electronics

MDPI

Article

Intelligent Detection of Cryptographic Misuse in Android Applications Based on Program Slicing and Transformer-Based Classifier

Lizhen Wang [1,2], Jizhi Wang [1,2,3,4,*], Tongtong Sui [1,2], Lingrui Kong [1,2] and Yue Zhao [1,2]

[1] School of Computer Science and Technology, Qilu University of Technology (Shandong Academy of Science), Jinan 250353, China; 13954323783@163.com (L.W.)
[2] Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (National Supercomputing Center in Jinan), Jinan 250014, China
[3] Quancheng Laboratory, Jinan 250100, China
[4] Jinan Institute of Supercomputing Technology, Jinan 250301, China
* Correspondence: wangjzh@sdas.org

Abstract: The utilization of cryptography in applications has assumed paramount importance with the escalating security standards for Android applications. The adept utilization of cryptographic APIs can significantly enhance application security; however, in practice, software developers frequently misuse these APIs due to their inadequate grasp of cryptography. A study reveals that a staggering 88% of Android applications exhibit some form of cryptographic misuse. Although certain tools have been proposed to detect such misuse, most of them rely on manually devised rules which are susceptible to errors and require researchers possessing an exhaustive comprehension of cryptography. In this study, we propose a research methodology founded on a neural network model to pinpoint code related to cryptography by employing program slices as a dataset. We subsequently employ active learning, rooted in clustering, to select the portion of the data harboring security issues for annotation in accordance with the Android cryptography usage guidelines. Ultimately, we feed the dataset into a transformer and multilayer perceptron (MLP) to derive the classification outcome. Comparative experiments are also conducted to assess the model's efficacy in comparison to other existing approaches. Furthermore, planned combination tests utilizing supplementary techniques aim to validate the model's generalizability.

Keywords: slicing; android; cryptography; application security; transformer; clustering; active learning

Citation: Wang, L.; Wang, J.; Sui, T.; Kong, L.; Zhao, Y. Intelligent Detection of Cryptographic Misuse in Android Applications Based on Program Slicing and Transformer-Based Classifier. *Electronics* **2023**, *12*, 2460. https://doi.org/10.3390/electronics12112460

Academic Editors: Juan M. Corchado, Carlos A. Iglesias, Byung-Gyu Kim, Rashid Mehmood, Fuji Ren and In Lee

Received: 21 March 2023
Revised: 22 May 2023
Accepted: 25 May 2023
Published: 30 May 2023

1. Introduction

Due to its open source nature and expansive features, Android has emerged as the predominant operating system for smart mobile devices, owing to the rapid advancements in technology witnessed in recent years. Java offers an extensive array of user-friendly function libraries and third-party resources, making it a preferred choice among developers who appreciate its open source nature and versatility.

According to the Android app dataset Androzoo [1], there exists a global community of over a million Android developers, and this dataset has amassed a sample of more than 10 million Android applications from the worldwide app marketplace. However, Android applications are not without their flaws. Subsequently deployed on users' devices, these applications may harbor exploitable vulnerabilities and defects, primarily due to developer limitations and a lack of thorough code review and security checks during the development phase. Although these vulnerabilities are generally not introduced intentionally by the developers, malicious actors can exploit them through man-in-the-middle attacks, compromising user system security and pilfering sensitive user data.

Given the significant amount of personal user information stored on smart devices, such vulnerabilities have the potential to inflict irreparable harm upon consumers.

Cryptographic security assumes a pivotal role in software security, with the use of system-provided APIs offering explicit encryption standing as a conventional technique for safeguarding critical information within Android applications. Although these interfaces aid developers in comprehending the intricacies of cryptographic primitives and mitigating workload, it remains imperative to ensure the accuracy and dependability of these API parameters during utilization.

A study conducted by researchers revealed that a staggering 88% [2] of the Android applications they examined exhibited at least one instance of misuse, defining six prevalent categories of faults when employing cryptography APIs. To address this issue, numerous researchers have been devising innovative methods for detecting cryptographic API misuse in Android applications. Noteworthy examples of static detection tools include CryptoLint [2], CMA [3], CogniCrypt$_{SAST}$ [4], and CryptoGuard [5]. Although these tools have achieved some success, they predominantly rely on manually constructed rules or models, often requiring a substantial understanding of cryptography and being susceptible to errors. Furthermore, various machine learning techniques, such as probabilistic model-based approaches and support vector machine (SVM) classification [6], have been employed to explore cryptographic APIs. These techniques have improved detection efficacy while reducing false positives. Consequently, the proper utilization of system-provided APIs for encryption stands as an exception rather than the norm. This underscores the necessity for further research to uncover the underlying causes of frequent misuse.

In this research, we offer a method for identifying intelligent cryptographic misuse based on program slicing, which combines program slicing with deep learning to create a neural network model to examine the usage of cryptographic APIs in Android applications. The following are the contributions made by this paper:

1. In this paper, we employ program slicing to identify the locations of cryptographic APIs and acquire code slices about cryptographic APIs with more focused data samples.
2. Irregularly sliced code lengths and contextual links between codes are resolved using the transformer model, which can accommodate varying-sized inputs.
3. Active learning and a new deep learning model are incorporated into the training. The experimental results demonstrate that the system outperforms the models in other relevant studies regarding cryptographic misuse detection.

2. Related Work

This section encompasses several contemporary research studies that are relevant to our work. The concept of program slicing, which involves decomposing a program into its smallest constituents to achieve a desired behavior, was initially elucidated by Weiser [7] in their seminal research conducted in 1984. Hoffmann [8] subsequently employed this method to scrutinize malware and applied it to Smali code to attain slicing outcomes for Android applications. In response to the inadequacy of the conventional attention mechanism to achieve parallelism, the Google team, led by Vaswani, proposed a model architecture named the transformer [9], which hinges on the attention mechanism. Transformer significantly broadens the application of the attention mechanism by introducing a multi-headed attention mechanism. The k-means approach, originally delineated by Macqueen [10] in 1967, offers an effective means to handle vast datasets by enabling efficient clustering. In order to iteratively refine the model and incorporate human expertise into the machine learning framework, active learning entails the acquisition of challenging sample data that defies categorization by machine learning methods. This process facilitates manual validation and review followed by retraining the manually labeled data utilizing supervised or semi-supervised learning models. Settles [9] provided a comprehensive exposition on active learning, encompassing a review of relevant literature, solutions, a framework for

query strategy, analysis of active learning, considerations in variable setting, applicability to real-world problems, and related areas of study.

In order to scrutinize the ramifications of improper SSL/TLS utilization in Android applications, Fahl et al. [11] initially divulged a plethora of vulnerabilities associated with SSL/TLS in Android applications in 2012. Six cryptographic API protocols were established, and Egele et al. [2] subsequently assessed the accurate implementation of these cryptographic APIs in congruence with encryption principles. To adhere to the notion of IND-CPA security, CryptoLint was devised and developed, leveraging the Androguard [12] static analysis framework tool. In order to present CMA, a detection tool predicated on pattern matching for identifying cryptographic API misuse, Shao et al. [3] meticulously examined and analyzed instances of password misuse in Android applications. They encapsulated their findings in a password misuse model, employing a combination of static and dynamic analysis. Ma et al. [13] proposed the employment of a tool named CDRep to automatically rectify defects resulting from cryptographic misuse in Android applications. CDRep harnesses an existing program [2,3] to identify such misuse and generates a set of manual patch templates for the targeted code alteration. Drawing inspiration from the work of CryptoLint, Muslukhov et al. [14] constructed an advanced BinSight, a static detection tool that primarily focuses on ascertaining whether the program itself or third-party libraries are culpable for the misuse of cryptographic APIs. In their quest to detect instances of cryptographic abuse in Android applications, Krüger et al. [4] proposed CogniCrypt$_{SAST}$, a static tool that redefined the rule set by utilizing the definition language CrySL in the Java Cryptographic Architecture document. This tool leverages a whitelist to manually define the appropriate utilization of cryptographic APIs. Unfavorable outcomes were discovered by Gao et al. [15] when they employed the CogniCrypt$_{SAST}$ tool to evaluate whether API misuse in Java programs had been adequately addressed in a recent software upgrade. Rahaman et al. [5] introduced CryptoGuard, a static analysis tool that examines Java projects and Android applications for cryptographic API misuse by employing a novel slicing method along with 16 specific usage rules. Fischer et al. [15] employed a support vector machine (SVM) model as a learning mechanism to extract and scrutinize all security-related code snippets from Stack Overflow posts that discussed Android encryption. Xu et al. [16] devised a probabilistic model-based technique to explore how Android applications employ cryptographic APIs. Grounded in hidden Markov and n-gram models, their study generated a dataset of cryptographic API sequences, incorporating parameters for intelligent detection of Android applications.

In addition to the detection of cryptography in Android applications, there have been recent advancements in identifying cryptographic misuse in various languages and platforms. Zhang et al. [17] devised and implemented CRYPTOREX, an innovative tool capable of detecting cryptographic abuse in IoT devices. Their study revealed that a staggering 24.2% of firmware images violate cryptographic regulations. Li et al. [18] on the other hand, developed CryptoGo utilizing sophisticated techniques in static contamination analysis, employing 12 cryptographic rules. Their findings indicated that 83.33% of cryptographic elements in the Go programming language demonstrate at least one instance of cryptographic misuse. An et al. [19] proposed a BiLSTM-based approach for cryptographic detection, which facilitates bidirectional learning. They devised the CryptoDetection tool specifically for Java source code and achieved an impressive accuracy rate of 92%. Wickert et al. [20] created a static analysis tool catered to Python projects, encompassing five distinct cryptographic rules known as LICMA. Their research revealed that 52.26% of Python projects exhibit at least one instance of cryptographic misuse. Rahaman et al. [21] developed a static tool named TAINTCRYPT, tailored for detecting cryptographic implementations in C/C++ programming languages. They utilized a specification language of deterministic finite automata (DFA).Rodrigues et al. [22] proposed an innovative methodology that combines graph embedding techniques with machine learning models to detect instances of cryptographic abuse in source code. They leveraged the Node2vec [23] and Bag of Graphs [24] models as embedding generators for representing source code graphs.

3. Methods

In this section, we present a method for classifying cryptographic API abuse in Android applications and specify the implementation details and model selection for the method.

3.1. Overview

Figure 1 illustrates the overarching workflow of the system. This study endeavors to develop an intelligent detection system, centered on program slicing, to identify instances of cryptographic misuse within Android applications. More specifically, the focus is on identifying cryptographic misuse APIs. Initially, we preprocessed the dataset to extract Smali code files. Subsequently, the Smali files underwent static analysis to extract a compilation of code slices related to cryptography, in line with the defined slicing criteria. Lastly, the collected code slices were utilized to train our deep learning model, ensuring that it met our accuracy requirements for classifying the code slices. The specific steps are as follows:

1. Preprocessing: we processed the Dalvik bytecode [25] (.apk) Android application file into a Smali code file and parsed it to create an object representation.
2. Static slicing: a standard appropriate for this work was created based on the security criteria established by abstract slicing patterns. By executing static backtracking, all potential execution routes were tracked, and the set formed by all reachable nodes was determined and represented by a graph.
3. Security classification: then, unsupervised learning was applied to the slice set using the k-means clustering to obtain several classes. Lastly, active learning was used to label the code segments in various classes in order to train the model.

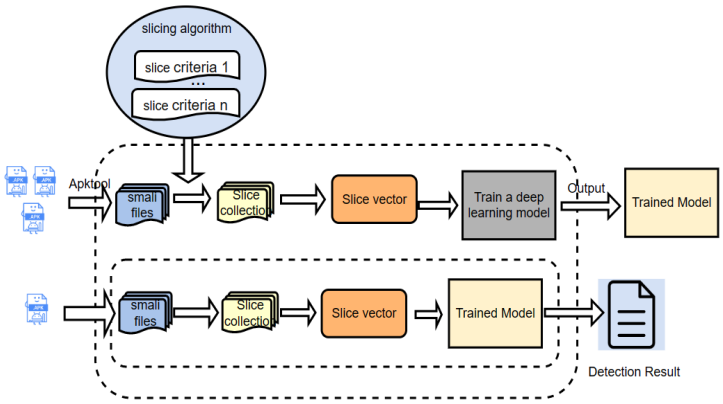

Figure 1. Overall system workflow.

We provide further details on each stage in the sections that follow.

3.2. Preprocessing

To mitigate the risk of failure, we utilize the Dalvik bytecode decompilation technique, which transforms the Android program into register-based intermediate code represented as Smali code. Direct decompilation to Java source code carries a certain probability of failure. In this stage, the .apk file undergoes decompilation using the Apktool [26] tool, resulting in the production of the classes.dex file. Subsequently, the baksmali6 tool processes this file, generating a list of files that resemble Java code. We then parse these files into an object representation, encompassing all fundamental attributes and blocks.

3.3. Locating API Abuse in Android Apps

The program slicing methods we employ, the slicing criteria we derive from defining slicing patterns, and the quick ways to obtain slices will all be discussed further below.

3.3.1. Slicing Criteria

We have implemented the slicing schema proposed by Johannes [27], which serves as a conceptual representation of the resource or object to be monitored in XML format. This choice is crucial, as program slicing heavily relies on the specific criteria employed for a given line of program code. The adoption of this schema enables us to effectively identify use–def chains by examining calling statements that conform to the designated schema. Consequently, we can derive our slicing criterion. To achieve this, we meticulously inspect each line of code within the application, subjecting the specified method signature to scrutiny. We consider any supplementary program statements associated with each matching instance, akin to the initial slice. By establishing a connection between the index of each occurrence register and the designated parameter of interest, we are able to determine the name of the register that requires tracing. Ultimately, this process yields an optimal set of slicing criteria.

3.3.2. Backtracking

We employed a generalized static backtracking approach for Smali codes, as proposed by Hoffmann et al. [8], to perform forward slicing. During the execution of a static traceback of registers, we employ the use–def method for code traversal, meticulously documenting all pertinent program statements as nodes and incorporating them into the slicing tree. The tracing halts, and the slicing does not proceed further down the tree when a constant value is assigned to the register currently under scrutiny. Additionally, the slicing process terminates when the traced register is overwritten or becomes inaccessible. This procedure generates one or more slicing criteria based on the slicing pattern, which are subsequently added to the queue, also known as a first in, first out (FIFO) queue. Each register is backtracked until the queue is depleted, provided that the registers discovered during the initial search for a matching call opcode fill the queue. The forward and backward slicers obtain their input from the to-do list, encompassing all monitored registers, fields, return values, and arrays. Furthermore, the to-do list ensures that no monitored items are reprocessed, as their references are retained. When requested by the slicer, the queue yields the next object to be traced, containing the registers to be monitored and the location of the subsequent opcode.Pseudocode for the backtracking algorithm is shown in Algorithm 1.

Algorithm 1: Backtracking Algorithm

```
Data: Initialising the queue:Q;
      Slicing criteria:C
Result: Slicer
r = C.p[index]; visited[r] = 1; r Join the queue Q;
while Queue Q is not empty do
    Queue head element of r_temp = Q out of queue;
        visited[r_tem] = 0;
        Search up the opcode to find statement S ;
    if S Existence then
        Register for operating r_temp = Q in w = S;
    else
        if wnot visited then
            w Join the queue Q;
            Search up the opcode to find statement S ;
        end
    end
end
```

We consider the Smali code example fragment in Figure 2. The diagram depicts a symmetric encryption algorithm. The developer creates a *Cipher* object, calls the *getInstance*

method of the *Cipher*, and passes it the value of the requested conversion as a parameter. The conversion value usually includes the name of the encryption algorithm, the mode of operation, and the padding scheme. The figure "$AES/CBC/PKCS5Padding$" describes the conversion value for this method, where the algorithm name is AES, the mode of operation is CBC, and the padding scheme is $PKCS5Padding$. The ECB mode has been proven to be an insecure encryption mode and should no longer be used. The developer should provide a non-random IV when CBC mode is chosen for encryption. An encryption vulnerability will occur if no IV is given or if the IV value is predictable.

```
.method public a([BI)[B
    new-instance v1, Ljavax/crypto/spec/SecretKeySpec;
    invoke-virtual {v0}, Lcom/nesurv/nestudy/kh;->a()[B
    move-result-object v2
    const-string v3, "AES"
    invoke-direct {v1, v2, v3}, Ljavax/crypto/spec/SecretKeySpec;><init>
([BLjava/lang/String;)V
    new-instance v2, Ljavax/crypto/spec/IvParameterSpec;
    invoke-virtual {v0}, Lcom/nesurv/nestudy/kh;->b()[B
    move-result-object v0
    invoke-direct {v2, v0}, Ljavax/crypto/spec/IvParameterSpec;-><init>([B)V
    const-string v0, "AES/CBC/PKCS5Padding"
    invoke-static {v0}, Ljavax/crypto/Cipher;-
>getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
    move-result-object v0
    invoke-virtual {v0, p2, v1, v2}, Ljavax/crypto/Cipher;-
>init(ILjava/security/Key;Ljava/security/spec/AlgorithmParameterSpec;)V
    invoke-virtual {v0, p1}, Ljavax/crypto/Cipher;->doFinal([B)[B
    move-result-object v0
    return-object v0
.end method
```

Figure 2. Example Smali code snippet.

For the encryption algorithm in Figure 2, there will be two backtracks. Firstly, determine all calls to the method $Cipher \rightarrow getInstanceinit()$ and backtrack the first parameter, register $v1$, to save the conversion value for each call. Find all possible execution paths where the endpoint is similar to the conversion or specifies a symmetric block cipher, e.g., AES, until the path reaches the end of the constant endpoint. Then, for all $Cipher \rightarrow init()$ calls that use the $AlgorithmParameterSpec$ object as the second parameter operation mode, backtrack the value of that parameter. Using the list of constants found, verify that an object of type $IvParameterSpec$ is created by calling its constructor. If it is not found, abort. A subpath from each available slice path starting with the iv parameter is extracted and passed to the constructor of the $IvParameterSpec$ object.

3.3.3. Collection of Slices

We construct a tree encompassing all encountered slicing nodes by aggregating all reachable nodes. The foremost node in the specified slicing pattern invariably serves as the slicing criterion, akin to the point of origin for all potential execution paths. The lines of code that have been excised reside in the lowermost nodes. In situations where multiple execution paths exist, as seen in conditional evaluations, a slicing node may have connections from various predecessor nodes. When code statements undergo numerous iterations, such as those within a for or while loop, loops are formed among the vertices. Each intermediate node retains a catalog of all precursor nodes, accompanied by the initial register names and register names associated with the current program statement.

A slicing tree is comprised of one or more terminal nodes, with each terminal node representing either a constant value or an abruptly terminated slicing process. When a constant value (such as an integer or a string) is copied into a trace register, the slicing process typically concludes due to the alteration of the register's value. This implies that, in the context of backward slicing, the operation has modified the value of one or more slicing conditions. In the case of forward slicing, it indicates that the data stream has reached its endpoint and will be inserted into the terminal node to ensure the slicer maintains continuity. Consequently, the tracked register will not impact subsequent operations.

3.4. Transformer-Based Label Efficient Error Detection Model

The overall procedure is depicted in Figure 3. Subsequently, the features are extracted from the slicing code associated with cryptography using the transformer, which then channels these features into the MPL layer for detection. Following the input of annotation features into the MPL layer utilizing the k-means clustering-based active learning approach, the detection outcomes are produced. After being fed into the MPL layer for detection, the results are subsequently output. A loss computation is then performed on the results and annotated samples. To ensure adherence to our accuracy standards, the process is terminated after a predetermined number of iterations.

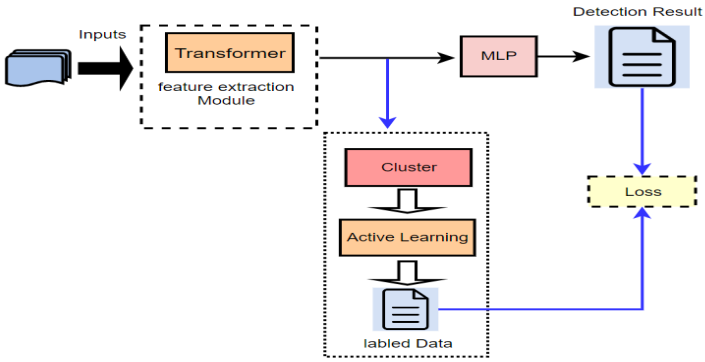

Figure 3. Classification model with active learning for data label.

In this study, we employ a transformer [28] as the feature extractor. The assortment of slices related to cryptographic abuse in Section 3.3 is obtained, serving as our input. The transformer initially converts the lexicon into a vector of the same embedding dimension. Because attention lacks the ability to recognize the order of the sequence, position embedding is introduced to assign numerical values to each position. Each word vector is now accompanied by a position vector, aiding in its identification. The creation of the new vector is achieved by combining the location and word vectors. The position encoding algorithm is as follows:

$$\begin{cases} PE_{2i}(p) = \sin\left(p/10000^{2i/d_{pos}}\right) \\ PE_{2i+1}(p) = \cos\left(p/10000^{2i/d_{pos}}\right) \end{cases} \quad (1)$$

Suppose the word embedding is d_{pos} characters long. In that case, we must create a position encoding vector PE that is d_{pos} characters long, where p stands for the word position, $PE_i(p)$ is the value of the i th element in the position vector of the p th word position vector, and p is the current word position. The word vector and the position vector are then added together. In this work, we use sine and cosine functions of different frequencies. Each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10{,}000 \cdot 2\pi$. This function was chosen because we assumed that it would allow the model to learn easily by relative position, because for any fixed offset k, PE_{pos+k} can be expressed as a linear function of PE_{pos}. Using positional encoding ensures that the positional relationships are maintained after our code slices are converted to vectors, which benefits the results. The sinusoidal version was chosen because the length of the slicing code produced by program slicing is variable, and the sinusoidal version allows the model to infer longer sequences than those encountered during training.

The transformer obtains the attention value through the multi-headed attention module. First, we generate three corresponding vectors for the input word vector X: $query$, key, and $value$. With Q, K, and V vectors, we first calculate the similarity of K and Q by point accumulation. To prevent the similarity from being too large in SoftMax, the point of the point is divided by $\sqrt{d_k}$, of which $\sqrt{d_k}$ is the dimension of K dimension. The output of the self-attention at that position is created using the SoftMax normalized values, which

are then multiplied by the "V" vector matrix and added together. The formula is recorded as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2}$$

Among the variables, Q, V, and K represent the queries, keys, and values matrix of the input sentence; each line of the matrix corresponds to the *query*, *key*, *value* vector corresponding to each word; and $\sqrt{d_k}$ represents the vector length.

Multi-headed attention consists of numerous sets of attention, similar to the one described above, but it uses various heads to obtain distinct feature expressions. Afterwards, the features will be combined, but the weights assigned to each attention group will not be distributed equally. The following is the calculation formula:

$$head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right). \tag{3}$$

$$MultiHead(Q, K, V) = Concat(head_1, \cdots head_h)W^o. \tag{4}$$

The following equation provides a multiple-dimensionality reduction by optimizing full connectedness:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2. \tag{5}$$

In order to obtain the results, we finally input the MLP with the properties of the transformer.

3.5. Active Learning Based on K-Means Clustering

We employ active learning [29] based on clustering [30] because manually annotating each sample requires much time and effort. By training the model with fewer annotated examples, active learning seeks to improve results. We first discuss k-means clustering and then explain how to obtain high-quality labeled data using the clustering results.

We obtain the dimensionality reduction features using the procedures in Section 3.4 and subsequently group the dimensionality reduction features using the k-means algorithm.

This is demonstrated by choosing random locations as the initial clustering centers in the K feature space of the vector.

$$Euclidean\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}. \tag{6}$$

$$d = \sqrt{(x_n - x_1) + (y_n - y_1)}. \tag{7}$$

The distance to the K centers is calculated for each of the remaining points, after which the remaining points select the cluster center closest to them as their own labeled category. After that, each cluster's new center point is calculated.

$$P_y = \sum_{i=1}^{n} P_{iy}/n. \tag{8}$$

$$P_x = \sum_{i=1}^{n} P_{ix}/n. \tag{9}$$

The repeat ends if the calculated new center point is the same as the original one. Otherwise, the second step of the process will be repeated. When the result of each iteration is unchanged, the algorithm is considered to converge, and clustering is complete. Otherwise, the second step of the process is repeated. The method is said to have converged when the outcome of each iteration stays the same.

In order to obtain our one-pass classification results in this paper, we first choose a small number of samples from each center of clusters. Then the labeled data are used as supervised information to train the transformer model. The cluster coreset algorithm is used to re-select the labeling samples if the entropy output is high, until the model satisfies

our accuracy standards. In order to conduct experiments, we designate 200, 400, 600, 800, and 1000 training samples in this work.

4. Experimentation and Evaluation

This study assesses the suggested technique utilizing the APK dataset as an experimental subject to validate the efficacy of the program slicing-based cryptographic misuse analysis method for Android applications presented in Section 3.

4.1. Dataset Preprocessing

The dataset used in this experiment consists of a selection of APKs downloaded from Androzoo between 2020 and the present, totaling 1255 applications. We concentrated on the libraries "javax. crypto", "java. security", and "java.net.ssl" because we are studying cryptographic APIs. We kept a total of 1178 APKs after performing a fast review of the gathered APKs to remove those that did not use any cryptographic APIs from the downloaded application libraries. When the APK files were deleted, 1146 viable APKs remained. All samples were examined using the decompiler Apktool, and our criteria determined their slices. A total of 3788 slices about cryptography were acquired, and this dataset served as the basis for our tests.

4.2. Evaluation Criteria for Cryptographic Misuse Detection Model

The following measures were employed in this study to evaluate the effectiveness of the cryptographic misuse detection and classification system.

Accuracy means the proportion of samples that were properly categorized by a classifier to all samples. In general, improved detection or classification results from higher accuracy. This indicator's expression is as follows,

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{10}$$

Precision measures the percentage of samples that the system correctly predicts cryptographic misuse, i.e.,

$$P = \frac{TP}{TP + FP}. \tag{11}$$

Recall rate measures the proportion of samples of cryptographic misuse that the detection system believes to be present as a percentage of the actual samples of cryptographic misuse,

$$R = \frac{TP}{TP + FN}. \tag{12}$$

F1 value combines accuracy and recall and is thought to provide a more accurate view of the system's ability to detect cryptographic misuse, i.e.,

$$F1 = \frac{2 \cdot P \cdot R}{P + R}, \tag{13}$$

where TP is the number of samples the system determined to have cryptographic misuse and are misused; FP is the number of samples the system decided were misused but were not misused in reality; and FN is the number of samples the system determined to not have misuse but contained misuse in actuality.

4.3. Experimental Details

Our work's primary optimization goal is a focal loss of $\gamma = 6$. We utilize the command line utility "gpustat" with a tiny batch size of 1 and forgo generating any gradients to assess the model's GPU memory utilization. Note that we only use one GPU card for training and inference. We conducted our trials using the PyTorch framework. For the training, we used tiny batches of size 6. All tests were carried out on a workstation with two NVIDIA

3090 GPU cards. The transformer model we employed has a depth of four layers, which is relatively simple to obtain and can handle input of varying lengths. It was pre-trained in natural language processing.

4.4. Label Standards

Due to the active learning methodology used in this study, only minimal annotation was required for training in the clusters split by clusters. We annotate them in accordance with the norms of cryptographic algorithms, and our annotation criteria are as follows. This system aims to classify cryptographic methods and assess whether there is misuse.

(1) The following situations will be considered instances of SSL/TLS abuse: (a) The developer's implementation of the X509Trust Management interface's Verify Server Trusted method accepts all certificates without verifying the server certificate. (b) The developer's implementation of the verify method under the hostname verifier interface accepts all domain names. It does not check if the hostname associated with the URL matches the server's hostname. The server's hostname and the hostname associated with the URL match, but it allows all domain names. (c) SSLSocket Factory, which sets the domain name verification parameter to ALLOW ALL HOSTNAME VERIFIER and creates sockets for secure connections. (d) No use of SSL or a combination of secure and insecure connections. (e) Use of an outdated SSL/TLS version.
(2) Under the following scenarios, the symmetric encryption algorithm will be deemed to have been abused: (a) The use of unsafe symmetric encryption techniques, such as DES, 3DES, RC2-64-bit cipher, RC4-stream cipher, etc. (b) The ECB algorithm is selected while the AES algorithm is under-configured. (c) The block mode's initialization vector (IV) is not randomly produced. (d) Oracle chooses PKCS1.5 for padding, etc. (e) The mechanism of encrypting keys is hard-coded.
(3) Under the following situations, asymmetric encryption techniques will be deemed to be improperly applied: (a) The RSA algorithm is insecure when the key length is less than 3072; then, asymmetric encryption algorithms will be deemed to be misused. (b) The ECC algorithm is insecure when the key length is less than 224, and optimal asymmetric encryption padding is not used by the RSA method (OAEP).
(4) The following situations constitute misuse of a hash function: (a) employ of a cryptographic hash method that is unsafe (e.g., MD2, MD5, SHA-1, etc.). (b) Use of the less secure KDF algorithms PBKDF2 and Bcrypt.
(5) Under the following circumstances, random number generation will be deemed to have been abused: Using the random number generator random.

4.5. Experiments and Analysis of Results

4.5.1. Model Effect Analysis

Of the data used for the experiments in this paper, 80% were used as the training set, and the remaining 20% were used as the test set. The objective was to determine whether using program slicing and constructing a cryptographic misuse detection system based on the transformer model could better detect cryptographic misuse. This article evaluates the model's precision, recall, accuracy, and F1 values at different classification head levels. This article consider insecure samples as positive and secure ones as negative. Thus, the precision score measures how many of the predicted unsafe segments are indeed unsafe, the recall score assesses how unsafe many of the true unsafe segments retrieved from all unsafe elements are, and the accuracy score measures the overall classification performance considering both positive and negative samples.

Based on the experimental results shown in Table 1, the best results obtained by this method when considering all evaluation criteria were: accuracy (0.953), precision (0.978), recall (0.935), and F1 value (0.956), with three classification head layers. These results will be used in subsequent experiments to compare with other detection systems.

Table 1. Comparing outcomes of this system's indicators at various classification head levels.

Number of Layers in the Classification Head	Accuracy	Precision	Recall	F1
1	0.873	0.953	0.733	0.869
3	0.953	0.978	0.935	0.956
5	0.944	0.965	0.901	0.932
10	0.876	0.923	0.811	0.863

4.5.2. Comparative Analysis of Our Approach and the SVM Model

The literature [15] provides an SVM-based approach to cryptographic misuse detection. A comparison between the system in this paper and the SVM system was conducted to ascertain whether the system in this research has better cryptographic misuse detection performance compared to other detection systems based on the Android cryptographic misuse detection model. The current method and SVM methods were compared based on various metrics when different sample sizes were selected. The results are presented in Tables 2–5.

Table 2. Comparison of recall scores.

Model	200	400	600	800	1000
SVM	0.581	0.645	0.711	0.737	0.763
Transformer	0.742	0.789	0.822	0.887	0.935

Table 3. Comparison of Accuracy scores.

Model	200	400	600	800	1000
SVM	0.844	0.874	0.881	0.887	0.889
Transformer	0.921	0.944	0.954	0.921	0.953

Table 4. Comparison of Precision scores.

Model	200	400	600	800	1000
SVM	0.827	0.822	0.831	0.829	0.826
Transformer	0.934	0.953	0.962	0.943	0.978

Table 5. Comparison of F1.

Model	200	400	600	800	1000
SVM	0.835	0.844	0.854	0.858	0.856
Transformer	0.925	0.945	0.935	0.940	0.970

We used our dataset as input to the tuned SVM model, as has been suggested in the literature [15], and tuned the model to the best possible parameters. Table 2 shows that our system's average F1 value is 9.7% higher than that of this system, which shows that our system's capacity to identify sample cryptographic misuse thoroughly is significantly better than Fischer's model. This system's accuracy in determining whether cryptographic misuse of samples has dramatically increased, as it is 8.3% greater than that of this model. As demonstrated in Table 4, when comparing accuracy, our model performs better than the SVM model, with an average improvement of roughly 6%. As shown in Table 5, when precision is compared, our model surpasses the SVM model by an average of 13.1%. Furthermore, when we label the data as 200, our system's F1 value is 9% higher than the SVM model's, along with its recall rate of 16%, the accuracy rate of 7.7%, and precision of 10.7%. Based on the data above, it can be inferred that when we perform a precise small amount of annotation, the system's effectiveness in this paper is significantly higher than

that of the SVM model put forth by Fischer. This finding demonstrates the effectiveness of our active learning to select annotated samples, i.e., we only need to annotate a small number of samples to obtain good results.The SVM model is chosen to use TF-IDF for feature extraction, and finally the code fragment is fed directly into the SVM model as text input, so that the contextual relationship in the code is blurred, while this method proposes the experimental transformer model for feature extraction, using the position encoding method to represent the contextual relationship to enhance the detection effect.

4.5.3. Comparison with Rule-Based Matching Tools

To validate the performance of the proposed detection method, the tools in this paper were compared with the state-of-the-art rule-based matching password misuse detection techniques CogniCrypt$_{SAST}$ [4] and CryptoGuard [5]. We manually analyzed a random selection of 50 APKs from Androzoo downloaded applications and found 138 cryptographic misuses, of which CogniCrypt$_{SAST}$ and CryptoGuard found 108 and 101, respectively. A comparison of the performance of the detection tool proposed in this paper with existing detection tools is shown in Figure 4. From the figure, it can be seen that the detection accuracy, precision, recall, and F1 scores of the detection tool proposed in this paper were 0.925, 0.917, 0.92, and 0.91, respectively. In comparison, the best detection tools in CogniCrypt$_{SAST}$ and CryptoGuard were 0.842, 0.84, 0.732, and 0.783, respectively. This is because different tools use different sets of cryptographic rules, and any one tool is limited in its identification methods and recognition capabilities and cannot cover the full range of rules, resulting in performance differences between the evaluation metrics. Our tool, on the other hand, is unlimited by the encryption rule set.

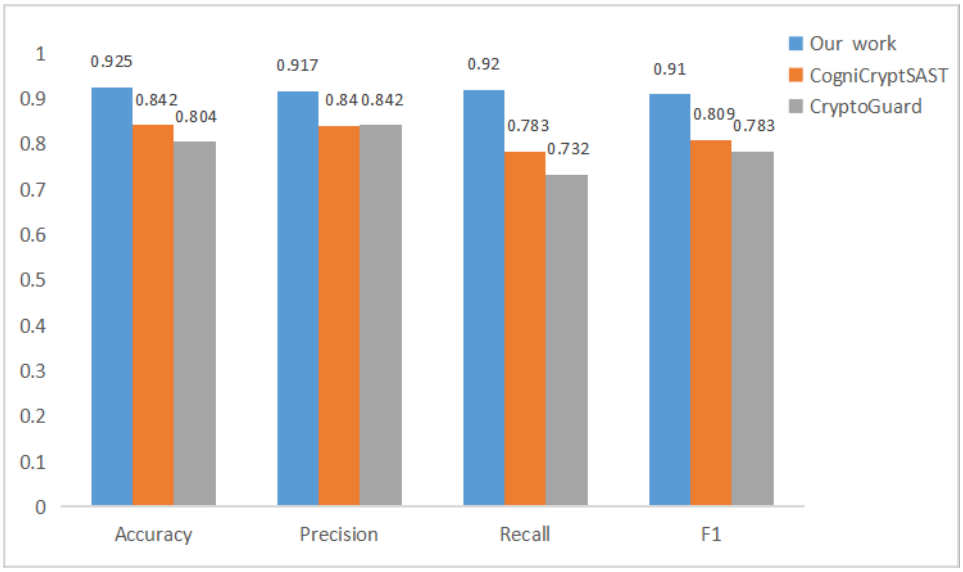

Figure 4. Performance comparison of the proposed detection tool and the existing ones.

4.5.4. Integration with Other Methods

This study also aims to integrate the techniques in this research with other machine learning techniques in order to anticipate better classification outcomes. To this end, we fix the MLP at two and conduct trials combining the methods.

The results shown in Table 6 demonstrate that the model created in this work performs noticeably better than the model developed using other methods, proving that our model was more successfully created. The results of the experiments demonstrate how much more effective our approach is in detecting Android cryptography misuse. The results of classifying our data using the SVM model after two layers of MLP are superior to those of Fischer's suggested model without using MLP, as seen in the table. Additionally, the transformer-based method integrates well with other methodologies, demonstrating the generalizability of transformer-based feature extraction.

Table 6. Integration with other methods.

Model	Accuracy	Precision	Recall	F1
MLP	0.953	0.978	0.935	0.956
MLP + Decision Trees	0.844	0.903	0.902	0.902
MLP + SVM	0.923	0.934	0.911	0.922
MLP + Random Forest	0.812	0.823	0.723	0.770

5. Discussion

In this paper, we analyze how Android applications abuse cryptography APIs using a neural network model-based methodology. We developed a dataset of static program-slice-analyzed Smali code slices about cryptography. Through comparative tests, we tested the performance of our model, and the results showed that our model outperforms SVM-based models and rule-based matching detection tools in detecting cryptographic API abuse in Android applications.

Future work will involve extending binary classification to multi-classification in order to more precisely identify the abuse of cryptographic APIs. Additionally, by making the granularity of the slices finer, we aim to increase the accuracy of the detected samples.There are problems with the computational effort involved in using our model for detection, and in subsequent work, we will continue to adapt the model with with the aim of reducing its computational effort.

Author Contributions: Methodology, L.W.; software, L.W.; investigation, T.S., L.K. and Y.Z.; writing—original draft, L.W.; writing—review and editing, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Key R&D Plan of Shandong Province (No. 2022CXGC020106), Major Innovation Project of Science, Education and Industry of Shandong Academy of Sciences (No. 2022JBZ01-01).

Data Availability Statement: The experimental data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 13th International Conference on Mining Software Repositories, Austin, TX, USA, 14–15 May 2016; pp. 468–471.
2. Egele, M.; Brumley, D.; Fratantonio, Y.; Kruegel, C. An empirical study of cryptographic misuse in android applications. In Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 73–84.
3. Shuai, S.; Guowei, D.; Tao, G.; Tianchang, Y.; Chenjie, S. Modelling analysis and auto-detection of cryptographic misuse in android applications. In Proceedings of the IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, Dalian, China, 24–27 August 2014; pp. 75–80.
4. Krüger, S.; Späth, J.; Ali, K.; Bodden, E.; Mezini, M. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2382–2400. [CrossRef]
5. Rahaman, S.; Xiao, Y.; Afrose, S.; Shaon, F.; Tian, K.; Frantz, M.; Kantarcioglu, M.; Yao, D.D. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019.
6. Cortes, C.; Vapnik, V. Support vector machine. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
7. Weiser, M. Program slicing. *IEEE Trans. Softw. Eng.* **1984**, *10*, 352–357. [CrossRef]
8. Hoffmann, J.; Ussath, M.; Holz, T.; Spreitzenbarth, M. Slicing droids: Program slicing for smali code. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1844–1851.
9. Settles, B. *Active Learning Literature Survey*; University of Wisconsin: Madison, WI, USA, 2009.
10. MacQueen, J. Classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*; University of California: Berkeley, CA, USA, 1967; pp. 281–297.
11. Fahl, S.; Harbach, M.; Muders, T.; Baumgärtner, L.; Freisleben, B.; Smith, M. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In Proceedings of the ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 50–61.

12. Desnos, A.; Gueguen, G. Androguard Documentation. 2011. Available online: https://androguard.readthedocs.io/ (accessed on 11 October 2022).
13. Ma, S.; Lo, D.; Li, T.; Deng, R.H. Cdrep: Automatic repair of cryptographic misuses in android applications. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, Xi'an, China, 30 May–3 June 2016; pp. 711–722.
14. Muslukhov, I.; Boshmaf, Y.; Beznosov, K. Source attribution of cryptographic api misuse in android applications. In Proceedings of the Asia Conference on Computer and Communications Security, Incheon, Republic of Korea, 4–8 June 2018; pp. 133–146.
15. Fischer, F.; Böttinger, K.; Xiao, H.; Stransky, C.; Acar, Y.; Backes, M.; Fahl, S. Stack overflow considered harmful? The impact of copy&paste on android application security. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 121–136.
16. Xu, Z.; Hu, X.; Tao, Y.; Qin, S. Analyzing cryptographic api usages for android applications using hmm and n-gram. In Proceedings of the International Symposium on Theoretical Aspects of Software Engineering (TASE), Hangzhou, China, 11–13 December 2020; pp. 153–160.
17. Zhang, L.; Chen, J.; Diao, W.; Guo, S.; Weng, J.; Zhang, K. CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In Proceedings of the RAID—22nd International Symposium on Research in Attacks, Intrusions and Defenses, Beijing, China, 23–25 September 2019; pp. 151–164.
18. Li, W.; Jia, S.; Liu, L.; Zheng, F.; Ma, Y.; Lin, J. CryptoGo: Automatic Detection of Go Cryptographic API Misuses. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; pp. 318–331.
19. An, C.; Zhang, D.; Gao, X.; Zhu, X. CryptoDetection: A Cryptography Misuse Detection Method Based on Bi-LSTM. In Proceedings of the IEEE 8th International Conference on Computer and Communications (ICCC), Chengdu, China, 9–12 December 2022; pp. 1244–1249.
20. Wickert, A.K.; Baumgärtner, L.; Breitfelder, F.; Mezini, M. Python Crypto Misuses in the Wild. In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Bari, Italy, 11–15 October 2021; pp. 1–6.
21. Rahaman, S.; Cai, H.; Chowdhury, O.; Yao, D. From Theory to Code: Identifying Logical Flaws in Cryptographic Implementations in C/C++. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 3790–3803. [CrossRef]
22. Rodrigues, G.E.d.P.; Braga, A.M.; Dahab, R. Using graph embeddings and machine learning to detect cryptography misuse in source code. In Proceedings of the 19th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 14–17 December 2020; pp. 1059–1066.
23. Grover, A.; Leskovec, J. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 855–864.
24. Silva, F.B. Bag of Graphs = Definition, Implementation, and Validation in Classification Tasks. 2014. Available online: https://repositorio.unicamp.br/acervo/detalhe/932510 (accessed on 26 May 2023).
25. Wognsen, E.R.; Karlsen, H.S.; Olesen, M.C.; Hansen, R.R. Formalisation and analysis of Dalvik bytecode. *Sci. Comput. Program.* **2014**, *92*, 25–55. [CrossRef]
26. Winsniewski, R. Android–apktool: A tool for reverse engineering android apk files. *Retrieved Febr.* **2012**, *10*, 2020.
27. Feichtner, J. A Comparative Study of Misapplied Crypto in Android and iOS Applications. In Proceedings of the ICETE (2), Hyderabad, India, 22–23 March 2019; pp. 96–108.
28. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*; NeurIPS: Vancouver, BC, Canada, 2017; Volume 30.
29. Prince, M. Does active learning work? A review of the research. *J. Eng. Educ.* **2004**, *93*, 223–231. [CrossRef]
30. Urner, R.; Wulff, S.; Ben-David, S. Plal: Cluster-based active learning. In Proceedings of the Conference on Learning Theory—PMLR, Princeton, NJ, USA, 12–14 June 2013; pp. 376–397.