets using the Weka implementation of the Support Vector Machine algorithm. **Second**, the trained classifiers have then be applied to the verified test sets. The evaluation results for the symmetric and asymmetric classifiers are presented in the left sub-tables of Figure 3. The following observations can be made: Although, rather small training sets have been used for both classifiers, a very high accuracy can be achieved on the verified test sets. Only one method of the implementation of the Camellia cipher (*setKey* of `org.bouncycastle.crypto.engines.CamelliaEngine`) was wrongly classified as normal method by the symmetric classifier. The symmetric classifier is capable of identifying hash functions as cryptographic code, albeit such methods have not been included in the training set.

The obfuscated Bouncy Castle library has been evaluated with the symmetric and asymmetric classifiers. The results are presented in the right sub-table of Table 3. It can be observed, that for both classifiers the number of analyzed obfuscated methods is larger than for the standard library. This is explained by obfuscation procedures, such as method inlining. The differences in total methods for the symmetric and asymmetric classifiers is caused by the different opcode filters used in the instance extraction process. The good results for the obfuscated libraries are expected, because apart from rearranging the package and method structure (e.g., changing names, inlining) the executed code remains the same. However, if certain features, such as internal method calls or variable types are included in the training process, then the obfuscation of the same code would significantly impact classification results.

4.4 E3 – Password Safes (Symmetric Classifier)

In this evaluation scenario the classifier trained for symmetric key cryptography is applied to the 98 password safes downloaded from Google Play. In total, 59,215 methods have been analyzed.

Overall, the classifier labeled 55,534 methods as “normal” code, and the remaining 3,653 methods as symmetric key cryptography. 849 out of these 3,653 methods have been obfuscated and due to the time-consuming process of manually analyzing obfuscated code, their implemented functionality has not been

symmetric	S-C	N	Total	Correct
S-C	99	1	100	99%
Normal	0	900	900	100%

asymmetric	A-C	N	Total	Correct
A-C	20	0	20	100%
Normal	0	980	980	100%

Analysis	Classification	Standard Library	Obfuscated Library
Symmetric Key Cryptography	Cryptography	293	344
	Normal	1513	1570
	Total	1806	1914
Asymmetric Key Cryptography	Cryptography	27	32
	Normal	6941	7026
	Total	6968	7058

Figure 3: Classifier performance on the verified test sets for the asymmetric classifier (lower left) and symmetric classifier (upper left). On the right, the results for the standard and the obfuscated Bouncy Castle libraries are shown.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: 3,653 out of 55,534 methods were classified as symmetric cryptographic code, which can be categorized in various sub-categories.

Classification	Category	%	Count	# methods
Symmetric key cryptography	Cryptography	47.6	1740	3653
	Obfuscated	23.2	849	
	Encoding schemes and data notations	20.9	765	
	Other	6.6	240	
	Non-cryptographic hash functions	1.3	46	
	Compression functions	0.4	13	
Normal				55534
Total				59215

analyzed. The manual analysis of the remaining 2,804 methods reveals that their functionality can be assigned to the following categories (overview in Table 1):

4.4.1 Cryptographic Functions (1,740 Methods)

The methods containing cryptographic code can be assigned to further sub-categories (overview in Table 2):

Table 2: Manual evaluation results for the category cryptography.

Cryptography (1739 methods)		
Category	%	Count
Symmetric Ciphers	66.6	1158
Hash functions	15.4	268
Modes of operation	5.6	98
MACs	5.1	89
Key and parameter generation	4.1	72
Random number generation	3.1	54

- **Symmetric Ciphers (1,158 Methods):** The majority of identified cryptographic methods are used to build symmetric ciphers.
  - **Bouncy Castle Implementations**: Some applications include the original Bouncy Castle library or a renamed version, like *Spongy Castle*[10]. The latter is a repackaged Bouncy Castle library, which avoids naming conflicts with the Android Bouncy Castle API. Furthermore, some developers renamed Bouncy Castle packages, e.g., by using the package name `org.bownzycastle.*`.
  - **Custom Implementations:** A small number of custom AES implementations could be identified. Furthermore, a few applications included cryptographic code found in libraries. One example for such cryptographic code is *UnixCrypt*[11], which is included in some Apache libraries. According to its documentation, this class implements DES.
  - **Parts of Ciphers:** Many implementations of encryption algorithms are split into several methods. For example, the AES algorithm consists of four parts: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRowKey*. Many implementations of this algorithm, including those in the Bouncy Castle library, have a similar structure where each of these steps is encapsulated within a separate method. Not only does the analysis correctly recognize the main method

Table 3: Manual evaluation results for the category encoding schemes.

Encodings (765 methods)		
Category	%	Count
Base64	31.8	243
UTF-8	19.3	148
JSON	13.2	101
Readers and Writers	13.1	100
ASN1	9.3	71
Other	8.2	63
ISO 9796-1	3.3	25
HEX	1.8	14

[10] http://rtyley.github.io/spongycastle/

[11] http://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/UnixCrypt.html

that performs the encryption and calls all subsequent methods as cryptographic code, but it also tags all parts involved in this encryption process as cryptography.

  - **Round Key Generation:** Many ciphers, like AES, require so-called round keys, which have to be derived from the encryption key. Most of these key generation methods are recognized as well.

- **Hash Functions (268 Methods):** Most of the identified methods related to cryptographic hash functions are Bouncy Castle implementations. Furthermore, 10 custom SHA implementations (both SHA-0 and SHA-1) have been found, contributing with a total of 20 methods, since different parts of the implementations are found separately.
- **Message Authentication Codes (MACs) (89 Methods):** The origin of all identified methods is the package `org.bouncycastle.crypto.macs.*` included in the Bouncy Castle library.
- **Modes of Operation (98 Methods):** Various modes of operation, as for example cipher-block chaining (CBC), Galois/Counter Mode (GCM), or several output feedback modes have been found. All of these methods are located in the Bouncy Castle package `org.spongycastle.crypto.modes.*`.
- **Key and Parameter Generation (72 Methods):** We found two custom implementations of password-based key derivation functions: BCrypt (Provos and Mazieres, 1999), which is based on Blowfish, and scrypt (Percival and Josefsson,). The custom BCrypt implementation has been found in one password safe, whereas the scrypt implementation is included in two other safes, as part of the Spongy Castle library. Furthermore, key generators for DES, IDEA, and LOKI91 have been detected by the analysis, as well as key factories and parameter generators included in the Bouncy Castle library.
- **Random Number Generation (54 Methods)**: Bouncy Castle includes several pseudorandom number generators (PRNGs), found in the package `org.spongycastle.crypto.prng.*`. For instance, the VMPC random number generator included in this package has been detected by the analysis. VMPC itself is a one-way function and stream cipher. In addition to VMPC, `org.spongycastle.util.test.FixedSecureRandom` has been detected as well. Finally, the analysis also detected a custom random number generator not included in bouncy castle, which implements the R250 random number generator.
- **LFSR:** Linear feedback shift registers have similar structures to symmetric-key ciphers. Therefore, methods implementing this functionality could also be identified.

4.4.2 Non-cryptographic Hash Functions (46 Methods)

Methods implementing non-cryptographic hash applications used by data structures like hash maps are also detected by the classifier. An example is *MurmurHash3*[12], which was found in three password safes as part of a Google library, which includes two implementations of this hash function for different digest sizes of 32 and 128 bits.

4.4.3 Encoding Schemes and Data Notations (765 methods)

Various encoding mechanisms and some data notations are recognized as cryptographic code. Table 3 shows the different identified encoding schemes. The most prominent is Base64, with 243 methods, followed by UTF-8 and JSON.

- **Base64 (243 Methods):** Base64 is a binary-to-text encoding scheme utilized by many applications. The encoding/decoding process requires shift operations as well as bitwise AND calculations. In addition, it utilizes an encoding table, similar to the substitution boxes (S-boxes) found in many block ciphers, including AES.
- **JSON (101 Methods):** JSON[13] is short for for JavaScript Object Notation and is used to interchange data between computers. Some methods required for serializing objects, to read and write objects, and to convert data types are recognized as cryptographic code.
- **Readers and Writers (100 Methods)**: Input stream readers and output stream writers have to transform the date read from and written to their respective streams. One example found in three password safes is `org.spongycastle.bcpg.ArmoredOutputStream`, which performs Base64 encoding and CRC computations that have a high similarity to cryptographic code.

[12]https://code.google.com/p/smhasher/wiki/MurmurHash3

[13]http://www.json.org/

- **ISO 9796-1 Padding (25 Methods):** ISO 9796-1 is a padding scheme used for asymmetric ciphers. According to (Menezes et al., 1997), the message is padded, extended, and redundancy is added. Thus, despite being used for asymmetric cryptography, the padding itself has similarities to symmetric cryptography. The code of this padding can be found in Bouncy Castle, in `org.bouncycastle.crypto.encodings.ISO9796d1Encoding`.
- **Other Encoding Schemes (63 Methods)**: Similar to Base64 encoding, other encoding schemes like hex encoding included in Bouncy Castle utilizes an encoding table and shift operations. Also, the BER and DER encoding schemes, as well as some other notations found in the ASN.1 standard are labeled as cryptography.

4.4.4 Compression Functions (13 Methods)

The opcodes used for the compression process are similar to those used in cryptographic operations. The classifier identified a *ZIP* implementation included in one application, and *bzip2* implementations in two other applications.

4.4.5 Others (240 Methods)

These methods include implementations for calculating checksums, as well as other methods containing many mathematical operations similar to symmetric encryption mechanisms.

- **Checksums:** Similar to non-cryptographic hash functions, some checksum operations also have comparable structures. The classifier identified the Adler-32 algorithm in one of the applications.
- **False Positives:** Finally, the analysis also incorrectly classified a few methods, mainly concerned with many mathematical functions, as for example the method *findCornerFromCenter* found in the *MonochromeRectangleDetector*[14], which tries to find a corner of a barcode.

4.5 E4 – Password Safes (Asymmetric Classifier)

In this evaluation scenario the trained asymmetric classifier has been applied to the same 98 password safes as in E3. In total, 2,415,513 methods were analyzed by the classifier. 512 of these methods were identified by the classifier as asymmetric code. 64 of those methods were obfuscated. This leaves 448 methods subject to a detailed manual analysis, which reveals various categories presented in Table 4. In general, it can be observed that a majority of the detected methods are implemented in the Bouncy Castle library or its repackaged versions.

Classification	Category	%	Count	# methods
Asymmetric key cryptography	ECC	22.3	114	512
	GOST	15.6	80	
	Obfuscated	12.5	64	
	DSA	10.0	51	
	RSA	10.0	51	
	NaccacheStern	6.3	32	
	Key Agreement	5.5	28	
	SRP6	4.3	22	
	ElGamal	3.1	16	
	ECNR Signer	3.1	16	
	Others	3.1	16	
	BigInteger Math	2.3	12	
	NTRU	2.0	10	
Normal				2415001
Total				2415513

Figure 4: 512 out of 2,415,513 methods were classified as asymmetric cryptographic code, which can be categorized in various sub-categories.

- **RSA (51 Methods):** 8 applications included RSA code from Bouncy Castle. Also, two additional custom RSA implementations could be identified, which provide methods to perform RSA en- and decryption, as well as to sign content and to verify signatures. In addition, a test case with the name *TestRSA* has been detected by our analysis, as well as an RSA key pair generator.
- **DSA (51 Methods):** The identified DSA implementations are mainly related to (repackaged) Bouncy Castle libraries. However, there were also two custom implementations found in two applications. The first one can be found in Apache libraries, namely *SHA1withDSA_SignatureImpl*[15]. The second implementation is called *RawDSASignature* and implements a standard DSA algorithm.
- **ElGamal (16 Methods):** Similar to DSA, most ElGamal implementations are from repackaged Bouncy Castle libraries. Only one single other custom implementation has been detected, which offers methods for en- and decryption and signature creation/verification. Furthermore, a test case for this implementation has also been detected and a class to generate ElGamal key pairs is included in this custom implementation as well.

[14] https://code.google.com/p/zxing/source/browse/trunk/core/src/com/google/zxing/common/detector/MonochromeRectangleDetector.java?r=1003

[15] org.apache.harmony.security.provider.crypto.SHA1withDSA_SignatureImpl

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

- **Other Signature Algorithms:** Another class of methods found in Bouncy Castle are related to signing operations. The package `org.bouncycastle.crypto.signers` contains several of these signature algorithms. For the classes *DSASigner* and *GOST3410Signer* (based on the GOST R 34.10-94 Signature Algorithm) the two methods *generateSignature* and *verifySignature* have been labeled as asymmetric cryptography. For *ECDSASigner*, *ECNRSigner*, and *ECGOST3410Signer* the *generateSignature* methods have been detected. All of these methods really perform operations similar to asymmetric cryptography.
- **NaccacheStern Cryptosystem (32 Methods):** The public-key cryptosystem is based on the higher residuosity problem. This cryptosystem also uses RSA modulus n, which also consists of two large prime numbers multiplied together. The encryption process itself utilizes the Chinese Remainder Theorem. From the 98 password safes, 8 include the Bouncy Castle Naccache-Stern implementation, and for each of these implementations, several methods involved in the en- and decryption process, as well as key generation mechanisms have been detected.
- **Elliptic Curve Cryptography (114 Methods):** Since the Bouncy Castle ECC implementation relies heavily on the *BigInteger* type, these methods have been correctly classified as asymmetric-key cryptography.
- **SRP (22 Methods):** The *Secure Remote Password Protocol* (Wu, 1998) is a key exchange protocol that utilizes asymmetric key exchange. Bouncy Castle includes the SRP-6a protocol, which improves the original SRP-protocol. The classifier labeled several method of this implementation as asymmetric cryptography. A total of 6 password safes include methods from this SRP implementation. Furthermore, another password safe includes a custom SRP implementation, that operates on a custom *BigInteger* class, called *UBigHexInteger*.
- **Key Agreement (28 Methods):** Key-agreement protocols are used in order to create a shared key between two parties. One very common key-agreement protocol is the Diffie-Hellman key exchange. It is based on the same operations as RSA, and thus detected by the classifier. Furthermore, other key-agreement protocols, like Elliptic Curve Menezes-Qu-Vanstone (ECMQV), are detected as well. Again, 8 applications include the respective Bouncy Castle implementations, which have been detected. In addition, one application includes a custom SSL Diffie-Hellman implementation.
- **Zero-knowledge Proof (ZKP):** In order to implement zero-knowledge proofs, strategies similar to asymmetric encryption can be used (Schneier, 1996). One analyzed application includes such a custom ZKP implementation by Mozilla, called *JPakeCrypto*[16]. Three methods found in this class, namely *createZkp*, *checkZkp*, and *round2* have been labeled as asymmetric cryptography by the analysis. The operations used for creating and verifying the ZKP are very similar to asymmetric cryptography.
- **Fractions:** Another method labeled as asymmetric cryptography is a method called *addSub* implemented in `org.apache.commons.lang3.math.Fraction`. This implementation heavily utilizes mathematical operations based on *BigInteger* types for operations similar to asymmetric encryption algorithms.
- **Others:** Finally, 38 other methods have been tagged, 12 of which are part of BigInteger helper libraries. The other 26 methods also are several helper methods that operate on BigIntegers.

5 CONCLUSIONS

We have developed and evaluated Semdroid, a tool to facilitate Android app security analysis by identifying code implementing cryptographic functionality. We will now conclude by considering Semdroid's overall performance, the accuracy of its detection (false positives/negatives), its capabilities and its applicability for Android app security analysis.

Overall: Considering the fact rather small training data sets were used for the symmetric and asymmetric classifiers, the gained results are very promising. The application to the verified test sets and the password-safe applications reveals that the detector is capable in detecting cryptographic code with a high accuracy. Although many implementations are based on the standard Bouncy Castle library, we were also able to detect custom implementations of cryptographic functionality, which is especially important for the envisaged application analysis scenario.

False Negatives: Since the manual analysis of all methods within the 98 password-safe applications is not feasible, the empirical scenarios E3 and E4 do not

[16]http://dxr.mozilla.org/mozilla-central/source/mobile/android/base/sync/jpake/JPakeCrypto.java

let us draw detailed conclusions on the false negative rates. Conclusions on these rates can only be drawn from the evaluation results on the verified test sets. These results indicate a very low false negative rate, but due to the rather small number of verified cryptographic methods, more accurate false negative rates need to be gained in real application analysis projects.

False Positives: Here, the results from all evaluation scenarios can be taken into account. Due to the manual analysis of all detected crypto methods, we know in detail which type of method the detector is able to identify. When using strict definitions for the nature of cryptographic code, then methods related to Base64 encoding or checksum calculations could be considered as false positives. However, our experience with application analysis shows that such methods are often used in combination with real cryptographic code, and unfortunately, are sometimes used as security mechanism by developers. In that sense such methods have not been considered as false positives. The only real false positives were related to the implementation of mathematical operations.

Capabilities: By analyzing the gained results, we learn more about the capabilities of the detection system. This is especially important when analysing new applications where no a priori knowledge is available. Also, knowing the type of code the detector can find simplifies the analysis of obfuscated code, where we cannot rely on variable or method names to find out more about the implemented functionality.

Semdroid: The evaluation shows that the architecture of Semdroid related to method filters, model generation, instance generation etc. is flexible enough to be quickly adapted to heterogeneous analysis processes. Also, the deployment of the Semantic Patterns concept enabled us to evaluate a wide range of feature sets without the requirement to apply complex post processing steps.

Future Work: Since the gained results are very promising, we aim to use the crypto detection system in upcoming application analysis projects, and – where possible and reasonable – extend the machine learning based detection system to other application security aspects, such as key derivation functions or secure communication and management facilities.

REFERENCES

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11*, page 15, New York, New York, USA. ACM Press.

Cortes, C. and Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, 20(3):273–297.

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 73–84, New York, New York, USA. ACM Press.

Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. (2012). Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *CCS*, pages 50–61. ACM.

Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. (2012). The most dangerous code in the world. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, page 38. ACM Press.

Ghorbanzadeh, M., Chen, Y., Ma, Z., Clancy, T. C., and McGwier, R. (2013). A neural network approach to category validation of Android applications. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 740–744. IEEE.

Menezes, A. J., Oorschot, P. C. V., and Vanstone, S. A. (1997). *Handbook of Applied Cryptography*, volume 106.

Percival, C. and Josefsson, S. The scrypt Password-Based Key Derivation Function.

Provos, N. and Mazieres, D. (1999). A Future-Adaptable Password Scheme. *USENIX Annual Technical Conference, ...*, pages 1–12.

Schneier, B. (1996). Applied Cryptography. *Electrical Engineering*, 1([32):429–455.

Shabtai, A., Fledel, Y., and Elovici, Y. (2010). Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In *2010 International Conference on Computational Intelligence and Security*, pages 329–333. IEEE.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2011). Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190.

Teufl, P., Leitold, H., and Posch, R. (2013). Semantic Pattern Transformation. In *Proceedings of the 13th International Conference on Knowledge Management and Knowledge Technologies - i-Know '13*, pages 1–8, New York, New York, USA. ACM Press.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition (The Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann.

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. (2012). DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE.

Wu, T. (1998). The Secure Remote Password Protocol. In *Proceedings of the Symposium on Network and Distributed Systems Security NDSS 98*, pages 97–111. Internet Society.

SCIENCE AND TECHNOLOGY PUBLICATIONS