Analyzing Cryptographic API Usages for Android Applications Using HMM and N-Gram

Zhiwu Xu*, Xiongya Hu*, Yida Tao* and Shengchao Qin‡*
*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
‡School of Computing, Engineering and Digital Technologies, Teesside University, UK

***Abstract*—A recent research shows that 88% of Android applications that use cryptographic APIs make at least one mistake. For this reason, several tools have been proposed to detect crypto API misuses, such as CryptoLint, CMA, and CogniCryptS_{AST}. However, these tools depend heavily on manually designed rules, which require much cryptographic knowledge and could be error-prone. In this paper, we propose an approach based on probabilistic models, namely, hidden Markov model and n-gram model, to analyzing crypto API usages in Android applications. The difficulty lies in that crypto APIs are sensitive to not only API orders, but also their arguments. To address this, we have created a dataset consisting of crypto API sequences with arguments, wherein symbolic execution is performed. Finally, we have also conducted some experiments on our models, which shows that (i) our models are effective in capturing the usages, detecting and locating the misuses; (ii) our models perform better than the ones without symbolic execution, especially in misuse detection; and (iii) compared with CogniCryptS_{AST}, our models can detect several new misuses.**

***Index Terms*—Cryptographic API, Android, API analysis, hidden Markov model, n-gram**

I. Introduction

According to a report from IDC [1], Android is the most popular platform for smartphones, with almost 87% the market share in 2019. As a result, Android users started to accumulate a large amount of sensitive and private data on their personal mobile devices that requires confidentiality protection [2]. Besides filesystem-level encryption, one typical way to achieve data confidentiality is to implement custom, application-specific solutions using supported cryptographic APIs (crypto APIs for short). For example, developers can encrypt user data before storing it on the device or transmitting it over the network.

However, developers can easily make mistakes in implementing and using cryptography in their Android applications due to either a lack of cryptographic knowledge or human error, and such mistakes often lead to a false sense of security. A recent research shows that 88% of Android applications that use crypto APIs make at least one mistake [3]. What's worse, as shown in [4], usage updates are unlikely to fix crypto API misuses.

To address this problem, several tools have been proposed to detect crypto API misuse, such as CryptoLint [3], CMA [5], and CogniCryptS_{AST} [6]. However, these tools depend heavily on manually designed rules or models, which require much cryptographic knowledge and could be error-prone. Some existing heuristic approaches on general API usage can be applied on crypto APIs, such as hidden Markov model [7]–[10], n-gram model [11]–[13], and deep learning [14]. But these approaches focus on API orders, leaving arguments out of consideration, so that they are insufficient for crypto APIs, because crypto APIs are sensitive to not only the API orders, but also their arguments. For example, if AES algorithm is used, then the key size should be 128, 192, or 256, but not the others. There are some existing approaches targeting at API arguments [15], [16], but API orders are not considered.

In this paper, we propose an approach to analyzing crypto API usages in Android applications, based on probabilistic models, namely, hidden Markov model and n-gram model, which require less cryptographic knowledge. Different from existing work, we create a dataset consisting of crypto API sequences with arguments. For that, we perform symbolic analysis (*i.e.*, a simple variant of symbolic execution) on program traces extracted from Android applications, and preserve the crypto ones. In order to capture the usage as correct as possible, we also leverage the error reports from CogniCryptS_{AST} [6] to classify the API sequences. Moreover, using these probabilistic models, we propose a new approach to analyzing crypto API usage, which is based on the assumption that *the correct API entails a higher probability of an API sequence while the misuse one does the opposite*.

Several experiments have been conducted to evaluate our approach. Firstly, we used our models to analyze the crypto API usage, including the usage detection and the misuse location. The experimental results have shown that our models can not only capture the correct usages, but also detect the misuses. Secondly, we compared our models with existing approaches using probabilistic models, which have demonstrated that our models perform better than the ones without symbolic execution, especially in misuse detection. Finally, we also compared our models with a state-of-the-art tool CogniCryptS_{AST} [6], and found that our models can detect several new misuses, most of which are due to a wrong argument.

The remainder of this paper is organized as follows. Section II describes our approach, followed by the experimental results in Section III. Section IV presents the related work, followed by some concluding remarks in Section V.

II. Approach

In this section, we present our approach to analyzing crypto API usages in Android applications. Figure 1 shows the

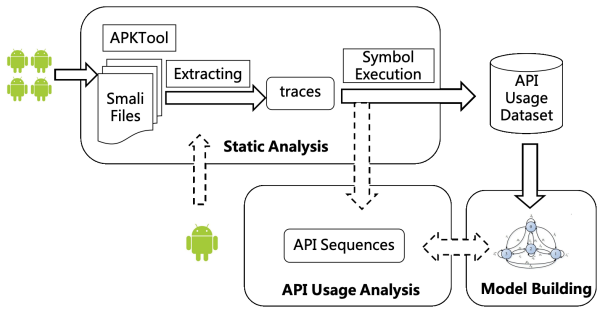

Fig. 1. Framework of Our Approach

framework of our approach, which consists of three tasks: *static analysis*, *model building* and *API usage analysis*. The first task, namely, static analysis, is meant to extract program traces from Android applications and then perform symbolic analysis on these traces, yielding crypto API sequences. The model building task aims to build probabilistic models based on the dataset that is created from an Android application dataset via the first task. Finally, using the models built above, the last task tries to analyze the crypto API sequences for a given Android application, including the usage detection and the misuse location. In what follows, we depict each task of our approach in detail.

A. Static Analysis

This section is devoted to crypto API sequence extraction from an Android application in the instruction level, which consists of three steps: pre-processing, trace extraction, and symbolic analysis.

Pre-Processing. Android applications are distributed in markets in the form of APK. An APK is a compressed archive of Dalvik bytecode for execution, resources, assets, certificates, and an XML manifest file. Among them, the Dalvik bytecode for execution, namely, the file named *classes.dex*[1], will be extracted for further analysis.

For ease of extracting traces, we leverage the disassembler Apktool [17] to disassemble the *dex* files. After disassembling, the *dex* files are converted to *smali* files, which give us the readable code in the *smali* language. We use *smali* code, instead of *Java* code, because the disassembling is lossless in that the *smali* files support the full functionality of the original *dex* files.

For simplicity, we focus on the API sequences that would be possibly executed, so we use FlowDroid [18] to extract the reachable methods for each APK. Moreover, to create an API usage dataset, we also perform a state-of-the-art tool CogniCrypt$_{\text{SAST}}$ [6] on each APK to detect possible crypto API misuses, which are used to classify the crypto API sequences.

Trace Extraction. Generally, a *smali* file contains the definition of a separate class, either a general class or an inner class, in the Java source code. So we extract the traces class by class and method by method for all smali files contained in an APK.

To do this, we first use the static analysis of CDGDroid [19] to identify all the instructions in a method and build its control-flow graph (CFG) with the instructions as nodes and the first instruction as the starting node. Then we traverse the CFG to extract the possible traces for each reachable method.

The procedure of method traversing is given in Algorithm 1, which takes a method m as input and returns a set trs of possible program traces. The algorithm first initializes the set trs as an empty set (line 1) and puts into an empty set S the initial state with the starting node of the method m, an empty trace, and an empty set of backward edges (*i.e.*, edges that go back to some visited nodes) (lines 2-3). It then iterates the set S state by state (lines 4-19). For each state, the algorithm continues to append the current instruction ins into the current trace tr and updates the current instruction as its unique successive one, once the out-degree of the current instruction is 1 (*i.e.*, only one successive instruction) and it is not a *return* instruction (lines 6-8). Otherwise, the current instruction ins is appended into the current trace tr (line 9). If ins is a return one, then the current trace tr is added into the trace set trs (line 10-11). Finally, for each successive instruction, the algorithm checks whether the edge to it exists in the current backward set or not (line 13). If not, the algorithm makes a copy of the trace and the backward set, respectively (lines 14-15). Moreover, if the successive instruction nxt has been visited before, the edge to it would be recorded into the backward set to avoid repeated looping (lines 16-17). After that, the state with the successive instruction nxt, the new trace ntr and the new backward set nbw is added into the state set S, if the state does not exist in S (lines 18-19). Note that, we traverse any loop at most once for any trace, which is ensured by the backward edge checking.

Some APKs encapsulate the Crypto APIs for ease of use or as a part of a common utility. Then when encrypting or decrypting, the encapsulated APIs are called instead. Hence, to address this situation, we consider inter-procedural analysis for trace extraction. Moreover, we also found method calls in such a situation would not be too deep, so we limit ourselves with a depth d for method calls.

The inter-procedural analysis is based on the intra-procedural analysis (*i.e.*, Algorithm 1), which is shown in Algorithm 2. To start with, the algorithm invokes Algorithm 1 to extract the set trs of the intra-procedural traces for the method m (line 1). If the current depth d is not larger than 0, then the intra-procedural trace set trs is returned immediately (line 2-3). Then the algorithm iterates on each trace tr in trs to identify and expand the possible method calls (lines 5-12). And for each instruction ins of the target trace tr, the algorithm performs the following analysis (lines 7-11): if the instruction ins calls a method m and its definition exists in the APK, then the algorithm extracts a possible inter-procedural trace set from the called method m by a recursive call with a depth $d-1$, and expands the called instruction ins by the traces

[1]There may be several additional *dex* files with the name "classes*i*.dex" in large APKs.

Algorithm 1: `Trace_Extraction`

input : a reachable method m
output: a possible trace set trs

```
1  trs ← ∅;
2  S ← ∅;
3  S.add((m.start, [], {}));
4  while S is not empty do
5      (ins, tr, bw) ← pop S;
6      while |m.cfg(ins)| == 1 and ins is not return do
7          tr.append(ins);
8          ins ← m.cfg(ins);
9      tr.append(ins);
10     if ins is return then
11         trs.add(tr);
12     for each nxt in m.cfg(ins) do
13         if (ins, nxt) ∉ bw then
14             ntr ← copy tr;
15             nbw ← copy bw;
16             if nxt ∈ ntr then
17                 nbw.add((ins, nxt));
18             if (nxt, ntr, nbw) ∉ S then
19                 S.add((nxt, ntr, nbw));
20 return trs
```

Algorithm 2: `Trace_Extraction_inter`

input : a reachable method m and a depth d
output: a possible trace set trs

```
1  trs ← Trace_Extraction(m);
2  if d ≤ 0 then
3      return trs;
4  inter_trs ← {};
5  for each trace tr in trs do
6      temp_trs ← {[]};
7      for each instruction ins in tr do
8          if ins calls m and m's definition exists then
9              temp_trs ← temp_trs · {[begin ins]} ·
                 Trace_Extraction_inter(m, d − 1) ·
                 {[end ins]};
10         else
11             temp_trs ← temp_trs · {[ins]};
12     inter_trs.add(temp_trs);
13 return inter_trs;
```

extracted from m via *language product*[2] (lines 8-9), where $begin$ and end are used to mark the beginning and the ending of a method call, respectively; otherwise, the instruction ins remains the same (lines 10-11). Finally, the expanded trace set is returned (line 13).

Symbolic Analysis. After the traces are extracted, we perform symbol analysis, that is, a simple variant of symbolic execution, on each trace and then choose to keep those APIs that are interested, that is, related to crypto APIs.

The procedure of symbolic analysis is given in Algorithm 3,

[2] *E.g.*, $\{[a,b],[c]\} \cdot \{[d],[e,f]\} = \{[a,b,d],[a,b,e,f],[c,d],[c,e,f]\}$.

Algorithm 3: `Symbol_Analysis`

input : a trace tr
output: an API sequence

```
1  env ← initialize parameters from declarations;
2  seq ← [];
3  for each inst in tr do
4      (opcode, args_r, args_w) ← parse inst;
5      for each arg in args_r do
6          read arg from env;
7      syminst ← concat(opcode, args_r);
8      if syminst calls m then
9          if m is related to crypto then
10             seq.append(syminst);
11         else
12             syminst ← return_type(m);
13     for each arg in args_w do
14         write arg as syminst into env;
15 return seq;
```

which takes a trace tr as input and returns a possibly empty API sequence. The algorithm starts with an environment env that are constructed from the declarations of the corresponding methods where the trace tr is extracted (line 1) and an empty sequence seq of APIs (line 2). Then the algorithm iterates on each instruction $inst$ in the target trace tr to update the instruction and the environment and identify the interesting APIs (lines 3-14). Specifically, for each instruction $inst$, the algorithm first parses it into a triple consisting of the opcode $opcode$, those arguments $args\_r$ that $inst$ would read from, and those arguments $args\_w$ that $inst$ would write into (line 4). There are 222 opcodes in total listed in Android Dalvik-bytecode list [20], and for simplicity, we group them into several categories according to their semantics and arguments. Next, the algorithm updates each argument in $args\_r$ with respect to the environment env (lines 5-6) and constructs a symbolic expression $syminst$ from the opcode and the updated arguments (line 7). The interesting case is that a method m is called by the current instruction (line 8). If m is related to crypto APIs, then it is appended into the current sequence seq (lines 9-10). Otherwise, the symbolic expression $syminst$ is replaced by the return type of m (lines 11-12). This means that symbolic execution is only performed on the instructions or expressions that are related to crypto APIs, while the others are abstracted as their types, which is mainly to avoid user-defined methods or complex expressions. Finally, the collected API sequence seq is returned (line 15).

Let us consider the snippet code shown in Figure 2. The API sequence extracted from this snippet code is given in Table I, which are reformatted for easy understanding, that is, erasing interface information and reordering method names and arguments. For example, in our dataset, the last API of Table I is "invoke-virtual { invoke-static { *"AES/CBC/PKCS5Padding"* }, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;, *[B* },

```
.method public static a(Ljava/lang/String;Ljava/lang/String;)[B
1     new-instance v0, Ljavax/crypto/spec/SecretKeySpec;
2     invoke-virtual {p1}, Ljava/lang/String;->getBytes()[B
3     move-result-object p1
4     const-string v1, "AES"
5     invoke-direct {v0, p1, v1}, Ljavax/crypto/spec/SecretKeySpec;-><init>([BLjava/lang/String;)V
6     const-string p1, "AES/CBC/PKCS5Padding"
7     invoke-static {p1}, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
8     move-result-object p1
9     new-instance v1, Ljavax/crypto/spec/IvParameterSpec;
10    invoke-virtual {p0}, Ljava/lang/String;->getBytes()[B
11    move-result-object p0
12    invoke-direct {v1, p0}, Ljavax/crypto/spec/IvParameterSpec;-><init>([B)V
13    const/4 p0, 0x2
14    invoke-virtual {p1, p0, v0, v1}, Ljavax/crypto/Cipher;-><init>(ILjava/security/Key;Ljava/security/spec/AlgorithmParameterSpec;)V
15    invoke-virtual {p1, p2}, Ljavax/crypto/Cipher;->doFinal([B)[B
16    move-result-object p0
17    return-object p0
.end method
```

Fig. 2. Snippet Code from an APK

Ljavax/crypto/Cipher;->doFinal([B)[B". The symbol expressions show that both API orders and API arguments are collected.

Two different traces from an identity method could yield two API sequences of the same, which should be the same usage, so only one sequence is collected. However, if traces are from different methods, then all the copies are retained, as they are clearly different usages.

As a notice, to make it easy to follow, we present trace extraction and symbolic analysis separately. But this way could give rise to inefficient algorithms to use in practice. In fact, trace extraction and symbolic analysis can be carried out together: just let *states* in Algorithm 1 maintain the API sequences directly as well as environments.

B. Model Building

In this section, we use hidden Markov model [21] and n-gram model [22] to build our model, respectively.

Hidden Markov Model. Hidden Markov model (HMM) is a generative probabilistic model that describes the process of generating sequences, and has been applied in speech recognition, speech synthesis, machine translation, and so on. Recently, HMM is used to analyze API usages as well [7]–[10].

A HMM for API usage [9] can be formalized as a 5-Tuple (Q, V, π, A, B), where Q is a set of K hidden states, V is a set of M interesting APIs (in our case, APIs are associated with some symbolic arguments or types), π is the initial state distribution specifying that each state q_i has a probability π_i to be selected as the starting state of the model, A is the transition matrix of size $K \times K$ specifying the state transition probabilities (*e.g.*, $a_{i,j}$ is the probability that the model changes from state q_i to state q_j), and B is the generating matrix of size $K \times M$ specifying that the emission probabilities of each state (*e.g.*, $b_{i,n}$ is the probability to call method m_n when the model is in state q_i). As seen, the HMM for API usage has $K + K^2 + K \times M$ parameters.

We employ a modified version of Baum-Welch algorithm presented in [9] to train our HMM for API usage, wherein forward and backward probabilities are used. In detail, the forward probability $\alpha_{i,t}$ is defined as the probability of seeing partial method sequence $m_1, m_2, \ldots, m_t$ and being in state q_i at time t given the model λ:

$$\alpha_{i,t} = P(m_1, m_2, \ldots, m_t, q_t = q_i \mid \lambda)$$

and the backward probability $\beta_{i,t}$ is defined as the probability of seeing the ending partial sequence $m_{t+1}, \ldots, m_T$ given state q_i at time t and the model λ:

$$\beta_{i,t} = P(m_{t+1}, \ldots, m_T \mid q_t = q_i, \lambda)$$

where q_t denotes the hidden state at time t. Based on the forward and backward probabilities, the probability of an API sequence $s = m_1, m_2, \ldots, m_T$ can be computed by any position t in it:

$$P(s) = \sum_{i=1}^{K} \alpha_{i,t}\beta_{i,t}$$

N-Gram Model. An n-gram model (NGM) is a type of probabilistic language model for predicting the next item in such a sequence in the form of a (n-1)-order Markov model. N-gram models are now widely used in probability, communication theory, computational linguistics, computational biology, and data compression. And there exist some research work on API usage analysis that uses n-gram [11]–[13].

In an n-gram model for API usage, the probability of an API sequence is estimated by generating the sequence word by word[3]; and the probability of each word in a sequence is only determined by the conditional probabilities of the previous $n-1$ tokens. Given a sequence $s = m_1, m_2, m_3, \ldots, m_T$, its probability is estimated as:

$$P(s) = \prod_{i=1}^{T} P(m_i|h_i)$$

where h_i denotes the history sequence $m_{i-n+1}, \ldots, m_{i-1}$ of size at most $n-1$. For example, if $n = 3$ (*i.e.*, using a 3-gram model) and $T = 4$, then the probability of the sequence $s = m_1, m_2, m_3, m_4$ is

$$P(s) = P(m_1)P(m_2|m_1)P(m_3|m_1, m_2)P(m_4|m_2, m_3)$$

We use the procedure presented in [13] to build our n-gram model.

C. API Usage Analysis

In this section, we present how to analyze a crypto API sequence, including the usage detection and the misuse location. The first task aims to detect whether a given crypto API sequence is a correct usage or not. If not, then the second task would locate the possible misused APIs. Both tasks are based on the assumption that *the correct API could higher the probability of an API sequence while the misused one does the opposite*. The approach works for both HMM and NGM, so we do not specify the model in the following.

The first task is intuitive and simple: just check whether the probability of a given API sequence is higher than the

[3]In our situation, words are crypto APIs with some symbolic arguments or types.

TABLE I
CRYPTO API SEQUENCE FOR THE SNIPPET CODE

LN	ID	Symbol Expression
5	se1	invoke-direct Ljavax/crypto/spec/SecretKeySpec;->init(*Ljavax/crypto/spec/SecretKeySpec;, [B, "AES"*)
7	se2	invoke-static Ljavax/crypto/Cipher;->getInstance(*"AES/CBC/PKCS5Padding"*)
12	se3	invoke-direct Ljavax/crypto/spec/IvParameterSpec;->init(*Ljavax/crypto/spec/IvParameterSpec;, [B*)
14	se4	invoke-virtual Ljavax/crypto/Cipher;->init(*se2, 0x2, Ljavax/crypto/spec/SecretKeySpec;, Ljavax/crypto/spec/IvParameterSpec;*)
15	se5	invoke-virtual Ljavax/crypto/Cipher;->doFinal(*se2, [B*)

threshold. So the key is to set up a reference probability for the threshold, which is computed based on the distribution of probabilities of all unique API sequences used for training our models. We currently use the 80% percentile as the threshold.

Once a sequence is detected as a misuse (*i.e.*, lower than the threshold), we shall identify the possible misuse locations. For that, we define the probability of an API m used in a location l of an API sequence $s = m_1, m_2, m_3, \ldots, m_T$ with $l \leq T$ as the probability of the sequence obtained by replacing m_l with m:

$$P(m \mid s, l) = P(s[m_l \mapsto m])$$

These probabilities clearly can be used as the scores for all the APIs such that we can sort all the APIs according to their probabilities for any location of any sequence. Then based on the sorting, we define the distance of an API m used in a location l of a sequence s as the order in the corresponding ordered list for l and s:

$$dis(m \mid s, l) = order(\{P(m' \mid s, l) \mid m' \in APIs\})$$

Generally, the misuse APIs would have a lower probability and thus have a large distance. Accordingly, we identify the misuse locations for a misuse sequence s as the ones that have a top-k distance among all the locations of s, where k is a small integer.

III. EXPERIMENTS

This section presents the experimental results, including model selection, API usage analysis, and comparison with CogniCryptS_{AST}.

A. Dataset and Evaluative Criteria

To create our dataset, we collect 87375 APKs from the benign samples of AndroZoo [23], released from 2016 to 2019. We currently focus on the libraries of "javax.crypto". So we perform a quick scan on the collected APKs to filter out that have not used any API from the target libraries, and keep the remaining 19766 ones for our dataset. Next, we extract the API sequences from the selected APKs via the analysis in Section II-A. Then we filter out the sequences of length smaller than 5, which we think should not be a complete usage, and identify the reported errors by CogniCryptS_{AST} among the remaining sequences, yielding the dataset for crypto API usages. Moreover, among these selected APKs, we take 11856 APKs as the training set, 3957 as the validation set, and 3953 as the testing set. Table II lists some results of the dataset, where Positive (resp, Negative) denotes the sequences that contain no error (resp, some errors) reported by CogniCryptS_{AST}, Num denotes the total number of sequences, Min, Max, and Ave denote the minimum length, the maximum length, and the average length for the extracted API sequences, respectively.

TABLE II
API SEQUENCES FROM APK DATASET

Dataset	Positive				Negative			
	Num	Min	Max	Avg	Num	Min	Max	Avg
Training	8890	5	23	6.96	17509	5	41	7.36
Validation	3381	5	23	8.24	4747	5	41	7.34
Testing	3478	5	30	6.82	5296	5	41	7.38

In our experiments, we use the following performance measures to quantitatively validate the experimental results. *Accuracy* is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. *Precision* is the ratio of correctly predicted positive observations to the total predicted positive observations, and *Recall* is the ratio of correctly predicted positive observations to all observations in actual class. *F1 score* is the weighted average of *Precision* and *Recall*, that is, $(2 \cdot Precision \cdot Recall)/(Precision + Recall)$. Intuitively, the higher the measures above, the better the model.

B. Model Selection

As shown in Section II-B, there is a hyper-parameter K (*i.e.*, the state number) for HMM that needed to be fixed. For that, we take the range $[5, 15]$ as the candidate for K. This is due to that the minimum length is 5 and that the average length of the extracted API sequences ranges from 6 to 8. And for each value in the candidate, we build a HMM taking it as the state number K from the positive sequences[4] in the training set. After all the models are built, we then perform them on the validation set and select a model according to the results.

The results for HMM with different state numbers on the validation set are given in Table III, which show that the performances of the models are quite close, although with different state numbers. In particular, the HMM with 12 states obtains the highest accuracy and the highest precision, while the HMM with 8 states achieves the highest recall and the highest F1 score. As a result, the HMM with 8 states is selected.

[4]We take only the positives as we would like the models to capture the correct usages.

TABLE III
RESULTS FOR HMM WITH DIFFERENT STATE NUMBERS

K	Accuracy	Precision	Recall	F1
5	66.68%	58.54%	68.17%	62.99%
6	66.60%	58.44%	68.29%	62.98%
7	65.92%	57.71%	67.61%	62.27%
8	67.23%	59.15%	**68.58%**	**63.52%**
9	66.83%	58.73%	68.08%	63.06%
10	66.70%	58.57%	68.20%	63.02%
11	66.80%	58.72%	67.99%	63.02%
12	**67.45%**	**59.51%**	68.05%	63.50%
13	67.37%	59.43%	67.90%	63.39%
14	67.37%	59.40%	68.11%	63.46%
15	67.19%	59.24%	67.73%	63.20%

TABLE IV
RESULTS FOR NGM WITH DIFFERENT GRAM SIZES

N	Accuracy	Precision	Recall	F1
3	68.08%	60.09%	69.26%	64.35%
4	69.68%	62.10%	69.56%	65.62%
5	**70.90%**	**63.39%**	**71.10%**	**67.02%**

Likewise, we need to fix a hyper-parameter N (*i.e.*, the gram size) for NGM. The selection procedure is similar to the above. We currently set the gram size ranges from 3 to 5. This is because (i) 3-gram, 4-gram, and 5-gram models, as shown in [13], generated reasonable results; and (ii) the average length of the extracted API sequences ranges from 6 to 8.

The results for NGM with different gram sizes on the validation set are given in Table IV. From the results, we can see that 5-gram model performs best in all the evaluative criteria. Hence, 5-gram model is selected. Compared with HMM, NGM performs slightly better on the validation set.

C. API Usage Analysis

Using the selected models, we perform experiments to analyze the crypto API usages, including the usage detection and the misuse location, on the testing set.

Usage Detection. The usage detection results are given in the top half part of Table V, where *Sym-HMM* and *Sym-NGM* denote our selected HMM and NGM, respectively; and *Baseline* denotes the results of a random predictor wherein half the positives and half the negatives are assumed to be predicted correctly. The results indicate that both our models obtain a better result than the baseline on the testing set. Compared with Sym-NGM, Sym-HMM achieves (a better recall and) a better F1 score.

In addition to the correct usages, we also concern about the misuse usages. For that, we turn over the positives and the negatives, and recompute the corresponding evaluative criteria (marked with a superscript "T"), which is shown in Table V as well. From the results, we can see that both our models perform well in terms of all the evaluative criteria as well. And Sym-NGM performs better than Sym-HMM.

Moreover, we also compare our models with the no-symbolic-execution versions, namely, a variant of Nguyen et. al.'s approach [9] on crypto APIs (using HMM but no symbolic execution, denoted as No-HMM) and a variant of Wang et. al.'s approach [13] on crypto APIs (using n-gram model but no symbolic execution, denoted as No-NGM). As our symbolic analysis has already captured the dependences between the objects in the libraries of "javax.crypto", we drop off the symbolic expressions in the API sequences of our dataset and use them to build No-HMM and No-NGM in the same way as shown in Section III-B.

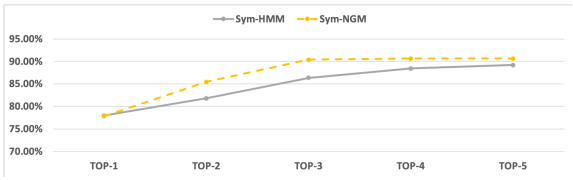

Fig. 3. API Misuse Location

The results are given in the bottom half part of Table V, from which, we can see that No-HMM and No-NGM can capture the correct usages as well, which is in accordance with existing work [9], [13]. However, the usage information they capture contains only the API dependence orders, such that they are not good at detecting the misuse crypto APIs. As shown in Table V, both the F1 scores of No-HMM and No-NGM on the misuse detection are quite close to the random baseline.

Compared with No-HMM (resp. No-NGM), Sym-HMM (resp. Sym-NGM) performs 11.77% (resp. 16.82%) better on the usage detection in term of F1 score. Moreover, our models still perform well on the misuse detection, and can improve 35.35% and 28.29% in term fo F1 score for HMM and NGM, respectively. This indicates that our approach is preferable to crypto APIs, even to domain-specific API usage analysis, because domain-specific APIs are sensitive to not only API orders, but also their arguments.

Nevertheless, there are still some rooms for improvement in our solution. For example, when creating an *IvParameterSpec* object, it needs to ensure that the byte array passed to the object should be generated randomly. But in our symbolic analysis, the methods that are not in the libraries of "javax.crypto" are abstracted, so that such methods for random generation are lost. We can enhance our abstraction to address this problem, which is left for future work.

Misuse Location. To evaluate our misuse location, we performed it on the negatives in the testing set and checked whether our reported top-k locations hit the ones reported by CogniCryptS$_{\text{AST}}$. The results are shown in Figure 3, which demonstrated our location approach is effective. In detail, top-1 can achieve about 78% accuracy for both Sym-HMM and Sym-NGM. And the accuracy increases rapidly when k is not larger than 3; after that, it goes slowly.

D. Comparison with CogniCryptS$_{AST}$

Finally, we compare our models using top-1 as the misuse location with a state-of-the-art tool CogniCryptS$_{\text{AST}}$ [6]. For

TABLE V
DETECTION RESULTS ON TESTING SET FOR DIFFERENT MODELS

Model	Accuracy	Precision	Recall	F1	Precision^T	Recall^T	F1^T
Sym-HMM	70.38%	59.93%	76.28%	**67.12**%	81.02%	66.50%	73.05%
Sym-NGM	71.23%	61.60%	72.83%	66.75%	79.73%	70.19%	**74.65**%
Baseline	50.00%	39.64%	50.00%	44.22%	60.36%	50.00%	54.69%
No-HMM	57.23%	47.68%	81.11%	60.05%	77.00%	41.54%	53.97%
No-NGM	57.67%	47.73%	71.19%	57.14%	72.06%	48.79%	58.19%

TABLE VI
COMPARISON WITH COGNICRYPTS$_{AST}$

APK	Sym-HMM			Sym-NGM			CogniCryptS$_{AST}$	
	N	AP	E	N	AP	E	N	E
*runy	0	0	0	0	0	0	3	3
*Client	0	0	0	0	0	0	1	1
*collage	2	0	2	1	0	1	0	0
*Hero	1	0	0	1	0	0	1	0
*bonso	0	0	0	0	0	0	3	3
*dictionary	0	0	0	0	0	0	3	3
*Erudit	10	0	8	10	0	8	8	8
*client	8	2	6	8	2	6	6	6
*diner	6	1	5	6	1	5	6	6
*cake	0	0	0	0	0	0	3	3
*retro	0	0	0	0	0	0	0	0
*stock	6	1	5	6	1	5	7	6
*byapps	0	0	0	0	0	0	0	0
*game	0	0	0	0	0	0	3	3
*4791	2	0	1	1	0	0	0	0
*Apps	6	1	5	6	1	5	7	6
*diamond	4	3	1	1	0	1	2	2
*kingvip	7	2	5	7	2	5	7	6
*troll	6	1	5	6	1	5	0	0
*charles	1	0	1	1	0	1	0	0
*pvsh	1	0	1	1	0	1	0	0
*sytadin	1	0	0	1	0	0	1	1
*artkiss	0	0	0	0	0	0	3	3
*input	11	7	4	13	8	5	10	10
*scope	6	2	4	6	2	4	4	4
*ventura	8	4	4	8	4	4	4	4
*racks	12	3	9	12	3	9	11	11
*pushtan	4	0	4	4	0	4	5	5
*wallet	4	0	4	4	0	4	5	5
*newspapers	6	2	4	6	2	4	6	5
*babyplan	8	2	4	8	2	4	6	6
*share	4	0	4	4	0	4	9	8
*sounds	4	1	3	4	1	3	3	3
*travel_eng	4	2	2	4	2	2	2	2
*awadcar	3	0	3	3	0	3	3	3
*white	1	0	1	1	0	1	1	1
*metro	1	0	1	1	0	1	1	1
*sogno	2	0	2	2	0	2	1	1
*compass	4	2	1	4	2	1	1	1
*monster	3	2	1	3	2	1	2	2
Total	146	38	100	143	36	99	138	132

that, we perform all the tools on 40 APKs. The results are given in Table VI, where N denotes the number of errors that are reported by any tool, E denotes the number of actual errors by manual inspection, AP denotes the errors that are caused by argument passing in our models.

In total, Sym-HMM, Sym-NGM, and CogniCryptS$_{AST}$ report 146, 143 and 138 errors for these 40 APKs, respectively. On these reported errors, we perform a manual inspection for further analysis. Firstly, we found there are 88 errors in common for all tools, and 2 more common errors reported by both Sym-NGM and CogniCryptS$_{AST}$. Secondly, among the reported errors, 100, 99, and 132 ones are actual for Sym-HMM, Sym-NGM, and CogniCryptS$_{AST}$, respectively, that is, the accuracies for the tools are 68.49%, 69.23%, and 95.65%, respectively. One reason for this is that, in our models, an API taking a misuse API as an argument could be reported simultaneously, as it could have the same large distance as the misuse one. We call this reason as argument passing. Clearly, these errors caused by argument passing are duplicated. There are 38 and 36 such kind of errors for Sym-HMM and Sym-NGM, respectively. Setting these duplicated errors aside, the accuracies of our models can reach 92.59% and 92.52%, respectively, both of which are quite close to the one of CogniCryptS$_{AST}$. Finally, we also found that 12 (resp. 10) actual errors reported by Sym-HMM (resp. Sym-NGM) are not reported by CogniCryptS$_{AST}$, most of which are due to a wrong argument. Conversely, there are 44 (resp. 43) actual errors reported by CogniCryptS$_{AST}$ are not reported by Sym-HMM (resp. Sym-NGM), among which, 75.0% (resp. 88.9%) are due to the abstraction as discussed in Section III-C.

IV. RELATED WORK

This section presents some recent related work on crypto API usage analysis.

Android. Egele et al. [3] proposed a light-weight static tool CryptoLint to identify misuses of crypto APIs for Android applications, taking six rules as a guide. Later, Shao et al. [5] proposed a systematic tool CMA, combining both static and dynamic analysis, based on a manually built model of crypto misuse vulnerabilities. And a similar analysis to Shao et al.'s is presented by Chatzikonstantinou et al. [24] as well. Ma et al. presented a tool CDRep for automatically repairing crypto misuse defects in Android applications, wherein CryptoLint [3] and CMA [5] is used in detection, and a set of manually created patch templates is used in repairing. Muslukhov et al. [25] developed a static tool BinSight to attribute crypto APIs misusage to its source, wherein the rules in CryptoLint [3] is used. Krüger et al. [6] proposed a static tool CogniCryptS$_{AST}$ to detect the crypto misuses in Android applications, using a designed rule set rewritten in a definition language CrySL from the Java Cryptography Architecture documentation. Gao et al. [4] tried to infer crypto API usage rules from the developer updates but obtained a negative result, wherein CogniCryptS$_{AST}$ [6] is used to identity misusage.

Java. Nadi et al. [26] performed an empirical investigation into the obstacles developers face while using the Java crypto APIs. Krüger et al. [27] provided a toolkit CogniCrypt to support Java developers with the use of crypto APIs. Paletov et al. [28] proposed an approach to infer crypto API rules for Java programs from code changes. Duncan et al. [29] presented a methodology for automatically checking security properties in JavaScript code. Wickert et al. [30] built a dataset of parametric crypto misuses from real-world java projects, with the help of CogniCryptS_{AST} [6].

Apart from the Android platform or Java-specific applications, there are some other approaches for other platforms or languages. Li et al. [31] developed a tool iCryptoTracer to detect the crypto misusage for iOS applications, using the rules similar to CryptoLint [3]. Gorski et al. [32] presented an approach to help Python developers on crypto API misuse. Mindermann et al. [33] performed an exploratory study on the usage of Rust crypto APIs.

V. Conclusion

In this paper, we have proposed an approach based on probabilistic models to analyzing crypto API usages in Android applications. To build the models, we have created a dataset consisting of crypto API sequences with arguments. We also have carried out some interesting experiments to evaluate our models, which show that our models are capable of capturing the usages, detecting and locating the misuses; and perform better than some existing approaches based on probabilistic models.

As for future work, we will enhance our abstraction to detect more misuses and consider more domain-specific API libraries. We will also consider the crypto API recommendation as well as the usage rule mining from existing codes.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grants No. 61972260, 61772347, 61836005; Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577; and Guangdong Science and Technology Department under Grant No. 2018B010107004.

References

[1] *Report from IDC*, https://www.idc.com/promo/smartphone-market-share/os.

[2] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov, "Know your enemy: the risk of unauthorized access in smartphones by insiders," in *MobileHCI*, 2013, pp. 271–280.

[3] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *CCS*, 2013, p. 73–84.

[4] J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein, "Negative results on mining crypto-api usage rules in android apps," in *MSR*, 2019, pp. 388–398.

[5] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling analysis and auto-detection of cryptographic misuse in android applications," in *DASC*, 2014, pp. 75–80.

[7] D. Chen, Yanduo Zhang, Rongcun Wang, W. Wei, Huabing Zhou, Xun Li, and Binbin Qu, "Mining api protocols based on a balanced probabilistic model," in *FSKD*, 2015, pp. 2276–2282.

[6] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *ECOOP*, 2018, pp. 10:1–10:27.

[8] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *ASE*, Nov 2015, pp. 795–800.

[9] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: a statistical approach," in *ICSE*, 2016, pp. 416–427.

[10] B. Rashidi, C. Fung, and E. Bertino, "Android resource usage risk assessment using hidden markov model and online learning," *Comput. Secur.*, vol. 65, no. C, pp. 90–107, 2017.

[11] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *MSR*, 2013, pp. 319–328.

[12] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*, 2014, p. 419–428.

[13] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *ASE*, 2016, pp. 708–719.

[14] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *FSE*, 2016, p. 631–642.

[15] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring api method parameter recommendations," in *ICSME*, 2015, pp. 271–280.

[16] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *OOPSLA*, pp. 104:1–104:22, 2017.

[17] R. Wiśniewski and C. Tumbleson, *Apktool: A tool for reverse engineering Android apk files*, https://ibotpeaches.github.io/Apktool/.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI*, 2014, pp. 259–269.

[19] Z. Xu, K. Ren, S. Qin, and F. Craciun, "CDGDroid: Android Malware Detection Based on Deep Learning Using CFG and DFG," in *ICFEM*, 2018, pp. 177–193.

[20] *Dalvik Bytecode*, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[21] L. R. Rabiner and B.-H. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[22] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Comput. Linguist.*, vol. 18, no. 4, p. 467–479, 1992.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *MSR*, 2016, pp. 468–471.

[24] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis, "Evaluation of cryptography usage in android applications," in *BICT*, 2016, pp. 83–90.

[25] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source attribution of cryptographic api misuse in android applications," in *ASIA CCS*, 2018, p. 133–146.

[26] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do java developers struggle with cryptography apis?" in *ICSE*, 2016, pp. 935–946.

[27] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and et al., "Cognicrypt: Supporting developers in using cryptography," in *ASE*, 2017, p. 931–936.

[28] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," in *PLDI*, 2018, p. 450–464.

[29] D. Mitchell, L. T. van Binsbergen, B. Loring, and J. Kinder, "Checking cryptographic api usage with composable annotations (short paper)," in *PEPM*, 2017, p. 53–59.

[30] Wickert, Anna-Katharina and Reif, Michael and Eichberg, Michael and Dodhy, Anam and Mezini, Mira, "A dataset of parametric cryptographic misuses," in *MSR*, 2019, pp. 96–100.

[31] Y. Li, Y. Zhang, J. Li, and D. Gu, "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications," in *NSS*, 2014, pp. 349–362.

[32] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl, "Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse," in *SOUPS*, 2018, pp. 265–281.

[33] K. Mindermann, P. Keck, and S. Wagner, "How usable are rust cryptography apis," in *QRS*, 2018, p. 143–154.