Beyond Decision: Android Malware Description Generation through Profiling Malicious Behavior Trajectory

CHUNLIAN WU, College of Intelligence and Computing, Tianjin University, Tianjin, China
SEN CHEN, College of Cryptology and Cyber Science, Nankai University, Tianjin, China
JIAMING LI and RENCHAO CHAI, College of Intelligence and Computing, Tianjin University, Tianjin, China
LINGLING FAN, College of Cryptology and Cyber Science, Nankai University, Tianjin, China
XIAOFEI XIE, Singapore Management University, Singapore, Singapore
RUITAO FENG, Southern Cross University, Lismore, Australia

Malware family labels and key features used for the decision-making of Android malware detection models fall short of precise comprehension of malicious behaviors due to their coarse granularity. To solve these problems, in this article, we first introduce the concept of the malicious behavior trajectory (*MBT*) and propose an innovative approach called *ProMal*. *ProMal* aims to automatically generate malware descriptions with fine granularity through extracted *MBTs* from malware for users. Specifically, a labeled dataset of *MBTs* is constructed through substantial human efforts to build a behavioral knowledge graph (*BxKG*). The *BxKG* is scalable and can be automatically updated using two strategies to ensure its completeness and timeliness: (1) taking into consideration the evolution of Android SDKs and (2) mining new *MBTs* by leveraging the widely-used malware datasets. We highlight that the knowledge graph is essential in *ProMal*, which can reason new *MBTs* based on existing *MBTs* because of its structured data representation and semantic relation modeling, and thus helps effectively extract real *MBTs* in Android malware. We evaluated *ProMal* on a recent malware dataset where researcher-crafted malware descriptions are available, and the Precision, Recall, and F1-Score of *MBT* identification based on *BxKG* reached 96.97%, 91.43%, and 0.94, respectively, outperforming the state-of-the-art approaches. Taking *MBTs* identified from Android malware as inputs, precise, fine-grained, and human-readable descriptions can be generated using the large language model, whose readability and usability are verified through a user study. The generated descriptions play a significant role in interpreting and comprehending malware behaviors.

CCS Concepts: • **Security and privacy** → **Software security engineering**;

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62472309 and 62332005), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

Authors' Contact Information: Chunlian Wu, College of Intelligence and Computing, Tianjin University, Tianjin, China; e-mail: chunlian@tju.edu.cn; Sen Chen (corresponding author), College of Cryptology and Cyber Science, Nankai University, Tianjin, China; e-mail: tigersenchen@163.com; Jiaming Li, College of Intelligence and Computing, Tianjin University, Tianjin, China; e-mail: jiamingli@tju.edu.cn; Renchao Chai, College of Intelligence and Computing, Tianjin University, Tianjin, China; e-mail: chairenchao@tju.edu.cn; Lingling Fan, College of Cryptology and Cyber Science, Nankai University, Tianjin, China; e-mail: linglingfan@nankai.edu.cn; Xiaofei Xie, Singapore Management University, Singapore, Singapore; e-mail: xfxie@smu.edu.sg; Ruitao Feng, Southern Cross University, Lismore, Australia; e-mail: ruitao.feng@scu.edu.au.

ACM 1557-7392/2025/8-ART196
https://doi.org/10.1145/3715909

Additional Key Words and Phrases: Android Malware, Malicious Behavior Analysis, Knowledge Graph

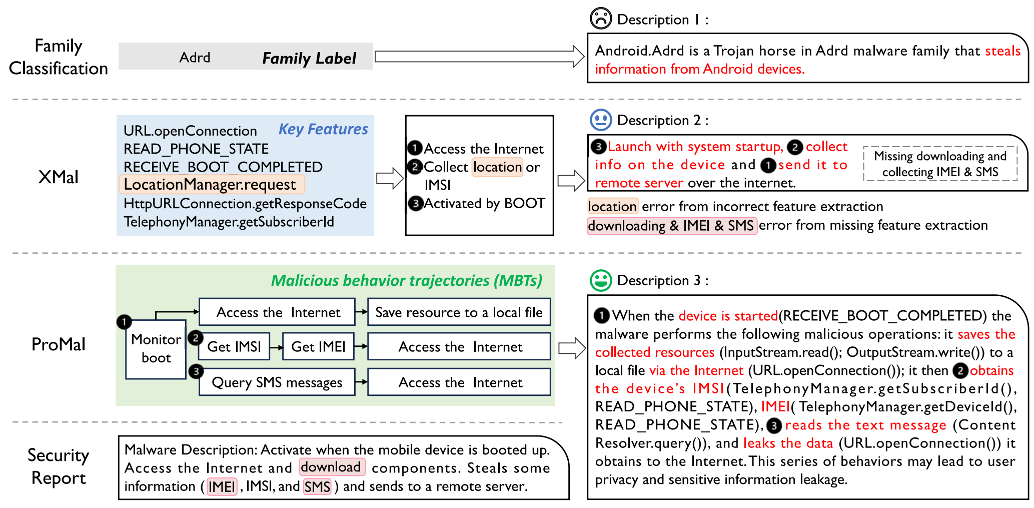
ACM Reference format:
Chunlian Wu, Sen Chen, Jiaming Li, Renchao Chai, Lingling Fan, Xiaofei Xie, and Ruitao Feng. 2025. Beyond Decision: Android Malware Description Generation through Profiling Malicious Behavior Trajectory. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 196 (August 2025), 39 pages.
https://doi.org/10.1145/3715909

1 Introduction

Android malware, specifically designed to infiltrate Android devices, poses significant security threats to millions of users worldwide. As Android malware becomes more prevalent, the need for effective detection approaches to protect users and their devices grows. Different approaches including machine learning-based [3, 4, 7, 8] and deep learning-based approaches [13, 64] have been proposed to detect Android malware. While these approaches have shown promising accuracy in identifying malware, they struggle with limited comprehension of their malicious behaviors [26, 37, 41, 54].

There is a growing need for advanced and explainable techniques that offer detailed explanations for identified malware. Taint analysis [5], shows promise in explaining specific malicious behaviors such as sensitive data leakage by utilizing dataflow but is limited in scope, focusing on common malicious behaviors. Combining machine learning models with explainable AI techniques (e.g., Drebin [4], LIME [41], LEMNA [19], and XMal [54]) can predict key malicious behaviors based on extracted key features used for the decision-making of detection models. However, these approaches have some limitations. (1) For Drebin, the key features for decision-making are completely dependent on the training data, and the impact of each feature on the classification is fixed after the model training is complete, leading to inability to adapt to the dynamic nature of malicious behaviors in malware. (2) Compared with Drebin, although LIME and LEMNA are model-agnostic, they leverage linear or simple models to approximate the local part of the original complex model, inevitably missing crucial features that may also play a vital role in predicting malicious behaviors. (3) For XMal, the reliability of key features relies on the robustness of the underlying machine learning models, which can lead to inaccurate or unstable results if the models lack robustness. Moreover, the above work is not able to provide a detailed explanation of how malicious behavior is executed. While XMal attempted to describe the execution of malicious behavior by sorting malware operations that constitute the malicious behavior, the ordering rules it uses are predefined based on the experience of manual analysis of malicious behaviors, neglecting the control flow and dataflow of malware. This means the generated explanation may deviate from the real execution of malware.

In conclusion, these machine learning-based approaches are unreliable since the key features used to predict primary malicious behaviors may not be complete or accurate, as well as a lack of an adequate analysis of interrelations between different malware operations that indicate the specific implementation process of malicious behavior, preventing a comprehensive understanding of malware. For example, as shown in Figure 1, although the intermediate result of XMal contains three malware operations (i.e., *“Access the Internet,” “Collect IMSI,”* and *“Activated by Boot”*), it could not present a significant malicious behavior of leaking data until relations between malware operations are taken into account and they are ordered as *“Activated by Boot⟶Collect IMSI⟶Access the Internet”* as described in Description 2. Besides, XMal misses key malware operations (i.e., *Missing downloading* and *collecting IMEI and SMS messages*) and introduce incorrect malware operations (i.e., *“Collect location”*).



Android.Adrd is a Trojan horse in Adrd malware family that steals information from Android devices.

3 Launch with system startup, 2 collect info on the device and 1 send it to remote server over the internet.

location error from incorrect feature extraction

downloading & IMEI & SMS error from missing feature extraction

1 When the device is started(RECEIVE_BOOT_COMPLETED) the malware performs the following malicious operations: it saves the collected resources (InputStream.read(); OutputStream.write()) to a local file via the Internet (URL.openConnection()); it then 2 obtains the device's IMSI(TelephonyManager.getSubscriberId(), READ_PHONE_STATE), IMEI(TelephonyManager.getDeviceId(), READ_PHONE_STATE), 3 reads the text message (Content Resolver.query()), and leaks the data (URL.openConnection()) it obtains to the Internet.This series of behaviors may lead to user privacy and sensitive information leakage.

Malware Description: Activate when the mobile device is booted up. Access the Internet and download components. Steals some information (IMEI , IMSI, and SMS) and sends to a remote server.

Fig. 1. An example of malware profiling result of *ProMal* compared with family classification and XMal.

To overcome these limitations in the existing approaches, we aim to design a new approach to effectively profile the malicious behaviors of Android malware, however, encounter the following challenges: ***C1***: The existing granularity of malware family labels and key features is too coarse to accurately profile malicious behaviors. They fail to capture the relations among independent operations, which are crucial for understanding and interpreting malware behaviors. Consequently, the primary challenge lies in devising fine-grained representations of malicious behaviors that are specifically tailored for accurate interpretation. This is vital because malicious behaviors typically consist of various operations, necessitating a thorough analysis of their interrelations for effective interpretation. ***C2***: A major obstacle in developing a fine-grained representation of malicious behaviors is the absence of a widely-recognized and adequately labeled dataset for such a detailed representation. This shortage significantly hinders the accurate profiling of malicious behaviors. Therefore, there is an urgent requirement to carefully design and create a well-labeled dataset tailored to this advanced behavior representation. This endeavor is particularly challenging due to its time-consuming and resource-intensive nature, requiring the specialized skills of experts for in-depth analysis of real-world malware samples. ***C3***: Although it is feasible to design and construct fine-grained representations using a limited set of sampled behaviors, it is still insufficient due to the dynamic nature of malicious behaviors in malware. Therefore, a significant challenge lies in developing a scalable methodology that can automatically reason about new operations as they emerge. ***C4***: Compared to code-level features, fine-grained representations offer more informative insights. However, there is still a notable semantic gap between these detailed behavioral representations and their translation into human-readable malware descriptions. Bridging this gap presents a distinct challenge.

To this end, we propose a knowledge graph-based approach named *ProMal* to generate *precise, fine-grained, and human-readable* behavior descriptions of Android malware. Specifically, to address *C1*, we first define a novel concept of **malicious behavior trajectory (*MBT*)**, which will be used to help empower our approach to profile malware by identifying independent malware operations as well as their execution relations and further narrow down the semantic gap between code-level representation and behavior description. Based on the design of *MBT*, to address *C2*, we first take substantial human efforts to label *MBTs* of malware. In particular, we collected hundreds of representative malware samples along with high-quality expert-crafted reports in different ways. Although these reports have precise malware descriptions, the labeling process is also a non-trivial task due to the need for verifying *MBTs* through a comprehensive code review, which requires more than two man-months of effort. Such a human-labeled dataset of *MBT* is further used to build a **behavioral knowledge graph (*BxKG*)**, where nodes may present specific malware operations or code-level features that are used to implement these malware operations and each edge refers to the relation between two entities represented by nodes, e.g., the call relation between a malware operation and its relevant API, or the request relation between API and permission. To address *C3*, to adapt to the dynamic and ever-changing nature of Android malware, on the one hand, we use a knowledge graph to present *MBTs*, which is essential because it can help reason variant *MBTs* based on the original ones. On the other hand, we further design two innovative strategies to dynamically augment and update the initial *BxKG*. Specifically, we consider the evolution of Android **software development kits (SDKs)** based on the official documents and, meanwhile, propose *self-updating* to automatically update the initial *BxKG* based on the widely-used malware datasets, including Genome [66], Drebin [4], AMD [51], and GP Malware [6]. To address *C4*, we use *MBTs* as well as their corresponding code-level features as the input presentations of **large language models (LLMs)**, which significantly mitigates the problem of the semantic gap that occurred in the previous works. The example of extracted *MBTs* and generated description by *ProMal* is shown in Figure 1, which achieves the best performance of malware profiling compared with family labels and XMal.

To demonstrate the effectiveness of *ProMal*, we conducted a series of experiments. Firstly, we evaluated *MBT* extraction on 105 real-world malware samples from the GP Malware dataset [6]. We spent one month manually labeling the *MBTs* in these samples based on their corresponding expert-crafted reports. The experimental results indicated that *ProMal* achieved Precision, Recall, and F1-Score of 96.97%, 91.43%, and 0.94 respectively in *MBT* extraction. Meanwhile, when compared with the state-of-the-art approach XMal [54], *ProMal* outperformed with a 0.35 higher F1-Score on 121 malware samples. Secondly, we demonstrated the real impact of self-updating and graph reasoning on *MBT* extraction, which established the necessity of knowledge graph construction. Thirdly, we evaluated the contribution of different individual components in *ProMal*. The experimental results unveiled that the completion of **call graph (CG)**, the analysis of control flow, the analysis of dataflow, and the reasoning capability contribute 0.97%, 75.95%, 6.82%, and 13.52% Precision improvements in *MBT* extraction on our experimental dataset, respectively. Note that, despite the completion of the CG, which resulted in only a modest 0.97% improvement in Precision, it significantly increased the Recall by 22.86%. Last but not least, *ProMal* can explain how malicious behaviors are executed with the help of malware descriptions generated by LLMs like GPT-3.5 [32] according to these extracted *MBTs*, whose readability and usability are further verified through a user study. The study showed that malware descriptions generated by *ProMal* have an average satisfaction score of 9.67, while descriptions generated by XMal and baseline have an average score of 3.37 and 6.32.

In summary, we made the following contributions:

- We proposed *ProMal* based on the constructed *BxKG*, which achieves precise, fine-grained, and human-readable malware descriptions, and outperforms the state-of-the-art approach in profiling malware.
- We proposed a new concept of MBT and manually constructed a human-labeled dataset from four different resources, including 399 malware samples along with well-labeled malicious behavior trajectories, which takes three man-months of effort.
- We proposed two novel methods to dynamically and automatically update the build knowledge graph based on SDK evolution and widely used malware datasets in the wild.

— We have released the relevant data on GitHub [2]. Meanwhile, we are currently in the process of rolling out all the capabilities of malware profiling through an online service (see Section 6.3).

Lastly, it is important to note that ProMal is specifically designed to analyze and profile malicious behaviors within identified Android malware. While existing machine learning and deep learning-based methods can achieve very high detection accuracy, often surpassing 99%, users may still have doubts about the reliability of classification results. This is where ProMal comes in. ProMal's application scenario is to further analyze and dissect malicious behavior of malware after high-performance malware detection tools have identified the software as malicious. Moreover, since our analysis is conducted on samples that have been pre-verified as malware, the likelihood of encountering such false positives is significantly reduced. This precondition also ensures that the MBTs extracted from malware are more likely to represent actual malicious behaviors rather than benign operations mistakenly flagged as malicious. Consequently, it requires an Android malware as an input rather than an unlabeled one during our profiling process.

2 Overview

2.1 Malware Operation

In the field of mobile security, a malware operation refers to the action or activity performed by malware on infected devices, e.g., *"Get contact information," "Send SMS messages,"* or *"Obtain root privileges on the device."* In other words, malware operations are operations required to execute malicious behavior. In the prior work [48], malicious behaviors are classified into 12 categories, such as privacy stealing, abusing SMS/CALL, remote control, etc. If a piece of malware intends to steal users' private information, it should generally at least perform two operations sequentially: data collection and data transmission.

Since malware operations are usually implemented by coding, we take three types of code-level features into account to represent them in this article:

(1) Manifest Information: Android malware often requests unreasonable or unnecessary permissions, such as reading contacts (*READ_CONTACTS*) or accessing the network (*INTERNET*). Also, Intent may contain essential information to identify malware operations. For example, *android.intent.action.USER_PRESENT* may be used to monitor the device screen.
(2) API Call: Android malware may invoke specific APIs to execute malware operations, such as sending SMS messages (*SmsManager.sendTextMessage()*) or establishing a network connection (*URL.openConnection()*).
(3) String Constant: Some string constants as API parameters may also serve as a crucial role for spotting malware operations. For example, "accessibility_enabled" and "android_id", constants of *Secure.getInt()*, determine the actual use of this API, namely whether it is to check the enabled status of accessibility services on the device or to obtain the unique identifier of the device.

Most importantly, given that the ultimate goal of our work is to generate fine-grained and human-readable behavioral textual scripts for Android malware that explain why an Android app is classified as malware, to bridge the gap between code-level features and malware behavioral descriptions, annotating malware operations with behavioral semantic tags is a viable solution. As shown in Figure 2, there are two tags corresponding to two malware operations: "Get user's phone number" (belonging to data collection) and "Access the Internet" (belonging to data transmission). And the relevant code-level features of malware operations are listed below their respective tags. Note that some of these tags are manually summarized from expert analysis reports, while others

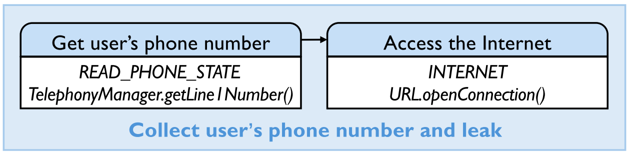


Fig. 2. An example of *MBT* that demonstrates privacy stealing (i.e., phone number).

are automatically generated during the knowledge graph update process using LLMs based on detailed functional descriptions of code-level features from the Android developer documentation (see Section 5).

2.2 MBT

It is emphasized in the Introduction that interrelations among malware operations are crucial to the understanding of malware at a finer level, since a sequence of malware operations can indicate the detailed execution process of malware's malicious behaviors step by step. Considering malware operations, their code-level features, as well as their relations with each other, we describe malicious behavior using *MBT*, based on which we can generate expected behavioral descriptions of Android malware.

Definition 1. Definition 1. Formally, *MBT* is defined as an ordered list of malware operations, i.e., $O = \{o_1 \rightarrow o_2 \rightarrow \cdots o_i \cdots \rightarrow o_n\}$, representing the complete sequential execution process of malicious behavior, where each malware operation o_i $(1 \leq i \leq n)$ corresponds to a step within this process. Most importantly, each malware operation is assigned a semantic tag that indicates its functionality and is linked to the code-level features that implement the operation in code, such as API calls, Permissions, Manifest Intents, and String Constants.

Figure 2 illustrates a case of a *MBT*, i.e., *"Get user's phone number → Access the Internet,"* which represents the malicious behavior that collects and leaks the user's phone number, comprising two distinct malware operations. Specifically, the operation *"Get user's phone number"* is implemented by the API *"TelephonyManager.getLine1Number()"* and the permission *READ_PHONE_STATE*. The collection of these code-level features is partly determined during the manual labeling of *MBTs* from reports (see Section 3) and partly updated through the *BxKG* self-updating process (see Section 5).

When compared with existing concepts such as API call sequence or subgraph used in the previous works [27, 33, 49], the *MBT* shows two differences: (1) Specifically designed to represent the execution process of malicious behavior, constructed initially based on manually analyzed malware reports. It is a targeted sequential set aimed at addressing the understanding of malicious behaviors. Unlike existing approaches that extract sensitive API sequences or subgraphs—which often introduce numerous irrelevant APIs that do not contribute to malicious operations—these methods frequently fail to accurately and effectively pinpoint the sensitive APIs within a malware sample that trigger malicious behavior, leading to an incomplete understanding of the malicious behaviors. This issue is underscored in our experiments in Section 7.1.1. In contrast, the *MBT* specifically targets malicious behaviors that align with our objectives, providing a more precise and meaningful representation of the malware execution process, thereby making it more effective for use in LLMs to generate accurate descriptions of these behaviors. (2) Carries a stronger semantic meaning. Each malware operation is assigned a semantic tag that indicates its function and is linked to corresponding code-level features (shown in Figure 6). By ordering these malware operations sequentially, *MBT* helps bridge the gap between source code and natural language. In Section 7.4, we compared the results of generating descriptions using only code-level features as input to LLMs with those generated using *MBTs* along with their corresponding code-level features. The results demonstrated that descriptions based on *MBTs* were more readable and accurate because *MBTs* provide a clearer and more structured representation of malicious behaviors, enhancing the overall quality and precision of the generated descriptions.

2.3 Motivating Example

Figure 1 illustrates the limitations of existing approaches that profile malware and generate malware descriptions. For family classification, users can only obtain a short and extremely coarse-grained text related to the malware family (i.e., Adrd), thus they only know that information will be stolen from Android devices but are unaware of the specific information being stolen. Unlike malware family classification, XMal [54], a machine learning-based approach, can extract key features (e.g., APIs and permissions) from malware and then match these features with simple semantics. After that, each semantic will be converted into a brief description (e.g., "Activated by Boot" is transformed into "Launch with system startup"), which will be further arranged and combined into a more complete description (as indicated in Description 2) according to ordering rules defined by authors. However, XMal has the following shortcomings: (1) The extracted key feature might be a false positive (i.e., *LocationManager.request()*) since their reliability is determined by the robustness of the underlying machine learning models, leading to an error in malware operation detection (i.e., "*Collect location*"). (2) XMal only considers several key features, which may not cover the complete range of malicious behaviors. In other words, it may lose other key features that are also helpful for understanding malware. For example, the lack of API *TelephonyManager.getDeviceId()* and permission *READ_SMS* leads to the failure of malware operation identification of "*Get IMEI*" and "*Query SMS messages*", respectively, which is also the key to interpreting the malicious behavior (reflected in the security report). (3) As mentioned in the Introduction, malware operations and their relations can detail the execution process of malicious behavior. However, XMal determines these relations based on heuristics, ignoring the specifics of individual malware instances, which raises concerns about the accuracy of the analysis of malicious behavior.

To address these limitations, we propose an effective approach called *ProMal*, which can automatically extract *MBTs* from Android malware and describe them in a human-readable way to inform users of malicious behaviors and remind users of potential security threats in detail. As shown in Figure 1, *ProMal* performs well in *MBT* extraction, i.e., it not only performs well in malware operation detection but also accurately captures relations between malware operations. When compared to the ground truth, the description generated by *ProMal* are semantically close and comprehensive. Additionally, it can generate more precise and informative descriptions than XMal or family classification.

In conclusion, our approach attempts to achieve the following design goals:

- *Precise and Fine-Grained*: The approach should precisely identify *MBTs* from malware, which has richer semantic information in terms of understanding malware behaviors.
- *Scalable*: The approach is designed to be scalable and capable of addressing various types of malware and variants, especially adapting to the rapid iteration of malware.
- *Human-Readable*: The malware description should be easily readable, facilitating user acceptance and understanding of how malicious behaviors are implemented.

2.4 Architecture Overview

Figure 3 illustrates an overview of *BxKG* construction and *ProMal*, *BxKG* construction consists of the following three major phases: *MBT* Labeling is to manually label *MBTs* from expert-crafted security reports of malware samples. Knowledge Graph Construction is to build a *BxKG* based on the labeled *MBTs*. Knowledge Graph Update is to automatically update the initial *BxKG* to

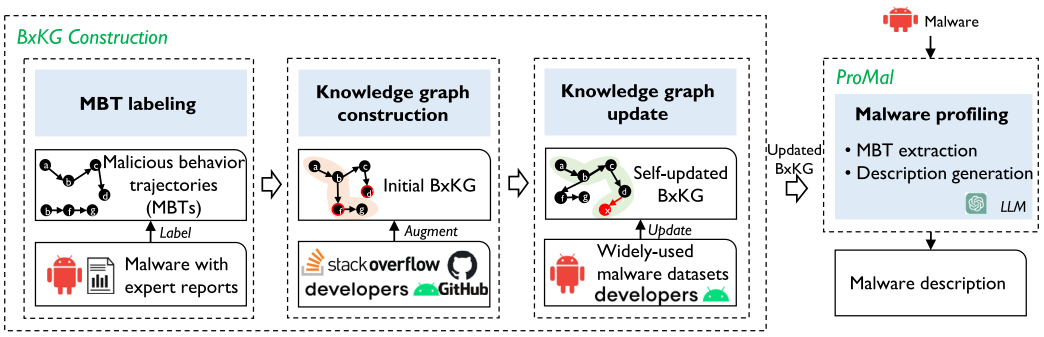


Fig. 3. An overview of *BxKG* construction and *ProMal*. The new *MBTs*, highlighted in pink, are the product of graph reasoning. The red circles (d' and f') represent that the malware operation of this node can be accomplished by other code-level features. The red solid node x represents the new node derived from widely-used malware datasets and the new *MBTs*, highlighted in green, are augmented by self-updating.

Table 1. The Two Datasets of Security Reports Used for Manual Analysis and *MBT* Extraction

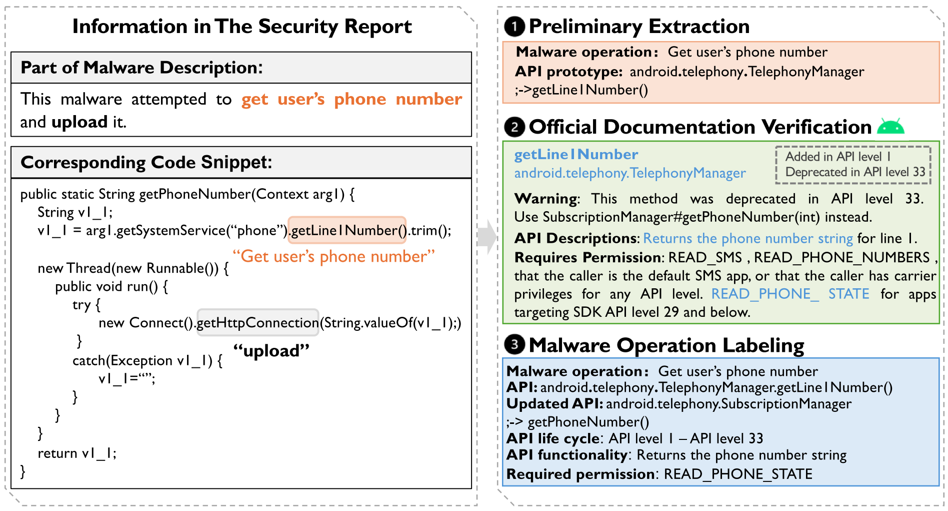
Source	MALRADAR	Anonymous Center
# of Reports	178	100
# of Malware	4,535	100
# of Families	148	23
Collection time	2014–2021	2008–2018

ensure the scalability of *ProMal*. *ProMal* takes an Android malware as an input and outputs a human-readable malware description based on the extracted *MBTs* along with relevant features from the pre-constructed *BxKG*.

3 MBT Labeling

3.1 *MBT* Labeling from Malware Reports

Currently, there are no credible and high-quality labeled *MBTs* available. Therefore, we first collected 278 well-maintained security reports from MALRADAR [48] and our long-standing industry partner (i.e., Anonymous National Computer Emergency Center) for manual labeling. The details of the datasets are shown in Table 1. MALRADAR [48] harvested 178 superior security reports that contained sufficient details about malware (e.g., their malicious behaviors) and provided at least one IoC (e.g., MD5 or SHA256). Importantly, these security reports were selected from leading security companies (e.g., TrendMicro, Kaspersky, and McAfee) launched by AV-Comparatives [1]. Afterwards, malware samples, published between 2014 and 2021, were collected according to the IoCs given in these reports. In addition, the industry partner provided us with 100 expert analysis reports, each corresponding to malware that was released between 2008 and 2018. Specifically, 4,635 (4,535 + 100) malware samples related to these security reports belong to 170 (148 + 23−1) malware families in total since one family label was repeated. Consequently, these reports provide a wide range of malicious phenomena, making them an appropriate source of data for us to label malware operations as well as *MBTs*. Next, we carried out the following steps:


Information in The Security Report
Part of Malware Description:
Corresponding Code Snippet:
❶ Preliminary Extraction
❷ Official Documentation Verification
❸ Malware Operation Labeling
"Get user's phone number"
"upload"

This malware attempted to **get user's phone number** and **upload** it.

```
public static String getPhoneNumber(Context arg1) {
    String v1_1;
    v1_1 = arg1.getSystemService("phone").getLine1Number().trim();

    new Thread(new Runnable()) {
        public void run() {
            try {
                new Connect().getHttpConnection(String.valueOf(v1_1););
            }
            catch(Exception v1_1) {
                v1_1="";
            }
        }
    }
    return v1_1;
}
```

Malware operation: Get user's phone number
API prototype: android.telephony.TelephonyManager ;->getLine1Number()

getLine1Number
android.telephony.TelephonyManager
Added in API level 1
Deprecated in API level 33

Warning: This method was deprecated in API level 33. Use SubscriptionManager#getPhoneNumber(int) instead.
API Descriptions: Returns the phone number string for line 1.
Requires Permission: READ_SMS , READ_PHONE_NUMBERS , that the caller is the default SMS app, or that the caller has carrier privileges for any API level. READ_PHONE_ STATE for apps targeting SDK API level 29 and below.

Malware operation: Get user's phone number
API: android.telephony.TelephonyManager.getLine1Number()
Updated API: android.telephony.SubscriptionManager ;-> getPhoneNumber()
API life cycle: API level 1 – API level 33
API functionality: Returns the phone number string
Required permission: READ_PHONE_STATE

Fig. 4. An example of labeling malware operations and verifying the accuracy of code-level features with the official documents from Google Android Developers. Verification information is highlighted in blue.

3.1.1 Operation Identification and Code-Level Feature Extraction. By meticulously reviewing these reports, we can initially identify malware operations and relevant code-level features (described in Section 2.2) based on malware descriptions and code snippets provided by reports. The effectiveness of *MBT* depends on the precision of the code features. However, the mapping between malware operation and its code-level features may not be accurate enough in these reports. For example, sometimes only the method name of the API can be obtained. Considering that other APIs with the same method name may exist, a complete API prototype (including package name, class name, method name, etc.) that can identify the unique API, is necessary. Furthermore, in some cases, even the method name cannot be determined because of obfuscation. As a solution to these problems, malware samples are decompiled into Java source codes using JADX [46], which also provides anti-confusion capabilities. After getting the API prototype from the source code, we then query the Google Android Developers documentation for: (1) collecting more details about the API to build a richer knowledge base, such as its functionality, required permissions, evolution, and life cycle; (2) ensuring the API precisely aligns with the reported malware operation and establishes a relation with them; and (3) optimizing the malware operation's tag to make it more concise and accurate if possible.

As shown in Figure 4, the malware description indicates two malware operations, i.e., "get user's phone number" and "upload," and we can locate relevant APIs (framed with boxes) in the code snippet. Take the first API named "getLineNumber" as an example, we can get it with a more complete representation (i.e., *TelephonyManager.getLine1Number()*) from the malware's source code. Next, according to the official documentation, we can learn that there is a call relation between this API and the malware operation "Get user's phone number" since this API *returns the phone number string*. Meanwhile, there is a request relation between this API and *READ_PHONE_STATE*. Note that, *HttpURLConnection* is a class used to handle HTTP connections rather than an API. Since this class is obtained through *URL.openConnection()*, which can return an instance representing a connection to the remote object referred to by the URL, we consider that malware can upload information using *URL.openConnection()*, and we update the malware operation's tag from "upload" to "Access the Internet" to make the malware operation and the API more consistent.

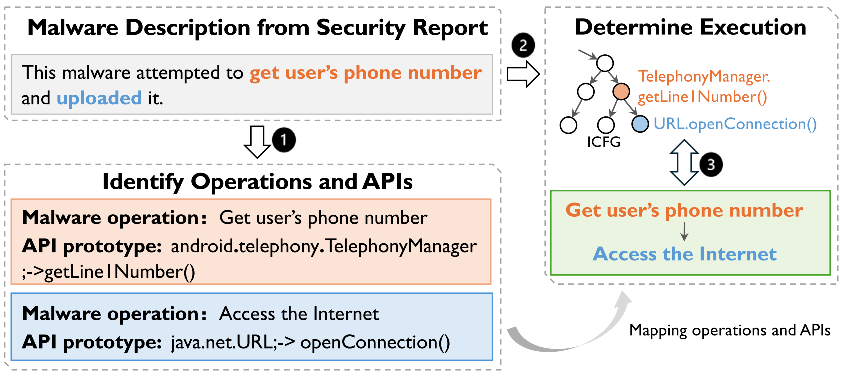

Malware Description from Security Report
This malware attempted to get user's phone number and uploaded it.
1
Identify Operations and APIs
Malware operation: Get user's phone number
API prototype: android.telephony.TelephonyManager ;->getLine1Number()
Malware operation: Access the Internet
API prototype: java.net.URL;-> openConnection()
2
Determine Execution
TelephonyManager. getLine1Number()
URL.openConnection()
ICFG
3
Get user's phone number
Access the Internet
Mapping operations and APIs

Fig. 5. An example of relation construction between malware operations. The final execution relation is determined by a reachability analysis between APIs corresponding to the extracted malware operations.

3.1.2 Relation Construction between Malware Operations. Malware operations and the order in which they are executed are the key components of an *MBT*. Therefore, we also need to extract the execution relations between malware operations from the reports. Unfortunately, sometimes the relations are missing or too complicated to be directly extracted. In addition, the execution process described in the reports may deviate from the actual one. In order to obtain a more accurate execution relation between malware operations rather than relying solely on reports, we perform a time-consuming manual analysis to build a reliable knowledge base. We first assess the execution order based on descriptions provided in reports, and then refer to the analysis results from some mature tools (e.g., Androguard and FlowDroid) to help determine the exact relations between different malware operations. Specifically, we use Androguard to generate CGs and **control flow graphs (CFG)** of malware, and then investigate the **inter-procedural control flow graphs (ICFG)** to trace the execution paths of malware operations through an examination of dependencies among their corresponding APIs.

As illustrated in Figure 5, based on the identified malware operations and APIs, we locate APIs in ICFG and perform reachability analysis. Since there is a path from *TelephonyManager.getLine1Number()* to *URL.openConnection()*, the execution relation between the two malware operations can be determined and the *MBT* is "Get user's phone number → Access the Internet," where "→" indicates the order of execution.

3.1.3 Consistency Check and Collaborative Revision. In light of the risk of malicious operations loss and relational misinterpretation due to manual analysis, three co-authors conducted cross-verification after each author had completed their independent annotations. Any conflict in the results was promptly resolved during the whole process.

Finally, we collected a set of human-labeled *MBTs*, including malware operations and relevant precise code-level features. In addition, these labeled data are used to construct the *BxKG* (see Section 4).

4 Knowledge Graph Construction

A knowledge graph is a structured representation of knowledge that interconnects entities and concepts through well-defined relations [10, 20, 21, 31, 35]. Entities are specific objects or things, such as "Albert Einstein" and "Theory of Relativity." Concepts are abstract ideas or categories like "Scientist" and "Scientific Theory." The knowledge graph captures both entities and their semantic relations within a specific domain, facilitating reasoning, inference, and knowledge discovery by leveraging the graph's structure. For example, a knowledge graph might link "Albert Einstein" (entity) to "Theory of Relativity" (entity) with the relation "developed." Knowledge graphs are employed to integrate information from various sources, providing a unified framework for data analysis and decision-making processes.

In this article, we utilize a knowledge graph to model malicious behaviors, and each malware operation in the graph is characterized by three distinct types of code-level features: permissions and Intent from the manifest file, API calls, and string constants introduced in Section 2.2. The constructed graph is a *BxKG*.

4.1 BxKG

The *BxKG* is a directed labeled graph defined as $BxKG = (V, E, f, g)$, where V is the set of nodes and $E \subseteq V \times V$ is the set of directed edges indicating relations between two nodes. Each node $v \in V$ is assigned a label from the set of node labels $L_V = \{Malware\ Operation, API, Permission, Intent, String\ Constant\}$. The labeling function $f : V \longrightarrow L_V$ maps each node to its corresponding label. Similarly, each edge $e \in E$ is assigned a label from the set of edge labels $L_E = \{call, request, proceed, register, align, input\}$, defined by the labeling function $g : (L_V \times L_V) \longrightarrow L_E$. The label of the edge will be determined according to the label of two nodes. Specifically:

— *Request*: e.g., an API may request a permission.
— *Call*: e.g., a malware operation may call an API.
— *Proceed*: e.g., one malware operation may proceed to another operation in execution order.
— *Register*: e.g., a malware operation may register an intent.
— *Align*: e.g., two APIs may execute in parallel to achieve the same malicious operation.
— *Input*: e.g., a string constant used as input to an API.

A triplet (n_i, e, n_j) is used to represent two nodes and the directed edge between them, where $n_i, n_j \in V$ and $e \in E$, indicating that there is a relation e from node n_i to node n_j. The labeling functions $f : V \longrightarrow L_V$ and $g : (L_V \times L_V) \longrightarrow T_E$ map nodes to their respective types and determine the edge type based on the types of the connected nodes. For example, if $f(n_i)$ = Malware Operation and $f(n_j)$ = Malware Operation, the labeling function $g(f(n_i), f(n_j))$ determines that the edge label of e should be "proceed." Then this triplet (n_i, e, n_j) can therefore be described in natural language as "after completing n_i, the malware proceeds to execute n_j."

4.2 Graph Construction Using Labeled *MBTs*

We construct the *BxKG* using labeled *MBTs*, which consist of various malware operations and their relations. As shown in Figure 6, there are five types of entities (i.e., graph nodes): malware operation, API, permission, Intent, and string constant which we can all extract from the collected *MBTs*. As for relations (i.e., graph edges) between two entities, they are also determined according to *MBTs*, such as the execution relation between "*Get user's phone number*" and "*Access the Internet*," or the request relation between *TelephonyManager.getLine1Number()* and *READ_PHONE_STATE*. Figure 6 shows a part of the initial *BxKG*. During this process, different *MBTs* will be connected by the common malware operations, e.g., "*Access the Internet*." The ability of graph reasoning is additionally demonstrated in the generation of new *MBTs* during the construction process of *BxKG*. For example, a new *MBT* involves the following behaviors including "*Get IMEI*"→"*Access the Internet*"→"*Download*," which was augmented by two initial *MBTs* (i.e., ① "*Get IMEI*"→"*Access the Internet*" ② "*Access the Internet*"→"*Download*").

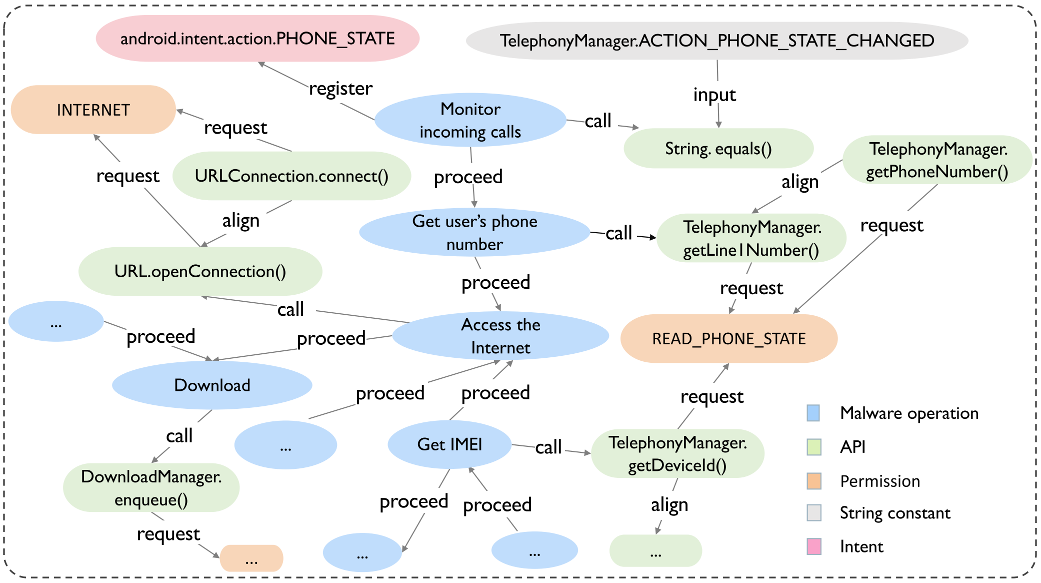
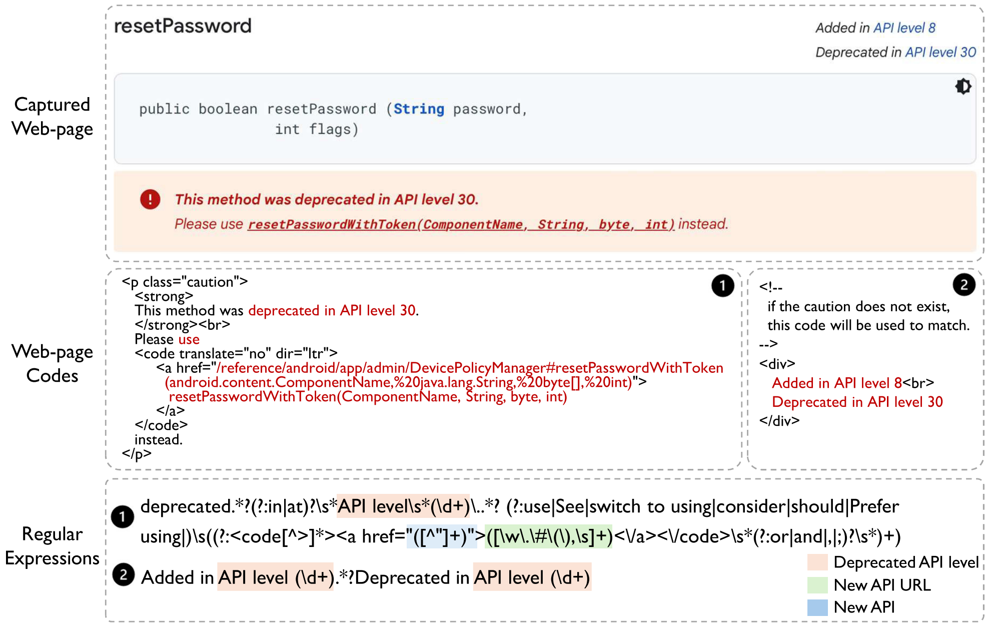


Fig. 6. An example of graph construction using three *MBTs*. There are four types of entities, i.e., malware operations, API, permission, Intent, and string constant.

4.3 Knowledge Augmentation for Initial *BxKG*

4.3.1 Manual Augmentation. Utilizing the gathered *MBTs*, we have built an initial version of *BxKG*. However, due to the limited number of security reports (i.e., 278 reports) that we used to construct a human-labeled dataset of *MBTs*, the code-level features describing malware operations may not be comprehensive. Particularly, APIs with analogous functionalities might be used interchangeably to perform similar malware operations. To address it, we attempt to manually incorporate a broader range of functionally analogous or even equivalent APIs by referring to public resources (e.g., Stack Overflow, GitHub) to enrich the comprehensiveness of the *BxKG*. For example, by referring to the Android Developer documents for API *openConnection().getInputStream()*, it can be found that this API links to another API, i.e., *URLConnection.getInputStream()*, that performs the same functionality. Consequently, we update *URLConnection.getInputStream()* as a new API feature into the *BxKG* and establish a parallel relation between this updated API and the API *openConnection.getInputStream()*.

4.3.2 Automated Augmentation. Moreover, certain APIs will adjust as the SDK version is updated. Specifically, the method name or parameters of some APIs will be replaced, deprecated, or altered, e.g., API *DevicePolicyManager.resetPassword()* has been deprecated in API level 30 and was replaced by *DevicePolicyManager.resetPasswordWithToken()*. As a result, *BxKG* might not be able to recognize malware operation efficiently when it is implemented by calling the latest API. Consequently, *BxKG* should also be continuously updated rather than fixed. To address this issue, we propose a method based on crawler technology to automatically update the initial *BxKG* in response to API changes during SDK updates by targeting the Android Developer documentation. As shown in Algorithm 1, the algorithm starts by loading a list of target API URLs (marked in the *MBT* labeling process), which point to the API description web pages in the official website, and a set of pattern matching rules to detect changes in APIs. These patterns include:



resetPassword

Added in API level 8
Deprecated in API level 30

```
public boolean resetPassword (String password,
                int flags)
```

This method was deprecated in API level 30.
Please use `resetPasswordWithToken(ComponentName, String, byte, int)` instead.

1
```
<p class="caution">
  <strong>
  This method was deprecated in API level 30.
  </strong><br>
  Please use
  <code translate="no" dir="ltr">
     <a href="/reference/android/app/admin/DevicePolicyManager#resetPasswordWithToken
      (android.content.ComponentName,%20java.lang.String,%20byte[],%20int)">
       resetPasswordWithToken(ComponentName, String, byte, int)
     </a>
  </code>
  instead.
</p>
```

2
```
<!--
 if the caution does not exist,
 this code will be used to match.
-->
<div>
  Added in API level 8<br>
  Deprecated in API level 30
</div>
```

1
```
deprecated.*?(?:in|at)?\s*API level\s*(\d+)\..*? (?:use|See|switch to using|consider|should|Prefer
using|)\s((?:<code[^>]*><a href="([^"]+)">([\w\.\#\(\),\s]+)<\/a><\/code>\s*(?:or|and|,|;)?\s*)+)
```

2
```
Added in API level (\d+).*?Deprecated in API level (\d+)
```

🟧 Deprecated API level
🟩 New API URL
🟦 New API

Fig. 7. The deprecation mark detection in automated augmentation. The captured web page, a screenshot from the official website, shows that there was a deprecation along with a caution box alerting the user.

— *Method Signature Change Detection*: A pattern $P_{signature}$ is used to detect changes in method signatures, including the class name, method name, and the number or types of parameters. The pattern is represented as $P_{signature}$ = `Class_Name.Method_Name(Param_Type`$_1$`,Param_Type`$_2$, ...), where both the renaming of the methods and any change in the parameter list (addition or removal of parameters) will trigger the detection method.
— *Deprecation Tag Detection*: When a deprecation occurs, the deprecated API level and the replacement information of the new API (if any) will be explicitly mentioned on the web page, we thus can use different regular expressions to extract the deprecated API level, the new API (if any) and its URL from the web page's source code. The pattern $P_{deprecation}$ = `Regular_Expressions(Deprecated_API_Level, New_API, New_API_URL)`, as shown in Figure 7.

For each fetched web-page content, we applied these predefined pattern matching rules to identify any modifications or deprecations. When an API change is detected, the algorithm extracts the relevant key information (e.g., method's signature, API level at which the change occurred, API's description, URL pointing to the API, etc.) and integrates it into *BxKG*. A new API will be added to *BxKG* if an API has been replaced or deprecated in the latest version, while the old API will be marked as deprecated. To ensure that *BxKG* stays continuously updated, a periodic task is created to run the crawling and updating process on a regular basis.

Meanwhile, we randomly sampled 100 sensitive APIs from *BxKG* and examined them for changes. A manually constructed ground truth was used to investigate the accuracy of our crawler technology method. The results show that 83% of the sensitive APIs remained unchanged, 11% were deprecated, and 6% had signature changes. Our method achieved 100% accuracy on these 100 sampled APIs to detect these changes, successfully identifying all signature changes and deprecated APIs, which includes deprecation levels and alternative APIs (if any).

Algorithm 1: Automated Augmentation of *BxKG*

```
   Input: url_list: List of API URLs;
          patterns: Pattern matching rules;
          kg_current: Current BxKG.
   Output: kg_updated: Updated BxKG with latest API information.
1  kg_updated ← kg_current
2  foreach url ∈ url_list do
3      content ← FetchContent(url)
4      detected_changes ← ∅
5      foreach pattern ∈ patterns do
6          matches ← ApplyPattern(content, pattern)
7          if matches found then
8              detected_changes.add(matches)
9      foreach change ∈ detected_changes do
10         if change is a modification then
11             kg_updated.modify(change)
12         if change is a deprecation then
13             kg_updated.deprecate(change)
14         if change is a new API then
15             kg_updated.add(change)
   // Set a periodic task to execute the algorithm regularly for continuous updates
16 Schedule periodic execution of this algorithm
17 return kg_updated
```

Subsequently, APIs adapted to different API levels will be extracted and added to the initial *BxKG*. As a result, this dynamic updating mechanism allows *BxKG* to accommodate a greater variety of malware operations, even when they are implemented by APIs from different SDK versions. In the initial *BxKG*, the APIs are from API Level 30 or even earlier. However, with continuous updates in Android SDK versions, the API Level has now advanced to at least 33. Therefore, taking API Level 30 as the baseline, we observed the API changes in the graph after automated augmentation and verified whether these APIs were up to date by referring to Google Development Documents. Figure 8 illustrates the performance of automated API augmentation. The horizontal lines in the figure represent the API levels. For example, API *TelephonyManager.getLine1Number()* was added to the SDK at API Level 1 but was deprecated at API Level 33 and replaced by *SubscriptionManager.getPhoneNumber()*. It is clear that the APIs on the initial *BxKG* are dynamically changing and can be automatically updated, ensuring that *BxKG* can adapt to malware under various Android SDK versions.

5 Knowledge Graph Update

As constructed using a limited set of labeled *MBTs*, it also poses a challenge for identifying *MBTs* composed of new malware operations since they are absent from the initial *BxKG*. To resolve this problem, we propose an automated method called *self-updating*, which allows the initial *BxKG* to automatically enlarge its scale, thereby assisting in the discovery of future *MBTs*.

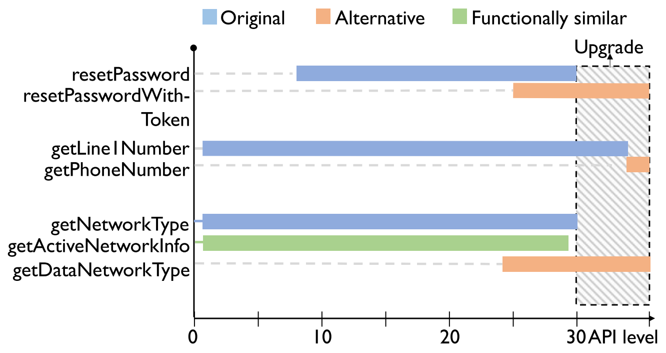


Fig. 8. Cases of automated augmentation for APIs and their life cycle.

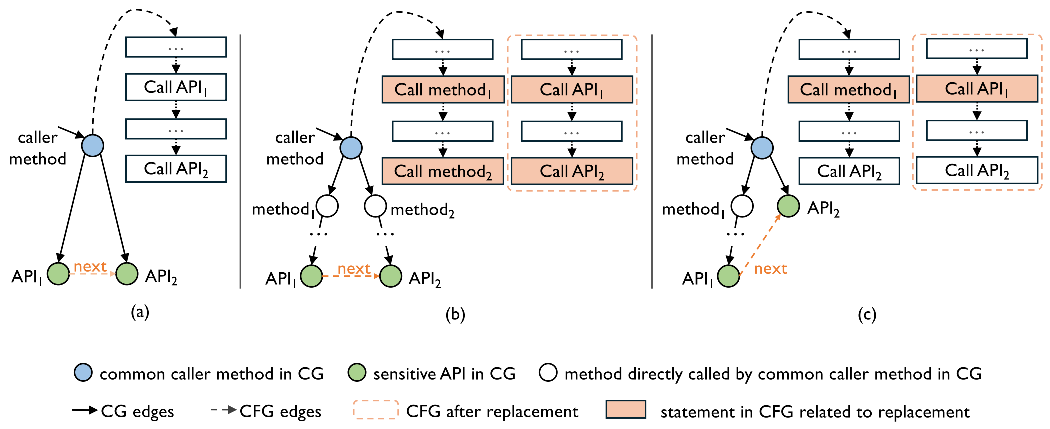


Fig. 9. Three cases that may be encountered when using control-flow analysis for relation search.

5.1 Generation of Code-Level Features for Malware

As mentioned in Section 2.2, code-level features are important indicators for identifying malware operations. Therefore, the first step is to obtain them from malware. Taking the malware as input and using Androguard, we can extract all the permissions that the malware is applying for and parse Intent from the AndroidManifest.xml file, and obtain all sensitive APIs in CG. In addition, when analyzing malware, it is important to understand the relations between different sensitive APIs, because it can map the relation between malware operations.

The relations between APIs can be divided into three distinct cases summarized from our in-depth observation as illustrated in Figure 9. Let $len(P_{(A,B)})$ be the shortest path length from nodes A to B. Assuming that we need to find the relation between API_1 and API_2, and the *method* is a common caller method (direct or indirect) of API_1 and API_2 whose CFG is used to determine relations. Three cases that may be encountered during relation extraction can be formalized as follows: *(a)* Two APIs are called directly by *method*, i.e., $len(P_{(method,\ API_1)}) = len(P_{(method,\ API_2)}) = 1$; *(b)* Two APIs are called indirectly by *method*, i.e., $len(P_{(method,\ API_1)}) > 1\ \&\ len(P_{(method,\ API_2)}) > 1)$; and *(c)* Only one API is called directly in *method*, i.e., $(len(P_{(method,\ API_1)}) > 1\ \&\ len(P_{(method,\ API_2)}) = 1)|(len(P_{(method,\ API_1)}) = 1\ \&\ len(P_{(method,\ API_2)}) > 1)$.

From CG, we can learn which methods directly or indirectly call sensitive APIs. However, for the specific call order of APIs, we need to use CFG, which is composed of **basic blocks (BBs)** connected by control flow edges. Assuming that we need to establish the call order between the two objects (either sensitive API or user-define method) within the CFG, we proceed as follows: ① If two objects reside within the same BB, the objects in the front position are called first since the statements in the block are executed sequentially. ② If two objects are located in different BBs, their order is determined by analyzing the control dependencies between these two blocks, i.e., the object in the controlling block (staring point of the control flow edge) is called before the object in the controlled block (ending point of the control flow edge). If the blocks are not reachable, the two objects are considered to be called in parallel.

As for (a), both APIs are called by the same method, therefore, we can directly determine the relation between these APIs in the common caller method's CFG. For (b) and (c), the issue lies in the fact that the common caller method may invoke an API through multi-layer methods, making it not feasible to directly establish the relations between two APIs in its CFG. To address this issue, we trace back along the call edges in CG from these APIs to find methods that are directly invoked by the common caller method, such as $method_1$ and $method_2$ shown in Figure 9. Then we record the mapping between APIs and found methods, and replace the position of the methods in the common caller method's CFG using corresponding APIs, so as to directly determine the relations between two APIs in the CFG. By traversing all sensitive APIs (including those identified by [57] and those confirmed during manual labeling in Section 3.1) on the CG and using CFG to calculate the relations between sensitive APIs in real time, we can ultimately construct a dependency graph of sensitive APIs (i.e., *SADG*).

Meanwhile, since there is no interface available to extract string constants that determine the specific use of APIs from malware, for each sensitive API that requires a constant string constant, we will look at the CFG (also through Androguard) of the method that calls this API to check whether this string constant actually used by this API. If so, the string constant will be integrated into this sensitive API node on the *SADG*. As a result, the *SADG* assists us in narrowing down the scope of malware operations identification and relation search rather than the entire code within the Android bytecode program.

5.1.1 *Completion of CG.*

The construction of the CG serves as the foundation for updating the *BxKG* and identifying the *MBTs* in this work. Particularly, the accuracy of the CG directly impacts the ability to capture program flows, thus influencing the extraction of the relations between sensitive APIs. Many existing studies rely on Androguard for generating CGs and CFGs [27, 33, 49, 57, 61]. However, we found some limitations in Androguard's default CG construction, particularly in handling Android-specific event-driven and callback mechanisms—critical aspects for accurately capturing program flow. Given the importance of CG completeness for our analysis, we addressed the gaps in Androguard's CG generation by improving it at four key points: *(1)* lifecycle callback methods, *(2)* event-driven callback methods, *(3)* asynchronous calls, and *(4)* **inter-component communication (ICC)** relations [9].

Firstly, we utilize Androguard to construct an initial CG based on invocation statements. Subsequently, for the discontinuity of lifecycle callback methods in CG, we introduce a virtual main method named "dummyMain" for each Android component (e.g., Activity, Service, Broadcast Receiver, and Content Provider). Within this method, we establish invocation edges between lifecycle callback methods according to the order specified by the Android development documentation. Next, to handle the discontinuity of event-driven callback methods, we utilize FlowDroid [5] to collect callback methods within each Android component. Referring to the callback methods results of FlowDroid, we can add invocation edges to CG. As for asynchronous calls, based on the predefined invocation rules (as shown in Table 2), we first capture statements related to asynchronous operations (e.g., *Thread.start()* and *AsyncTask.execute()*). These statements are further analyzed to

Table 2. Calling Convention of Asynchronous Calls

Class	Calling Convention
android.os.AsyncTask	execute() ⟶ onPreExecute() ⟶ doInBackground() ⟶ onPostExecute()
java.lang.Runnable	start() ⟶ run()
java.lang.Thread	start() ⟶ run()
android.os.Message	sendMessage() ⟶ handleMessage()

identify the caller and the actual callee. Subsequently, invocation edges between the caller and the callee are dynamically created and added to the CG, thus representing asynchronous call relations. For example, in the statement "Thread t = new Thread(new MyRunnable())," the caller is identified as *Thread.start()*, while the parameter type specified during the creation of the *Thread* instance is *MyRunnable*. From this, the actual callee is inferred to be the *MyRunnable.run()*. In the case of *AsyncTask*, type analysis is performed to locate its specific implementation class, allowing identification of other lifecycle methods (e.g., *doInBackground()*, *onPreExecute()*, and *onPostExecute()*). Invocation edges are then added between these lifecycle methods. This approach helps the CG obtain the implicit relations inherent in asynchronous calls. Finally, our analysis leverages existing analysis [24, 60] to identify ICC relations, which include explicit intents (e.g., *startActivity()*) and implicit intents resolved through attributes such as action or category. Using these data, we map the invoking methods in source components to the lifecycle entry points (e.g., *onCreate()*, *onStart()*) of the target components. By adding edges that represent both explicit and inferred implicit ICC relations, we dynamically enhance the CG to capture the communication flow between components, ensuring a comprehensive representation of the application's behavior.

For example, in some cases, the malware first calls API *TelephonyManager.getDeviceId()* to retrieve the device's IMEI and then calls API *AsyncTask.execute()* to start a asynchronous task. Therefore, we can get a execution flow from *TelephonyManager.getDeviceId()* to *AsyncTask.execute()*. However, it is unknown what this asynchronous task is and how this device information will be handled since the specific task are usually performed in the method *doInBackground()* of AsyncTask, while there is no call edge between *AsyncTask.execute()* and *doInBackground()* in initial CG. Therefore, in this case, we add the missing edge (i.e., between *AsyncTask.execute()* and *doInBackground()*) to the initial CG. In the method *doInBackground()*, the malware sends the collected data to a remote server using *HttpURLConnection.connect()*, thus we can get the complete execution flow (i.e., from *TelephonyManager.getDeviceId()* to *HttpURLConnection.connect()*) and deduce that the malware will leak the device ID. Our enhanced CG construction ensures that the CG accurately captures the entire execution flow of malware by introducing virtual edges.

5.2 Update of Knowledge Graph

As shown in Figure 10, based on the *SADG* generated by both CFG and CG, along with the initial *BxKG*, new graph entities and relations, contributing to new *MBTs*, could be extracted and then inserted into the initial *BxKG*. The self-updating takes widely-used malware samples as inputs and leverages the following three modules to automatically update the initial *BxKG*.

— *Relation Analyzer*: Applied to extract new relations between entities.
— *Entity Miner*: Used to mine new entities including code-level features and malware operations.
— *KG Updater*: Responsible for integrating new entities and relations into the current *BxKG*.

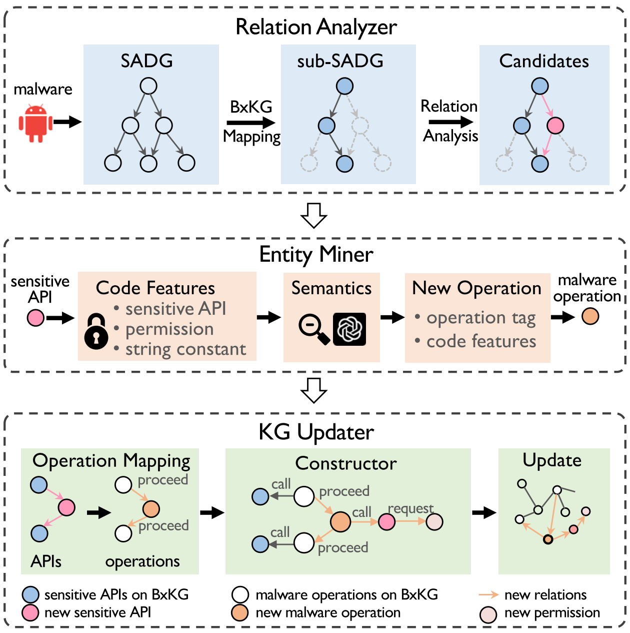


Fig. 10. Overview of self-updating.

We have implemented the self-updating in 2K lines of Python code. Figure 10 provides an overview of how the initial *BxKG* is updated through the above three modules. In this work, we update the *BxKG* using sources such as Genome [66], Drebin [4], AMD [51], and GP Malware [6]. Meanwhile, users also have the flexibility to incorporate their own malware datasets for updating purposes.

5.2.1 Relation Analyzer. The input for this module is widely used malware samples, from which we can extract *SADGs* for them as described in Section 5.1. A comparison with APIs in malware operations of *BxKG* and their relations will yield a subgraph of *SADG* (denoted as sub-*SADG*). Using *SADG* for relation extraction and API selection (relating to malware operation) is primarily driven by the following considerations. As malicious behavior is typically formed by multiple operations in succession, the new malware operation may be an extension or subsequent trigger operation of the existing *MBT*. It may be reasonable to introduce the new malware operation if there is evidence of a correlation or dependence between the proposed malware operation and the known *MBT*, and this relation can be proven by program analysis (i.e., based on *SADG*). In light of this premise, by linking it with the existing *MBT*, we will also be able to represent the full picture of the MBT more completely, which will improve our ability to detect and understand malicious behavior.

Specifically, for each leaf node (called checked API) in sub-*SADG*, check whether it has immediate successors or predecessors in *SADG*. If it has successors, the checked API, its successors, and the relation between them will be entered into the Entity Miner. If it has a predecessor, we need to further investigate whether at least one of its ancestors can reach this predecessor. If so, the checked API, the predecessor, the ancestor, and their relations will all act as the input of the Entity Miner. One special case is when there are two sensitive APIs on *SADG* that exist in *BxKG*, but the relation between them in *SADG* is absent from *BxKG*. This indicates that the malicious behavior pattern composed of these two APIs is lacking in *BxKG*. As the data (e.g., relevant malware operation, requested permission) pertaining to these two APIs already recorded by *BxKG*, this missing relation between APIs can be directly input into the KG Updater. With this selective strategy, we can find sensitive APIs that could potentially be used for malicious purposes and avoid blindly adding irrelevant APIs. In addition, we can capture the relations that exist between sensitive APIs.

5.2.2 Entity Miner. From the Relation Analyzer, we learn which sensitive APIs may contribute to building new malware operations. In this module, we aim to mine as complete entities (as defined in Section 4.1) as possible so as to prepare for the update of the current *BxKG*. On the one hand, code-level features are associated with the detection of malware operations. On the other hand, the semantics of code-level features are related to the semantics of newly introduced malware operations, which ultimately affects the quality of the final malware description. Therefore, firstly, for each of the extracted sensitive APIs that do not exist in the current *BxKG*, we will query its required permissions through an interface (i.e., *"load_api_specific_resource_module()"*) provided by Androguard. Next, both the API and permissions will be further used to model a new malware operation. To determine the tag (i.e., semantics) of new malware operation, we attempt to query the descriptive text of each API or permission from the Android Developer documents to obtain their semantics. And then the semantics of these features will be automatically summarized by a LLM (e.g., GPT-3.5 [32]), which can identify the core actions, recognize the object, and combine them into a concise phrase. Finally, the novel mapping between features and malware operation will act as the output of Entity Miner and be input into the KG Updater. By adopting this approach, we can make the representation of malware operations as accurate and comprehensive as possible so that they can be consistent with expert reports.

5.2.3 KG Updater. With a reasonable relation derivation and malware operation representation, the effectiveness of *BxKG* updates can be assured to some extent. The relation between malware operations can be mapped from the relation between their corresponding sensitive APIs. As for the input directly comes from the Relation Analyzer, we add a "proceed" relation between existing malware operations on the *BxKG* based on the missing relation between two sensitive APIs on *SADG*. For the entities (including malware operations and their features) and relations entered by the Entity Miner, we add the new malware operation and its relations between existing malware operations, which forms a new *MBT*, as well as the relations between it and its code-level features. Note that, before inserting an API or a permission into *BxKG*, it is necessary to check whether they have been on the initial *BxKG*. If they do not exist in the *BxKG*, we proceed with adding them.

Figure 11 gives an example to illustrate how we step-by-step add new graph entities and relations: *(1) Relation Analyzer*: First, we generate the *sensitive API dependency graph* (i.e., *SADG*) of input malware sample, from which we can identify sensitive APIs and relations between them defined by *BxKG* through comparing *SADG* and *BxKG*, thus generating the sub-*SADG* mentioned in Figure 10, i.e., "$A_1 \rightarrow A_2 \rightarrow A_3$," where $A_n (n = 1, 2, 3)$ represents a sensitive API. Next, we can find the new sensitive API (absent from *BxKG*) that connects to both the leaf node and non-leaf node of sub-*SADG*, i.e., A_4*: AccountManager.getAccounts()*, which acts as the cornerstone of self-updating and will be input into the Entity Miner. *(2) Entity Miner*: Afterwards, the permission that this input API needs to request, i.e., *GET_ACCOUNTS*, will be queried *via* an interface provided by Androguard. Based on the usage of *AccountManager.getAccounts()* is "Lists all accounts visible to the caller regardless of type" and the explanation of *GET_ACCOUNTS* is "Allows access to the list of accounts in the Accounts Service," ChatGPT summarizes these semantics, identify the core actions (i.e., "List"), recognize the object (i.e., "all accounts"), and combine them into a concise phrase. As a result, a new malware operation tagged *"List all accounts"* is created with *AccountManager.getAccounts()* and *GET_ACCOUNTS* as code-level features. *(3) KG Updater*: As relations between "*List all accounts*" and other malware operations can be mapped from the corresponding API sequence, a new *MBT* is constructed, i.e., *"Request permission → List all accounts → Send SMS messages."* Finally, the

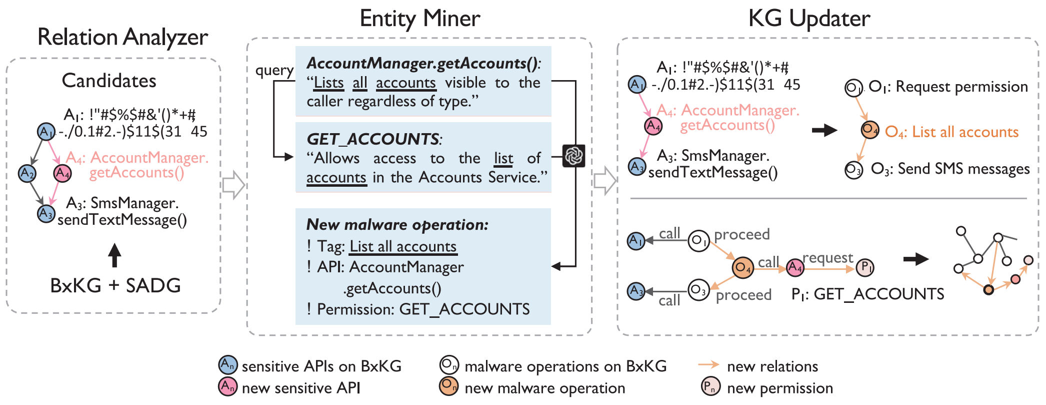


Fig. 11. Example of self-updating.

newly constructed *MBT* and the code-level features (including *AccountManager.getAccounts()* and permission *GET_ACCOUNTS*) of new malware operation will be inserted into *BxKG*.

Finally, our *BxKG* after self-updating comprises approximately 1,000 entities (including 347 malware operations, 622 code-level features) and 1,500 relations (including 554 relations between malware operations, 347 relations between malware operations and APIs, and 574 relations between code-level features), covering a diverse range of *MBTs*. The detailed results of the self-updating are shown in Section 7.2.1 (Evaluation of the Expansion of *BxKG* through the Self-updating).

Following the self-updating, we identified 2,119 meaningful *MBTs* within *BxKG*, encompassing a variety of malicious behavior types. Referring to the classification of malicious behaviors in [48], we categorized these *MBTs* into their 12 categories. The distribution of *MBTs* across different categories, considering overlaps, is as follows: "Aggressive Advertising" and "Privacy (Data) Stealing" each make up 19.4% of the *MBTs*. "Abusing SMS/CALL" follows with 18.7%, and "Stealthy Downloading" accounts for 15.3%. "Tricky Behavior" comprises 10.6%, while "Privilege Escalation" and "Abusing Accessibility" together make up 6.2%. Other categories include "Remote Control" at 4.0%, "Premium Service" at 2.2%, "Miner" at 0.8%, "Ransom" at 0.7%, and "Bank/Financial Stealing" at 0.5%. It is important to note that there is overlap among these categories, meaning that a single *MBT* could be classified under multiple categories.

6 Malware Profiling Based on *ProMal*

6.1 Extraction of *MBTs*

An *MBT* mainly consists of malware operations and relations between them. Thus, the first step to generate *MBTs* is to map the code-level features (i.e., manifest information collected from AndroidManifest.xml file and sensitive APIs extracted from *SADG*) of given malware to malware operations defined by the *BxKG* via pattern matching. After malware operation identification, relations between identified malware operations will be queried based on both *BxKG* and *SADG* to construct potential *MBTs*. Additionally, to ensure that the extracted *MBTs* about privacy leakage are as close as possible to the program's real execution, these relations will be filtered by dataflow analysis.

6.1.1 Malware Operation Detection and Relations Matching. Firstly, because *SADG* contains all sensitive APIs invoked by malware, we first remove sensitive APIs that have not been labeled by *BxKG* and the relations (start with this API or end with this API) associated with these APIs from *SADG*. Next, as mentioned in Section 2.2, features of malware operations include APIs, permissions,

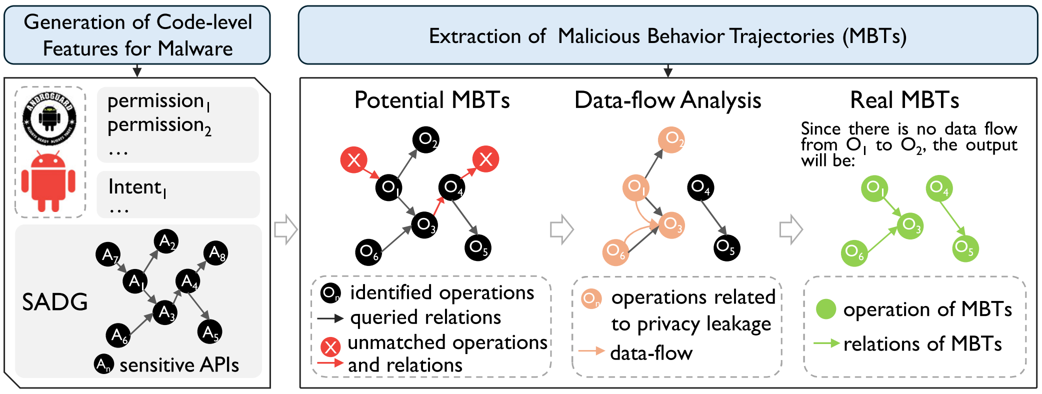
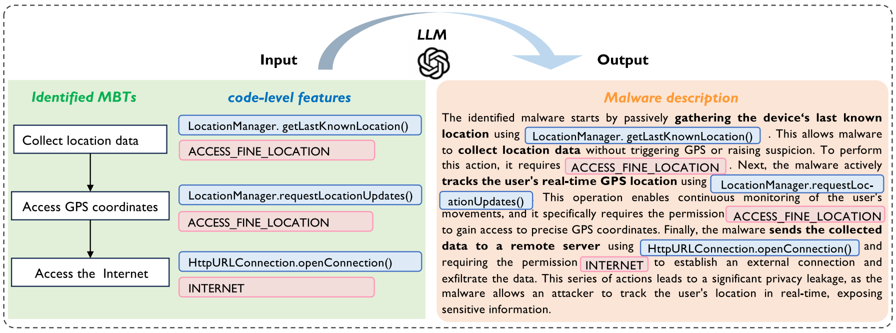

Generation of Code-level Features for Malware
Extraction of Malicious Behavior Trajectories (MBTs)
permission$_1$
permission$_2$
...
Intent$_1$
...
SADG
sensitive APIs
Potential MBTs
Data-flow Analysis
Real MBTs
Since there is no data flow from O$_1$ to O$_2$, the output will be:
identified operations
queried relations
unmatched operations and relations
operations related to privacy leakage
data-flow
operation of MBTs
relations of MBTs

Fig. 12. Overview of generation of *MBTs* based on *BxKG*.

Intent, and string constants, which are key metrics for determining whether a certain malware operation was performed. Specifically, a malware operation is only considered to be detected when all its features are matched by the malware's code-level features. As shown in Figure 2, the malware operation "*Get user's phone number*" consists of two features. If both of them exist in the extracted code-level features, it can be assumed that the malware will attempt to obtain the user's telephone number. Based on identified malware operations, their relations on *SADG* that potentially contribute to *MBTs* can be checked using the *BxKG*. If the relation between malware operations mapped from *SADG* differ from *BxKG*, this relation will be deleted, such as the relation from O_3 to O_4 shown in Figure 12. In addition, the *MBTs* associated with privacy leakage will be further verified through dataflow analysis, without which a false positive might occur.

6.1.2 Additional Verification. Considering that relying solely on control-flow dependencies without dataflow tracking may not effectively distinguish between legitimate and malicious API calls, potentially leading to a high false-positive rate, especially when identifying these *MBTs* that indicate privacy leakage, we further refine the relations between APIs based on Flowdroid's [5] taint analysis. In FlowDroid, a *Source* typically refers to the input point of sensitive information, and common *Sources* include APIs used to obtain sensitive user data, such as reading location data (i.e., *LocationManager.getLastKnownLocation()*). A *Sink*, on the other hand, represents the output point of the data, often involving potentially dangerous operations, such as transmitting personal data over a network or writing it to a file (i.e., *FileOutputStream.write()*). By using static analysis, FlowDroid detects potential leakages of sensitive information from *Source* to *Sink*. Notably, we select those APIs related to privacy leakage from our *BxKG* and then utilize them as supplements to FlowDroid's Sources and Sinks API list. In particular, we do not inspect if there is dataflow between all APIs but only screen for APIs related to information leakage to improve the efficiency of this verification. During this step, we filter out the Sources and Sinks APIs that lack data dependencies. Consequently, corresponding relations between APIs are also excluded. For example, in the process from dataflow analysis to the output of the final *MBTs*, O_2 and its relation with O_1 will be deleted as illustrated in Figure 12. In this step, we filter out relations extracted from malware that are related to data leakage but do not actually have data dependencies, to enhance the accuracy of relation extraction.



The identified malware starts by passively **gathering the device's last known location** using LocationManager. getLastKnownLocation() . This allows malware to **collect location data** without triggering GPS or raising suspicion. To perform this action, it requires ACCESS_FINE_LOCATION . Next, the malware actively **tracks the user's real-time GPS location** using LocationManager.requestLocationUpdates(). This operation enables continuous monitoring of the user's movements, and it specifically requires the permission ACCESS_FINE_LOCATION to gain access to precise GPS coordinates. Finally, the malware **sends the collected data to a remote server** using HttpURLConnection.openConnection() and requiring the permission INTERNET to establish an external connection and exfiltrate the data. This series of actions leads to a significant privacy leakage, as the malware allows an attacker to track the user's location in real-time, exposing sensitive information.

Fig. 13. An example of malware description generated by LLM based on extracted *MBT* and code-level features.

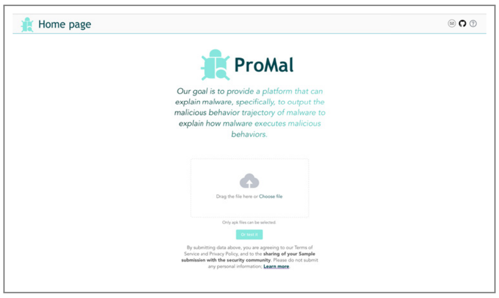
6.2 Description Generation

While *MBTs* offer a rich semantic understanding of malware, their technical nature can be still difficult for non-technical users to fully grasp. To make this information more accessible, we propose generating natural language descriptions of these malicious behaviors through LLMs (i.e., GPT-3.5 [32]). Given the need to translate technical details into accessible explanations, we employ a Chain-of-Thought prompting approach [53]. This method systematically guides LLMs through the process of generating detailed descriptions based on the extracted *MBTs*, simulating human reasoning by presenting a series of structured questions that help break down and explain the key aspects of malware behavior. This prompting approach consists of four parts: *(1)* Preamble: The Preamble provides the description and context of the task to guide the LLM. In our task, the Preamble ensures that the LLM understands the objective of generating a clear and accessible description of malware behavior from the *MBTs*, focusing on making the explanation more understandable to users. *(2)* One-shot Example: A concrete demonstration is given to show how to analyze each step of *MBTs*. This helps the LLM adhere to the reasoning process and ensures consistency in how it approaches the task. *(3)* Task Input: The task input consists of the *MBTs* along with their associated code-level features. *(4)* Chain of Thoughts: In this step, the LLM is guided through a sequence of structured questions designed to capture all critical aspects of the *MBTs*. These questions ensure that the LLM fully comprehends the malware operations, the underlying logic of each operation, and how these operations together form a coherent malicious behavior.

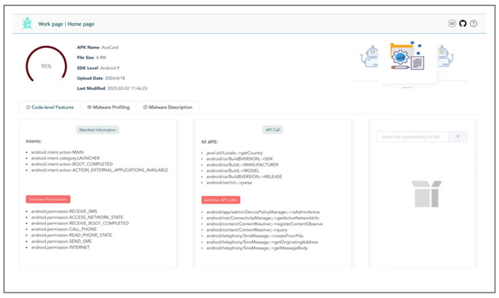
As shown in Figure 13, a detailed illustration of how the input *MBTs* map to the generated output was given. On the left side of Figure 13, we provide the *MBT* consisting of three malware operations and two relations indicating the execution order, along with the relevant APIs and permissions of each malware operation. The right side shows the corresponding coherent and detailed description, which is generated by the LLM based on the input. For instance, it first describes the malware collects device location (correspond to *"Collect location data"*) and user's location (correspond to *"Access GPS coordinates"*), and then sends them to a remote server *via* Internet (correspond to *"Access the Internet"*), showing the potential privacy violations. The technical details, i.e., APIs and permissions of malware operations, have been highlighted in blue and pink. Additionally, we can find that the malware description explains how the malware steals privacy information in an accurate, informative, and readable way.

6.3 Online Service of *ProMal*

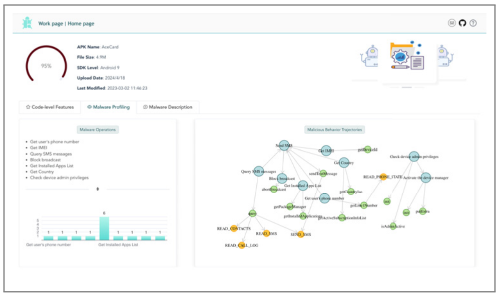
To make the work more useful to the Android mobile security community, we developed and have been maintaining an online website to provide the capabilities of *ProMal*. As shown


Home page
ProMal
Our goal is to provide a platform that can explain malware, specifically, to output the malicious behavior trajectory of malware to explain how malware executes malicious behaviors.

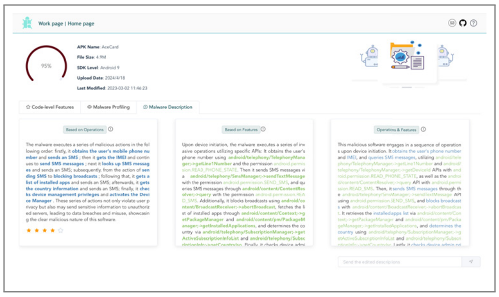
(a) The presentation of the homepage of *ProMal*.



(b) The presentation of code-level features.



(c) The presentation of the results of malware profiling.



(d) The presentation of the results of malware description.

Fig. 14. The online website of *ProMal*.

in Figure 14, an automatic analysis of malicious programs will begin once the user uploads the file.

7 Experimental Evaluation

7.1 Evaluation on *MBT*

To evaluate the *MBT* used in *ProMal*, *(1)* we conducted the first experiment to assess the necessity of the *MBT* by comparing with several existing approaches that focus on extracting sensitive API call subgraphs and sequences; *(2)* the second experiment is to investigate whether the extracted *MBTs* through *ProMal* match a human-labeled testing dataset based on security reports; and *(3)* we compared the capability of *MBT* extraction with a state-of-the-art approach.

Note that, these experiments were all conducted using the updated BxKG.

Dataset. We used the dataset comprising 105 Android malware from the GP Malware [6] to observe the performance of *MBT* extraction in *ProMal*. These applications penetrated the official Android Google Play Store between January 2016 and July 2021. In particular, these 105 malware samples were selected from 105 distinct families, which makes it possible to test whether our approach is universal. Additionally, detailed and manual reports of each sample are provided by Cao et al. [6], from which we labeled malware operations and *MBTs* by using the same method introduced in Section 3.1 to build the human-labeled testing dataset for this experiment. Finally, we manually labeled 122 *MBTs* in total.

It is important to note that this dataset is distinct from the Android malware dataset employed for the self-updating process, as well as the manually labeled dataset of 278 malware samples for graph construction in Section 3.1.

Table 3. Statistics of Subgraph, Sequences, and *MBT* Quantities across SeGDroid, S3Feature, LSCDroid, and *ProMal*

Metric	**SeGDroid**	**S3Feature**	**LSCDroid**	**MBT**
Average nodes per subgraph	7,045.57	7.03	N/A	N/A
Average edges per subgraph	11,118.37	8.20	N/A	N/A
Average subgraphs per APK	1	945.53	N/A	N/A
Average sequences per APK	N/A	N/A	1,013.67	N/A
Average APIs per sequence	N/A	N/A	5.156	N/A
Average *MBTs* per APK	N/A	N/A	N/A	1.26

7.1.1 Evaluation of MBT Necessity on Human-Labeled Testing Dataset. To assess the necessity of the *MBT*, we replicated and analyzed several existing methods that focus on extracting subgraphs involving sensitive API call or sensitive API sequences, to assess whether these methods could replace *MBT* for extracting malicious behaviors and inputting them into LLMs for malicious behavior description generation.

Setup. The methods chosen for comparison include SeGDroid [27], S3Feature [33], and LSCDroid [49]; they collect subgraphs involving sensitive API calls or sensitive API sequences to detect malicious behaviors. For S3Feature [33], we directly used their open sourced code to extract the corresponding subgraphs. While the others lack available code, we follow up the extraction methods introduced in SeGDroid and LSCDroid and reproduce their implementation based on the sensitive APIs used in MalScan [57] to extract the corresponding subgraphs and sequences, respectively.

Results. As shown in Table 3, the results demonstrate that the subgraphs and sequences extracted by SeGDroid, S3Feature, and LSCDroid were too large to be directly processed by LLMs and further used to generate a malware description.

Specifically, both SeGDroid and S3Feature generated excessively large number of subgraphs despite pruning. On average, the subgraphs extracted by SeGDroid contained 7045.57 nodes and 11118.37 edges, including a huge number of irrelevant to malicious behaviors, making it difficult to isolate critical behaviors. Similarly, S3Feature produced 945.53 subgraphs per APK and struggled to directly distinguish malicious from non-malicious operations based on these subgraphs, leading to unnecessary complexity and impracticality for generating behavior descriptions with LLMs. LSCDroid, while extracting sensitive API sequences (averaging 1013.67 sequences with 5.156 APIs per sequence), generates a large number of sequences, most of which are irrelevant. This abundance of irrelevant sequences also limits LSCDroid's ability to capture critical operations, similarly making it challenging for LLMs to produce coherent and accurate descriptions of malicious behaviors.

The experimental results highlight several key issues with existing extraction methods of subgraph and sequences for generating malware description: *(1) Overly large subgraphs or sequences*: These methods extract an overwhelming number of nodes, edges, subgraphs, and sequences, far exceeding the capacity needed for LLMs to describe malware behaviors. *(2) Overly much irrelevant information*: By focusing solely on direct and all relevant sensitive API calls, often producing subgraphs or sequences filled with irrelevant or redundant information.

In contrast, 1.26 *MBTs* on average were extracted per APK, which is a fine-grained representation of malicious behaviors that are specifically tailored for accurate interpretation. The number of *MBTs* extracted by *ProMal* is more practical for generating malware description by using the LLMs. It is important to clarify that this does not imply that each APK exhibits only 1.26 distinct malicious behaviors. During the extraction process, if a malware sample includes multiple malware operations that are closely linked, they are considered part of a single coherent malicious operation. For

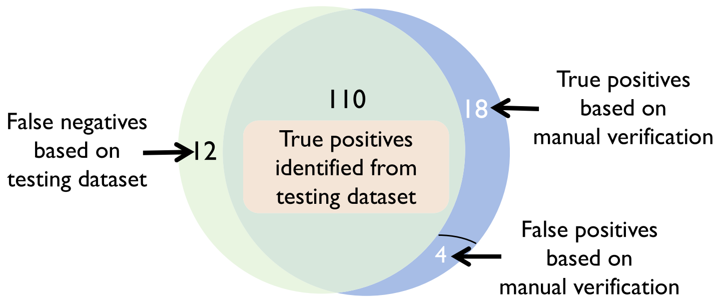

110
True positives identified from testing dataset
False negatives based on testing dataset
12
18
True positives based on manual verification
4
False positives based on manual verification

Fig. 15. The results of *MBT* extraction on testing dataset.

example, an *MBT* might simultaneously capture the leakage of a phone number, device information, and SMS content.

7.1.2 Evaluation of MBT Extraction on Human-Labeled Testing Dataset. The ability of *MBT* extraction plays a vital role in generating malware descriptions; therefore, we first investigated the effectiveness of *MBT* extraction.

Setup. We used *ProMal* to extract *MBTs* from 105 malware samples, subsequently comparing them with human-labeled *MBTs* from the same samples. For evaluating this comparison, we used Precision, Recall, and F1-Score as evaluation metrics.

Results. As shown in Figure 15, out of the 122 manually labeled *MBTs* within these 105 malware samples, *ProMal* successfully identified 110 of them but missed 12 cases. In addition, *ProMal* detected 22 *MBTs* that were not present in the human-labeled testing dataset. To further validate whether these detected *MBTs* were false positives, we carried out an in-depth manual analysis. The analysis revealed that 18 of these *MBTs* were indeed correct, while the remaining 4 were false positives. This phenomenon highlights that despite experts can disclose the majority of malware behaviors in their released security reports, some real malicious behaviors may still be overlooked. Such occurrences further underscore the necessity and significance of automated malware profiling tools. Based on the experimental results, *ProMal* demonstrates accurate identification of *MBTs* with a Precision of 96.97% ($\frac{110+18}{128+4}$) and a Recall of 91.43% ($\frac{110+18}{128+12}$), resulting in a corresponding F1-Score of 0.94.

We further analyzed the reasons for the false alarms. First, *ProMal* relies on static analysis to collect code-level features, rendering it unable to interpret dynamically loaded classes [36]. Besides, the implementation of malware operations is sometimes independent of API calls, such as by calling a third-party library. We cannot extract complete APIs so far, causing a malware operation matching failure, which then leads to 12 false negatives. The AndroidOSXavierAXM malware evaded our detection by using a library to dynamically download and drop a payload. Lastly, the actual functionality of some sensitive API calls is related to specific API parameters. *ProMal* may fail to correctly distinguish *MBTs* due to the lack of parameter, thus resulting in four false positives. For example, malware usually uses *PackageManagerService.setComponentEnabledSetting()* to hide its icon (belongs to Activity). This API can be used to set the input component's enabled state. However, the component encompasses not only Activity but also Service, Broadcast Receiver, etc. Thus, a false positive can occur when the actual controlled component is not necessarily an icon. These challenges result in the non-extraction or incorrect extraction of *MBTs*. Despite these challenges, *ProMal* maintains high consistency with the human-labeled testing dataset in *MBT* extraction and contributes to generating accurate and detailed malware descriptions.

Remark: *We compared the MBTs extracted by ProMal with the human-labeled testing dataset to quantitatively assess the effectiveness of MBT extraction. The F1-Score of MBT extraction using ProMal reaches 0.94.*

7.1.3 Evaluation of MBT Extraction by Comparing with XMal. To further assess the superiority of *ProMal* in terms of the *MBT* extraction compared to state-of-the-art approaches, we compared our work with XMal [54], which is an explainable ML-based approach that can classify malware with high accuracy as well as explain the classification decision. Furthermore, the XMal has been compared with the state-of-the-art explainable ML-based methods (i.e., Drebin and LIME [41] which achieved better explanation results compared to work such as SHAP [28], Anchor [42], LORE [17], and LEMNA [19] in the scenario of Android malware [12]) to demonstrate its effectiveness. Additionally, in these works, only XMal provides malware descriptions for users based on the identified key features. Therefore, we choose XMal as the state-of-the-art approach in this work.

Dataset. For this experiment, we used the same dataset from Section 7.1, which contains 105 samples with expert-labeled ground truth, as well as the 16 malware samples released by XMal. These 16 malware samples were randomly selected from the top 16 malware families with the largest number of samples according to the malware family tags provided by Drebin [4]. The ground truth for these samples was provided through manual analysis by experts in XMal's original work.

Setup. For the 121 malware samples (16 from XMal + 105 from our dataset, 121 samples in total), we used *ProMal* to extract *MBTs* for each sample. For XMal, we obtained the trained model from the first author of XMal and further deployed it based on its open source code under their experimental configurations used in their paper. Unfortunately, the evaluation process encountered challenges due to limitations in XMal's output format: XMal did not provide results in the form of sequences of malware operations, making a fair assessment based on *MBT* difficult. While XMal could produce ordered descriptions of malware operations through predefined orders, directly comparing the descriptions may result in erroneous results due to the semantic merging of similar features inherent in XMal processing. To bridge this gap, we manually enhanced the granularity of XMal's descriptions of *MBTs* based entirely on their results. For instance, in the case of XMal, the description "*collect info on the device and send it to the remote server over the internet*" was represented using the semantics of "*Collect IMSI/location*," generated by the feature *TelephonyManager.getSubscriberId()* Thus, we corresponded it to the *MBT* "*Get IMSI→Access the Internet*," which is identical to the form we extracted. Additionally, although the ground truth provided by XMal was verified by expert reports, it is still limited due to concise descriptions, failing to fully reflect the actual *MBTs*. To complement the ground truth, we manually reverse-engineered the 16 samples and analyzed the *MBTs* based on XMal's ground truth results. The results were cross-validated.

Results. After determining the corresponding *MBTs* for XMal (semantically identical to the XMal results), we compared their performance on the 121 malicious samples and the results about the comparison with XMal are accessible on our website [2]. Firstly, the experimental results for the extraction of *MBTs* are shown in Table 4. *ProMal* outperforms XMal significantly in the extraction of *MBTs*, achieving a F1-Score 0.35 higher than XMal (0.93 vs. 0.58). It is important to note that the lower precision of XMal here does not conflict with the original accuracy in the paper of XMal. This discrepancy arises because the evaluation criterion in the original paper of XMal is the detection accuracy in malware classification, not the accuracy of detecting *MBTs* within the malware. We further analyzed the reason for the low Recall in XMal. This is attributed to its approach of extracting only partial key features to generate malicious behavior descriptions, leading to its lack of comprehensive detection capability for *MBTs* of malicious behaviors.

For example, in the case of the BaseBridge0 sample, XMal failed to detect malware operations like "*Get installed app list*," "*Get mode*," and "*Get Android OS version*" compared to Ground Truth. This is

Table 4. The Evaluation Results of *MBT* Extraction between *ProMal* and XMal

Tool	Precision	Recall	F1-Score
ProMal	95.03% **(32.09%↑)**	90.90% **(36.49%↑)**	0.93 **(0.35↑)**
XMal	62.94%	54.41%	0.58

The bold values in the table indicate the improvements of *ProMal* over XMal in terms of Precision, Recall, and F1-Score for *MBT* extraction performance.

Table 5. Datasets for Automated Self-Updating

Publication Year	**Datasets**	**Samples (Files)**
2012	Gemome	1,200
2014	Drebin	5,560
2017	AMD	23,217
2022	GP Malware	1,133
	# Total Number	31,110

due to the trade-off that XMal made between improving interpretability in classification models and identifying more malware operations, which also indicates that accurately pinpointing *MBTs* in malware is a significant challenge. In comparison, our approach performs better, achieving a Precision of 95.03% and maintaining a Recall of 90.90%. We further analyzed the reasons that *ProMal* cannot accurately extract all *MBTs* in several samples. In the case of sample Kmin0, which extracts phone numbers by parsing SMS message contents rather than through system API calls, resulting in it not being included in the *MBTs*. Meanwhile, distinguishing certain malware operations is associated with specific API parameters as mentioned above, *ProMal* may struggle to precisely determine which malware operation is occurring, leading to false positives. Despite these challenges, our approach maintains significantly higher accuracy in *MBTs* extraction compared to XMal, laying a foundation for providing clearer malware descriptions in profiling malware.

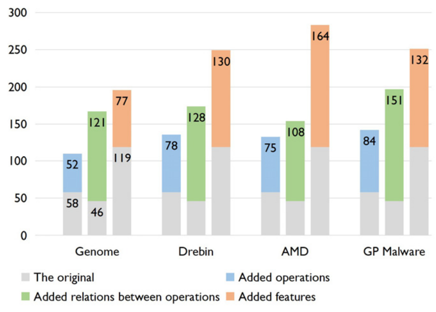
Remark: *By comparing the results of MBT extraction between ProMal and XMal, it is evident that ProMal extracts MBT more accurately and comprehensively (0.93 vs. 0.58 on F1-Score), laying the foundation for a more detailed profiling of malicious behaviors.*

7.2 Evaluation on Self-Updating and Graph Reasoning

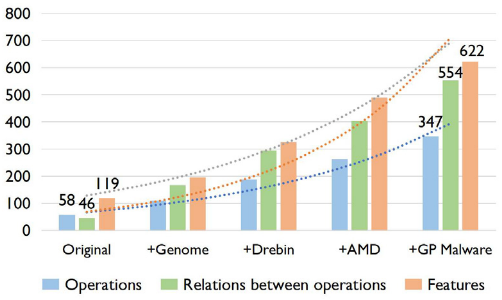
7.2.1 Evaluation of the Expansion of BxKG through the Self-Updating. A key aspect of our methodology is its automated self-updating component, which is capable of detecting and integrating malware operations, new features, and various types of relations into the existing knowledge graph. This ensures the ongoing enlargement of the knowledge graph. The objective of this experiment is to evaluate the expansion of *BxKG* before and after the update.

Dataset. To evaluate the effectiveness of the automated self-updating, we collected malware samples from Genome [66], Drebin [4], AMD [51], and GP Malware [6] (*excluding the 105 samples used in the experiments in* Section 7.1) which are the most commonly used malware datasets. Table 5 illustrates that these datasets encompass 31,110 samples.

Setup. The widely-used malware datasets were utilized one by one based on their release date to increase the number of update features, malware operations, and relations of the initial *BxKG*. We randomly selected 1,100 malware samples from each dataset to cross-compare the experimental results, where 1,100 is determined by the smallest dataset (i.e., GP Malware). As four datasets are



(a) When using different datasets to update the graph, the number of malware operations, relations, and features increases respectively.



(b) Overview of scale changes before and after self-updating.

Fig. 16. The update of *BxKG*.

utilized, the initial *BxKG* iteratively undergoes four consecutive automated self-updates, with each subsequent update building upon the previous one. To monitor the self-update, we tracked changes in the amounts of malware operations, relations between malware operations, and features about malware operations after each update.

Results. We show the updated results of self-upgrading in Figure 16(a) and (b). In Figure 16(a), we can observe that the initial *BxKG* consists of 58 malware operations, 119 features, and 46 relations between two malware operations. However, when these four popular datasets (i.e., Genome, Drebin, AMD, and GP Malware) are used to augment and update the initial *BxKG*, the scale of malware operations, features, and relations on the knowledge graph has subsequently changed four times. For instance, the initial *BxKG* gained 77 new features, 52 new malware operations, and 121 new relations between malware operations following the first round of automated self-updating. As shown in Figure 16, the scale of the graph keeps expanding during the whole process of self-updating. As a result, the number of malware operations, features, and relations between malware operations on the updated *BxKG* has increased significantly. After the last round of automated self-upgrading using the GP Malware dataset, the number malware operations, features, and relations has reached 347, 622, and 1,475 (including 554 relations between malware operations, 347 relations between malware operations and APIs, and 574 relations between features), respectively. Note that the number of features is significantly higher than the number of malware operations, as a malware operation can be implemented by multiple APIs. Additionally, the part of graph nodes automatically created from various datasets are displayed in Table 6. Columns 1 and 2 list various features for different malware operations and their semantics. The third column provides the semantics of malware operations summarized by LLM.

Remark: *To enhance the profiling ability, we introduce an automated self-updating method to expand BxKG, resulting in a significant increase in its scale. The approach successfully extracts practical MBTs from given datasets. Simultaneously, the effective updating and reasoning capability of BxKG enhances the flexibility and reliability of ProMal.*

7.2.2 Evaluation of the Contributions of Self-Updating and Graph Reasoning.

To evaluate the contributions of self-updating and graph reasoning of *BxKG* for extracting *MBTs*, we observed the differences in results between the two groups (i.e., “before self-updating” vs. “after self-updating” and “without graph reasoning” and “with graph reasoning”).

Dataset. The used dataset is the same as that used in Section 7.1.2.

Table 6. Examples of the Newly Mined Graph Nodes after Automated Self-Updating

Code-Level Features	**Features' Semantics Obtained from Documentations**	**Operations' Semantics Summarized by LLM**
Self-Updating by the Dataset Genome		
android/app/AlertDialog$Builder;→setTitle android/app/AlertDialog$Builder;→setMessage android/app/AlertDialog$Builder;→setPositiveButton android/app/AlertDialog$Builder;→create android/app/AlertDialog;→show	Set the title displayed in the Dialog. Set the message to display. Set a listener to be invoked when the button is pressed. Create an AlertDialog to this builder. Start the dialog and display it on the screen.	Show alter dialog
java/net/URL;→getHost java/net/URL;→getPort java/net/URL;→getPath	Get the hostname of this URL, if applicable. Get the port number of this URL. Get the path part of this URL.	Get url
Self-updating by the dataset Drebin		
android/os/Build$VERSION;→SDK	Get the user-visible SDK version of the framework.	Get SDK version
java/lang/Class;→getField 581 java/lang/reflect/Field;→get android/os/Build;→MODEl	Retrieve the object for a specified field. Return the value of the field represented by this Field. Get the end-user-visible name for the end product.	Get model ID
Self-updating by the dataset AMD		
android/content/Context;→getResources android/content/Context;→getPackageName android/content/res/Resources;→getIdentifier android/content/res/Resources;→getText	Return a resource instance for the application's package. Return the name of this application's package. Return a resource identifier for the given resource name. Return a string value associated with a particular resource ID.	Get application name
android.permission.ACCESS_NETWORK_STATE android/net/ConnectivityManager;→getActiveNetworkInfo android/net/NetworkInfo;→getTypeName	Allow applications to access information about networks. Return details about the currently active default data network. Return a resource identifier for the given resource name.	Get network type name
Self-updating by the dataset GP Malware		
android.permission.ACCESS_WIFI_STATE android/net/wifi/WifiManager;→getConnectionInfo android/net/wifi/WifiInfo;→getBSSID	Allow applications to access information about networks. Return the Wi-Fi information. Return the basic service set identifier of the current access point.	Get bssid
android/hardware/Sensor;→getName android/hardware/Sensor;→getVersion android/hardware/Sensor;→getVendor	Get the name string of the sensor. Get the version of the sensor's module. Get the vendor string of this sensor.	Get sensor info

Setup. (1) For the first group, we extracted *MBTs* by using the initial *BxKG* and then compared them to the results obtained in Section 7.1.2 (i.e., *MBT* extraction results based on the updated *BxKG*). We aim to investigate the differences in results between "before self-updating" and "after self-updating." For the self-updating, we used four different kinds of widely used malware datasets including Genome [66], Drebin [4], AMD [51], and GP Malware [6]. *(2)* For the second group, we investigated which *MBT* extraction results benefited from the *BxKG*'s reasoning capability, meaning that one experiment relied solely on all existing labeled *MBTs* in *BxKG*, while the control group depended on all existing labeled *MBTs* as well as new *MBTs* reasoned from the original ones. We aim to investigate the differences in results between "with graph reasoning" and "without graph reasoning."

Results. Table 7 shows that if only relying on the initial *BxKG* to extract *MBTs* on the same dataset, the F1-Score is only 0.66, and the Precision and Recall are only 68.18% and 64.29%, respectively. This indicates that knowledge obtained from a fixed and limited malware dataset cannot guarantee the robustness of malware profiling. Furthermore, based on Table 7, we noticed that the self-updated *ProMal*, without graph reasoning capability, resulted in a decrease of 13.52% and 9.12% in Precision and Recall for *MBT* extraction, along with 0.11 F1-Score reduction. For instance, consider the GhostTeam malware sample: *ProMal* extracted a *MBT* "*Look for installed App→Access the Internet→Download File→Install Apps → Launch Apps*." Here, "*Look for installed App→Access the Internet*" and "*Access the Internet→Download File→Install Apps→Launch Apps*" could each serve as a *MBT*. However, our *BxKG*, based on a common node "*Access Internet*" and the actual control and data dependencies between these two *MBTs* in the malware sample, deduced that they can actually

Table 7. The Evaluation Results on Self-Updating and Graph Reasoning

Metrics	Before Self-Updating	After Self-Updating	With Graph Reasoning	Without Graph Reasoning
Precision	68.18%	96.97% **(28.79%↑)**	96.97%	83.45% **(13.52%↓)**
Recall	64.29%	91.43% **(27.14%↑)**	91.43%	82.31% **(9.12%↓)**
F1-Score	0.66	0.94 **(0.28↑)**	0.94	0.83 **(0.11↓)**

The bold values in the table represent the improvements in *ProMal*'s *MBT* extraction ability after self-updating with the *BxKG* compared to before self-updating. Additionally, the changes in *ProMal*'s *MBT* extraction ability are shown when graph reasoning of *BxKG* is not utilized.

function as one new *MBT*. In other words, this complete *MBT* involves checking for the installation of specified (antivirus) software, then connecting to the network, downloading, installing, and running malicious software. Through such graph reasoning capability, we can uncover intricate relations between seemingly disparate *MBTs* within the malware, aiding in the identification of previously undetected *MBTs*, while also leading to more accurate malware profiling. Although the proportion of *MBTs* obtained through this graph reasoning capability appears relatively small in the results, this is partly due to the limited size of our testing dataset. We believe that testing on a more larger dataset would yield more *MBTs* through the graph reasoning capability.

***Remark:** Our evaluation showed the real impact of self-updating and graph reasoning on MBT extraction. We also established the necessity of knowledge graph construction, especially its reasoning ability for MBT extraction.*

7.3 Ablation Study

Besides the effectiveness evaluation above, we further conducted an ablation study to highlight the effectiveness of individual components in the *MBT* extraction, including *(1)* completion of CG (Section 5.1.1), *(2)* analysis of control flow, and *(3)* analysis of dataflow.

Dataset. The used dataset is also the same as that used in Section 7.1.2.

Setup. To thoroughly evaluate the efficacy of each component, we systematically removed each of them from the *MBT* extraction one by one. Our full approach, incorporating all components, served as the baseline for comparison. This approach allowed us to compare the performance of each component with and without its inclusion, thereby assessing their individual impact on the overall effectiveness.

Results. Each component's effects are shown in Table 8, with the most optimal results achieved when all components are combined. The following will elaborate on the results of different components in the ablation study.

7.3.1 Completion of CG. Table 8 shows that without CG completion, the Recall decreases by 22.86%. In particular, the utilization of the completion of CG significantly reduces false negatives. When comparing the results between applying CG completion and not, we found that over 22% of the *MBTs* leverage specific Android architecture features, such as asynchronous calls, life-cycle callbacks, and even inter-component collusion, to execute malicious payloads. This observation hints at a potential strategy used by malware developers to avoid detection: they may divert payload execution from the main application execution flow, potentially complicating call paths to evade detection mechanisms.

Table 8. The Impact of the Three Components on the *MBT* Extraction

CCG	CF	DF	Precision	Recall	F1-Score
✓	✓	✓	96.97%	91.43%	0.94
✗	✓	✓	96.00% **(0.97%↓)**	68.57% **(22.86%↓)**	0.80 **(0.14↓)**
✓	✗	✓	21.02% **(75.95%↓)**	91.43%	0.34 **(0.60↓)**
✓	✓	✗	90.15% **(6.82%↓)**	90.84 **(0.59%↓)**%	0.90 **(0.04↓)**

In this table, CCG, CF, and DF, respectively, stand for completion of CG, Analysis of control flow (CF), and Analysis of dataflow (DF). The bold values in the table represent the difference in performance compared to the best results, which are achieved when all three components—CCG, CF, and DF—are used.

7.3.2 Analysis of Control Flow. Table 8 reveals that the Precision decreases by 75.95% when control-flow analysis is not applied. This demonstrates the importance of control-flow analysis, specifically how the absence of API call sequence information might affect the accurate identification of *MBTs*. The relations between different malware operations are crucial for *MBT* construction and enriching semantic information. Therefore, obtaining API call sequence information through control-flow analysis is particularly crucial in *MBT* identification and extraction.

7.3.3 Analysis of Dataflow. Table 8 illustrates a 6.82% improvement in the Precision rate upon incorporating dataflow analysis. However, the effectiveness of the dataflow analysis in improving the performance of our approach is not as pronounced as expected. We analyzed the reasons behind this. In the manual analysis of malware, it is observed that if APIs that may execute malicious behavior are control-flow related, they are likely to execute malicious behavior together. Nevertheless, for APIs involving information leaks, applying dataflow analysis indeed yields more accurate results. For instance, in the BKotlindHRX malware sample, without dataflow analysis, we would identify a *MBT* of "*Get IMSI→Send SMS*." Dataflow analysis filters out this *MBT* due to the absence of data transfer, which aligns with the human-labeled testing dataset. Through manual analysis, we found that this APK simply sends a text message after retrieving IMSI, with the message content unrelated to the retrieved data.

Remark: *The ablation study demonstrated that all three components contribute positively to the results on MBT extraction, with analysis of control flow making the most significant contribution, followed by completion of CG, and finally analysis of dataflow.*

7.4 Practicality Evaluation of Description Generation

To verify the advantages of our *MBTs* in assisting users in understanding why malware is classified as malware, a user study was conducted to explore the respondents' assessment of the malware descriptions generated by *ProMal*.

Dataset. We used 16 malware samples that have malware descriptions in XMal for this experiment.

Setup. The malware descriptions of these 16 samples are used to design our user study. Furthermore, we configured the following settings to assess the readability of the generated malware descriptions:

- XMal: We directly leveraged the malware description results generated by XMal as presented in their paper.
- Baseline: We used code-level features corresponding to all *MBTs* as inputs to the LLM.
- ProMal: We extracted all *MBTs* along with their corresponding code-level features, feeding them into the LLM to produce corresponding interpretations.

Table 9. Taking Malware MobileTx0 as an Example, Three Types of Malware Descriptions will be Read and Rated According to Their Readability in the Study

	Malware Description	Score (1–10)
Type 1	This malicious program continues to request sensitive permissions, including **reading phone state** (`READ_PHONE_STATE`) and **sending SMS** (`SEND_SMS`). It utilizes key Android APIs, such as `Telephony- Manager.getSubscriberId()` method to **obtain the device's subscriber identifier** (IMSI), `URL. openConnection()` method for **network communication**, and `SmsManager.sendTextMessage()` method for **sending SMS**. These actions involve potential privacy and security risks, including the disclosure of user information and possible misuse for malicious activities.	6
Type 2	This malware **initially** **acquires the IMSI** (International Mobile Subscriber Identity) using the `READ_ PHONE_STATE` permission and the `TelephonyManager.getSubscriberId()` method. **Subsequently**, it **transmits this information** to a remote server over the Internet by utilizing `URL.openConne- ction()` method. **Following this**, the malware proceeds to **send SMS messages** through the `SEND_ SMS` permission and `SmsManager. sendTextMessage()` method. This sequence of actions may give rise to significant security risks.	9
Type 3	**Send SMS messages** to premium-rate numbers, **collect info** on the device, and **send it** to a remote server over the Internet	3

In this case, "Type 1" denotes baseline, "Type 2" refers to *ProMal*, and "Type 3" denotes XMal

Thus, each malware sample has three different styles of interpretations. To conduct our experiment, 30 participants with domain expertise from industrial companies and universities are recruited *via* email. A minimum of 1 year of experience in Android mobile security is required for all participants. Within the pool, 10% people represent industry professionals, while the remaining are affiliated with academic entities.

The survey started with a concise introduction. We informed participants that our task was to measure the readability of two distinct categories of description. To statistically evaluate the quality of the generated descriptions, we specified a scoring range of 1 to 10, with a higher score indicating that the interpretation of malware in this manner makes it easier to understand and accept. Participants were required to compare and rate three malware descriptions. Throughout the survey, the name of XMal, baseline, and *ProMal* were randomly replaced by "Type 1," "Type 2," and "Type 3" to reduce bias. Table 9 illustrates a part of the survey. As well, we also investigated with respondents their preferred style of description and why. The survey took an average of 25 minutes to complete.

Results. Finally, 30 valid survey results were collected. The survey results show that descriptions generated by *ProMal* have an average satisfaction score of 9.67, while descriptions generated by XMal and baseline have an average score of 3.37 and 6.32. Furthermore, 100% of respondents indicate that they prefer *ProMal*'s description. In particular, these descriptions interpret the specific process that may cause malicious attacks, which is easier to comprehend. Furthermore, it is logical and easier to form a *MBT* in the reader's mind, whereas another type of description is a patchwork of behaviors, making it difficult for readers to grasp the target.

Remark: *The user study indicates that 100% of respondents prefer the description generated by ProMal with an average satisfaction score of 9.67, which can generate a more human-readable description based on the extracted MBTs as well as the corresponding code-level features.*

8 Discussion

8.1 Special Case

During our experiment, we found that SendPay malware in Drebin attempted to monitor and send SMS messages, but neither XMal nor *ProMal* detected this malware operation. Particularly, it is confirmed to send messages after being scanned with over 70 different antivirus scanners by VirusTotal [47], an online service widely used by researchers to identify security threats. Further analysis finds that *SEND_SMS*, dangerous and hard-restricted permission, is absent from SendPay's AndroidManifest.xml file, leading to the failure of malware operation detection using *ProMal*. This indicates that, by bypassing permissions, malware can avoid detection, which is why the runtime permission model is introduced in Android 6.0 (i.e., API level 23) and higher. Therefore, the number of such cases results in a limited impact in our experiment 7.1.2 when analyzing malware with higher Android versions.

8.2 Limitations of *ProMal*

Despite comprehensive experiments demonstrating advancements in malware profiling, the current version of *ProMal* still exhibits certain limitations.

(1) One limitation of our method is the relatively small size of the dataset, despite it being sourced from 105 distinct malware families, which ensures a high degree of diversity. The limited dataset size is primarily due to the scarcity of high-quality, manually analyzed reports-a common challenge in Android malware behavior analysis. Constructing such datasets demands substantial expertise and effort, given the complexity and nuanced nature of behavior analysis in this domain. To address this, future work will focus on leveraging semi-supervised or unsupervised learning to efficiently generate additional high-quality labeled data. We also plan to collaborate with domain experts to expand and refine the dataset, ensuring the broader applicability of our method to real-world scenarios. *(2)* As mentioned in the experiments, *ProMal* faces its initial limitation in handling dynamically loaded classes [36]. However, our experiments revealed that the prevalence of malware utilizing dynamically loaded classes is limited. Moving forward, we aim to integrate dynamic information into *BxKG* to enhance its comprehensiveness. *(3)* The third limitation arises from our observation that some malware is able to leverage third-party components to execute malicious behaviors, including third-party libraries and cross-language libraries (i.e., *.so* files). Currently, we lack the capability to handle such cases. Similarly, within representative testing datasets, we found that cases of this nature are relatively very rare. Nevertheless, to enhance the robustness and reliability of *BxKG*, we plan to incorporate the influence of third-party components in the future. *(4)* The fourth limitation is relevant to the parameters of APIs. API calls play a critical role in identifying malware operations. Sometimes the parameters of API calls determine their intention. For instance, the API *PackageManagerService.setComponentEnabledSetting()* can control the enabled state of the input component. When the input is the MainActivity of malware, the given Activity will be disabled to hide the application icon, which is a common malware operation. The input components, however, will vary according to the actual situation, which may involve Services rather than Activities, i.e., the input of this API is a variable. As a result, *ProMal* is unable to determine whether the use of this API is malicious or benign due to the lack of automatic analysis of variable parameter pointing content. We also plan to incorporate this analysis capability into our future work. *(5)* The final limitation stems from the use of static analysis tools such as Androgurad and FlowDroid. Despite our best efforts to enhance the capabilities of these tools, such as substantially strengthening the CG, we still encounter limitations inherent to the used static analysis tools. We will continue to address potential impacts on profiling results arising from limitations imposed by static analysis tools.

9 Related Work

9.1 Android Malware Detection

Researchers from both industry and academia have developed various methods for malware detection, including signature-based approaches [11, 14, 67], behavior-based methods [45, 50, 58], and program analysis-based techniques [5, 16, 24, 52]. Currently, machine learning-based techniques [3, 4, 15, 39, 55], particularly deep learning [22, 30, 59, 63], have become promising solutions for malware detection and classification because they can defend against zero-day attacks and potentially keep up with the creation and evolution of malware. For example, Aafer et al. [3] trained a KNN classifier by learning relevant features extracted at the API level, achieving up to 99% accuracy and as low as 2.2% false positive rate. Wu et al. [55] used a k-nearest neighbors classification model with dataflow APIs as classification features to detect Android malware. Other machine learning algorithms, such as SVM [4], Random Forest [39], XGBoost [15], and Least Square Support Vector Machine [29], have also been applied to malware detection and proven effective. Regarding deep learning, Yu et al. [63] proposed using Artificial Neural Networks to train malware detection models. McLaughlin et al. [30] designed a malware detection system using deep convolutional neural networks to learn raw opcode sequences from disassembled programs. Kim et al. [22] utilized multimodal deep learning methods to learn various features, maximizing the advantages of different types of features. Xu et al. [59] applied Long Short Term Memory networks on the semantic structures of Android bytecode and used multilayer perceptrons on XML files to efficiently identify malware. In addition, Li and Li [23] combined permissions, components, system calls, and IP addresses through adversarial training to improve the robustness of their malware detection model. Şahin et al. [44] proposed a multiple linear regression-based malware detection model that used permission features and enhanced performance by employing ensemble learning, particularly bagging (majority voting). Rustam et al. [43] developed an image-based malware detection system using transfer learning and machine learning algorithms. Their hybrid model, which combined VGG-16 and ResNet-50, achieved 100% accuracy on 25 malware classes in the Malimg dataset by sequentially extracting hybrid feature sets through a stacking approach.

While machine learning and deep learning methods have significantly improved malware detection accuracy, they primarily emphasize classification performance over interpretability. These methods can detect whether an application is malicious but fail to explain why it is classified as such or what specific malicious behaviors it exhibits. This stems from limitations in machine learning and deep learning models, such as their black-box nature [40] and the risk of models learning irrelevant features like temporal variances in datasets [26]. *ProMal* aims to bridge this gap by generating clear and precise descriptions of malware behavior, enabling users to not only identify an application as malicious but also understand the specific malicious actions it performs, addressing the interpretability shortfall in existing approaches.

9.2 XAI-Based Malware Profiling

In recent years, machine learning-based Android malware detection has shown excellent performance with an accuracy achieving 99% [26]. However, these approaches are usually black-box models, lacking interpretability, and making users uncertain about the results. To address this, several interpretation approaches have been recently proposed, such as Drebin [4], LIME [41], LEMNA [19], and XMal [54], but they still possess limitations, as we highlighted in the introduction. Apart from the above work, some survey papers [18, 25] also conducted studies on interpretability. However, overall, they cannot accurately interpret the output of models at a fine-grained level in Android malware detection. To solve this problem, we propose *ProMal*, an innovative approach for profiling Android malware, utilizing the concept of *MBT* to automatically generate fine-grained malware descriptions for enhanced interpretability and comprehension. Meanwhile, we chose XMal according to the interpretable performance in the scenario of malware profiling [12] as the baseline method and thoroughly compared it in Section 7.1.3.

9.3 Android Description Generation

A series of efforts have been made to generate descriptions for Android apps. WHYPER [34] is the pioneer in utilizing **natural language processing (NLP)** technology to interpret permission requests. AutoCog [38] goes a step further by associating descriptions with permission lists using NLP and machine learning algorithms. DescribeMe [65] analyzed internal program logic and translated security-sensitive code patterns into natural language scripts. Wu et al. proposed prescription [56], considering user preferences, and focal points to craft distinctive descriptions tailored to various user types. DescribeCTX [62] introduced a context-aware description synthesis approach, addressing privacy concerns in apps.

Existing works, such as DescribeMe [65] and DescribeCTX [62], generated *sensitive* behavior descriptions based on code analysis by using all sensitive APIs that are used in the app. Their goal is to enhance users' understanding of sensitive behaviors across the entire app market. This leads to, in the context of malicious behavior analysis, existing works extracting sensitive behaviors as sequences of all sensitive APIs, resulting in a high rate of false positives for the scenario of malware, since sensitive behaviors clearly do not necessarily mean malicious behaviors. Moreover, DescribeMe focuses solely on data-dependency-related behavioral operations, failing to fully capture extensive malicious behaviors and thus limiting its descriptions. For example, in Wu et al.'s work, privilege escalation (e.g., gaining ROOT access on the targeted system, requesting user administration privileges) and hiding icons are considered malicious behaviors, which could not be described by DescribeMe. In contrast, our work is specifically focused on malware, concentrating on the extraction and description of *MBTs* that clearly indicate malicious behaviors in Android malware. Additionally, our approach goes beyond data dependency to capture a broader range of malicious behaviors, leveraging the comprehensive definition of *MBT*.

Additionally, existing approaches for extracting sensitive behaviors based on *fixed* sensitive APIs or key features, cannot adapt to new types of sensitive behaviors, especially as malware evolves over time by using similar functionalities but different APIs to evade detection. As a comparison, *ProMal* stands out, by utilizing a self-updating knowledge graph of malicious behaviors, allowing it to update and reason novel *MBTs* from extensive malware datasets. Moreover, *ProMal* can adapt to the continuous updates of API levels by dynamically expanding the range of sensitive APIs from official documents to capture evolving patterns of *MBTs*. Last but not least, *ProMal* leverages the versatile updating and reasoning capabilities of the knowledge graph, providing a substantial advantage in constructing new *MBTs* as well as human-readable descriptions demanding more precise and fine-grained information. This enhancement significantly contributes to the efficacy of malicious behavior profiling.

10 Conclusion

In this article, we introduced *ProMal*, an innovative approach to generating a human-readable malware description by profiling malicious behavior trajectories of malware based on a self-updated knowledge graph. The graph is scalable and can help provide fine-grained *MBTs* of malicious behaviors, laying the foundation for a more precise malware description. *ProMal* shows superior performance in understanding what and how malware accomplishes harmful tasks.

References

[1] List of Mobile Security Vendors. 2020. Retrieved from https://www.av-comparatives.org/list-of-mobile-security-vendors-android/

[2] Description Generation through Profiling Malicious Behavior Trajectory. 2024. Retrieved from https://github.com/ProMal4Android/ProMal4Android

[3] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in Android. In *9th International ICST Conference on Security and Privacy in Communication Networks (SecureComm '13)*. Springer, 86–103.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Network and Distributed System Security*, Vol. 14, 23–26.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Notices* 49, 6 (2014), 259–269.

[6] Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. Rotten apples spoil the bunch: An anatomy of Google Play malware. In *Proceedings of the 44th International Conference on Software Engineering*, 1919–1931.

[7] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Computers & Security* 73 (2018), 326–344. DOI: https://doi.org/10.1016/J.COSE.2017.11.007

[8] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. StormDroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS '16)*. ACM, 377–388. DOI: https://doi.org/10.1145/2897845.2897860

[9] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 239–252.

[10] Lisa Ehrlinger and Wolfram Wöß. 2016. Towards a definition of knowledge graphs. In *Proceedings of the International Conference on Semantic Systems 2016 (Posters, Demos, SuCCESS)*, 2.

[11] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 235–245.

[12] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. 2020. Can we trust your explanations? Sanity checks for interpreters in Android malware analysis. *IEEE Transactions on Information Forensics and Security* 16 (2020), 838–853.

[13] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. 2021. A performance-sensitive malware detection system using deep learning on mobile devices. *IEEE Transactions on Information Forensics and Security*. 16 (2021), 1563–1578. DOI: https://doi.org/10.1109/TIFS.2020.3025436

[14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 576–587.

[15] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. 2016. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *Proceedings of the 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.

[16] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of android applications in droidsafe. In *Proceedings of the Network and Distributed System Security*, Vol. 15, 110.

[17] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Dino Pedreschi, Franco Turini, and Fosca Giannotti. 2018. Local rule-based explanations of black box decision systems. arXiv:1805.10820. Retrieved from https://arxiv.org/abs/1805.10820

[18] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[19] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 364–379.

[20] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. 2009. *OWL 2 Web Ontology Language Primer*. W3C Recommendation.

[21] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. 2021. *Knowledge graphs*. Vol. 12. Morgan & Claypool, 1–242 pages.

[22] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 773–788.

[23] Deqiang Li and Qianmu Li. 2020. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security* 15 (2020), 3886–3900.

[24] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of 37th International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[25] Zachary C. Lipton. 2018. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue* 16, 3 (2018), 31–57.

[26] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Explainable AI for Android malware detection: Towards understanding why the models perform so well? In *Proceedings of the 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 169–180.

[27] Zhen Liu, Ruoyu Wang, Nathalie Japkowicz, Heitor Murilo Gomes, Bitao Peng, and Wenbin Zhang. 2024. SeGDroid: An android malware detection method based on sensitive function call graph learning. *Expert Systems with Applications* 235 (2024), 121125.

[28] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 4768–4777.

[29] Arvind Mahindru. 2023. Anndroid: a framework for android malware detection using feature selection techniques and machine learning algorithms. In *Mobile Application Development: Practice and Experience: 12th Industry Symposium in Conjunction with 18th ICDCIT '22*. Springer, 47–69.

[30] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep android malware detection. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, 301–308.

[31] Natasha Noy, Yuqing Gao, Anshu Jain, Annamalai Narayanan, Ankit Patterson, and Jamie Taylor. 2019. Industry-scale knowledge graphs: Lessons and challenges. *Commun. ACM* 62, 8 (2019), 36–43.

[32] OpenAI. 2024. ChatGPT (GPT-3.5 architecture). Retrieved from https://www.openai.com/chatgptLargelanguagemodel

[33] Fan Ou and Jian Xu. 2022. S3Feature: A static sensitive subgraph-based feature for android malware detection. *Computers & Security* 112 (2022), 102513.

[34] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Security Symposium*, 527–542.

[35] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web* 8, 3 (2017), 489–508.

[36] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the Network and Distributed System Security*, Vol. 14, 23–26.

[37] Qijing Qiao, Ruitao Feng, Sen Chen, Fei Zhang, and Xiaohong Li. 2024. Multi-label classification for android malware based on active learning. arXiv:2410.06444. Retrieved from https://arxiv.org/abs/2410.06444

[38] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 1354–1365.

[39] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 329–334.

[40] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144.

[41] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should I trust you? Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144.

[42] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-precision model-agnostic explanations.In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'18/IAAI'18/EAAI'18).* AAAI Press, Article 187, 1527–1535.

[43] Furqan Rustam, Imran Ashraf, Anca Delia Jurcut, Ali Kashif Bashir, and Yousaf Bin Zikria. 2023. Malware detection using image representation of malware data and transfer learning. *Journal of Parallel and Distributed Computing*. 172 (2023), 32–50.

[44] Durmuş Özkan Şahın, Sedat Akleylek, and Erdal Kiliç. 2022. LinRegDroid: Detection of Android malware using multiple linear regression models-based classifiers. *IEEE Access* 10 (2022), 14246–14259.

[45] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing* 15, 1 (2016), 83–97.

[46] SkiaBuild. 2024. Jadx: Dex to Java decompiler. Retrieved August 30, 2024 from https://github.com/skylot/jadx

[47] VirusTotal. 2019. The website of VirusTotal. Retrieved from https://www.virustotal.com/gui/home/upload

[48] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. 2022. MalRadar: Demystifying Android malware in the new era. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2, Article 40 (Jun 2022), 27 pages. DOI: https://doi.org/10.1145/3530906

[49] Weiping Wang, Jianjian Wei, Shigeng Zhang, and Xi Luo. 2019. LSCDroid: Malware detection based on local sensitive API invocation sequences. *IEEE Transactions on Reliability* 69, 1 (2019), 174–187.

[50] Zhaoguo Wang, Chenglong Li, Zhenlong Yuan, Yi Guan, and Yibo Xue. 2016. DroidChain: A novel Android malware detection method based on behavior chains. *Pervasive and Mobile Computing* 32 (2016), 3–14.

[51] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep ground truth analysis of current Android malware. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*. Springer, 252–276.

[52] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[54] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R. Lyu. 2021. Why an Android app is classified as malware: Toward malware classification interpretation. *ACM Transactions on Software Engineering and Methodology* 30, 2, Article 21 (Mar 2021), 29 pages. DOI: https://doi.org/10.1145/3423096

[55] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. 2016. Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology* 75 (2016), 17–25.

[56] Tingmin Wu, Lihong Tang, Rongjunchen Zhang, Sheng Wen, Cecile Paris, Surya Nepal, Marthie Grobler, and Yang Xiang. 2019. Catering to your concerns: automatic generation of personalised security-centric descriptions for Android apps. *ACM Transactions on Cyber-Physical Systems* 3, 4 (2019), 1–21.

[57] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 139–150.

[58] Tobias Wüchner, Aleksander Cisłak, Martin Ochoa, and Alexander Pretschner. 2017. Leveraging compression-based graph mining for behavior-based malware detection. *IEEE Transactions on Dependable and Secure Computing* 16, 1 (2017), 99–112.

[59] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. 2018. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 473–487.

[60] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. Iccbot: fragment-aware and context-sensitive icc resolution for android applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 105–109.

[61] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *19th European Symposium on Research in Computer Security (ESORICS '14)*. Springer, 163–182.

[62] Shao Yang, Yuehan Wang, Yuan Yao, Haoyu Wang, Yanfang Ye, and Xusheng Xiao. 2022. DescribeCtx: context-aware description synthesis for sensitive behaviors in mobile apps. In *Proceedings of the 44th International Conference on Software Engineering*, 685–697.

[63] Wei Yu, Linqiang Ge, Guobin Xu, and Xinwen Fu. 2014. Towards neural network based malware detection on android mobile devices. *Cybersecurity Systems for Human Cognition Augmentation* (2014), 99–117.

[64] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. 2016. Droid detector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology* 21, 1 (2016), 114–123.

[65] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards automatic generation of security-centric descriptions for Android apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 518–529.

[66] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE*, 95–109.

[67] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In *Proceedings of the Network and Distributed System Security*, Vol. 25, 50–52.

Received 15 June 2024; revised 24 December 2024; accepted 9 January 2025