A machine learning approach to detect misuse of cryptographic APIs in source code

Gustavo Eloi de P. Rodrigues[1], **Alexandre M. Braga**[1], **Ricardo Dahab**[1],

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

g230218@dac.unicamp.br, alexbraga@ic.unicamp.br, rdahab@ic.unicamp.br

Abstract. *Cryptography is an indispensable tool for achieving security requirements such as software security. However, most software developers do not have enough knowledge regarding the proper use of cryptography and its APIs. This leads to incorrect use and exploitable vulnerabilities in software applications. Here, we propose an approach based on machine learning techniques to detect different kinds of cryptographic misuse in known java source code representations, achieving an average 52 percentage points improvement with respect to previous works.*

1. Introduction

With the increasing use of technologies and applications with stringent security requirements, such as integrity, confidentiality, authenticity, and data availability, cryptography has become a widely used tool to fulfill these objectives. However, most software developers responsible for building such applications have limited knowledge on how to properly use cryptographic primitives. Also, software libraries and application programming interfaces (APIs) offering cryptographic services are not easy to understand and use, and have limited documentation. All these factors inevitably lead to the incorrect use of cryptographic schemes and APIs [Lazar et al. 2014], and the introduction of software vulnerabilities during development [Braga et al. 2017]. Accordingly, and in a somewhat broader sense, we use the expression **cryptography (or cryptographic) misuse** to designate a bad programming practice that creates vulnerabilities and is also associated with design flaws and unsafe architectural choices [Braga et al. 2017].

In this context, most software companies rely on supporting tools to aid in the development of applications that use cryptography. Unfortunately, such tools are far from perfect: studies have shown that they can only detect, on average, 35% of cryptographic misuses with the combination of two or more tools [Díaz and Bermejo 2013, Antunes and Vieira 2014, Goseva-Popstojanova and Perhinschi 2015]. Thus, solutions that effectively help software developers incorporate cryptography in a simple and effective way in their applications are urgently needed [Nadi et al. 2016].

Static code analysis tools (SCATs) have been used with limited success in detecting cryptography misuses because they usually adopt pattern matching techniques. On the other hand, machine learning techniques for detecting such misuses, to the best of our knowledge, are scarcely reported in the literature. Accordingly, in this work, we present an application of existing machine learning techniques to build binary classifiers for the detection of cryptography misuse in java source code (which can be adapted to other

programming languages), according to different misuse categories, achieving on average 87% misuse detection. More specifically, we contribute with the following:

- an extension of a feature extraction technique, known as Bag of Graphs, to perform source code vectorization;
- the construction of machine learning models to detect cryptographic misuse with no need of massive amounts of data;
- an implementation that outperforms previously evaluated tools for cryptographic misuse detection by 52 percentage points on average (87% against 35%).

We believe that the use of machine learning in detecting cryptographic misuse is a promising approach and can be further explored with prompt results.

The remainder of this text is organized as follows. Section 2 introduces basic concepts and related work, while Section 3 explains the research methodology we used. Section 4 presents results and findings, which are further discussed in Section 5. Section 6 concludes the paper and points to future work.

2. Background and Related Work

This section gives background and presents related work to our research.

Braga and Dahab analyzed and classified cryptography misuse by collecting software developer contributions in related online forums [Braga and Dahab 2016]. In addition to online forums and literature review, their work was also based on industry initiatives. The resulting classification, shown in Table 1, is grouped in three columns: the first, **Groups**, aggregates misuses by their level in the software development cycle: architectural, design and coding issues; the second, **Categories**, specifies a misuse exact nature; and the third, **Sub-types**, further details the sub-types of each category. It was also observed that several types of cryptography misuses show up in pairs or triples. Braga and Dahab extended their work [Braga and Dahab 2017] with a longitudinal study of cryptography misuse in online communities, and showed that these are persistent and recurrent. Later, they built a cryptography misuse dataset and performed a benchmark of several static analysis tools [Braga et al. 2017, Braga et al. 2019]. They found out that only 35%, at most, of cryptography misuses contained in source code are detected by the combined use of the tools, all of which used pattern matching and simple data-flow analysis. Also, they enhanced their previous cryptographic misuse categorization dividing it into complexity groups.

To the best of our knowledge Fischer *et al.* [Fischer et al. 2019] are the first attempt at applying machine learning techniques to detect cryptography misuse in source code. The authors collected several cryptography misuses in Stack Overflow and built a deep neural network architecture to detect and classify the misuses based on categories defined by them. This neural network architecture achieved almost perfect classification metrics. However, the authors only covered simple cases of cryptography misuses which are already detected with pattern matching techniques used by SCATs.

As machine learning algorithms need numerical representation of data, code embedding (i.e, the transformation of source code into numerical representation) has been recently proposed. Alon *et al.* [Alon et al. 2019] built a deep learning neural network, called *code2vec*, that learns source code representation based on method calls and tries

Table 1. Classification of cryptography misuse in software. Adapted from [Braga et al. 2019]

Groups	Categories	Sub-types
Group 1: Code-level misuses	Weak Cryptography (WC)	- Risky or broken encryption - Proprietary cryptography - Determin. symm. encryption - Risky or broken hash/MAC - Custom implementation
	Coding and Implementation Bugs (CIB)	- Wrong configs for PBE - Common coding errors - Buggy IV generation - No cryptography - Leakage of keys
	Bad Randomness Handling (BRH)	- Use of statistic PRNGs - Predict., low entropy seeds - Static, fixed seeds - Reused seeds
Group 2: Design flaws	Program Design Flaws (PDF)	- Insecure behavior by default - Insecure key handling - Insecure use streamciphers - Insec. combo enc. w/ auth. - Insec. combo enc. w/ hash - Side-channel attacks
	Improper Certificate Validation (ICV)	- Absent validation of certs - Insecure SSL/TLS channel - Incomplete cert. validation - Absent host/user validation - Wildcards, self-signed certs
	Public-Key Cryptography (PKC) issues	- Deterministic encrypt. RSA - Insecure padding RSA enc. - Weak configs for RSA enc. - Insecure padding RSA sign. - Weak signatures w/ RSA - Weak signatures w/ ECDSA - Insecure DH or ECDH - Insecure elliptic curves
Group 3: Insecure architectures	IV and Nonce Management (IVM) issues	- CBC with non-random IV - CTR with static counter - Hard-coded or constant IV - Reuse nonce in encryption
	Poor Key Management (PKM)	- Short key, improper key size - Hard-coded or constant keys - Hard-coded PBE passwords - Key reuse in streamciphers - Use of expired keys - Issues in key distribution
	Crypto Architecture and Infrastructure (CAI) issues	- Issues in crypto agility - API misunderstanding - Multiple access points - Randomness source issues - PKI and CA issues

to classify such method calls. This is considered the state of the art in source code embedding. However, it represents an instance of source code as a collection of numerical vectors, one for each method call in the instance. This representation is not suitable to some classification tasks which require a one-to-one mapping of methods to vectors.

Several works in software engineering apply machine learning to source code analysis for vulnerability detection, software defect detection and similar tasks. In general, these works use an Abstract Syntactic Tree (AST) as source code representation, instead of text representation, before the source code embedding step. This method resulted more effective than natural language processing. The following are examples of this approach: the PROPHET [Long and Rinard 2016] system, that uses deep neural net-

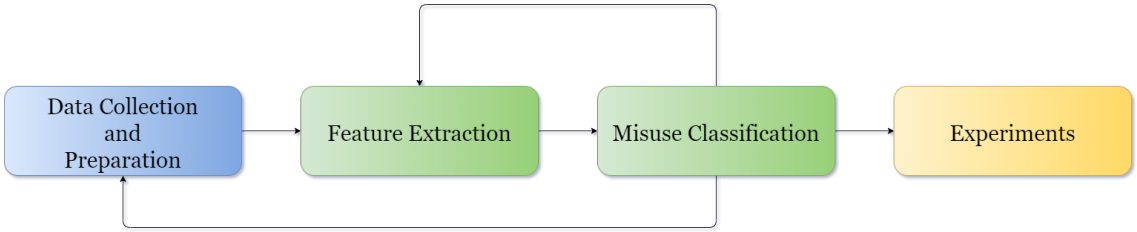

Figure 1. Work pipeline

works to detect differences in source code ASTs of different software patches in order to automatically generate correction patches; an LSTM model [Dam et al. 2018], based on trees, for the prediction of software defect using embeddings generated from source code ASTs; and a structure called AST-Ngrams [Shippey et al. 2019], used to build successful predictive models to detect software faults.

3. Research Methodology

This section describes the bulk of our methodology, step-by-step, summarized in Figure 1. It starts with the collection and preparation of the data used in our classifiers (in blue), then onto training and test configuration through feature extraction and misuse classification (in green), ending with experiments (in yellow). The following subsections detail the four steps, in order. All source codes developed and data used in this work will be available in https://gitlab.ic.unicamp.br/ra230218/mlmisusedetection.

3.1. Data Collection and Preparation

For data collection, we used datasets collected by Braga *et al.* [Braga et al. 2017] and Fischer *et al.* [Fischer et al. 2019]. The first one is a synthetic dataset, composed of source codes that emulate real world applications. The second dataset was created by collecting various source code snippets from programming forums. Thus, we were able to evaluate our classifiers in both synthetic (but realistic) data and real-world (but incomplete) data, resulting in a richer performance analysis. Both datasets are composed of Java source code that uses the Java Cryptography Architecture (JCA) [Oracle 2020] which is one of the most widely used cryptography API.

For each dataset, the distribution of instances among categories is presented in Table 2 and Table 3. In Table 2, column **Category** is the same as in Table 1; column **Secure** refers to the number of source codes in each category that are examples of correct use of the JCA cryptographic API; column **Insecure** gives the number of source codes that are examples of cryptographic misuse; and the last column, **Complexity**, ranks each category misuse into three levels of complexity, low, medium and high. In Table 3 we have analogous columns: **Category**, which gives the cryptographic misuse categories used by Fischer *et al.* [Fischer et al. 2019]; **Usage Pattern Definition**, which describes the tasks performed by the corresponding category; **Secure** and **Insecure** are analogous to Table 2, although we were able to use only a portion of the dataset, because it is composed of incomplete source codes (excerpts of source code that cannot be compiled) as well, and our technique requires complete (that can be compiled) pieces of code. The amount of actually used source codes is given by the two columns labeled **used**, whereas those labeled **full** list the full amount of source codes per category of the dataset.

Table 2. [Braga et al. 2017] dataset instances distribution by misuse category and misuse complexity.

Category	Secure	Insecure	Complexity
Weak Cryptography (WC)	10	20	Low
Poor Key Management (PKM)	32	19	High
Bad Randomness (BR)	8	12	Low
Program Design Flaws (PDF)	14	23	Medium
Improper Certificate Validation (ICV)	5	15	Medium
Coding and Implementation Bugs (CIB)	16	29	Low
Cryptography Architecture and Infrastructure (CAI)	7	8	High
Public-Key Cryptography (PKC)	58	68	Medium
IV/Nonce Management (IVM)	10	8	High

For source code representation, we decided to use ASTs (Abstract Syntax Trees) which are tree-shaped representations of tokens generated from expressions and declarations present in source code of a programming language. ASTs contain details about the structure of the source code and preserve its syntactic and semantic information [Mogensen 2017]. To generate an AST from code, we used *ANTLR4* [Parr 2013], which is a parser generator for reading, processing, executing or translating structured text or binary files. So, for every file present in the [Braga et al. 2017] dataset and in the [Fischer et al. 2019] dataset, we generated an AST representation and saved it in `.dot` file format. This is a commonly used file format for graph visualization and representation, which can properly store a graph's properties and structures.

3.2. Feature Extraction

In machine learning tasks, such as classification, clustering and others, there is the need to convert non-numerical input (in our case, ASTs) to a numerical representation, since machine learning algorithms can only handle numerical input. For this, we need a feature extraction and embedding method that can handle graphs, taking into consideration that our source code representation is a tree-shaped structure. At the same time, we also need that our method takes into account the content of ASTs nodes, as they contain text, lexical

Table 3. [Fischer et al. 2019] dataset instances distribution by misuse category.

Category	Usage Pattern Description	Secure (used)	Insecure (used)	Secure (Full)	Insecure (Full)
Cipher	Initialization of cipher, mode and padding	712	782	2764	2257
Key	Generation of symmetric key	384	325	1390	810
IV	Generation of IV	229	285	768	741
Hash	Initialization of cryptographic hash function	215	719	852	2487
TLS	Initialization of TLS Protocol	30	721	167	2319
HNV	Setting the hostname verifier	53	106	204	293
HNVOR	Overriding of hostname verification	6	71	28	180
TM	Overriding server certificate verification	51	443	73	1013

and structure information of the source code. We decided for the method of *Bag of Graphs (BoG)* [Silva et al. 2014]: this is a technique for the feature extraction and embedding of graphs that preserves the intrinsic structure and relationships present in the graph used as input [Silva et al. 2018]. It has been successfully used in malware detection in Android smartphones [Navarro et al. 2018].

We had to extend an existing BoG implementation to our use case. Figure 2 shows, step-by-step, our BoG implementation. As our digital object (source code) is already represented as a graph structure (AST), we can skip this part of the algorithm. After that, we define some nodes of interest (NoIs) that are used to build graphs of interest (GoIs) needed later on. In general, we used the following types of nodes which are source code structures present in the Java language definition: literal; variableDeclarator; methodInvocation_lfno_primary; methodInvocation; assignment; forStatement; basicForStatement; ifThenStatement; ifThenElseStatement; tryStatement. To adapt this to other programming languages than java, the only adjustment needed is that you need to generate ASTs with the corresponding language AST generator. Also the programming languages structures' names may differ from language to language.

Thereafter, from our nodes of interest, we built graphs of interest. We defined three types of GoIs:

- NoI source code text;
- Tree with the NoI as its root (AST structures labels and source code text);
- Shortest path from the root of the AST to a NoI.

After extracting GoIs from source code ASTs, we randomly sample a number of GoIs, since we need a fixed number of GoIs in the next step. We then apply a clustering step to create a codebook from the randomly sampled GoIs. For

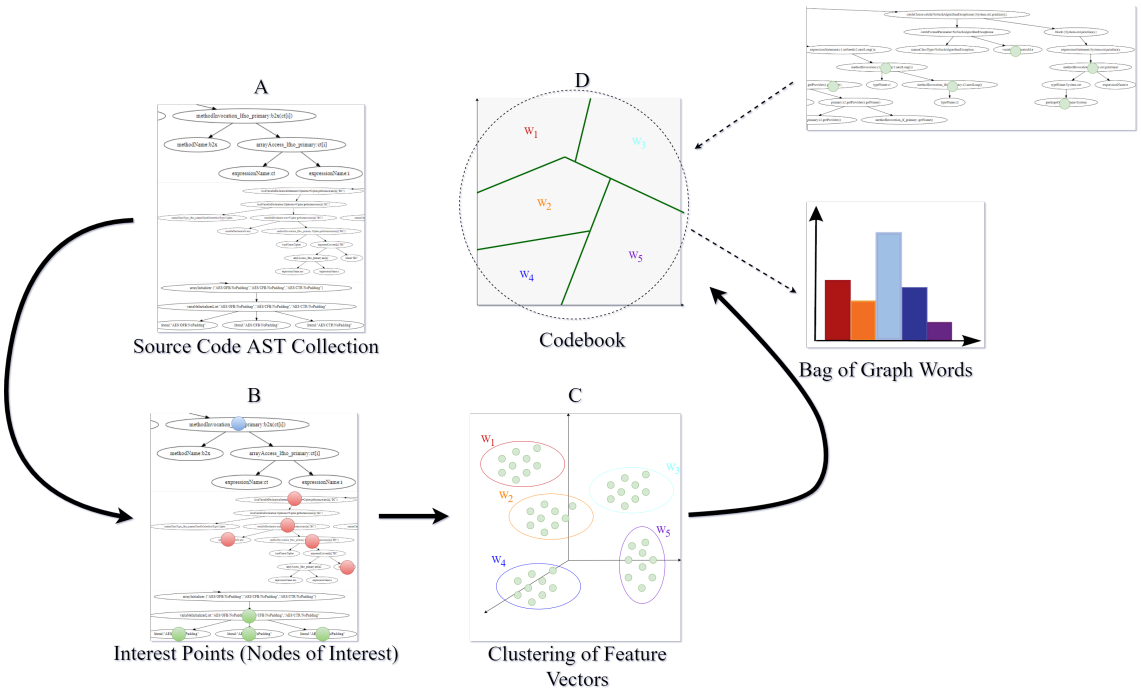

Figure 2. Bag of Graphs steps. Adapted from [Silva et al. 2018]

this, we use K-Means Clustering [Forgy 1965], which is a common clustering algorithm, with random centroids. Following the creation of a codebook, we can bag our GoIs by computing a histogram of the frequency of each codeword present in a source code. This histogram is the numerical representation of a source code that can be fed as input to a machine learning algorithm. All code in this part was written in Python 3 using the Networkx API [Hagberg et al. 2020] for graph manipulation and Scikit-Learn [Pedregosa et al. 2011] for K-Means. Also, the steps described in the following sections used the Scikit-Learn machine learning API to perform misuse classification.

3.3. Misuse Classification

Before using the numerical representation of our data as input to machine learning classifiers, we split the dataset into *train* and *test* datasets. We chose the 80/20 proportion, where 80% of the dataset comprised the train dataset, and 20% the test dataset. Note that there is no intersection between both datasets and the test dataset was used only once in our experiments, to get the final classification results.

In order to detect and classify misuses, we decided to build a binary classifier for each type of misuse, because each misuse has a respective good (correct) use, and, at the same time, misuses fall into different categories. As our datasets have labeled data, we were able to use a supervised classification approach. So, we trained different SVM (Support Vector Machine) classifiers for each type of cryptography misuse present in a dataset. We used a cross-validation approach to train and validate our classifiers, since our datasets are not big enough to make other approaches suitable. We also used GridSearchCV [Pedregosa et al. 2020] to search the best parameters of each classifier, which is a well used hyperparameter-tuning approach in machine learning tasks.

To evaluate our results, we used three widely used classification metrics for machine learning algorithms: Precision, Recall and F1-Score. We used different metrics to show the performance of our classifiers under different aspects. They are defined as follows:

$$\textit{Precision} \ = \ \frac{tp}{tp + fp}; \tag{1}$$

$$Recall \ = \ \frac{tp}{tp + fn}; \tag{2}$$

$$\textit{F1-score} \ = \ 2 * \frac{Precision * Recall}{Precision + Recall}. \tag{3}$$

Here, tp is the number of true positives (number of misuses correctly classified as misuses); fp is the number of false positives (number of good code misclassified as misuse) and fn is the number of false negatives (number of misuses misclassified as good code).

3.4. Experiments

As explained in 3.1, we have two datasets, with two different ways of categorizing cryptography misuse. So, we had to perform two distinct experiments, each with a different dataset. Both experiments followed the steps outlined in Section 3.2 and Section 3.3 for feature extraction and classification.

3.4.1. Experiment 1

For Experiment 1, we used the [Braga et al. 2017] dataset, with only 362 instances in total. This is a small dataset in comparison to the usual sizes in machine learning experiments. Given the limited amount of data, we had to select fewer types of NoIs in order to make our features less complex. Also, as previously stated, this dataset is composed of manufactured source codes that emulate real world applications, allowing us to evaluate our classifiers in a synthetic dataset setup. In this experiment, we selected the following types of NoIs and used our BoG implementation for feature extraction: literal; variableDeclarator; methodInvocation_lfno_primary; assignment. So, as the dataset has nine categories of cryptographic misuses, we built nine different binary classifiers, one for each type of misuse, and used the GridsearchCV [Pedregosa et al. 2020] to find the best classifier for each category of cryptographic misuse.

3.4.2. Experiment 2

For Experiment 2 we used the [Fischer et al. 2019] dataset, which is a larger dataset than that of Experiment 1, but with a different cryptographic misuse categorization. Naturally, this larger universe of instances, providing more data for both training and testing, gave us more confidence in the performance of our classifiers. Moreover, this dataset is comprised of real-world application source codes. We tried different combinations of types of NoIs (see Section 3.2); however, only the best were considered in the results. Finally, as the dataset has seven categories of cryptographic misuses, we built seven different binary classifiers, one for each type of misuse, and repeated the same procedure as in Experiment 1.

4. Results and Findings

This section presents results for the experiments described in the previous section. Results for Experiment 1 SVM classifiers, which use the [Braga et al. 2017] dataset, are shown in Figure 3. The results for Experiment 2, which use the [Fischer et al. 2019] dataset, are shown in Figure 4. Both images show results on test sets.

For Experiment 1, most of the classifiers achieved very good results when compared to [Braga et al. 2017] SCATs (Static Code Analysis Tools) results for F1-score, Precision and Recall, respectively: Weak Cryptography (86%, 75%, 100%); Poor Key Management (89%, 80%, 100%); Bad Randomness (100%, 100%, 100%); Improper Certificate Validation (100%, 100%, 100%); and IV/Nonce Management (100%, 100%, 100%). For the remaining four misuse categories, although better than those in [Braga et al. 2017], our results were not as good: Program Design Flaws (73%, 80%, 67%); Code Implementation Bugs (80%, 75%, 86%); Cryptography Architecture and Infrastructure (50%, 33.3%, 100%); and Public Key Cryptography (61%, 47%, 88%). We considered "poor" the results for the latter four categories, because they were the ones that had more false positives or false negatives given the Insecure/Secure instance distribution. Also, by checking the complexity of each cryptographic misuse in Table 2 and the previous results, both good and poor results are not well defined in the misuse complexity spectrum shown in Table 2 defined by [Braga and Dahab 2016], with good results (LOW:2; MEDIUM:1; HIGH: 2) and poor results (LOW:1; MEDIUM:2; HIGH:1). For

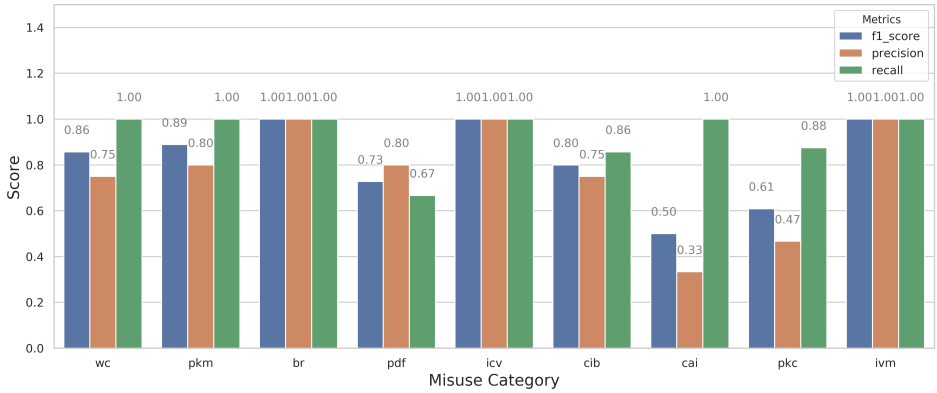

Figure 3. Results for Experiment 1

true positives (absolute number of cryptographic misuses detected), not shown in Figure 3, we have: Weak Cryptography (100%); Poor Key Management (100%); Bad Randomness (100%); Improper Certificate Validation (100%); IV/Nonce Management (100%); Program Design Flaws (66.6%); Code Implementation Bugs (85.7%); Cryptography Architecture and Infrastructure (100%) and Public Key Cryptography (87.5%). This represents an improvement of 55 percentage points compared to SCATs in misuse detection (90% of the misuses were detected) [Díaz and Bermejo 2013, Antunes and Vieira 2014, Goseva-Popstojanova and Perhinschi 2015].

Next, we present examples of classified source codes and illustrate how certain differences between them are hard to distinguish. All source codes are from the PKC misuse category. Listing 1 shows source code correctly classified as cryptography misuse (true positive). Listing 2 shows source code incorrectly classified as good (false negative). Finally, Listing 3 shows source code incorrectly classified as cryptography misuse (false positive).

In Listing 1, we have an instance of use of the RSA algorithm with insecure padding. In this example, the RSA key size used is secure (2048 bits) [Giry 2020]. However, in line 13, the algorithm does not use any padding, which is is a mandatory feature. In Listings 2 and 3, the issue is with the method parameter in line 5: in both cases (incorrectly classified), we are specifying an elliptic curve by its name; in Listing 2, the elliptic curve ”sect193r1” is insecure. In Listing 3, the elliptic curve specified is secure, but the machine learning model was not able to detect that difference. Both examples suffer from the same problem, that of distinguishing this parameter, which is the only difference between both source codes.

For Experiment 2, results are also shown for F1-score, Precision and Recall, respectively: Cipher (84%, 78%, 92%); Hash (81%, 89%, 74%); HNV (77% 88%, 68%); HNVOR (93% 93%, 93%); IV (77%, 88%, 68%); Key (79%, 89%, 71%); TLS (94%, 96%, 91%) and TM (96%, 99%, 94%). As such, HNVOR and TLS are the poorest results we obtained. The rest of the classifiers achieved good results, with Cipher and TM being the best ones. For true positive numbers, we have: Cipher (92.1%); Hash (74%); HNV (68.1%); HNVOR (93.3%); IV (68.2%); Key (71%); TLS (91%) and

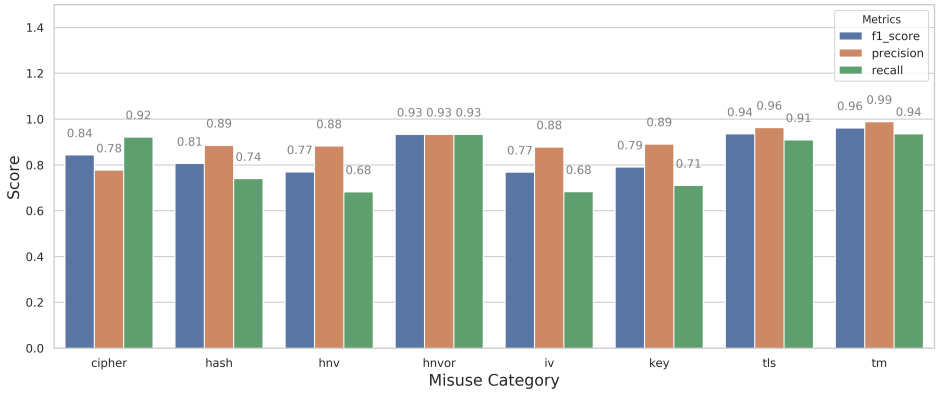

Figure 4. Results for Experiment 2.

TM (93.5%). This represents an improvement of 49 percentage points compared to SCATs in misuse detection (84% of the misuses where detected) [Díaz and Bermejo 2013, Antunes and Vieira 2014, Goseva-Popstojanova and Perhinschi 2015]. With this, averaging Experiment 1 and Experiment 2 results, we achieve an improvement of 52 percentage points compared to SCATs.

```
public final class InsecurePaddingRSA2 {

  public static void main(String args[]) {
    try {
      Security.addProvider(new BouncyCastleProvider()); // provedor BC
      byte[] msgAna = ("Cripto deterministica").getBytes();
      KeyPairGenerator g = KeyPairGenerator.getInstance("RSA", "BC");
      g.initialize(2048);
      KeyPair kp = g.generateKeyPair();

      U.println("Texto claro   : " + new String(msgAna));

      Cipher enc = Cipher.getInstance("RSA/None/NoPadding", "BC");
      enc.init(Cipher.ENCRYPT_MODE, kp.getPublic());
      Cipher dec = Cipher.getInstance("RSA/None/NoPadding", "BC");
      dec.init(Cipher.DECRYPT_MODE, kp.getPrivate());

      U.println("Encriptado com: " + enc.getAlgorithm());
      byte[][] ct = new byte[2][]; // ciphertext
      for (int i = 0; i < 2; i++) {
        ct[i] = enc.doFinal(msgAna);
        byte[] ptBeto = dec.doFinal(ct[i]);
        U.println("Criptograma   : " + U.b2x(ct[i]));
      }

    } catch (NoSuchAlgorithmException | NoSuchPaddingException |
           InvalidKeyException | IllegalBlockSizeException |
           BadPaddingException | NoSuchProviderException e) {
      System.out.println(e);
    }
  }
}

```

Listing 1. Example of cryptography misuse classified as misuse

```
public final class InsecureCurve_sect193r1 {

  public static void main(String argv[]) {
    try {
      ECGenParameterSpec ecps = new ECGenParameterSpec("sect193r1");
      U.println("EC parameters "+ecps.getName());

      KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
      kpg.initialize(ecps);
      KeyPair kp = kpg.generateKeyPair();

    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException|
          NoSuchProviderException e) {
      System.err.println("Error: " + e);
    }
  }
}

```

Listing 2. Example of cryptography misuse classified as good code

```
public final class SecureCurve_secp384r1 {

  public static void main(String argv[]) {
    try {
      ECGenParameterSpec ecps = new ECGenParameterSpec("secp384r1");
      U.println("EC parameters "+ecps.getName());

      KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
      kpg.initialize(ecps);
      KeyPair kp = kpg.generateKeyPair();

    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException|
          NoSuchProviderException  e) {
      System.err.println("Error: " + e);
    }
  }
}

```

Listing 3. Example of good code classified as cryptography misuse

5. Discussion of Findings

This section discusses the results presented in the previous Section 4, giving possible explanations and implications. We are, of course, aware of the limitations of our experiments, such as the amount of data in Experiment 1, class imbalance in datasets and others that will be further discussed. However, we argue that our experiments show promising results and interesting implications for future research.

In Experiment 1, for example, we notice that some cryptographic misuse categories occur in very few instances, thus hindering a proper cross-validation step and leading to not-so-good results from a machine learning perspective (like in the CAI misuse case). This occurs because the classifier is not fed enough data to be trained with a sufficiently large number of instances. In addition, some types of data are too complex to be distinguished based on few instances during the training step, leading to bad results. However, as stated in Section 4, our results still outperform common SCATs results shown in [Braga et al. 2017].

Another effect of the small dataset size in Experiment 1 is the overly optimistic results obtained in the BR, ICV and IVM categories. These categories achieved perfect results (100%) in all of the three metrics, which cannot be credited to a perfect classifier, as such a thing does not exist. However, these results indicate that our method can achieve good results in cryptography misuse detection.

An interesting aspect of Experiment 1 is that our results are spread over the complexity spectrum, i.e, we do not have 100% of good results in just one complexity category. This possibly means that our method extracted a variety of source code fea-

tures; however, those features are not specific for distinguishing between code complexity classes. Instead, they can distinguish code structures and code elements.

For Experiment 2, we see that our results are more "stable", as we have more instances in each category of the dataset. HNVOR category was the problem here, as we had a huge class imbalance (less than 10% of the instances were from the good class), that led our classifier to have 100% of false positives. However, the remaining categories presented good results, with the TM category reaching more than 90% in all metrics. As mentioned in Section 3.1, we could not use the dataset entirely because it contains incomplete source codes. However, the proportion of total instances per class is very similar, in most cases, to the proportion we were able to use. Therefore, it is fair to say that, were we able to use the full dataset, we would have likely obtained similar results, as we would have more instances to train and test respectively. In addition, our method does not need the same amount of data as a Deep Learning approach, which can be an advantage in a scenario where data is scarce.

With the results obtained in both of our experiments, we are confident in saying that our methodology shows promising (as well as similar) results in both synthetic and real-world data. Also, we argue that maybe synthetic source code and real world source are not so different, and we could use synthetic data to improve our datasets in the future, as in data augmentation, and, consequently, our classifier metrics.

We also realized, in Experiment 1, that most of the false positives and false negatives obtained in our classifiers' test were caused by the lack of similar source codes of the test sets in the training sets. As a result, some of the classifiers were not able to correctly classify some test instances. We experienced around 26% of misclassified code in Experiment 1, 21% of false positives but only 5% of false negatives. As an example we have Listing 3 (a false positive) that was the only code in the whole dataset that used the "secp384r1" as parameter (an insecure curve).

The final topic we would like to discuss is source code complexity. Here, we speculate that, in both our experiments, we did not achieve better results because of source code cyclomatic complexity (intrinsic dependencies within source code instructions, such as loops and conditionals, and data flow) contained in our data. By using an AST representation, it seems that it is not possible to extract all of the necessary information about cyclomatic complexity, even using AST nodes with loop information and conditionals.

6. Concluding Remarks

We are still far from a definitive solution to cryptography misuse detection. Nevertheless, the work presented in this paper has a few interesting and desirable features. First, it does not need a huge amount of training data while still achieving good results when compared to static analysis tools evaluated in literature. Moreover, although in some cases our classifiers cannot tell good from bad code, in most cases bad code is classified as bad code, and this is good from a security perspective. Also, most of the cases exhibit good balance between precision and recall, which means that our model's ability in finding relevant instances (i.e., tell when a piece of code contains a cryptography misuse or not) is well measured.

For future work, in addition to the AST structure in use, we intend to use alternative graph representations of source code that allow for more complete cycle information

and data flow information. This will enrich our feature extraction step and can lead to better classifiers. Also, we would like to enrich our dataset with more synthetic source codes which emulate a wider range of possibilities of use of cryptographic APIs. With this, we could evaluate the effect of data augmentation in source code context. Finally, we plan to build classifiers that can identify more than one misuse per source code, a likely event in real-world applications.

7. Acknowledgments

We thank CAPES for the financial support, grant number 88887.335984/2019-00. We thank Unicamp and LASCA (Laboratory of Applied Security and Cryptography) for the institutional support as well as Doctor Sandra Eliza Fontes de Avila for all support and good advice.

References

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

Antunes, N. and Vieira, M. (2014). Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. *IEEE Transactions on Services Computing*, 8(2):269–283.

Braga, A. and Dahab, R. (2016). Mining cryptography misuse in online forums. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 143–150. IEEE.

Braga, A. and Dahab, R. (2017). A longitudinal and retrospective study on how developers misuse cryptography in online communities. *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17), Brasília, DF, Brazil*.

Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2017). Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 170–181. IEEE.

Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2019). Understanding how to use static analysis tools for detecting cryptography misuse in software. *IEEE Transactions on Reliability*, 68(4):1384–1403.

Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T., and Kim, C.-J. (2018). A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*.

Díaz, G. and Bermejo, J. R. (2013). Static analysis of source code security: Assessment of tools against samate tests. *Information and software technology*, 55(8):1462–1476.

Fischer, F., Xiao, H., Kao, C.-Y., Stachelscheid, Y., Johnson, B., Razar, D., Fawkesley, P., Buckley, N., Böttinger, K., Muntean, P., et al. (2019). Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 339–356.

Forgy, E. W. (1965). Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *biometrics*, 21:768–769.

Giry, F. (2020). Keylenght - nist report on cryptographic key length and reccomendation. *URL: https://www.keylength.com/en/4/*.

Goseva-Popstojanova, K. and Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33.

Hagberg, A., Schult, D., and Swart, P. (2020). Networkx - network analysis in python. *URL: https://networkx.github.io/*.

Lazar, D., Chen, H., Wang, X., and Zeldovich, N. (2014). Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 7. ACM.

Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM.

Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.

Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946. ACM.

Navarro, L. C., Navarro, A. K., Grégio, A., Rocha, A., and Dahab, R. (2018). Leveraging ontologies and machine-learning techniques for malware analysis into android permissions ecosystems. *Computers & Security*, 78:429–453.

Oracle (2020). Java cryptography architecture (jca) reference guide. *URL: docs.oracle.com/javase/8/docs/technotes/guides/ /security/crypto/CryptoSpec.html*.

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2020). Scikit: Tunning the hyper-parameters of an estimator. *URL: https://scikit-learn.org/stable/modules/grid_search.html*.

Shippey, T., Bowes, D., and Hall, T. (2019). Automatically identifying code features for software defect prediction: Using ast n-grams. *Information and Software Technology*, 106:142–160.

Silva, F. B. et al. (2014). Bag of graphs= definition, implementation, and validation in classification tasks. *URL: http://repositorio.unicamp.br/handle/REPOSIP/275527*.

Silva, F. B., Werneck, R. d. O., Goldenstein, S., Tabbone, S., and Torres, R. d. S. (2018). Graph-based bag-of-words for classification. *Pattern Recognition*, 74:266–285.