Detecting Cryptography Misuses With Machine Learning: Graph Embeddings, Transfer Learning and Data Augmentation in Source Code Related Tasks

Gustavo Eloi de Paula Rodrigues, Alexandre M. Braga, and Ricardo Dahab

***Abstract*—Cryptography is a ubiquitous tool in secure software development in order to guarantee security requirements in general. However, software developers have scarce knowledge about cryptography and rely on limited support tools that cannot properly detect bad uses of cryptography, thus generating vulnerabilities in software. In this work, we extend the scarcely use of machine learning to detect cryptography misuse in source code by using a state of the art deep learning model (i.e., *code2vec*) through transfer learning to generate features that feed machine learning models. In addition, we compare this approach to previous ones in different types of binary models. Also, we adapt code obfuscation to serve as data augmentation in machine learning source code related tasks. Finally, we show that through transfer learning *code2vec* can be a competitive feature generator for cryptography misuse detection and simple code obfuscation can be used to generate data to enhance machine learning models training in source code related tasks.**

***Index Terms*—Code obfuscation, cryptography misuse, data augmentation, machine learning, misuse detection, transfer learning.**

I. Introduction

SOFTWARE services are consumed daily by people in different areas of activity. In each of these areas, security is an aspect that underlies services in order to protect users' data in general. On modern software, in general, cryptography is the tool used to achieve data protection. It provides requirements such as confidentiality, integrity, and authenticity, that cannot be achieved without the use of cryptography primitives in software development. However, the use of cryptography is not a simple task. Most software developers have limited knowledge of cryptography and do not use it properly. Also, most of the cryptographic application programming interfaces (APIs) have poor usability, with a documentation that is difficult to understand and to use [1], [2]. To address this problem, software development companies rely on software development supporting tools in order to detect incorrect use of cryptography. These tools, though, are limited and can only detect at most one-third of incorrect uses of cryptography (or, *cryptography misuses*) when including complex misuse cases [3] with little difference in performance when including only simple cases [4], [5]. With all this, these misuses persist in source code and, often, vulnerabilities are introduced. For example, it is estimated that most of android applications have at least one cryptography misuse in their source code [6]. These vulnerabilities can be exploited and bring damage to software companies and users as well. That said, there is an urgent need to improve cryptography misuse detection tools, as they perform an important role supporting secure software development. This will result in a reduction in security breach incidents as there will be fewer vulnerabilities to exploit. Finally, with robust cryptography misuse detection tools, software developers that do not have proper knowledge of cryptography will still be able to write secure software without the need for an expert all the time.

Recent work shows a trend of machine learning support in addressing the problem of detecting cryptography misuse in source code, achieving very good results, with additional room for improvements [7]. In this work, we expand the use of machine learning techniques in cryptographic misuse detection tasks. More specifically, we extend previous works applying a state-of-the-art deep learning source code feature extraction approach (namely, *code2vec*) through transfer learning. We do this because previous works do not take advantage of deep learning in cryptographic misuse detection and deep learning techniques are widely known to produce very good results in a diversity of areas [8], [9], [10]. In addition, we also implement data augmentation applied to the context of our problem, through the use of obfuscated source code. To the best of our knowledge, this is the first time this approach is used and implemented in this scenario. To evaluate our method, we build supervised binary classifiers to detect cryptography misuses in both the context of one classifier per misuse type and one classifier that distinguishes between all misuses/good uses (ignoring misuse types). Finally, we also evaluate a recent common detection tool, namely *Coverity Scan*, in order to measure our improvements compared to previous works.

Manuscript received 7 November 2021; revised 19 April 2022 and 7 October 2022; accepted 30 December 2022. Date of publication 7 February 2023; date of current version 1 December 2023. This work was supported by CAPES under Grant 88887.335984/2019-00. Associate Editor: H. Madeira.*(Corresponding author: Gustavo Eloi de Paula Rodrigues.)*

Gustavo Eloi de Paula Rodrigues and Ricardo Dahab are with the Institute of Computing, University of Campinas - Unicamp, Campinas 13083-970, Brazil (e-mail: gueloi95@gmail.com; rdahab@ic.unicamp.br).

Alexandre M. Braga is with the CPQD Foundation, Telecommunications Research and Development Center, Campinas 13083-970, Brazil (e-mail: alexmbraga2007@gmail.com).

Source code and other useful information will be available at https://gitlab.ic.unicamp.br/ra230218/mlmisusedetectiongraph.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2023.3237849.

Digital Object Identifier 10.1109/TR.2023.3237849

A. Summary of Contributions

We show that the use of transfer learning through *code2vec* can outperform previous techniques in some misuse case detection and in general misuse detection. Also, we provide evidence that classifiers that use data augmentation in their training step tend to have higher recall scores when compared to regular training, thus resulting in fewer false negatives. In addition, we show that the use of machine learning in cryptography misuse detection still outperforms common detection tools.

B. Organization of This Article

The rest of this article is organized as follows. Section II presents theoretical background and related work. Section III explains the methodology used in this work. Section IV shows the results obtained in the experiments described in Section III. Section V discusses the implications of the results presented in Section IV. Finally, Section VI concludes the article.

II. Background and Related Work

In this section, we introduce concepts used in this work and we discuss related work. Section II-A presents the concept of *cryptography misuse*; Section II-B presents concepts on graph representations of source code; Sections II-C and II-D show the definitions of *Bag of Graphs (BoG)* and *node2vec* graph embeddings techniques; Section II-E presents applications of deep learning in source code-related tasks; Section II-F discusses concepts on data augmentation and its use in source code-related tasks.

A. Cryptography Misuse

As defined in [7], *cryptography misuse*, in a broad sense, is the improper use of cryptography-related code. Examples of cryptography misuse are the adoption of broken or obsolete algorithms such as message-digest algorithm 5 and Data encryption standard; the inclusion of hard-coded passwords or static seeds when generating keys or other sensitive cryptography material such as initialization vectors (IVs), message authentication codes (MACs); the use of pseudorandom number generators (PRNGs); the reuse of encryption keys in contexts where they must be single-used; elliptic curve (EC) related problems; public key infrastructure (PKI) and certification authority issues; transport layer security (TLS) and secure sockets layer (SSL) misuse; and so on. Here, we are interested in misuse of cryptography APIs. We chose Java cryptography architecture (JCA) because it is a stable and well established API, one of the most used cryptography APIs in industry and also in literature [6], [11], [12], [13]. Fig. 1 displays an example of insecure use of the Rivest–Shamir–Adleman (RSA) algorithm with no *padding*, shown in lines 13 and 15. *Padding*, in the context of cryptography, is a set of techniques that adds data to an incomplete block of data [14]. This is applied to a message prior to an encryption task, in order to avoid issues such as *malleability* in the case of RSA.

On the other hand, Fig. 2 shows an example of secure use of the RSA algorithm, now with the using of padding as showed

```
1   public final class InsecurePaddingRSA2 {
2
3     public static void main(String args[]) {
4       try {
5         Security.addProvider(new BouncyCastleProvider());
6         byte[] msgAna = ("Cripto deterministica").getBytes();
7         KeyPairGenerator g = KeyPairGenerator.getInstance("RSA", "BC");
8         g.initialize(2048);
9         KeyPair kp = g.generateKeyPair();
10
11        U.println("Texto claro   : " + new String(msgAna));
12
13        Cipher enc = Cipher.getInstance("RSA/None/NoPadding", "BC");
14        enc.init(Cipher.ENCRYPT_MODE, kp.getPublic());
15        Cipher dec = Cipher.getInstance("RSA/None/NoPadding", "BC");
16        dec.init(Cipher.DECRYPT_MODE, kp.getPrivate());
17
18        U.println("Encriptado com: " + enc.getAlgorithm());
19        byte[][] ct = new byte[2][];
20        for (int i = 0; i < 2; i++) {
21          ct[i] = enc.doFinal(msgAna);
22          byte[] ptBeto = dec.doFinal(ct[i]);
23          U.println("Criptograma   : " + U.b2x(ct[i]));
24        }
25
26      } catch (NoSuchAlgorithmException | NoSuchPaddingException |
27            InvalidKeyException | IllegalBlockSizeException |
28            BadPaddingException | NoSuchProviderException e) {
29        System.out.println(e);
30      }
31    }
32  }
33
```

Fig. 1. Source code with a misuse. Here, the RSA algorithm is used with insecure parameters.

```
1   public final class OAEP_2048x256_2 {
2
3       public static void main(String args[]) throws NoSuchAlgorithmException,
4             NoSuchPaddingException, InvalidKeyException, BadPaddingException,
5             IllegalBlockSizeException, NoSuchProviderException,
6             InvalidAlgorithmParameterException {
7
8       Security.addProvider(new BouncyCastleProvider());
9
10      int ksize = 2048;
11      int hsize = 256;
12      int maxLenBytes = (ksize - 2 * hsize) / 8 - 2;
13
14      KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "BC");
15      kpg.initialize(ksize);
16      KeyPair kp = kpg.generateKeyPair();
17
18      Cipher c = Cipher.getInstance("RSA/None/OAEPwithSHA256andMGF1Padding","BC")
        ;
19
20      Key pubk = kp.getPublic();
21      c.init(Cipher.ENCRYPT_MODE, pubk);
22      byte[] ptAna = U.cancaoDoExilio.substring(0, maxLenBytes).getBytes();
23      byte[] ct = c.doFinal(ptAna);
24
25      Key privk = kp.getPrivate();
26      c.init(Cipher.DECRYPT_MODE, privk);
27      byte[] ptBeto = c.doFinal(ct);
28
29      U.println("Chave publica: " + pubk);
30
31
32      U.println("Encriptado com: " + c.getAlgorithm());
33      U.println("Texto claro  da Ana: " + U.b2s(ptAna));
34      U.println("Criptograma (A-->B): " + U.b2x(ct) + ", bits " + ct.length * 8);
35      U.println("Texto claro do Beto: " + new String(ptBeto));
36  }
37 }
38
```

Fig. 2. Source code with a correct use of the RSA algorithm with secure parameters.

on line 18. This is the secure version of Fig. 1 source code. Without the use of padding in RSA, an attacker can transform the encrypted message into another encrypted message, leading to a known transformation of the plain message [14]. For example, if x is the amount of money to transfer to an account, an attacker could double this amount by incorporating another encrypted message into the original one. This is one of the consequences of the *malleability* property of the RSA [14].

Many authors agree on what constitutes cryptography misuse; however they may not agree completely on how to categorize a cryptography misuse. The works of Braga and Dahab and Braga et al. [2], [3], [12], [15] resulted in a taxonomy composed of an extended list of different types of cryptography misuse, shown in Table I: The **Groups** column lists misuse sources; column **Categories** further details these sources; and **Subtypes**

TABLE I
CLASSES OF CRYPTOGRAPHY MISUSE IN SOFTWARE (ADAPTED FROM [3])

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 Password Based Encryption (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/ EC - Insecure Diffie-Helman - Insecure elliptic curves
Group 3: Insecure architectures	IV and Nonce Management (IVM) issues	- Cipher Block Chaining (CBC) with non-random IV - Counter mode 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

are subgroups of misuses present in a given category. Braga and Dahab's works begin by mining cryptography misuses in specialized programming forums and categorizing these misuses by complexity. Also, they checked for the joint occurrence of two or more different instances of cryptography misuse. Next, they performed a longitudinal study, in order to analyze the behavior of developers towards the use of cryptography in software development. Finally, they evaluated the performance of static code analysis tools (SCATs) in supporting the detection of cryptography misuses in common coding tasks. SCATs are support tools that do not need the execution of the source code in order to evaluate it [15]. These tools can be used for different activities from linting to misuse detection, and work at most on the syntactic and semantic layers of compilation.

Several other works in the literature address the problem of cryptography misuse. In [16], the authors conducted a study that concludes that there are no significant security flaws on cryptography APIs, but, rather, it is how developers use them that introduce flaws. In [6], the authors downloaded 49 Android applications from Google Play in order to find possible cryptography misuses. They divide these into four categories (weak cryptography, weak implementations, use of weak keys, and use of weak cryptography parameters). They show that around 80% of those applications have some kind of misuse, and also propose countermeasures to reduce the impact of these issues. In [17], a web platform called *Crypto Explorer*, with the objective of teaching correct use of cryptography APIs, was presented. This platform includes real world secure and insecure examples of cryptography API usage, showing the importance of the correct use of cryptography in software. In [1] and [13], the authors presented *CogniCrypt*, a support tool that detects cryptography misuse based on predefined strict rules. This tool is very good in detecting simple misuses cases; however, it does not detect those with temporal aspects. Also, there is the need to create new rules every time a new misuse case is discovered, thus not learning from data.

All that said, we contribute to cryptographic misuse detection by taking advantage of machine and deep learning, building models that can successfully detect misuses with temporal aspects by learning from data, without the need to create new rules every time.

B. Graph Representations of Source Code

Programming languages' source codes are frequently viewed as just a sequence of text tokens that represent variable names, function calls, loop instructions, data structures, and many other source code intrinsic structures. However, treating source code simply as linear sequences of tokens, regardless of their syntax and structure, may not retain important characteristics of source code that can be useful in machine learning tasks. As noted by Alon et al. [18], there is a tradeoff between the degree of program analysis required to extract a representation, and the necessary effort needed by a machine learning model to learn this representation. By using a representation like simple text may result in a learning effort that is prohibitive given the amount of data required. On the other hand, using a representation that is too specific, requiring deeper program analysis, may lead to language-specific/task-specific models [18]. Thus, a representation with intermediate complexity is needed in order to preserve important information about the source code, but also remaining easily learned by machine learning models. Graph representations of source code are a good example of this.

Common graph representations of source code include abstract syntax trees (ASTs), control flow graphs, and program dependency graphs. These representations vary on the degree and type of information they contain about a source code. We focus on ASTs because they are a well known representation that contains syntactic and semantic information about source codes, such as library/API method calls, control flows, and data flows, which are used to classify cryptographic misuse [12]. With that

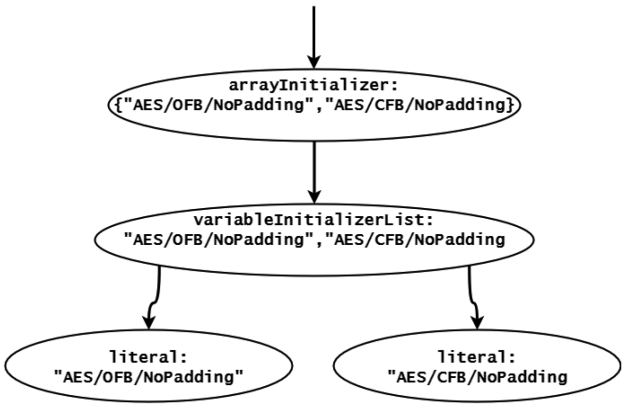

Fig. 3. Sample of an AST. Here the nodes contain the label of a source code grammar structure and the source code snippet related to it.

information, we can generate representative features of source code that can be used to feed machine learning models.

An AST is a tree-shaped representation of source code structures. It is comprised of tokens that are generated from declarations and expressions contained in the source code of a programming language. Such tokens vary from language to language according to the syntax. An AST describes the source code according to details within the source code structure and contents of variables, function calls, and others, as well as information that can be added later when considering compiled languages.

In an AST, each node represents a construction of the source code and, thus, syntactic and semantic information from the source code is preserved [19], as shown in Fig. 3. By construction, we mean the terminal and nonterminal symbols which characterize lexical elements of the grammar, that define the production rules of a formal grammar of programming languages [19].

C. Bag of Graphs

BoG [20] is a feature extraction and graph embedding technique that creates a vector representation of the local features and relationships present in a graph. This is achieved by creating a set of symbols known as *graph bag items* that represent elements or subgraphs that are part of the original graph [21]. This way, the intrinsic structure of the object (represented in graph form) and relationships present in that object are preserved. A general description of the *BoG* method is presented in [21].

Definition 1: Let $G = (V, E)$ be a graph and $w_i = f(G, i)$ a function that extracts a symbol defined as a sorted sequence $w_i = (p_1, ..., p_m)$ of elements $p_j \in G$ (elements of G are subgraphs, which, in particular, can be edges, nodes, or a combination of both). A *BoG* can be defined as a set of all symbols $B = \{w_1, ..., w_m\}$ extracted from G by f. The subgraphs $S \subset G$ can be described as a set of items $W \subset G$.

This technique was successfully used in feature extraction for two distinct contexts: Images [22] and Android permission files [21].

D. Node2vec

Node2vec [23] is a feature-generating graph embedding technique that tries to preserve the graph (or network) neighborhoods of nodes in a *d-dimensional* space. Based on a flexible neighborhood sampling strategy, this technique uses the concept of *random walks* that learn features from edges and nodes of a graph by interpolating between breadth-first search and depth-first search. A *random walk* is described as a walk of length l starting at vertex u and traversing the edges of the graph based on a transition probability π and a search bias α. Both of these parameters are used to guide the neighborhood sampling mechanism that generates node embeddings. The *search bias* influences the transitional probability that determines the next step of each walk, allowing the account of the network structure and the exploration of different neighborhoods in the graph [23].

E. Deep Learning and Code2vec

Deep learning is a field of artificial intelligence that uses artificial neural network models composed of many layers, with the objective of solving complex tasks that common machine learning algorithms cannot solve [9], [10]. The first models that were built using the deep learning approach were used in image classification and automatic speech recognition problems. Nowadays, deep learning can perform a range of different tasks, from transfer learning, neural style transfer, automatic source code generation, feature extraction, deep fakes, language translation, drug discovery, and many other applications from different fields [9], [10].

The use of deep learning in code-related tasks is recent. By code-related, we mean problems that use source codes (or a source code representation like an AST) as its initial input. Most of its applications are related to software engineering problems. Some applications are: Automatic patch correction [24]; software defect feature extraction [25]; software vulnerability detection [26], [27], [28], [29], [30]; software defect prediction [31]; and bug detection [8]. All of these applications use source code in a graph representation, instead of treating it as text and using natural language techniques to process it. It has been shown that common natural language process techniques are not well suited for machine/deep learning code-related tasks [18]. Alternative source code representations like ASTs as input to feature extraction methods perform better than using source code as text input.

1) Code2vec: Until 2019, there was no proposed deep neural network architecture for general source code feature extraction based on graph representations. *Code2vec* [18] is a deep neural attention model that represents source code snippets as code embeddings, i.e., continuous distributed vector representations that preserve semantic properties of source code represented as ASTs. This is a path-based approach that represents code snippets based on the number of methods contained within them. This approach can be summarized as follows.

1) Each method in a source code is parsed and an AST is constructed from it. Then, each AST is traversed and paths between its leaves are extracted, generating a tuple representation called *path-context*.

2) For each such path and its values in a *path-context*, a mapping to a numeric vector representation is built and an embedding is created. Then, three embeddings of each context are concatenated into a single one to represent a *path-context* as a vector.
3) Finally, a path-attention network using attention scores aggregates multiple path-context embeddings into a single vector that represents the body of the method. In *Code2vec*, this code vector is used to predict the method label. However, this architecture can also be used as a feature extractor by not using its final layer (the label prediction layer).

So, for a single source-code file, multiple embeddings can be extracted depending on the number of methods present in the file. Therefore, to generate a single vector representation of a source code using this neural network as a feature extractor, an aggregation mechanism is needed. One was proposed in [32].

F. Data Augmentation

Data augmentation is a set of techniques used in machine learning tasks during the training step of a model, with the objective of improving model generalization by reducing overfitting. These techniques work with the assumption that more information can be obtained from the original dataset by using augmentations [33]. One common use of data augmentation is *oversampling*, which adds to the training set synthetic created "copies" of its instances. This is mostly used in *computer vision* tasks by flipping, rotating, shearing, cropping, and zooming images of a small training set, with the objective of increasing its size depending on the problem's domain. The actual operations will depend on what makes sense in a solution. For example, if a model is trying to classify flowers, it does not make sense to add images of a flower rotated upside down. As in the real world, flowers are not often seen upside down. Every data augmentation technique assumes that the added objects follow the same semantics as the original ones [33].

In source-code machine learning-related tasks, to the best of our knowledge, there is no standard way of applying data augmentation to training sets. One way was suggested in [32]. The authors suggested using *obfuscated source code* as a data augmentation operation in source codes. *Code obfuscation* is a technique used by programmers with the objective of making source codes difficult to read/understand, thus guaranteeing security by obscurity, or hiding malicious code. Some obfuscation operations include: Indentation removal, variable name changing to meaningless names, introduction of null operations, and many others. With all of these operations, it is possible to create copies of a source code and apply data augmentation, as done with images in computer vision tasks.

III. Methodology

In this section, we briefly present the methodology used in this work. First, we discuss the process of data collection and data preparation to make our data suitable for machine learning tasks. Then, we describe the feature extraction techniques applied to our data in order to transform it into numeric form, which is then used during the process of training and testing of the classifiers. Last, we describe the four experiments that were done in order to compare feature extraction techniques, evaluate the impact of data augmentation, and briefly evaluate an updated SCAT.

TABLE II
Dataset Instances Distribution by Misuse Category[34]

Category	Usage pattern description	Secure (used)	Insecure (used)
Cipher	Initialization of cipher, mode and padding	712	782
Key	Generation of symmetric key	384	325
IV	Generation of IV	229	285
Hash	Initialization of cryptography hash function	215	719
TLS	Initialization of TLS protocol	30	721
HNV	Setting the hostname verifier	53	106
HNVOR	Overriding of hostname verification	6	71
TM	Overriding server certificate verification	51	443

A. Data Collection

For Data Collection, as there is no standard cryptography misuse dataset, we decided to use the same dataset used in [34]. Table II shows the cryptography misuse categorization of the dataset, as well its description and ratio of secure and insecure code with respect to the entire code. **Category** is the type of cryptography misuse; **Usage pattern description** is the description of the misuse category; **Secure (used)** is the number of good code instances used; **Insecure (used)** is the number of misuse code instances used. Also, we used the same dataset because we extended some of the experiments present in [7]. This dataset is composed of Java JCA source code snippets collected from *stack overflow* threads. JCA is one of the most used and evaluated cryptography APIs in the literature. As a number of snippets are not complete and thus, noncompilable code, we could not use the full original dataset in our experiments. Only the source codes that were compilable were used, and thus the ones that have an AST representation.

B. Data Preparation

For data preparation, we have to transform our source codes into a graph representation that will be used as input for the feature extraction step. We use the software *ANTLR4*, which is a parser software that can generate ASTs based on a programming language grammar definition. With this, we generate a `dot` file with the corresponding AST representation of a source code. For *code2vec*, we only need to pass the source code as input to the network. The *code2vec* architecture has a parser in its implementation that already generates an AST from the source code.

C. Feature Extraction

We have to divide the feature extraction step in two parts, one for graph feature extraction techniques (*BoG* and *node2vec*), and the other for transfer learning using *code2vec* neural networks. In the first one, we use *BoG* and *node2vec* separately and generate two distinct datasets, one for each feature extraction

technique. The step-by-step application of the *BoG* technique is as follows: First, we select nodes of interest (*NoI*) in the AST based on the label of a node, which corresponds to a structure defined by the language grammar (for example, *literal* in Java). Next, for each *NoI* selected in the previous step, we generate three graphs of interest (*GoI*) as follows:

1) the *NoI* itself;
2) the shortest path from the root of the AST to the *NoI*;
3) the tree generated from the *NoI*.

After that, we generate a hash representation for each of these graphs and concatenate the three vectors. It is worth noting that we have to do a random sampling of the selected *NoIs* in order to limit their number. Next, we use a clustering algorithm (in this case, *K-Means*) to generate our codebook of representations, where each concatenated *GoI* is associated to a feature vector. Finally, we calculate the histogram of the frequency of feature vectors per source code and generate a representation for it.

For *node2vec*, the step-by-step application of this technique is as follows: first, we label each AST node with its content, so as to be able to use *node2vec* properly, taking into account the text attributes of the graph. Next, we apply *node2vec* to each AST source code file. The algorithm generates a node embedding for each node in the AST graph, and we select only node embeddings based on the node label as a filtering mechanism. Next, we perform a random sampling on the filtered node embeddings in order to limit the number of node embeddings per file. Finally, we repeat the same final steps of the BoG application. We use *K-means* to produce the embeddings codebook and we calculate a histogram per file.

For *code2vec*, we apply *transfer learning* to generate code embeddings for each source code file. *Transfer learning* is a technique used in machine learning applications where a pretrained neural network is used as a feature extractor, generating embeddings that represent the object used as input of the network. We used the pretrained *code2vec* model presented in [35]. We also used its aggregation pipeline, as *code2vec* generates multiple embeddings per file.

D. Training and Testing of Classifiers

Now, we describe our machine learning models' training and testing steps. It is worth noting that we use 80% of the dataset for the training step and 20% for the testing step, with no intersection between the splits. The test dataset is used only once to obtain the final results. We use binary *support vector machines (SVM)* and *multilayer perceptron (MLP)* classifiers.

1) Hyperparameter Tuning and Model Selection: For the training step, we use *k-fold cross-validation*. This technique is well-known and used in the context of machine learning applications in the training step to evaluate a trained model without the explicit use of a validation dataset before hyperparameter tuning. We use *GridsearchCV* for hyperparameter tuning and model selection. This is an automated process, which incorporates cross-validation and searches for the best parameters and models giving a list of predefined parameters.

2) Data Augmentation Using Obfuscated Code: For one of the experiments in this work, we use data augmentation with obfuscated code. We use the obfuscation tool presented in [35], with the *random obfuscation* technique, where the names of the variables of a source code are changed to random names. We apply data augmentation to our training sets by adding obfuscated versions of the files in the train set, up to the point where the less representative class reaches the size of the most representative. We basically try to balance our training set, if possible, never letting the less representative class become the most representative.

E. Experiments

In this section, we describe the experiments that we have performed in our work. All of the experiments use the setup described in the previous sections for data preparation, feature extraction, and training/testing of classifiers.

1) Experiment 1: In Experiment 1, we extend the experiments in [7] and we use the set generated by the *code2vec* network feature extractor. We train and test models for each type of misuse, and we compare the results obtained by models trained with *code2vec* features against the results obtained in [7] and [15].

2) Experiment 2: In Experiment 2, we extend both Experiment 1 and the experiments in [7] by adding the data augmentation step during the training step for all types of generated sets (*BoG*, *node2vec*, and *code2vec* sets). We train and test models for each type of misuse, and we compare the results obtained by the models with data augmentation, with the results obtained in Experiment 1.

3) Experiment 3: In Experiment 3 we use a different approach, in the sense of how models are created. Instead of training a binary model for each type of misuse, we train a binary model that tries to distinguish between misuse and secure code. We generate models with data augmentation and with no data augmentation for the three types of feature extractors. We compare the obtained results with the previous experiments and with previous works.

4) Experiment 4: In Experiment 4, we extend the analysis of Braga et al. [15] by evaluating a tool called Coverity Scan [36] that can be used to detect cryptography misuse. For this evaluation, we use its free version. We wanted to perform this experiment to check if there is any improvement in SCATs over the previous evaluations in the literature. Also, Coverity Scan incorporates other SCATs in its analysis, so we do not need to evaluate all of them. In addition, this tool also incorporates both dynamic and static analysis to detect vulnerabilities in source code. So, to be able to use this tool, we need compilable source codes, their libraries and dependencies that the source code uses within it. We used the same dataset as Braga et al. [15], which uses the categorization in Table I, instead of the dataset used in previous experiments, as later dataset libraries and project dependencies are intractable.

IV. Results

In this section, we present the results obtained in all of the experiments described in Section III-E. Experiment 1 is the evaluation of *code2vec* compared to other feature extraction

TABLE III
RESULTS FOR EXPERIMENT 1 AND EXPERIMENT 2 PART ONE

	Misuse classification											
	Cipher			Hash			Key			IV		
Metrics (%) / **FeM**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**
BSVM	78	92	**84**	89	74	81	89	71	79	88	68	**77**
BMLP	82.9	87.1	85	88.5	89.7	89.1	75.3	75.3	75.3	82.7	76.1	79.3
NSVM	73	73	73	90	83	86	74	70	72	67	60	63
NMLP	74	75.9	75	85.5	93.1	89.1	73.4	68.1	70.6	62	57.1	59.5
CSVM	81.2	77.14	79.1	85.5	76.7	80.9	83.3	68.4	75.1	70.2	67.3	68.7
CMLP	80.6	86.4	83.4	86.2	86.2	86.2	85.4	72.6	78.5	61.4	71.4	66
BSVM-A	80.1	80.7	80.4	86.6	89.0	**87.8**	80.5	84	**82.2**	87.8	57.1	69.2
BMLP-A	80.9	87.8	84.2	88.8	87.6	88.2	77.3	84	80.5	88.8	76.1	82
NSVM-A	74.5	74.0	74.2	89.2	85.6	87.4	66.6	81.1	73.2	70.4	49.2	57.9
NMLP-A	75	74.0	74.5	90.6	92.4	91.5	70.8	73.9	72.3	63.8	47.6	54.5
CSVM-A	77.7	87.1	82.1	77.7	96.5	86.1	83.3	68.4	75.1	72.5	75.5	74
CMLP-A	80.1	83.5	81.8	85.9	84.4	85.2	85.4	72.6	78.5	63.7	75.5	69.1

Note: For each FeM (with data augmentation or not), in each Misuse category, Precision, Recall, and F1-Score Metrics are presented. Here, Cipher, Hash, Key, and IV categories results are presented. The best results considering only SVMs (one binary classifier for each misuse category) are in bold.

TABLE IV
RESULTS FOR EXPERIMENT 1 AND EXPERIMENT 2 PART TWO

	Misuse classification											
	HNV			HNVOR			TLS			TM		
Metrics (%) / **FeM**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**
BSVM	88	68	77	93	93	93	96	91	94	99	94	96
BMLP	88.2	68.1	76.9	100	80	88.8	95.7	94.4	95	97.6	89.2	93.2
NSVM	67	67	67	0	0	0	96	90	93	95	87	91
NMLP	80	80	80	92.8	92.8	92.8	96.2	92.8	94.5	92.5	94.9	93.7
CSVM	88.2	71.4	78.9	92.3	80	85.7	95.9	91.4	93.6	96.6	95.6	96.1
CMLP	95	90.4	92.6	93.7	100	96.7	96.8	94.5	95.6	96.7	97.8	97.2
BSVM-A	74	90.9	81.6	93.7	100	**96.7**	95	93.7	94.3	100	89.2	94.3
BMLP-A	78.9	68.1	73.1	93.7	100	96.7	95.1	95.1	95.1	95.5	91.3	93.4
NSVM-A	58.8	66.6	62.5	93.3	100	96.5	96.7	84.2	90	94.8	93.6	94.2
NMLP-A	80	80	80	92.3	85.7	88.8	96	87.1	91.3	90.9	88.6	89.7
CSVM-A	85.7	85.7	**85.7**	93.75	100	**96.7**	95.3	95.3	**95.3**	96.7	97.8	**97.2**
CMLP-A	88.2	71.4	78.9	91.6	73.3	81.4	95.9	92.2	94.0	96.6	95.6	96.1

Note: For each FeM (with data augmentation or not), in Each Misuse category, Precision, Recall, and F1-Score Metrics are presented. Here, HNV, HNVOR, TLS, and TM categories results are presented. The best results considering only SVMs (one binary classifier for each misuse category) are in bold.

methods. Experiment 2 is the analysis of the impact of data augmentation in different configurations of classifiers. Experiment 3 evaluates all feature extraction methods and data augmentation in the context of a binary classifier for all misuses. Finally, Experiment 4 is the evaluation of the Coverity Scan tool. Both Table III and Table IV contain the results of Experiments 1 and 2, where Table IV is the continuation of Table III; also, each of these contains the results for four types of misuses. Table V contains the results for Experiment 3, where we do not have a misuse categorization, as in this experiment we just divided the whole dataset into misuse and good uses, regardless of the type. In all of the tables, we have the following columns: 1) **feature extraction method** (FeM) which is the feature extraction method used (i.e., B for *BoG*, N for *node2vec*, or C for *code2vec*) plus the classifier used (i.e., *MLP* for multilayer perceptron and *SVM* for support vector machines), followed by "A", if data augmentation was used. For example, "BSVM-A" stands for *Bag of Graphs* ("B") using a *SVM* classifier ("SVM") with data augmentation ("A"); 2) **metrics**, which describes the metrics used in the experiment, i.e., **Prec**, **Rec**, and **F1**, which are Precision, Recall, and F1-Score,

TABLE V
RESULTS FOR EXPERIMENT 3

FeM	**Metrics** (%)		
	Prec	**Rec**	**F1**
BSVM	83.9	75.5	79.1
BMLP	78.7	83.9	81.2
NSVM	77.7	77.6	77.7
NMLP	83.8	77.5	80.5
CSVM	87.8	70.1	78
CMLP	81.7	81.4	81.5
BSVM-A	79.5	85.0	82.2
BMLP-A	78.3	83.4	80.8
NSVM-A	80	78.8	79.4
NMLP-A	80	78.8	79.4
CSVM-A	79.6	86.4	**82.9**
CMLP-A	80.5	81.4	80.9

Note: For each FeM, Precision, Recall, and F1-Score Metrics are presented. Here, the results were obtained from binary classifiers that distinguish between misuse/good use (with no misuse categorization).

respectively [9]. Here, we are using two classifiers only to verify the impact of data augmentation using obfuscation code in misuse detection. We are not comparing two types of classifiers.

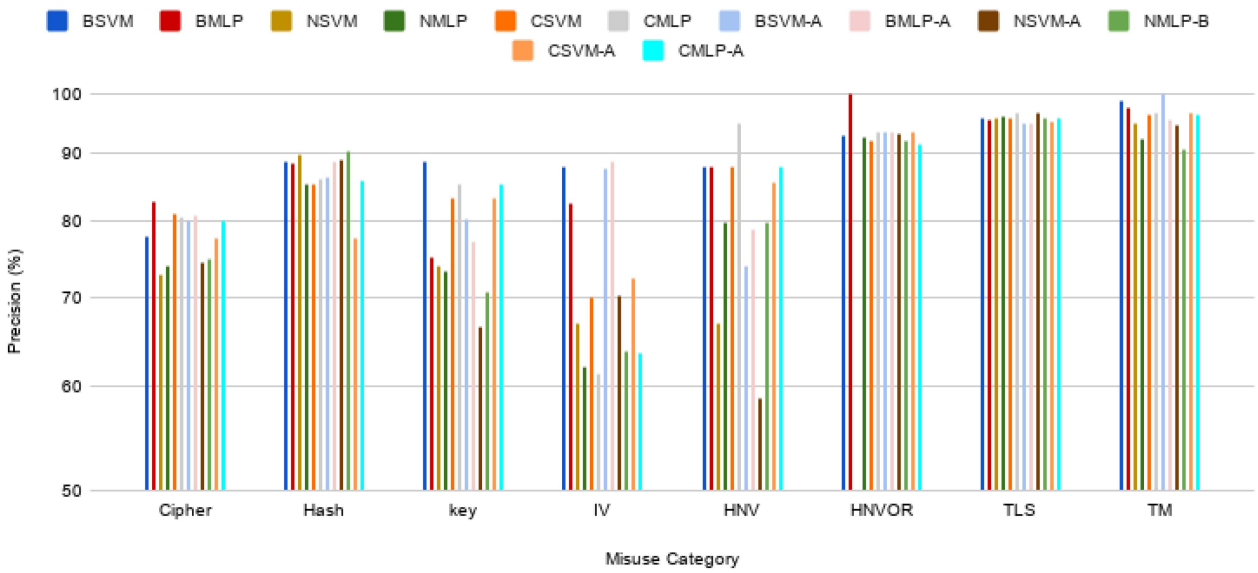

Fig. 4. Precision results for Experiments 1 and 2. Each column color represents a configuration setting of the feature extraction method, classifier (one binary classifier for each misuse category), and data augmentation (optional).

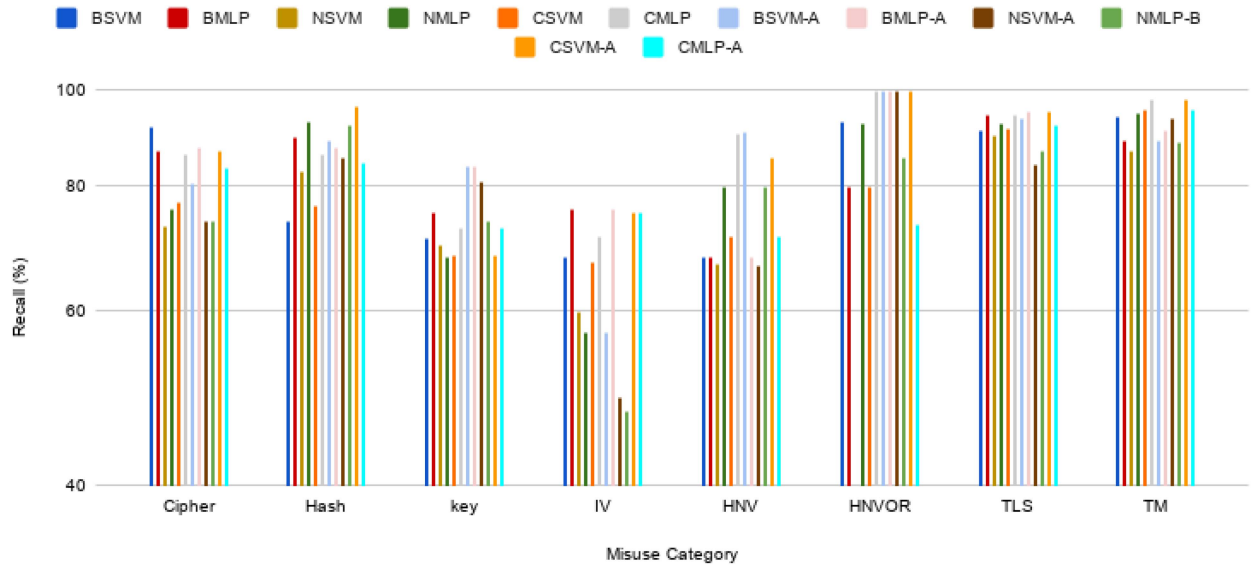

Fig. 5. Recall results for Experiments 1 and 2. Each column color represents a configuration setting of the feature extraction method, classifier (one binary classifier for each misuse category), and data augmentation (optional).

Also, Figs. 4, 5, and 6 present a graphical comparison of all feature extraction techniques in Precision, Recall, and F1-score, respectively.

We first discuss Experiment 1, whose results are shown in Tables III and IV. We show Precision, Recall, and F1-score for *code2vec* combined with CSVM, the SVM classifier (we are not taking into account the MLP classifier, as the work in [7] only uses SVMs). We obtain the following outcomes for the three metrics: Cipher (81.2%, 77.14%, 79.1%); Hash (85.5%, 76.7%, 80.9%); Key (83.3% 68.4%, 75.1%); IV (70.2%, 67.3%, 68.7%); HNV (88.2%, 71.4%, 78.9%); HNVOR (92.3%, 80%, 85.7%); TLS (95.9%, 91.4%, 93.6%); and TM (96.6%, 95.6%, 96.1%). Here, we chose to present our comparison using the F1-score metric because we need classifiers that have good performance in both Recall and Precision. Also, F1-score is the harmonic mean of these. By comparing these results with *BSVM*, regarding their F1-Score, we notice that *CSVM (code2vec)* only outperforms it at the HNV class (the first and fifth rows of the tables), and performs virtually the same in most of the remaining classes. By comparing *CSVM* with *NSVM*, we verify that *CSVM* only underperforms it in the *Hash* class and performs equally or better in the remaining classes. This shows that *code2vec* is a FeM that can be used to detect different categories of cryptography misuse.

Now, Experiment 2, whose results are shown in Tables III and IV, highlights the effect of data augmentation in both *MLP* and

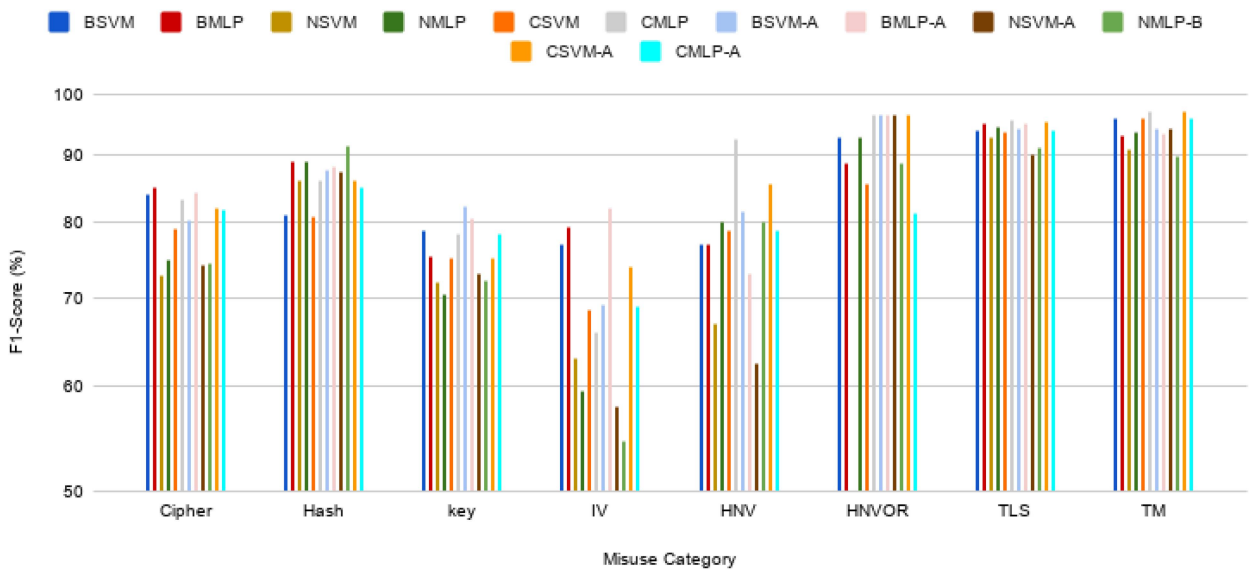

Fig. 6. F1-Score results for Experiments 1 and 2. Each column color represents a configuration setting of the feature extraction method, classifier (one binary classifier for each misuse category), and data augmentation (optional).

SVM classifiers. Here, we compare FeMs of the first six rows with their augmented ("A") version, as in BSVM with BSVM-A, *bag of graphs* with *SVM* classifier. We are also using F1-score as metric of comparison. In the case of *BoG*, the use of data augmentation improves misuse detection in the cases of *Hash*, *key*, *HVN*, and *HNVOR*; it underperforms in the cases of *Cipher* and *IV* and performs equally for *TLS* and *TM*. For *node2vec*, the use of data augmentation improves the misuse detection in *Hash*, *key*, *HNVOR*, and *TLS*. It underperforms in *IV* and *HNV*, and performs equally in *Cipher* and *TM*. For *code2vec*, the use of data augmentation improves misuse detection in *Cipher-svm*, *Hash-mlp*, *IV*, *HNV-svm*, and *HNVOR-svm*. It underperforms in *Cipher-mlp*, *Hash-svm*, *HNV-svm*, *HNVOR-mlp*, *TLS-mlp*, and *TM-mlp*; and performs equally in the remaining cases. In summary, for *BoG*, six out of eight categories have the same or better results when data augmentation is used. For *node2vec*, we also have six out of eight categories with equal or enhanced results. In *code2vec*, our analysis uses higher granularity, considering classifier types too, as the differences were more subtle. From a general perspective, in *code2vec* data augmentation makes virtually no difference, as the number of categories that benefit from it in the F1-score is the same compared to those that do not. On the other hand, we notice a difference when looking at the type of classifiers, as data augmentation tends to work better with *SVM* and worse with *MLP*. We could argue that those classifiers have distinct approaches in the way they are trained. In general, data augmentation makes classifiers perform equally or better.

Let us now consider Experiment 3, whose results are shown in Table V. In the case of classifiers without augmentation, *BoG* and *code2vec*'s performances are virtually the same on average, if we use the F1-score as a metric of comparison. *Code2vec* performs better if we use Precision as a comparison metric, and *BoG* performs better if we use Recall as the metric. The same pattern arises in the case of classifiers using data augmentation. However, we note that both *code2vec* and *BoG* have better Recall results and worse (but still good) Precision results. In general, the classifiers that use data augmentation attain better results with an improvement of 1.4 percentage points in average, with *code2vec* having the biggest improvement, 2.15 percentage points on average.

TABLE VI
RESULTS FOR EXPERIMENT 4

Category	Prec (%)	Rec (%)	F1 (%)
WC	94.1	80	**86.4**
PKM	63.1	63.1	63.1
Bad Randomness (BR)	66.6	50	57.1
PDF	100	25	40
ICV	75	100	85.7
CIB	75	10.3	18.1
CAI	50	25	33.3
PKC	61.5	47	53.3
IVM	69.2	47.1	56

Note: This table shows the results of Coverity Scan in [15] dataset. For each misuse category, Precision, Recall, and F1-score are presented.

Finally, for Experiment 4, we used the Coverity Scan tool to detect cryptography misuses in the dataset used in [15]. We achieve basically the same results as Braga et al.. Despite an overall improvement of 9 percentage points, from 35% to 44%, the Coverity Scan was able to detect only 44% of the cryptography misuses. This result is much worse than the 88% result, achieved in [7]. This experiment's results are shown in Table VI. It is worth noting that the WC and PDF categories achieve high Recall scores, but Coverity Scan falls behind in the detection of the other misuse categories.

V. DISCUSSION OF FINDINGS

In this section, we briefly discuss some interesting aspects of the results obtained in Section IV. We divide our discussion in three sections. The first discusses the impact of *code2vec* compared to previous works. The second section discusses the

impact of the use of data augmentation combined with each feature extraction method. Lastly, we consider implementation aspects of each feature extraction technique, their pros and cons.

A. Code2vec x Previous Feature Extractors

Code2vec is a deep learning neural network trained in approximately 14 million lines of Java source code with the purpose of predicting Java method names. Also, its hidden layers can be used to produce source code embeddings that can be used as features for machine learning classifiers through transfer learning. Thus, it is expected that its produced features would capture meaningful information about source code. That said, by analyzing our results, obtained in the case of one classifier per misuse presented in IV, we see that *code2vec* only outperformed *BoG* and *node2vec* (both at the same time) in one misuse category, namely HNV, by 1.9 percentage points compared to *BoG*, and 11.9 percentage points against *node2vec*. Compared to *BoG* in the other cases, *code2vec* performed similarly or worse (not by a great margin), and compared to *node2vec*, it performed worse only in the Hash category and better in the other cases. Here we argue that *code2vec* produces features that incorporate general aspects of source codes in comparison to BoG, as *code2vec* was not trained specifically in source code that contains cryptography misuses, in contrast to *BoG*, that was specifically trained in the cryptography misuse dataset. Also, as *code2vec* generates embeddings from paths between terminal nodes from the AST (literals), it does not filter structures based on their types; instead, all structures in the path are taken into account in order to produce embeddings. On the other hand, *BoG* produces specific features based on the chosen AST structures within the code and this reflects on the results obtained in Experiment 1, where the datasets have a very small size. Compared to *node2vec*, it is expected that *code2vec* performs better, as *node2vec* does not leverage text attributes of the source code and its syntax and semantics.

In Experiment 3, by dividing our dataset into misuses and good uses (by this we mean group the misuses into one class, and the good uses into another one), we ignore specific aspects that characterize certain types of misuses/good uses. Thus, the trained classifier looks for general patterns that can distinguish cryptography misuses from good uses in a broader sense, ignoring misuse and good use categories. As a result, since the amount of training and test instances was greater than in the case of Experiment 1, the differences between the obtained results are smaller. Still, the same best-to-worse order of the feature extractors, regarding the F1-score, is maintained, with *BoG* being the best technique, *code2vec* the second best, and *node2vec* the worst. However, we notice that each feature extraction technique give different priorities regarding false positives and false negatives, as reflected in the Precision and Recall metrics. For example, compared to the other two techniques, *code2vec* has the higher precision metric, which implies a low false positive rate. On the other hand, *BoG* has higher recall than *code2vec*, meaning that it has lower false negative rates. *Node2vec* has a more balanced score between recall and precision compared to the other two techniques. Thus, each technique has a different influence on the process of misuse detection and all of them can be further enhanced to present better results. Still, the results presented are very good and outperform common static analysis tools.

Finally, we have the results for Experiment 4. Although this experiment does not use any feature extraction method, using, instead, a free SCAT, its relevance is in showing how SCATs improved from previous works and how they compare to machine learning solutions previously proposed. As shown in Table VI, SCATs do not perform well in general. Of course, we can only compare them to the results of Experiment 3 as we used a different dataset. Here we argue that, although comparing the results in different datasets, SCATs have a stable behavior as they use handcrafted pattern rules. Also, by removing categorization in Experiment 3 and only training a classifier to distinguish between misuse and good uses, we can compare it to Experiment 4 results as we have a general misuse classifier. By averaging the F1-score results of Table VI, we observe that Coverity Scan obtains an average 54.7% F1-score, a 4.7 percent point improvement over evaluated tools in [15]. However, machine learning classifiers perform much better, with at least 13 percentage points of difference. Those results support the conclusion that machine learning can be a powerful tool to detect cryptography misuse. Nevertheless, it is worth noting that SCATs perform really well in low-complexity misuse categories like WCs. This suggests that hybrid approaches that use both rules and machine learning could be effective in some scenarios.

B. Impact of Data Augmentation

Now, we discuss the results presented by classifiers that use data augmentation as a tool to enlarge the training set, in order to train better classifiers. We use the *MLP* classifiers only to check the impact of data augmentation in the context of source codes as instances of a dataset. We start by analyzing the results obtained in Experiment 2, with the binary classifier for each category of misuse. Here, we notice mixed results with some cases of underperformance, but, in general, data augmentation helps classifiers perform better or virtually the same as classifiers that do not use it in the training step, as shown in Section IV. As those datasets were unbalanced in most of the cases, the predominance of enhanced results was expected, as we applied data augmentation in order to balance the datasets whenever possible. One example is the Hash category, where the unbalance was in the ratio of misuse instances, as shown in Table II. We were able to duplicate the good/bad use instances in the training set and, with that, the Hash category has some of the best results with data augmentation. On the other hand, misuse categories that already have an almost balanced ratio like Cipher and IV, tend to perform equally or worse than the normal classifiers, as the small number of instances added to the training set can confuse the classifiers' decision boundaries.

For the classifiers of Experiment 3, we notice some interesting effects on the application of data augmentation. As the size of the dataset is larger in Experiment 3, the results presented in Section IV show that we have little improvement with the use of data augmentation. However, as also shown, all of the data-augmented classifiers perform better than the normal classifiers.

Although some classifiers have a decrease in precision metrics, they have a noticeable increase in the recall metric, such as in the BSVM case, where we have 4.4 percentage point reduction in Precision but a 9.5 percentage point increase in Recall; the same goes for CSVM, with an 8.2 percentage point decrease in Precision but a 16.3 percentage point in Recall. This means that the number of false negatives is lowered in data-augmented classifiers and this is a desired behavior in this kind of application, where Recall is more important. Also, this could hint at a possible effect of data argumentation in source-code-related tasks.

C. Implementation Aspects

Now, we give some thought to the implementation aspects of each of the techniques. First, we mention the issue of Java versions that each implementation covers in the AST generation phase. For *BoG* and *node2vec*, we were only able to generate ASTs for Java 8 source code or lower versions, as ANTLR4 has only these versions of the Java parser available, as far as we know. As a consequence, we do not include test cases of newer versions of Java in this specific situation. Of course, there is a variety of Java parses that could replace ANTLR4 in this task of generating ASTs. However, some adaptations need to be made in the source code on the AST generation (the extractor part). Nevertheless, *code2vec* can handle Java 12 or lower version source codes, so it can handle the newer syntax and semantics of Java, with no need for an adaptation.

For the feature generation part, we did not benchmark each of the techniques. However, we notice that *code2vec* provides a more straightforward and faster feature generation approach than *BoG* and *node2vec*, as it does not need to train the network through transfer learning to generate those features. As a neural network approach, *code2vec* can take advantage of parallelism and can be executed faster than *BoG* or *node2vec*. For both *BoG* and *node2vec*, we need to train a clustering algorithm to generate a codebook in order to generate features. However, *code2vec* needs a GPU to be executed, which is not the case of *BoG* or *node2vec*. So depending on the hardware available, *code2vec* might not work.

Lastly, as we use a transfer learning approach, *code2vec* does not need to retrain the way we use the technique. Both *BoG* and *node2vec* would need retraining to expand their codebooks if we gather more data. *Code2vec* would only need retraining if we want to continue the training of a pretrained version of the network to incorporate training with cryptography misuse source code. These are some important implementation aspects to take into account if one wants to use these techniques.

D. Limitations of This Work

We close this section by talking about the possible limitations of this work. First, we talk limitations of our data augmentation application. The used technique, however simple, has similarities with widely used computer vision techniques such as image rotation and cropping. The technique we used in our work could be viewed as an analogy of the above in the source code context. Other techniques could be tested, but it seemed to us that, being simple enough and having produced the results it did, this would be an indication that this obfuscation works satisfactorily.

Now, we talk about the limitations of using transfer learning. In general, transfer learning is known to have the drawback of negative transfer, i.e., when the initial problem on which the model was trained is similar to the target problem [9]. In our case, the initial model was trained to recognize method names that are similar to ours in the sense that misuse detection depends also on method calls. Such method calls depend on the derivations of an AST, which extracts information about the source code itself. For source code, in particular, the issue is the new architectures that have not been tested enough.

Another possible limitation could be the size of the used dataset. However, as we used deep learning through transfer learning, by using a pretrained network trained in more than 14 million source code lines as a feature extractor, we do not need to train the whole model again. We used this model to generate features and, then, we trained SVM models to detect cryptographic misuses. So, we do not need to increase the size of the dataset as we did not train a deep learning Model.

Finally, we talk about the diversity of the used dataset. We were, indeed, limited to one language. But the language in question (Java) has the API that is the most studied language for the topic of cryptographic misuse (JCA). Thus, we cover very relevant data. Also, as each language has its grammar and we built our features based on the AST (which is constructed according to the language grammar), the method can be considered general. Thus, it would not need a diversity of languages in the dataset, but, instead, a diversity of misuses, which we currently have.

VI. Conclusion

We wish to conclude this work by saying that the use of machine learning and data oriented approaches to detect cryptography misuses is relatively new, so there is a lot to be done. With this work, we explored different types of machine learning approaches in order to detect cryptography misuses, each with its own advantages. All of these approaches surpassed common SCATs tools, in both previous and current analysis. It is important to note that we explored state-of-the-art methods, such as code2vec, and we applied state-of-the-art techniques such as transfer learning. We even adapted with success code obfuscation to serve as a source of data augmentation to source code related tasks. With this, we can safely state that machine learning can be used do detect cryptography misuse with a degree of success.

For future work and improvements, we intend to explore other machine learning paradigms such as unsupervised learning and semisupervised learning to detect cryptography misuses. Also, the creation of a standard dataset of cryptography misuses, where machine learning models can be evaluated is needed to better compare works in the literature. In addition, the possible combination of machine learning and other data oriented methods could provide another way of detecting cryptography misuses. Finally, the incorporation of explainable AI concepts is something we need to tackle in order to transform this approach into a tool that can easily support developers.

REFERENCES

[1] S. Krüger et al., "Cognicrypt: Supporting developers in using cryptography," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 931–936.

[2] A. Braga and R. Dahab, "*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, 2017.

[3] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira, "Understanding how to use static analysis tools for detecting cryptography misuse in software," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1384–1403, Dec. 2019.

[4] S. Afrose, Y. Xiao, S. Rahaman, B. Miller, and D. D. Yao, "Evaluation of static vulnerability detection tools with java cryptographic api benchmarks," *IEEE Trans. Softw. Eng.*, to be publihsed, doi: 10.1109/TSE.2022.3154717.

[5] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. D. Yao, and N. Meng, "Automatic detection of java cryptographic api misuses: Are we there yet," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 288–303, Jan. 2023.

[6] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis, "Evaluation of cryptography usage in android applications," in *Proc. 9th EAI Int. Conf. Bio-inspired Inf. Commun. Technol. (formerly BIONETICS). ICST (Institute Comput. Sci.)*, Social-Informatics and …, 2016, pp. 83–90.

[7] G. E. d. P. Rodrigues, A. M. Braga, and R. Dahab, "Using graph embeddings and machine learning to detect cryptography misuse in source code," in *Proc. IEEE 19th Int. Conf. Mach. Learn. Appl.*, 2020, pp. 1059–1066.

[8] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, p. 147, 2018.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[11] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do java developers struggle with cryptography apis?," in *Proc. 38th Int. Conf. Softw. Eng.*. ACM, 2016, pp. 935–946.

[12] A. Braga and R. Dahab, "Mining cryptography misuse in online forums," in *Proc. IEEE Int. Conf. Softw. Qual., Reliab. Secur. Companion*, 2016, pp. 143–150.

[13] S. Krüger, "Cognicrypt–the secure integration of cryptographic software." [Online]. Available: http://www.bodden.de/pubs/phdKrueger.pdf, 2020.

[14] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Berlin, Germany: Springer Science & Business Media, 2009.

[15] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira, "Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng.*, 2017, pp. 170–181.

[16] S. Das, V. Gopal, K. King, and A. Venkatraman, "Iv= 0 security: Cryptographic misuse of libraries," Massachusetts Institute of Technology, 2014.

[17] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz, "Cryptoexplorer: An interactive web platform supporting secure use of cryptography apis," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering*, 2020, pp. 632–636.

[18] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[19] T. Æ. Mogensen, *Introduction to Compiler Design*. Berlin, Germany: Springer, 2017.

[20] F. B. Silva et al., "Bag of graphs= definition, implementation, and validation in classification tasks," 2014. [Online]. Available: http://repositorio.unicamp.br/handle/REPOSIP/275527

[21] L. C. Navarro, A. K. Navarro, A. Grégio, A. Rocha, and R. Dahab, "Leveraging ontologies and machine-learning techniques for malware analysis into android permissions ecosystems," *Comput. Secur.*, vol. 78, pp. 429–453, 2018.

[22] F. B. Silva, R. d. O. Werneck, S. Goldenstein, S. Tabbone, and R. d. S. Torres, "Graph-based bag-of-words for classification," *Pattern Recognit.*, vol. 74, pp. 266–285, 2018.

[23] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 855–864.

[24] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. ACM SIGPLAN Notices*, 2016, vol. 51, no. 1, pp. 298–312.

[25] T. Shippey, D. Bowes, and T. Hall, "Automatically identifying code features for software defect prediction: Using ast n-grams," *Inf. Softw. Technol.*, vol. 106, pp. 142–160, 2019.

[26] J. A. Harer et al., "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*.

[27] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021.

[28] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197 158–197 172, 2020.

[29] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, "Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection," *IEEE Access*, vol. 10, pp. 32 595–32 607, 2022.

[30] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: Vulnerability detection with deep learning on a natural codebase for python," *Inf. Softw. Technol.*, vol. 144, 2022, Art. no. 106809.

[31] H. K. Dam et al., "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*.

[32] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, ser. MSR '20. New York, NY, USA:Association for Computing Machinery, 2020, pp. 243–253. [Online]. Available: https://doi.org/10.1145/3379597.3387445

[33] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *J. Big Data*, vol. 6, no. 1, p. 60, 2019.

[34] F. Fischer et al., "Stack overflow considered helpful! deep learning security nudges towards stronger cryptography," in *Proc. 28th USENIX Secur. Symp.* (USENIX Secur. 19), 2019, pp. 339–356.

[35] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 243–253.

[36] I. Synopsys, "Coverity scan," 2021. [Online]. Available: https://scan.coverity.com/