CryptoLLM: Harnessing the Power of LLMs to Detect Cryptographic API Misuse

Heewon Baek, Minwook Lee, and Hyoungshick Kim(✉)

Sungkyunkwan University, Suwon, Republic of Korea
{heewb9818,mwlee,hyoung}@skku.edu

Abstract. We propose CryptoLLM, a novel static analysis tool leveraging large language models (LLMs) to detect cryptographic API misuse vulnerabilities. Integrating optimized code slicing with fine-tuned LLMs, CryptoLLM achieves superior detection capabilities. After evaluating four models, we recommend CodeT5. CryptoLLM outperforms existing rule-based tools such as CryptoGuard, CogniCrypt, and SpotBugs on the CryptoAPI-Bench dataset (F1 score: 0.935). For unseen real-world Android apps, with a 20-minute analysis limit, CryptoLLM achieved the highest F1 score of 0.898, analyzing all apps without errors, while other tools failed to analyze a significant proportion, with CryptoGuard's highest F1 score at 0.645. Although CryptoLLM's performance initially dropped to 0.749 F1 score on mutated code, retraining with augmented data improved it to 0.988, demonstrating adaptability across diverse datasets.

1 Introduction

Cryptographic APIs are essential tools that enable developers to offer a standardized approach to implementing complex cryptographic functions, such as encryption and digital signing, without requiring extensive knowledge of cryptography. By leveraging secure implementations of cryptographic algorithms, developers can concentrate on building their applications while ensuring the protection of sensitive data and establishing secure communication channels. Consequently, cryptographic APIs play a crucial role in developing secure and trustworthy software systems.

However, the intricacy of cryptographic primitives often presents challenges for software developers, making it difficult to utilize cryptographic APIs correctly. The documentation for these APIs may lack comprehensive explanations or present information ambiguously, complicating their proper usage [9,17]. This scarcity and ambiguity in documentation prompt developers to seek examples from the open-source community, such as Stack Overflow, where instances of misuse are not uncommon [15]. Utilizing these APIs based on incorrect knowledge or flawed examples can often introduce vulnerabilities, potentially leading to data theft, information leakage, financial losses, and system failures.

J. Garcia-Alfaro et al. (Eds.): ESORICS 2024, LNCS 14982, pp. 353–373, 2024.
https://doi.org/10.1007/978-3-031-70879-4_18

In practice, the misuse of cryptographic APIs remains widespread. Egele et al. [13] discovered that 88% of 11,748 applications utilizing cryptographic APIs on the Google Play marketplace exhibited at least one instance of misuse. Similarly, Gajrani et al. [16] evaluated 7,000 apps from the seven most popular Android app stores and found that approximately 90% were vulnerable to exploits due to cryptographic weaknesses.

Numerous studies have proposed tools for detecting cryptographic API misuses, including BinSight [27], CDRep [22], CogniCrypt$_{\text{SAST}}$ [19], CryptoGuard [30], CryptoLint [13], and FixDroid [28]. Although these tools have shown effectiveness in identifying vulnerabilities, their dependence on manually configured rules can result in failures to detect anomalies that marginally deviate from these rules or produce false positives [7].

To overcome these limitations, we introduce CryptoLLM, a novel static analysis tool designed for flexible and efficient detection of cryptographic API misuses. CryptoLLM leverages the power of large language models (LLMs), which can dynamically learn patterns without the need for constant rule updates. By employing an LLM-based approach, CryptoLLM aims to provide a more adaptable and accurate solution for identifying cryptographic API misuses, reducing the risk of false negatives and false positives. This novel approach has the potential to significantly improve the security of software systems by enabling developers to quickly and reliably detect and remediate vulnerabilities arising from the improper use of cryptographic APIs.

A large-scale dataset is crucial for effectively applying LLMs to detect cryptographic API misuses. Previous studies have introduced datasets such as the 4,019 security-related code snippets from Stack Overflow [15] and the CryptoAPI-Bench benchmark [3]. However, these often have limitations for model training due to their size and occasional mislabeling issues. To address this, we present a new open cryptographic API misuse vulnerability dataset for Java, compiled by manually analyzing 80,000 real-world Android applications downloaded from AndroZoo [4]. This dataset was enhanced with mutated data generated by the MASC framework [7] to improve diversity and model robustness.

In CryptoLLM, we develop an LLM to detect cryptographic API misuse by learning from code snippets that capture the context and patterns useful for identifying code vulnerabilities. The code snippet generation process involves several steps. First, we identify relevant code segments and locate cryptographic API usage. Next, we create Control Flow Graphs (CFG) and Data Flow Graphs (DFG) to represent the code's structure and data dependencies. Relevant code snippets are then extracted using program slicing on the CFG and the DFG. We create an Abstract Syntax Tree (AST) and abstract user-defined functions and variables to generalize the snippets.

Through this code snippet generation process, we collected 97,962 manually labeled snippets, consisting of 41,018 benign snippets and 56,944 misuse snippets, covering five types of cryptographic API misuse vulnerabilities. This comprehensive dataset serves as a valuable resource for training and evaluating

Table 1. Cryptographic API misuse rules used in CryptoLLM.

Rule No.	Rule Description
Rule-1	Do not use a weak symmetric encryption algorithm/mode.
Rule-2	Do not use a predictable hardcoded cryptographic key.
Rule-3	Do not use a predictable hardcoded password for the keystore.
Rule-4	Do not use a short key (< 2048 bits) for RSA.
Rule-5	Do not use a predictable, static IV in block cipher modes.

cryptographic API misuse detection models, enabling the development of effective security tools.

We evaluated four LLM models–CodeBERT [14], CodeGPT [21], CodeT5 [33], and ELECTRA [10]–on our collected dataset. CodeT5 demonstrated superior performance with an F1 score of 0.942, leading us to recommend it for our approach. Without retraining, this model achieved a 0.935 F1 score on the publicly available CryptoAPI-Bench dataset. In real-world applications, CryptoLLM (utilizing CodeT5) significantly outperformed existing tools, achieving an F1 score 25.3% higher than CryptoGuard [30]. CryptoLLM successfully analyzed all code snippets, compared to 29–83% for other tools, underscoring its effectiveness in practical scenarios.

Our key contributions are as follows:

- We developed CryptoLLM, a high-accuracy tool leveraging large language models for cryptographic API misuse detection.
- We demonstrated CryptoLLM's effectiveness and robustness through comprehensive performance comparisons with existing tools on real-world Android applications and mutated code snippets, revealing significant improvements over traditional methods.
- We introduced a comprehensive dataset for cryptographic API misuse vulnerabilities, comprising 97,962 code snippets (17,661 original and 80,301 mutated samples). The CryptoLLM source code and dataset are publicly available at https://github.com/heewonB/CryptoLLM, with data curated in JSONL format, ensuring that decompiled source code and application names are not disclosed. This extensive dataset is poised to become a standard benchmark for future research in the field of cryptographic API misuse vulnerability detection.

2 Cryptographic API Misuse Vulnerabilities

In this section, we present the cryptographic API misuse vulnerabilities that CryptoLLM aims to detect. The misuse rules used in CryptoLLM were selected based on the common rules supported by three existing tools: CryptoGuard [30], CogniCrypt [18], and SpotBugs [2]. This allows for a fair comparison between CryptoLLM and these tools. Table 1 presents the cryptographic API misuse rules

for CryptoLLM, and the details of the vulnerabilities indicated by these rules are as follows.

(Rule-1) Do Not Use a Weak Symmetric Encryption Algorithm/-Mode. The Data Encryption Standard (DES) algorithm, a 56-bit symmetric cipher, is considered inadequate for modern security requirements. Due to its relatively short key length, DES is vulnerable to feasible straightforward attacks using modern computing resources, such as brute-force attacks [23]. In contrast, the Advanced Encryption Standard (AES) algorithm provides higher security levels by offering longer key lengths (128, 192, and 256 bits) and robust encryption techniques. Therefore, it is recommended to use AES instead of DES [26].

A secure encryption mode should be used instead of Electronic Code Book (ECB) mode, which does not guarantee confidentiality. ECB mode produces identical ciphertext blocks for identical plaintext blocks, exposing patterns and making it vulnerable to attacks. In contrast, block cipher modes like Cipher Block Chaining (CBC), Counter (CTR), and Galois/Counter (GCM) provide enhanced security. CBC and CTR modes improve confidentiality by generating different ciphertext blocks for identical plaintext blocks, while GCM, an authenticated encryption mode, ensures both confidentiality and integrity. To ensure data protection, it is highly recommended to use secure modes like CBC, CTR, or GCM instead of ECB [12].

(Rule-2) Do Not Use a Predictable Hardcoded Cryptographic Key. Predictable encryption keys should not be used. The attacker can easily predict these keys, allowing them to recover data and gain access to accounts. Additionally, since source code is often public or shared, hardcoding cryptographic keys within source code can pose a serious security threat. To maintain security, it is important to use unpredictable cryptographic keys and securely store them in separate configuration files or keystores rather than hardcoding them within the source code [25].

(Rule-3) Do Not Use a Predictable Hardcoded Password for the Keystore. Predictable passwords for the keystore should not be used. The attacker can easily guess these passwords and gain access to the keystore. This could seriously compromise the security of the entire system. Therefore, rather than hardcoding passwords to access the keystore, it is important to use unpredictable passwords and manage them securely (e.g., using separate configuration files or an external key management system to store and manage passwords) [24].

(Rule-4) Do Not Use a Short Key ($<$ 2048 Bits) for RSA. The length of the key used in the RSA algorithm directly affects the security level of encryption. The shorter the key length, the easier it becomes to decrypt encrypted data, and brute-force attacks become more feasible. National Institute of Standards and Technology (NIST) recommends using a key length of at least 2,048 bits

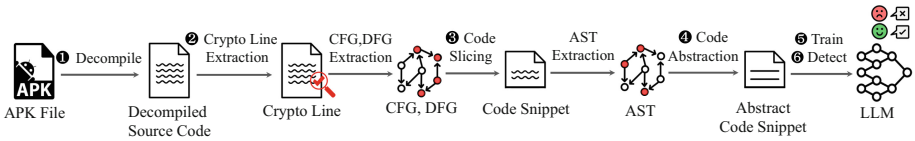

Fig. 1. Overview of CryptoLLM.

when using RSA encryption [8]. This ensures safety because it requires time and resources that are not realistically possible with current computing technology.

(Rule-5) Do Not Use a Predictable IV in Block Cipher Modes. In block cipher modes like CBC, CTR, and GCM, the Initialization Vector (IV) or nonce must be unpredictable. Using a predictable IV can lead to vulnerabilities, allowing attackers to bypass encryption and compromising confidentiality. Furthermore, reusing IVs across messages or sessions can result in attacks that exploit ciphertext patterns. To mitigate these risks, it is essential to use a secure random number generator to produce unique and unpredictable IVs for each message or session [12].

3 Overview of CryptoLLM

This paper presents CryptoLLM, a novel machine learning-based cryptographic API misuse detection tool. CryptoLLM aims to identify vulnerabilities in Android applications by analyzing the usage of cryptographic APIs in the source code. The tool consists of two main phases: (1) the training phase and (2) the detecting phase. In the training phase, CryptoLLM processes a dataset of APK files to extract relevant code snippets containing cryptographic API usage and trains an LLM on these code snippets. In the detecting phase, CryptoLLM takes a targeted APK file as input, extracts code snippets in the same manner as the training phase, and uses the trained model to detect vulnerabilities in the extracted code snippets.

Figure 1 illustrates the workflow of CryptoLLM. Essentially, the training phase and the detecting phase operate in the same manner. In the training phase, instead of a single sample, learning is conducted on many samples to optimize the parameters of the LLM in order to reduce the loss function for a specific task. The main steps involved in both phases are as follows:

1. **APK Decompilation (❶):** CryptoLLM uses Jadx[1] to decompile the APK files and obtain the decompiled source code.
2. **Crypto Line Extraction (❷):** CryptoLLM identifies lines of code that use cryptographic APIs through program slicing. It utilizes pre-defined slicing criteria that define the cryptographic APIs for this purpose.

[1] Command line tool for producing Java source code from Android DEX and APK files (https://github.com/skylot/jadx).

3. **Code Slicing (❸):** When crypto lines are detected, comments are removed from the source code. The Control Flow Graph (CFG) and Data Flow Graph (DFG) are constructed to analyze the control structures and relationships between data associated with the crypto lines. Backward slicing is performed on the DFG based on the parameters of interest in the crypto line, and forward and backward slicing is applied on the CFG for the extracted code lines to create a code snippet.
4. **Code Abstraction (❹):** The Abstract Syntax Tree (AST) is generated to identify and remove unnecessary code elements. User-defined functions and variables are replaced with abstract representations, simplifying the code snippets while preserving their essential structure and semantics.
5. **Model Training (❺):** In the training phase, the created code snippets are used to train an LLM.
6. **Vulnerability Detection (❻):** In the detecting phase, the trained LLM is employed to detect vulnerabilities in the code snippets extracted from the targeted APK file, and vulnerability results are provided for the APK file.

By following this workflow, CryptoLLM can effectively identify cryptographic API misuse vulnerabilities in Android applications, leveraging the power of machine learning and code analysis techniques.

4 Implementation of CryptoLLM

This section describes the implementation of CryptoLLM, the tool designed to detect cryptographic API misuse. The model is constructed by learning from code snippets that may contain misuse. To train the LLM, we performed a binary classification task on the generated code snippets, enabling the model to identify and understand patterns associated with cryptographic API misuse.

4.1 Code Snippet Generation

When training or detecting vulnerabilities in Java files using an LLM, many code segments unrelated to cryptographic APIs can degrade the model's performance and hinder effective vulnerability detection. To enhance learning efficiency and model performance, we extract only semantically related code segments to create compact code snippets through a three-step process: (1) crypto line extraction, (2) code slicing, and (3) code abstraction. The resulting code snippets are efficiently processed by the LLM models we considered, such as CodeBERT-base (512 tokens) [14], CodeGPT-small (1,024 tokens) [21], CodeT5-small (512 tokens) [33], and ELECTRA-base (512 tokens) [10]. To minimize the impact of token length, we set a maximum token length of 512 for all models, including CodeGPT-small. In cases where code snippets exceed this limit (approximately 19% for CodeT5, 24% for CodeBERT and CodeGPT, and 26% for ELECTRA), we use only the leading part of the snippet that satisfies the maximum token size. We also limit the maximum CFG and DFG graph generation time to 2 min during analysis, affecting only 5% of our dataset.

Table 2. Cryptographic APIs used as slicing criteria. The boldface indicates the parameters of interest.

Rule No.	Slicing Criteria APIs
Rule-1	<javax.crypto.Cipher: Cipher getInstance(**String**)>
	<javax.crypto.Cipher: Cipher getInstance(**String**, String)>
	<javax.crypto.Cipher: Cipher getInstance(**String**, Provider)>
Rule-2	<javax.crypto.spec.SecretKeySpec: void <init>(**byte[]**, String)>
	<javax.crypto.spec.SecretKeySpec: void <init>(**byte[]**, int, int, String)>
Rule-3	<java.security.KeyStore: void load(InputStream, **char[]**)>
	<java.security.KeyStore: void store(OutputStream, **char[]**)>
	<java.security.KeyStore: Key getKey(String, **char[]**)>
Rule-4	<java.security.KeyPairGenerator: void initialize(**int**)>
	<java.security.KeyPairGenerator: void initialize(**int**,SecureRandom)>
	<java.security.KeyPairGenerator: void initialize(**AlgorithmParameterSpec**)>
	<java.security.KeyPairGenerator: void initialize(**AlgorithmParameterSpec**, SecureRandom)>
Rule-5	<javax.crypto.spec.IvParameterSpec: void <init>(**byte[]**)>
	<javax.crypto.spec.IvParameterSpec: void <init>(**byte[]**, int, int)>

Crypto Line Extraction. Our goal is to detect cryptographic API misuses in code. To focus on the usage of cryptographic APIs, we first extract lines of code that use cryptographic APIs from the dataset. We define cryptographic APIs that could cause the vulnerabilities we want to detect and use these as slicing criteria. The slicing criteria are detailed in Table 2. To extract crypto lines, we first check whether the classes (e.g., *javax.crypto.Cipher*, *javax.crypto.spec.SecretKeySpec*) corresponding to the slicing criteria APIs are utilized in the Java file. If a specific class is used, we employ regular expressions to check whether the slicing criteria APIs corresponding to the class are used. For Rule-1, we use the class method (*Cipher.getInstance*), and for Rule-2 and Rule-5, we use the instance constructor (*new SecretKeySpec*, *new IvParameterSpec*) as regular expressions. However, Rule-3 and Rule-4 could be confused with functions from other libraries or user-defined functions if identified solely based on function names. So, we find a class instance (*KeyStore*, *KeyPairGenerator*) corresponding to the slicing criteria API and use the code where the instance and function are called together as a regular expression (e.g., *KeyStore.load()*, *KeyPairGenerator.initialize()*). Using these regular expressions, we extract crypto lines that could potentially serve as the starting point for vulnerabilities.

Code Slicing. We use COMEX [11] to find the code segments related to a crypto line. COMEX[2] is an easy-to-use tool built on top of a tree-sitter, a parser generator tool, and an incremental parsing library that provides a multi-code view by extracting structural and semantic properties from source code. We generate a Control Flow Graph (CFG) and a Data Flow Graph (DFG) for a Java file using COMEX to extract the program execution and data flow associ-

[2] https://github.com/IBM/tree-sitter-codeviews.

ated with the crypto line. If the Java file is too long, generating the graph may take too long, so we limit the graph generation time to 2 min. When the graph is successfully generated, DFG backward slicing is performed on the interest parameter of the slicing criteria API, starting from the crypto line. The interest parameters are parameters that could cause vulnerabilities, which means parameters that we pay close attention to. Interest parameters are indicated in bold in the slicing criteria API in Table 2. When first proceeding, only the code lines whose interest parameters of crypto lines are affected in terms of data are sliced backward. Afterward, regardless of the interest parameter, we conduct backward slicing on lines collected through the first backward slicing, identifying lines influenced by data aspects. To consider the code execution flow, we perform forward and backward slicing through the CFG for the code lines collected through DFG slicing. While performing CFG slicing, if the code line is an if statement, we also collect code lines using DFG backward slicing with a depth of 1 to check the data affecting the if statement. Finally, we create a code snippet by arranging the collected code lines obtained through code slicing using the CFG and DFG in the order of the original code.

Code Abstraction. User-defined functions, variables, and comments are not important for vulnerability analysis. Therefore, we aim to remove or replace unnecessary code to analyze the code structure and extract only the core code needed for vulnerability analysis. We remove comments at the input stage before creating the code snippets. To handle user-defined functions and variables, we generate an AST for the code snippets using COMEX. The AST generation time is also limited to 2 min, the same as the CFG and DFG generation time. In the AST generated by COMEX, functions and variables are defined with node types as the identifier. Therefore, we extract only values whose node type is the identifier from the AST. However, the identifier also contains classes (e.g., *java.util.Base64*, *java.lang.System*) provided by Java, so we do not extract any values in this case. Then, we compare the code snippet and identifier values. First, if there is an identifier value in the function declaration part of the code, the function name is replaced with FUNi (the ith ordered function name). Moreover, if there is an identifier value in the part where the variable in the code is used, the variable name is replaced with VARi (the ith ordered variable name). In this way, we abstract the code snippets through the AST and use the processed code snippets for model training or detection.

4.2 LLM Construction

To enhance the detection of cryptographic API vulnerabilities, we use a fine-tuned LLM that performs binary classification on code snippets. The LLM is trained on a dataset consisting of annotated code snippets, each labeled as either benign or containing cryptographic API misuse. By learning from these examples, the model develops a deep understanding of secure and insecure coding practices, enabling it to accurately identify potential vulnerabilities in new,

unseen code. The fine-tuning process allows the LLM to adapt its pre-existing knowledge of code semantics and context to the specific task of cryptographic API misuse detection, resulting in a powerful tool for identifying and mitigating security risks in software development.

During the fine-tuning phase, we employ a binary cross-entropy loss function, which is particularly suited for binary classification tasks. This loss function measures the discrepancy between the predicted probabilities and binary labels. The formula for binary cross-entropy loss is expressed as:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Listing 1. Misuse case for Rule-1.

```
Cipher.getInstance("D#ES".replace("#",""));
```

where N is the number of samples, y_i is the true label of the i-th sample, and $\hat{y}_i$ is the predicted probability of the i-th sample being vulnerable. Optimizing this loss function is crucial as it ensures that the models are precisely tuned to minimize errors in detecting vulnerabilities, thereby significantly enhancing the accuracy and reliability of cryptographic API vulnerability detection. This contributes to more secure software development practices and better protection against potential security threats. Additionally, to optimize training efficiency and avoid overfitting, we implemented early stopping and used dropout to improve the generalization capability of the model.

5 Experiments

5.1 Data Collection

Previous studies [3,5,15] have released datasets with limitations in size and potential mislabeling, making them inadequate for training advanced machine learning models. To address this, we created a comprehensive dataset for CryptoLLM through a multi-step process.

We began by randomly downloading 80,000 applications from AndroZoo [4], focusing on APK files dated between January 1, 2015, and November 17, 2023, and larger than 1,000 bytes to ensure potential cryptographic API usage. For vulnerability detection and initial labeling, we employed CryptoGuard [30], CogniCrypt [18], and SpotBugs [2]. We included only cases with a unanimous agreement or where two tools agreed, and one failed to analyze, limiting analysis time to 20 min per APK for efficiency.

The labeled Java data was converted into code snippets through code slicing, with duplicates removed using the SHA-2 algorithm. Two experts manually relabeled the snippets over a week to ensure reliability and accuracy. This process resulted in 17,661 code snippets, covering five cryptographic API misuse vulnerabilities: weak encryption algorithms, insecure modes of operation, insecure padding schemes, hard-coded keys, and improper initialization vectors.

To evaluate robustness, we used the MASC framework [6] to generate mutations. MASC modifies Java source code across three mutation scopes: *Main scope*, *Similarity scope*, and *Exhaustive scope*. Focusing on the *Similarity scope*, we created 32 benign and 45 misuse code patterns using 7 operators aligned with our detection rules. Listings 1 and 2 provide examples of these code patterns.

Table 3 presents the composition of the CryptoLLM dataset, comprising 97,962 code snippets in total: 17,661 original snippets (8,689 benign and 8,972 misuse cases) and 80,301 mutated snippets (32,329 benign and 47,972 misuse cases).

Listing 2. Benign case for Rule-2.

```
int minute = Integer.parseInt(new SimpleDateFormat("mm").format(new Date()));
byte[] tempBytes = new byte[minute];
new SecureRandom().nextBytes(tempBytes);
new SecretKeySpec(tempBytes, "AES");
```

Table 3. Composition of the CryptoLLM dataset: Distribution of original and mutated code snippets, categorized as benign or misuse, for each cryptographic API misuse rule.

Rule No.	Original		Mutated	
	Benign	Misuse	Benign	Misuse
Rule-1	1,992	3,552	9,509	22,997
Rule-2	1,913	1,295	4,494	2,874
Rule-3	2,273	1,191	8,408	8,781
Rule-4	638	645	7,398	8,179
Rule-5	1,873	2,289	2,520	5,141
Total	8,689	8,972	32,329	47,972

5.2 Experimental Setup

Experiments were conducted on an Ubuntu 18.04 server equipped with an Intel (R) Xeon (R) CPU at 3.10 GHz, 251.0 GB RAM, and an NVIDIA Tesla V100-PCIe GPU with 32 GB memory. All necessary packages were installed in an Anaconda virtual environment to ensure reproducibility.

5.3 Model Optimization

To optimize CryptoLLM's performance for cryptographic API misuse detection, we selected CodeBERT-base (125 M) [14], CodeGPT-small (124 M) [21], CodeT5-small (60 M) [33], and ELECTRA-base (110 M) [10] as our candidate LLMs. These models, pre-trained on extensive code corpora, were fine-tuned to provide the best detection accuracy.

We divided the original dataset into training, validation, and test sets using a 3:1:1 ratio, ensuring a balanced representation of each cryptographic rule across all sets. The models were fine-tuned on the training set using various combinations of batch sizes (16, 32, 64) and learning rates (1e-4, 1e-5, 1e-6). To prevent overfitting, we implemented early stopping, which resulted in varying numbers of training epochs for each model. Optimal hyperparameters were determined using the validation set, with the F1 score as the primary performance metric.

Table 4 shows the F1 scores for each model under the different parameter combinations. The best F1 score for each model is highlighted in bold. Except for CodeGPT, all models achieved their highest F1 score with a batch size of 16. Additionally, except for CodeBERT, all models achieved their best performance with a learning rate of 1e-4.

Table 4. Model optimization based on F1 score.

Parameters		Models			
Batch Size	Learning Rate	CodeBERT	CodeGPT	CodeT5	ELECTRA
16	1e-4	0.933	0.934	**0.945**	**0.927**
16	1e-5	**0.941**	0.938	0.937	0.723
16	1e-6	0.937	0.935	0.932	0.921
32	1e-4	0.933	**0.943**	0.939	0.880
32	1e-5	0.937	0.935	0.930	0.703
32	1e-6	0.937	0.936	0.929	0.924
64	1e-4	0.921	0.932	0.934	0.926
64	1e-5	0.937	0.934	0.925	0.795
64	1e-6	0.938	0.936	0.931	0.866

Table 5. Model Training Time and Epochs.

Model	Training Time (s)	Number of Epochs
CodeBERT	1,364	5
CodeGPT	3,976	13
CodeT5	810	4
ELECTRA	1,856	7

Table 5 presents the training time and the number of epochs for each model using the training dataset. CodeT5 had the shortest training time of 810 s (about 13 min) and required the least number of epochs (4) to converge, likely due to its smaller model size of 60 M parameters compared to the other models.

Table 6 presents a comparative analysis of each optimized model's performance on the test dataset. All models showed a slight decrease in performance compared to their validation set results. CodeT5 demonstrated superior capability in detecting cryptographic API misuses, achieving the highest F1 score of 0.942. While CodeGPT had a lower F1 score, it achieved the lowest perplexity (1.197), indicating its proficiency in predicting real data distribution. However, for our specific cryptographic API misuse detection task, the F1 score serves as a more direct performance indicator. Based on these results, we recommend CodeT5 as the primary LLM for further experiments and designate it as the CryptoLLM model.

6 Evaluation

In this section, we evaluate the performance and robustness of CryptoLLM on three datasets: (1) a benchmark dataset, (2) real-world Android apps, and (3) mutated datasets. We compare CryptoLLM with existing state-of-the-art tools to assess its effectiveness in detecting cryptographic API misuse vulnerabilities in the benchmark dataset and real-world Android apps, as well as its resilience against code mutations that existing tools struggle to detect [7].

Table 6. Performance comparison of LLM models.

Model	Accuracy	Precision	Recall	F1 score	Perplexity
CodeBERT	0.938	0.938	0.938	0.938	1.243
CodeGPT	0.937	0.937	0.937	0.937	**1.197**
CodeT5	**0.942**	**0.942**	**0.942**	**0.942**	1.206
ELECTRA	0.920	0.924	0.921	0.920	1.276

Table 7. Performance of tools for CryptoAPI-Bench.

Rule No.	CryptoLLM					CryptoGuard					CogniCrypt					SpotBugs				
	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1
Rule-1	9	8	0	1	**0.947**	10	2	6	0	0.769	7	6	2	3	0.737	2	6	2	8	0.286
Rule-2	6	2	0	0	**1.000**	5	1	1	1	0.833	6	0	2	0	0.857	1	1	1	5	0.250
Rule-3	4	3	0	2	0.800	6	2	1	0	**0.923**	6	0	3	0	0.800	2	3	0	4	0.500
Rule-4	4	1	0	0	**1.000**	3	0	1	1	0.750	4	0	1	0	0.889	1	1	0	3	0.400
Rule-5	6	2	0	1	0.923	6	1	1	1	0.857	7	0	2	0	0.875	7	1	1	0	**0.933**
All	29	16	0	4	**0.935**	30	6	10	3	0.822	30	6	10	3	0.822	13	12	4	20	0.520

6.1 Performance Comparison on the CryptoAPI-Bench Dataset

We conducted a comparative analysis of the CryptoLLM model against existing tools. To evaluate performance, we employed the publicly available CryptoAPI-Bench [3], a dataset comprising 171 Java test files. Our analysis focused on 49 Java files (16 benign and 33 misuse cases) specifically related to the five rules central to our study. These files served as unseen test cases, allowing us to assess CryptoLLM's vulnerability detection accuracy without retraining. This approach not only evaluated the model's performance but also demonstrated its generalization capabilities, addressing potential overfitting concerns and showcasing its robustness.

Table 7 presents the comparative performance results. CryptoLLM achieves the highest F1 score of 0.935 on the CryptoAPI-Bench dataset, outperforming existing tools such as CryptoGuard, CogniCrypt, and SpotBugs. It had only four false negatives and no false positives, while other tools had many false positives and false negatives, demonstrating CryptoLLM's superiority.

In particular, CryptoLLM performed well on Rule-1, Rule-2, and Rule-4. This superior performance might be attributed to the fact that these rules involve detecting specific, well-defined patterns in the code. For example, Rule-1 requires identifying weak symmetric encryption algorithms or modes that the model can easily recognize. Similarly, Rule-2 and Rule-4 involve detecting hard-coded cryptographic keys and short RSA keys, respectively, which are relatively straightforward patterns for the model to learn and identify.

However, CryptoLLM's performance on Rule-3 and Rule-5 was slightly lower compared to tools like CryptoGuard and SpotBugs. These rules involve more complex patterns and contextual information, such as detecting hardcoded passwords for keystores (Rule-3) and predictable IVs in block cipher modes (Rule-5). These scenarios require a deeper understanding of the cryptographic context and surrounding code, which may be more challenging for the model to learn accurately.

These results suggest that existing tools and LLM-based models like CryptoLLM can be used as complementary solutions, each leveraging their strengths in detecting specific types of vulnerabilities.

Table 8. Performance of LLM models for CryptoAPI-Bench.

Rule No.	CodeBERT					CodeGPT					CodeT5					ELECTRA				
	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1
Rule-1	7	7	1	3	0.778	9	8	0	1	**0.947**	9	8	0	1	**0.947**	10	4	4	0	0.833
Rule-2	5	2	0	1	0.909	3	1	1	3	0.600	6	2	0	0	**1.000**	3	2	0	3	0.667
Rule-3	6	3	0	0	**1.000**	6	3	0	0	**1.000**	4	3	0	2	0.800	6	3	0	0	**1.000**
Rule-4	3	0	1	1	0.750	4	1	0	0	**1.000**	4	1	0	0	**1.000**	4	1	0	0	**1.000**
Rule-5	7	1	1	0	0.933	7	2	0	0	**1.000**	6	2	0	1	0.923	7	2	0	0	**1.000**
All	28	13	3	5	0.875	29	15	1	4	0.921	29	16	0	4	**0.935**	30	12	4	3	0.896

Table 9. Performance of tools for real-world Android apps.

Model	F1 score	Complete
CryptoLLM	**0.898**	**100%**
CryptoGuard	0.645	47.3%
CogniCrypt	0.340	29.5%
SpotBugs	0.467	82.8%

Table 8 shows the performance of other LLMs. While CodeT5 achieved the highest F1 score of 0.935, all four LLM models outperformed the three existing tools, demonstrating the potential of LLM-based approaches in cryptographic API misuse detection.

6.2 Performance Comparison on Unseen Real-World APK Files

To assess CryptoLLM's real-world effectiveness and generalization capability, we evaluated it on 4,765 manually labeled code snippets from new, unseen Android applications. This dataset, comprising 2,480 benign and 2,285 misuse cases, enabled us to test CryptoLLM's performance in detecting cryptographic API misuse on previously unseen real-world Android apps.

We compared CryptoLLM against CryptoGuard, CogniCrypt, and SpotBugs, limiting analysis time to 20 min per app to simulate real-world constraints. Table 9 presents the results, with the "Complete" column indicating the percentage of analyses finished within the time limit.

CryptoLLM outperformed existing tools, analyzing 100% of APK files without errors and achieving an F1 score of 0.898. In contrast, CryptoGuard, CogniCrypt, and SpotBugs achieved completion rates of 47.3%, 29.5%, and 82.8%, respectively, with CryptoGuard's 0.645 F1 score being the next best after CryptoLLM.

Table 10. Performance of tools for mutated CryptoAPI-Bench.

Rule No.	CryptoLLM					CryptoGuard					CogniCrypt					SpotBugs				
	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1
Rule-1	52	44	4	8	**0.897**	56	10	38	4	0.727	10	36	12	50	0.244	30	10	38	30	0.469
Rule-2	9	4	0	3	0.857	10	3	1	2	**0.870**	12	0	4	0	0.857	2	3	1	10	0.267
Rule-3	36	5	7	0	0.911	36	9	3	0	**0.960**	36	0	12	0	0.857	16	9	3	20	0.582
Rule-4	35	9	1	5	**0.921**	23	3	7	17	0.657	29	3	7	11	0.763	13	3	7	27	0.433
Rule-5	10	4	0	4	0.833	10	4	0	4	0.833	14	0	4	0	0.875	14	4	0	0	**1.000**
All	142	66	12	20	**0.899**	135	29	49	27	0.780	101	39	39	61	0.669	75	29	49	87	0.524

Table 11. Performance of LLM models for mutated CryptoAPI-Bench.

Rule No.	CodeBERT					CodeGPT					CodeT5					ELECTRA				
	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1
Rule-1	52	38	10	8	0.852	57	48	0	3	**0.974**	52	44	4	8	0.897	56	31	17	1	0.868
Rule-2	9	4	0	3	**0.857**	6	2	2	6	0.600	9	4	0	3	**0.857**	7	4	0	5	0.737
Rule-3	36	5	7	0	0.911	36	9	3	0	**0.960**	36	5	7	0	0.911	30	10	2	6	0.882
Rule-4	35	8	2	5	0.909	39	10	0	1	**0.987**	35	9	1	5	0.921	38	10	0	2	0.974
Rule-5	10	2	2	4	0.769	14	4	0	0	**1.000**	10	4	0	4	0.833	12	4	0	2	0.923
All	142	57	21	20	0.874	152	73	5	10	**0.953**	142	66	12	20	0.899	146	59	19	16	0.893

These results demonstrate CryptoLLM's superior generalization capability and robustness in analyzing complex, real-world code snippets. The significant performance gap highlights CryptoLLM's potential to substantially improve cryptographic API misuse vulnerability detection in Android applications, providing strong evidence of its readiness for practical deployment.

6.3 Evaluation of Robustness on the Mutated Dataset

We analyzed the performance of CryptoLLM on mutated datasets because previous research has shown that existing rule-based detection tools are not effective in detecting vulnerabilities in mutated code samples. Ami et al. [7] discovered that 59.2% of the faults in the existing nine major tools, such as CryptoGuard [30], CogniCrypt [18], and SpotBugs [2], QARK [20], and ShiftLeft [1], occurred in the mutated dataset generated through MASC, revealing the limitations of existing tools for mutated datasets. These tools are all rule-based, and to overcome their limitations, we utilized large language models with a higher potential for discovering vulnerabilities by understanding the meaning and structure of the source code. This performance analysis is crucial because mutated code can be created in the real world through techniques such as code obfuscation.

We first generated the mutated dataset of CryptoAPI-Bench using MASC to verify the robustness of the tools. We generated 240 mutated Java files (78 benign and 162 misuse cases) for CryptoAPI-Bench. Table 10 shows the results.

CryptoLLM maintains high performance on the mutated CryptoAPI-Bench dataset, achieving an F1 score of 0.899 while existing tools show significant

Table 12. Model robustness evaluation with original and mutated datasets. "Original" shows F1 scores of models trained on original APKs; "Original + Mutated" shows F1 scores of models trained on both original and mutated (using the MASC) datasets.

Training Dataset	CodeBERT	CodeGPT	CodeT5	ELECTRA
Original	0.751	0.787	0.749	**0.790**
Original+Mutated	0.979	0.985	**0.988**	0.951

performance decreases. This demonstrates CryptoLLM's robustness compared to rule-based tools.

Interestingly, Table 11 shows CodeGPT surpassing CodeT5 in detection accuracy by over 5% (F1 scores: 0.953 vs. 0.899), highlighting CodeGPT's superior handling of code pattern variations. This advantage is likely due to CodeGPT's autoregressive nature, which excels at learning sequential code patterns and adapting to small variations in mutated code. CodeGPT accurately detected all rules except Rule-2 (hardcoded cryptographic keys) in mutated code, demonstrating its strong generalization ability across various code transformations. However, Table 8 reveals CodeGPT's consistent ineffectiveness in detecting Rule-2 in both original and mutated code. This weakness may arise from the diverse forms hardcoded keys can take and the contextual judgment required for their detection. Addressing this challenge could involve specialized training or additional data.

We extended our evaluation to 80,301 mutated datasets (32,329 benign and 47,972 misuse cases) derived from 17,661 real-world Android app code snippets (see Table 3). Table 12 reveals a significant performance decrease on this mutated data, with F1 scores ranging from 0.749 to 0.790. CodeT5, our base model for CryptoLLM, achieved only 0.749, while ELECTRA performed best at 0.790.

To mitigate this performance drop, we retrained the models on a combined dataset of original and mutated code samples. We ensured unbiased evaluation by considering the disparities in quantity, rules, labels, and dataset types. The total 97,962 code snippets were divided into training, validation, and test sets (3:1:1 ratio), maintaining balanced representation across cryptographic rules.

After retraining, CodeT5's performance improved dramatically, reaching an F1 score of 0.988. This significant enhancement emphasizes incorporating mutated code in the training process to boost model robustness. Although continuously learning new mutation code patterns can be challenging, it appears to be a more practical approach compared to adding new rules.

We reassessed these retrained models on the mutated CryptoAPI-Bench dataset (see Table 13). CodeBERT and CodeGPT achieved perfect detection across all rules, each with an F1 score of 1.000. CodeT5 excelled in all rules except Rule-4, which had 5 false negatives, resulting in an F1 score of 0.933 for that rule and an overall F1 score of 0.984. ELECTRA showed lower performance, particularly in Rule-1 and Rule-4, with 12 and 7 false negatives, respectively, leading to F1 scores of 0.889 and 0.904 for these rules. ELECTRA's overall F1 score was 0.938. Notably, none of the models produced any false positives across all rules, demonstrating high precision in their predictions after retraining on mutated code samples.

Table 13. Performance of LLM models on mutated CryptoAPI-Bench after training on both original and mutated datasets.

Rule No.	CodeBERT					CodeGPT					CodeT5					ELECTRA				
	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1	TP	TN	FP	FN	F1
Rule-1	60	48	0	0	**1.000**	60	48	0	0	**1.000**	60	48	0	0	**1.000**	48	48	0	12	0.889
Rule-2	12	4	0	0	**1.000**	12	4	0	0	**1.000**	12	4	0	0	**1.000**	12	4	0	0	**1.000**
Rule-3	36	12	0	0	**1.000**	36	12	0	0	**1.000**	36	12	0	0	**1.000**	36	12	0	0	**1.000**
Rule-4	40	10	0	0	**1.000**	40	10	0	0	**1.000**	35	10	0	5	0.933	33	10	0	7	0.904
Rule-5	14	4	0	0	**1.000**	14	4	0	0	**1.000**	14	4	0	0	**1.000**	14	4	0	0	**1.000**
All	162	78	0	0	**1.000**	162	78	0	0	**1.000**	157	78	0	5	0.984	143	78	0	19	0.938

7 Discussion

7.1 Comparison with Existing Approaches

CryptoLLM demonstrates several advantages over existing rule-based and machine learning-based approaches for detecting cryptographic API misuse vulnerabilities. Compared to rule-based tools like CryptoGuard [30], CogniCrypt [18], and SpotBugs [2], CryptoLLM achieves higher accuracy and robustness, particularly on mutated datasets, due to the use of LLMs that can understand the semantics and context of the code. However, rule-based tools have the advantage of being more interpretable and requiring less training data.

In contrast to previous machine learning-based approaches [15,29,31,32], CryptoLLM leverages state-of-the-art LLMs pre-trained on vast amounts of code, enabling it to capture more complex patterns and achieve higher accuracy. Unlike other ML-based tools that use support vector machines (SVM) and multilayer perceptrons (MLP), which do not inherently understand the context of code, our model performs better in vulnerability detection by comprehending the code itself. Moreover, while many have only trained their models with small datasets due to limited data availability, the reliability of our model is enhanced by increasing the data size. CryptoLLM provides a comprehensive evaluation using diverse benchmark datasets and real-world applications, demonstrating its practical applicability and robustness to variations in the data, which is unclear in other studies that do not conduct experiments with mutated data. Many papers proposed in academia do not properly provide their code and dataset, making direct comparisons with our work impossible despite our requests. To the best of our knowledge, our tool is the first publicly available and reproducible machine learning-based cryptographic API misuse detection tool.

7.2 Generalization of Experimental Results

The experimental results of CryptoLLM demonstrate its effectiveness in detecting cryptographic API misuse vulnerabilities across various datasets, including benchmark datasets, real-world Android apps, and mutated datasets, suggesting

that the approach can be generalized to different coding patterns and environments. However, the performance of CryptoLLM may vary depending on the specific characteristics of the target environment, such as the development framework and programming language.

One limitation of CryptoLLM is its focus on only five cryptographic API misuse rules out of the many possible vulnerabilities, which was necessary to ensure a fair comparison with other tools supporting these rules. Generalizing CryptoLLM to other programming languages or different types of misuse may be challenging, as the model is specifically trained on Java code and the five cryptographic API misuse rules. Further research could explore the adaptability of CryptoLLM to other programming languages and additional misuse types to assess its generalization capabilities.

7.3 Lack of Ground-Truth Labeling

CryptoLLM relies on manual labeling by two experts to create the ground-truth dataset, which may introduce errors despite collaborative discussions to ensure consensus and reduce individual bias. To address this limitation, we have made the dataset publicly available, encouraging the security community to further validate and refine the labels. Additionally, we evaluated CryptoLLM on the widely-used CryptoAPI-Bench dataset [3] to increase confidence in the results and further validate our approach's effectiveness.

8 Related Work

Rule-Based Vulnerability Detection. In previous studies [13,19,22,30], many rule-based vulnerability detection tools have been proposed and are currently widely used. Rahaman et al. [30] proposed CryptoGuard, a static analysis tool that detects 16 vulnerabilities in encryption and SSL/TLS APIs through flow-sensitive, context-sensitive, and field-sensitive analysis to reduce false positives. Kruger et al. [19] defined CrySL, a definition language that allows specifying safe use of cryptographic APIs, and proposed $CogniCrypt_{SAST}$, a CrySL compiler that performs flow-sensitive and context-sensitive static data-flow analysis by parsing CrySL rules and checking types. In addition, many tools such as BinSight [27], CrytoLint [13], and CDRep [22] have been proposed. These rule-based vulnerability detection tools manually inspect cryptographic APIs based on predefined rules, which can lead to false positives and false negatives. They also have the limitation of being unable to detect vulnerabilities in code written to violate the rules and require continuous updating and maintenance of rules whenever new vulnerabilities are discovered.

ML-based Vulnerability Detection. ML-based vulnerability detection is currently receiving significant attention in the security field, and existing studies [29,31,32] show that more flexible and accurate vulnerability detection is possible using artificial intelligence technology compared to traditional rule-based

methods. Rodrigues et al. [31] used the Bag of Graphs (BoG) algorithm and node2vec to extract features from the source code for nine rules and created a classification model using SVM. Rodrigues et al. [29] additionally performed oversampling using an obfuscation technique on the dataset, extracted features from the source code using code2vec, and created classification models using SVM and MLP. Wang et al. [32] preprocessed the APK file into a Smali code file and sliced the code through static backtracking. Using a transformer, they extracted features from the sliced code and created a classification model by training MLP using the k-means clustering-based active learning approach.

Unlike previous ML-based approaches, CryptoLLM leverages state-of-the-art LLMs such as CodeBERT [14], CodeGPT [21], CodeT5 [33], and ELECTRA [10] to detect cryptographic API misuse vulnerabilities. These LLMs have been pre-trained on a vast amount of code and can understand the semantics and context of the code. By fine-tuning these models on our carefully curated dataset, CryptoLLM achieves high accuracy in detecting vulnerabilities across a wide range of real-world applications. We also provide a comprehensive evaluation of CryptoLLM using diverse benchmark datasets and compare its performance with existing tools, demonstrating its effectiveness in detecting cryptographic API misuse. Furthermore, we have released the source code and dataset of CryptoLLM to support future research and development in this area.

9 Conclusion

We introduce CryptoLLM, a novel static analysis tool leveraging LLMs to detect cryptographic API misuses in Java source code. CryptoLLM outperforms existing rule-based tools, demonstrating high detection accuracy and robustness across various conditions, including public benchmark datasets, mutated datasets, and real-world Android applications. These results highlight CryptoLLM's significant potential to improve the detection of cryptographic API misuses, highlighting its practical applicability in real-world scenarios.

Future work will expand CryptoLLM to address a broader range of cryptographic API misuse vulnerability, focusing on complex cases requiring sophisticated analysis and deep code context understanding. By open-sourcing our tools and data, we aim to advance research in cryptographic API misuse vulnerability detection, fostering the development of more secure software across various domains.

Acknowledgments. The authors thank the anonymous reviewers for their valuable input. Their constructive feedback has been instrumental in improving this paper, leading to a more robust presentation of our research. Hyoungshick Kim is the corresponding author. This work was supported by the following grants: the IITP grants (No. 2022-0-00995, RS-2024-00439762; RS-2022-II221199, RS-2024-00438686), an NST grant (Global-23-001), and a KISA grant (No. 1781000003).

References

1. Shiftleft scan (2015). https://shiftleft.io/scan

2. Spotbugs (2024). https://spotbugs.github.io/
3. Afrose, S., Rahaman, S., Yao, D.: CryptoAPI-bench: a comprehensive benchmark on java cryptographic API misuses. In: Proceedings of the IEEE Cybersecurity Development (SecDev) (2019)
4. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the International Conference on Mining Software Repositories (2016)
5. Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M.: MUBench: a benchmark for API-misuse detectors. In: Proceedings of the International Conference on Mining Software Repositories (2016)
6. Ami, A.S., et al.: MASC: a tool for mutation-based evaluation of static crypto-API misuse detectors. In: Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2023)
7. Ami, A.S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., Nadkarni, A.: Why crypto-detectors fail: a systematic evaluation of cryptographic misuse detection techniques. In: Proceedings of IEEE Symposium on Security and Privacy (SP) (2022)
8. Barker, E., Roginsky, A., et al.: Transitions: recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Spec. Publ. **800**, 131A (2011)
9. Braga, A., Dahab, R.: A longitudinal and retrospective study on how developers misuse cryptography in online communities. In: Anais do XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (2017)
10. Clark, K., Luong, M.T., Le, Q.V., Manning, C.D.: Electra: pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555 (2020)
11. Das, D., et al.: COMEX: a tool for generating customized source code representations. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2023)
12. Dworkin, M.J.: SP 800-38A 2001 edition. Recommendation for block cipher modes of operation: methods and techniques. Tech. rep. (2001)
13. 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 and Communications Security (2013)
14. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
15. Fischer, F., et al.: Stack overflow considered harmful? the impact of copy&paste on android application security. In: Proceedings of IEEE Symposium on Security and Privacy (SP) (2017)
16. Gajrani, J., Tripathi, M., Laxmi, V., Gaur, M.S., Conti, M., Rajarajan, M.: sPECTRA: a precise framework for analyzing cryptographic vulnerabilities in android apps. In: Proceedings of the IEEE Annual Consumer Communications & Networking Conference (CCNC) (2017)
17. Green, M., Smith, M.: Developers are not the enemy!: the need for usable security APIs. IEEE Secur. Priv. **14**(5), 40–46 (2016)
18. Krüger, S., et al.: CogniCrypt: supporting developers in using cryptography. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2017)
19. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: an extensible approach to validating the correct usage of cryptographic APIs. IEEE Trans. Software Eng. **47**(11), 2382–2400 (2019)

20. LinkedIn: Introducing qark: An open source tool to improve android application security - linkedin engineering (2015). https://engineering.linkedin.com/blog/2015/08/introducing-qark
21. Lu, S., et al.: Codexglue: a machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021)
22. Ma, S., Lo, D., Li, T., Deng, R.H.: CDRep: automatic repair of cryptographic misuses in android applications. In: Proceedings of the ACM on Asia Conference on Computer and Communications Security (2016)
23. MITRE: CWE-1240: Use of a cryptographic primitive with a risky implementation (2024). https://cwe.mitre.org/data/definitions/1240.html
24. MITRE: CWE-259: Use of hard-coded password (2024). https://cwe.mitre.org/data/definitions/259.html
25. MITRE: CWE-321: Use of hard-coded cryptographic key (2024). https://cwe.mitre.org/data/definitions/321.html
26. MITRE: CWE-327: Use of a broken or risky cryptographic algorithm (2024). https://cwe.mitre.org/data/definitions/327.html
27. Muslukhov, I., Boshmaf, Y., Beznosov, K.: Source attribution of cryptographic API misuse in android applications. In: Proceedings of the ACM on Asia Conference on Computer and Communications Security (2018)
28. Nguyen, D.C., Wermke, D., Acar, Y., Backes, M., Weir, C., Fahl, S.: A stitch in time: supporting android developers in writingsecure code. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2017)
29. de Paula Rodrigues, G.E., Braga, A.M., Dahab, R.: Detecting cryptography misuses with machine learning: graph embeddings, transfer learning and data augmentation in source code related tasks. IEEE Trans. Reliab. **72**, 1678–1689 (2023)
30. Rahaman, S., et al.: CryptoGuard: high precision detection of cryptographic vulnerabilities in massive-sized java projects. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2019)
31. 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 IEEE International Conference on Machine Learning and Applications (ICMLA) (2020)
32. 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 **12**(11), 2460 (2023)
33. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 (2021)