Computers & Security 152 (2025) 104364

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

The silence of the phishers: Early-stage voice phishing detection with runtime permission requests

Chanjong Lee [a], Bedeuro Kim [b], Hyoungshick Kim [a,*]

[a] Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea
[b] Department of Electrical and Computer Engineering, Sungkyunkwan University, Republic of Korea

ARTICLE INFO

Keywords:
Voice phishing
Phishing detection
Mobile security

ABSTRACT

Voice phishing (vishing) is a sophisticated phone scam that causes significant financial harm to victims. Recently, vishing attacks have become more effective due to the use of vishing malware installed on victims' devices. Conventional anti-malware solutions, which rely on static analysis of app code and permissions at install time, are circumvented by vishing malware that requests additional code and permissions after installation. We introduce `VishielDroid`, a novel system for real-time detection of vishing malware on Android devices. By dynamically tracking apps' *runtime permission* requests, a critical indicator of malicious behavior specific to vishing malware, `VishielDroid` outperforms state-of-the-art systems in detection accuracy. Using only 98 features, `VishielDroid` achieved an F1-score of 99.78% with systematic testing, surpassing other solutions that achieve lower F1-scores (69.27% to 80.25%). The system demonstrated superior robustness across various scenarios: maintaining high performance with reduced training data and imbalanced datasets, achieving a 99.57% F1-score with a reduced feature set despite evasion attempts, and operating effectively across Android versions 8.1 to 12 with minimal modifications. We validated `VishielDroid`'s practicality through deployment on real devices, confirming marginal memory and battery consumption overheads.

1. Introduction

Voice phishing, often referred to as *vishing*, is a rapidly growing cybersecurity threat causing significant financial losses worldwide, particularly in East Asia (Bernama, 2022; Hao, 2023; Mayfield, 2023; Shuang, 2023; Ryall, 2021). In 2023, 56.2 million Americans fell victim to these scams, losing $25.4 billion (LaMont, 2024). Voice phishing scammers have enhanced their tactics by employing *vishing malware* — malicious apps that steal personal data and redirect calls, making it appear as if victims are communicating with legitimate entities. This deception is facilitated by manipulating caller display IDs (Song et al., 2014; Sahin et al., 2017; Pandit et al., 2018).

Recently, Kim et al. (2022) introduced HearMeOut, a system designed to identify the malicious activities of vishing malware at runtime, such as call redirection and fake call voices. However, our experimental results indicate that HearMeOut has limitations in its capability, as it only detects certain sequences of system activities. This limitation leads to the failure to recognize many instances of vishing malware. Another major drawback of HearMeOut is its detection of attacks only at the moment of making a phone call, posing a risk of actual voice phishing attacks occurring. Additionally, HearMeOut requires modifying each function of interest at the Android operating system level to capture different information obtained via each API invocation, necessitating continual rule updates to detect new attack patterns. While governments and smartphone manufacturers are exploring machine learning-based solutions (Anon, 2023b), these approaches face challenges in early prevention and raise privacy concerns.

To address these drawbacks, we propose `VishielDroid`, an Android tool that detects vishing malware by analyzing runtime behaviors, addressing previous drawbacks. `VishielDroid` comprises: (1) a system service monitoring app behavior and extracting detailed features from permission requests, and (2) a system app-implemented classifier determining vishing malware based on these features. Unlike previous studies (Kouliaridis et al., 2020; Li et al., 2018; Onwuzurike et al., 2019; Cai et al., 2018; Li et al., 2021) analyzing permissions at install time, our approach uniquely incorporates *runtime permission* information, introduced in Android 6.0 (API level 23) (Android Developers, 2024c).

`VishielDroid`'s system service tracks these runtime permissions, allowing users to grant or revoke specific permissions while apps are running. Since vishing malware typically requires the execution

* Corresponding author.
E-mail address: hyoung@skku.edu (H. Kim).

https://doi.org/10.1016/j.cose.2025.104364
Received 18 July 2024; Received in revised form 6 December 2024; Accepted 4 February 2025
Available online 14 February 2025

of protected operations, it frequently requests runtime permissions. VishielDroid leverages this characteristic by tracking these real-time permission requests and using them as features to detect the malicious behavior of vishing malware.

Previous approaches significantly overlooked runtime permission requests, a crucial aspect of vishing malware identified by VishielDroid. Vishing malware often demands excessive runtime permissions (*e.g.*, PROCESS_OUTGOING_CALLS, WRITE_CONTACTS, and RECORD_AUDIO) to control call redirection, access user data, or record audio/video for spying after installation. By registering a system service to trace runtime permission requests, VishielDroid observes voice phishing apps' most characteristic runtime behavior. Unlike HearMeOut, which detects vishing malware during actual voice phishing attacks, VishielDroid detects suspicious vishing malware before the execution of attacks, enabling more effective user warnings.

Our key contributions are as follows:

- **Highly accurate, lightweight vishing detection:** We have designed and implemented VishielDroid, a lightweight system for the real-time, on-device detection of vishing malware. The source code is publicly available at https://github.com/Talleyrand323/VishielDroid. VishielDroid's efficient design makes it suitable for practical deployment on real-world Android smartphones. We have successfully demonstrated its practicality by deploying VishielDroid on a Pixel 2 smartphone, demonstrating its ability to operate efficiently. A demo video featuring VishielDroid's real-world performance is available at https://youtu.be/WUO5dBV-AuM.
- **Superior performance over state-of-the-art solutions:** We evaluated the performance of VishielDroid against three prominent existing methods, and the results demonstrate its superiority in both detection accuracy and efficiency. MalScan (Wu et al., 2019) achieved an F1-score of 80.25% while requiring 21,986 features, whereas VishielDroid achieved a significantly higher F1-score of 99.78% with only 98 features. Similarly, Li et al.'s method (Kim et al., 2022) achieved an F1-score of 69.27% with 379 features, and Singh et al.'s method (Singh et al., 2024) achieved an F1-score of 78.89% using 146 features, both of which were substantially outperformed by VishielDroid. Additionally, VishielDroid demonstrated its exceptional capability for early detection by successfully identifying all vishing malware samples before any phone calls were initiated.
- **Robustness under diverse conditions:** We evaluated the robustness of VishielDroid under several practical challenges, including severe class imbalance (fewer vishing samples than benign ones) and adaptive attacks. VishielDroid demonstrated strong robustness in these scenarios, showing nearly similar detection performance even when trained with half the size of the original set. It exhibited minimal performance degradation, with the F1-score dropping to 97.78% in situations where the number of benign samples was 20 times that of vishing samples. Additionally, we assessed VishielDroid's resilience against evasion attacks that manipulate permission requests. While the detection performance dropped by 9.57% for adversarial instances, using a reduced feature set consisting of permissions frequently appearing in vishing malware allowed VishielDroid to maintain a high F1-score of 99.57%, demonstrating its robustness.

2. Attack model and requirements

In this section, we introduce the attack model for vishing malware attacks. We then list the crucial criteria that tools developed to detect vishing malware must meet to effectively mitigate this growing threat.

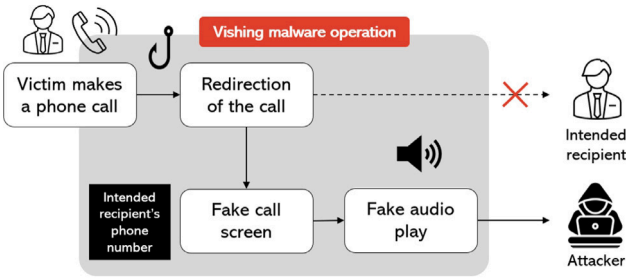


Fig. 1. When a victim makes a call, vishing malware can redirect it to the attacker instead of the intended recipient, like a bank. Despite being connected to the attacker, the victim's phone will still display the original number, concealing the redirection. Additionally, the malware may use pre-recorded fake audio to deceive the victim, imitating a bank's call voice. This misleads the victim into thinking they are speaking with the intended party, significantly increasing the success of a voice phishing attack.

2.1. Attack model

The profitability of voice phishing continues to drive attackers to refine their techniques. A notable recent trend involves the use of social engineering tactics to install malicious Android apps on victims' devices. These apps grant attackers control over the devices, enabling them to manipulate caller display IDs and more effectively deceive users. This evolution in vishing methods has been demonstrated to be highly effective in enhancing the success rate of such attacks.

In our attack model, we assume the attacker aims to deceive victims into divulging sensitive information or transferring funds by impersonating legitimate entities during phone calls. The attacker can create malicious Android apps that mimic legitimate ones and distribute them through various channels, such as SMS phishing or third-party app stores. These apps request extensive permissions and manipulate the device's call functionality to facilitate the attacker's deception.

Vishing malware often masquerades as legitimate applications, such as those related to banking, shopping, shipping, or government services. These apps facilitate deception by hijacking incoming and outgoing phone calls on the infected device. As illustrated in Fig. 1, attackers deceive victims by posing as bankers or government officials, driving them to reveal private information (*e.g.*, bank account passwords) or transfer money to the attackers' accounts.

Furthermore, vishing malware typically compromises victims' personal data by eavesdropping on phone calls and SMS messages and accessing the camera to record and monitor victims' behaviors (Liu et al., 2023). These apps alter the displayed caller ID to deceive victims and execute attacks by stealing personal information such as phone book entries, messages, and account details (Maggi, 2010). They disguise themselves as legitimate and employ various methods to reach their victims. For example, an SMS might falsely claim that a parcel or gift has arrived or that a bank has introduced a lower-interest loan service, providing a URL that installs vishing malware. Users are then lured into clicking on the URL and installing the downloaded app (Nahapetyan et al., 2024). Vishing malware not only exploits stolen personal information but can also execute unauthorized actions by launching apps on the victim's device, such as illicitly transferring funds to attackers using banking apps installed on the victim's device.

Although malicious apps can infiltrate legitimate app stores like the Google Play Store, our research focuses on apps from sources outside these official platforms. The Google Play Store typically enforces stricter security measures and vetting processes, making it difficult for overtly malicious apps to be published. As a result, attackers frequently use social engineering tactics to convince users to download vishing malware APKs via URLs, bypassing the controls of legitimate app stores.

Vishing malware operations typically consist of the following three phases:

Fig. 2. Five app icon examples show the visual similarity between benign apps and vishing malware samples.

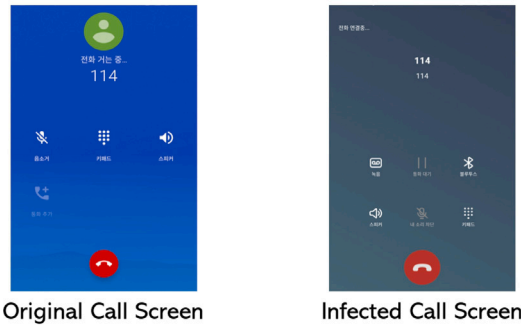


Fig. 3. Visual comparison of Google Pixel 2's call screen before and after vishing malware infection. The malware sample has replaced the default dialer app, tricking victim users into believing they are placing legitimate calls.

- **Phase 1. Installation through an illegitimate route:** Vishing malware is often installed via a URL in a fake SMS. It mimics benign apps from the outset, inducing victims to engage with the app by clicking its icon and advancing to the next phase (see Fig. 2). Vishing malware detects if the target device is in a virtual machine environment or if a debugger or runtime manipulation tool such as Frida (Anon, 2023a) is used. In such cases, it halts execution, making runtime behavior analysis challenging.
- **Phase 2. Privilege escalation by requesting permissions:** The malware then seeks to escalate privileges by requesting numerous permissions (*e.g.*, access to phone calls, contacts, location, SMS, camera, microphone) and settings during runtime. It also requests device configuration setup and permissions (*e.g.*, setting the app as the default dialer, allowing installation from unknown sources, enabling accessibility service) to gain sufficient control for espionage or device manipulation. The installation of nested APK files under the guise of a 'security update' is often observed during this phase. These files are used to gain access to various permissions while maintaining a low profile by not displaying app icons after installation.
- **Phase 3. Deceitful operations during phone calls:** This phase occurs when the victim makes an outgoing call with the infected device. The call screen often changes from the original one after infection. To accomplish this, vishing malware changes the device's 'default dialer app' to a fake dialer app or uses a screen overlay feature to cover the screen with a fake dialer app, hiding that the call is being directed to the attacker. Fig. 3 shows how the attacker's dialer app has a different outgoing call UI from the original dialer app on an infected device. To enable this, the malware requests to change the default dialer app or enable the accessibility service during Phase 2.

2.2. Key requirements for vishing detectors

It is crucial to establish clear system requirements to develop effective machine learning-based solutions for detecting vishing malware. Our approach integrates insights from previous work on Android malware detection (Kim et al., 2022; Roy et al., 2015; Naqvi et al., 2023) while also considering the unique characteristics of vishing malware. This comprehensive perspective allows us to derive key system requirements that address both general malware detection principles and the specific challenges posed by vishing attacks.

- **Early-Stage Detection.** HearMeOut (Kim et al., 2022) was designed to detect call redirection to an attacker's phone number during phone calls. However, its effectiveness is primarily limited to scenarios that occur after a call has been initiated, leaving users vulnerable to attacks that may compromise their data or device control before any call redirection activity takes place. Furthermore, even if HearMeOut detects vishing attacks in real-time, notifying users during an ongoing call may prove challenging, as the user is already engaged in a conversation and may be subject to ongoing deception. To address these limitations and provide more comprehensive protection, our objective is to detect vishing malware at the earliest possible stage, ideally before any attack is executed. As discussed in Section 2.1, our specific goal for early-stage detection is to identify potential threats during Phase 1 or 2, prior to the execution of an actual attack.
- **Effectiveness Against Unseen Samples.** As malware developers generate new samples by analyzing existing machine-learning solutions, maintaining high accuracy against such unseen samples is crucial. Models trained with past samples must preserve high detection accuracy for newly emerging, unseen samples over time.
- **High Accuracy Model with Imbalanced Dataset.** In Android malware detection using machine learning, the number of benign samples usually exceeds malware samples (Roy et al., 2015). This also applies to vishing malware samples. Ensuring high accuracy in cases of imbalanced classes is crucial due to significantly fewer malware samples than benign ones.
- **On-Device Detection.** While numerous methods (*e.g.*, (Bai et al., 2021; Onwuzurike et al., 2019; Yang et al., 2021)) exist for detecting or analyzing Android malware, many focus on analyzing malicious APK files on an external device rather than the actual victim smartphone, often requiring additional tools for feature extraction. This approach is less feasible as it involves uploading APK files to an external device, such as a cloud server, for installation and analysis, thus reducing its practicality.
- **Robustness Against Anti-Analysis Attacks.** Vishing malware has become more sophisticated, making it harder to detect. These apps often use techniques to bypass analysis frameworks and can even use tools to delete themselves to avoid detection (Naqvi et al., 2023). Therefore, it is essential to develop and implement detection solutions that are specifically designed to combat the anti-analysis strategies used by vishing malware.

In this work, we aim to develop a practical machine learning-based detection solution that meets these system requirements.

3. Understanding key characteristics of vishing malware

We thoroughly analyze samples of vishing malware to understand its fundamental characteristics. We specifically analyze the permissions requested by vishing malware because the permissions are associated with the actions that vishing malware intends to perform. Research on detecting malware using permissions has already been extensively conducted. However, unlike previous research that focused on permissions at install time, we also consider the permissions requested at runtime when the app is executed. This is because vishing malware typically acquires various permissions after installation and during execution.

Table 1
Overview of vishing malware and benign datasets used in this study, categorized by year. The number inside parentheses represents the count of distinct families.

Class	2021	2022	2023	Total
Benign	392	381	475	1,248
Vishing malware	356 (8)	27 (2)	230 (3)	613 (10)

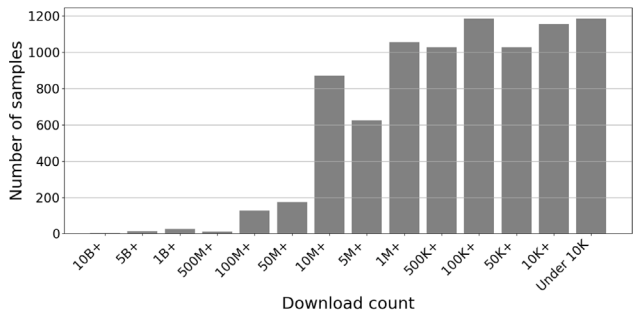


Fig. 4. Distribution of benign apps by download count.

3.1. Dataset

In our study, we collected APK files from both vishing and benign categories, spanning from 2021 to 2023. Table 1 summarizes the dataset, organized by year. We utilized the datasets from 2021 and 2022 to identify features and train the vishing malware detection model, whereas the 2023 dataset was exclusively used for testing. This method aligns with the requirement for **Effectiveness Against Unseen Samples** presented in Section 2.2. Below is the process we followed to gather the vishing and benign samples:

Vishing Malware Dataset. Our study analyzed 4803 expert-verified vishing malware samples: 4538 from a financial security institution in 2021 and 265 from a cyber threat management company (30 in 2022, 235 in 2023). To manage time constraints in dynamic analysis while addressing the disproportionate yearly distribution (4538 in 2021, 30 in 2022, 235 in 2023), we randomly selected 535 samples from 2021 and included all samples from 2022 and 2023. This approach ensured comprehensive representation, particularly of recent threats.

To further analyze the vishing malware samples, we utilized AVClass2 (Sebastián and Caballero, 2020) to extract family information from the VirusTotal reports. The results showed that the most frequently appearing label among the 613 samples was "fakecalls" (185 samples), followed by "fakenocam" (94 samples) and "wroba" (79 samples).

Notably, 98 samples (42.61%) out of 230 from the 2023 dataset were categorized as malware samples that were not present in the malware families found in the 2021 and 2022 datasets. This finding indicates that new vishing malware variants are emerging over time, posing increasing challenges for their identification.

Benign Apps. We initially considered 50,000 APKs of Google Play apps, available from 2021 to 2023, randomly selected from the `AndroZoo` dataset (Allix et al., 2016). We constrained their size to under 50MB to align with vishing samples, as the maximum size of vishing malware samples in our collected dataset is 41.29MB (mean: 11.62MB, standard deviation: 0.12MB). To verify that these apps were free from malicious software, they were scanned using `VirusTotal` (VirusTotal, 2023). Additionally, we considered the number of downloads for the benign apps to ensure a diverse range of popularity levels. Among the apps with available download information, we made a balanced selection of 8487 benign apps across various categories, defined by their number of downloads, as detailed in Fig. 4.

To determine the Android platform for `VishielDroid`, we utilized `Androguard` (Androguard, 2024) to analyze the minimum and target SDK versions used in the benign and vishing malware samples we collected. In Android, the minimum SDK version is the earliest version of the Android platform on which the app can run, while the target SDK version is the version for which the app is mainly developed and tested. While `Androguard` could analyze all benign apps except for just 1 sample, it failed to analyze many vishing malware samples — Only 2025 out of 4803 vishing malware samples were analyzed. Fig. 5 illustrates the distribution of both benign apps and vishing malware samples' minimum and target SDK versions.

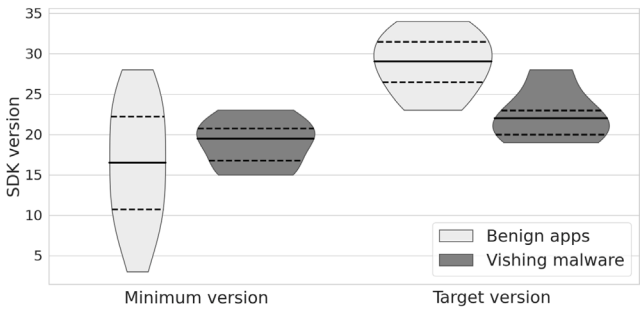


Fig. 5. SDK versions for benign and vishing malware. In each violin plot, the solid line in the center represents the median of the data. The upper and lower dotted lines of the violin indicate the third quartile (Q3) and the first quartile (Q1), respectively, signifying that the middle 50% of the data is contained within this range.

The benign samples exhibited a broader range in the minimum SDK version and a relatively higher range in the target SDK version compared to the vishing malware samples. Notably, all vishing malware samples targeted SDK levels no higher than 28 (Android 9), with the third quartile (Q3) of their target SDK level at 23 (Android 6). This trend reveals two key insights. First, vishing malware predominantly targets lower SDK versions, enabling attacks on a diverse array of devices, including older models. Second, attackers often deliberately choose lower SDK versions to circumvent the security mechanisms in newer, higher SDK versions. In light of this trend and to ensure a fair comparison with HearMeOut, which operates on Android 8.1, we selected Android 8.1 (SDK level of 27) as the implementation platform for `VishielDroid`.

3.2. Analysis of permissions requested

We analyzed the official Android Manifest information and identified 144 types of permissions that could be supported in SDK level 27 or lower, suitable for Android 8.1. Next, we excluded permissions with the protection level of signature or privileged (as they are intended for system apps, not for user-level apps where vishing malware is typically installed), and permissions common across all Android apps, ultimately selecting 123 permissions as important to distinguish vishing malware. To understand how these 123 permissions are utilized in vishing malware and benign apps, we analyzed the samples collected in 2021 and 2022. We used the 2023 samples solely for testing as unseen apps.

3.2.1. Install vs. Runtime permissions

Android's permission model significantly changed with the introduction of runtime permissions in Android 6.0 (API level 23) (Android Developers, 2024c), affecting how apps request and obtain permissions, especially "dangerous" ones. The two approaches are:

- **Install time permissions** ($\mathbf{P_{install}}$): Declared in `AndroidManifest.xml` and granted upon installation. Before Android 6.0, all permissions, including dangerous ones, were granted at this stage, requiring users to accept all permissions during installation.

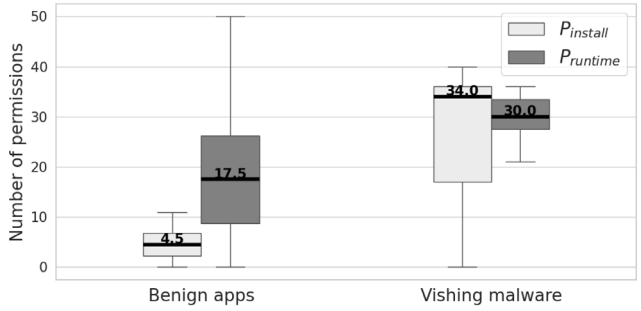


Fig. 6. Comparative distribution of permission counts in benign apps vs. vishing malware.

- **Runtime permissions** ($\mathbf{P_{runtime}}$): Introduced in Android 6.0, this model requires apps to request certain permissions during execution via user prompts. It applies to "dangerous" permissions (*e.g.*, `READ_PHONE_NUMBERS`, `READ_CONTACTS`, `WRITE_EXTERNAL_STORAGE`). Runtime permissions offer granular control, allowing users to grant, deny, or revoke specific permissions even post-installation. Additionally, some apps exploit nested APK installations to request additional permissions during runtime. In our work, we also categorize these nested APK-based permission requests as runtime permissions, recognizing their dynamic nature and potential security implications.

Our analysis of 2022 vishing samples showed that 40% relied on nested APK installations during execution to obtain further permissions at runtime, demonstrating how malicious apps adapt to the new model to gain permissions dynamically.

3.2.2. Permission usage patterns between vishing malware and benign applications

We delve into the distinctive permission usage patterns of vishing malware compared to benign applications, highlighting the crucial role of request timing in detection. Our analysis reveals a stark contrast in permission requests. As shown in Fig. 6, vishing malware samples overall request more permissions than benign apps, both during installation and at runtime. Interestingly, in terms of $\mathbf{P_{install}}$ and $\mathbf{P_{runtime}}$, benign apps and vishing malware exhibit opposite trends. For $\mathbf{P_{install}}$, the interquartile range (Q3-Q1) for benign apps is significantly smaller than that for vishing malware. However, for $\mathbf{P_{runtime}}$, the trend is completely reversed. This indicates that while there are many vishing malware samples, they consistently request a specific set of $\mathbf{P_{runtime}}$ permissions.

We specifically observed a suspicious behavior unique to 2022 vishing samples: the use of nested APK installation. This involved requesting runtime permissions ($\mathbf{P_{runtime}}$) while simultaneously declaring none in the manifest ($\mathbf{P_{install}}$), deviating from typical benign app permission usage. This anomaly suggests an attempt to mask the malicious intent hidden within nested APKs.

Considering the different patterns in permission requests between vishing malware samples and benign apps for $\mathbf{P_{runtime}}$ and $\mathbf{P_{install}}$, we need to consider the 123 permissions we selected, divided into installation time and runtime, totaling 246 combinations as potential candidate features for vishing malware detection.

3.3. Feature selection for `VishielDroid`

Starting with the initial feature set defined above, we excluded permissions not observed in our inspection of 1156 samples, leaving 90 features from $\mathbf{P_{install}}$ and 52 from $\mathbf{P_{runtime}}$. To identify the most relevant permissions for vishing detection, we calculated the point-biserial correlation (r_{pb}) (Kornbrot, 2014) between each permission feature and the class labels (vishing malware: 1, benign: 0). The point-biserial correlation is formalized as:

$$r_{pb} = \frac{\overline{X_1} - \overline{X_0}}{s_X} \cdot \sqrt{\frac{n_1 \cdot n_0}{n^2}} \tag{1}$$

Where $\overline{X_1}$ and $\overline{X_0}$ are the means of the feature values for vishing malware and benign apps, respectively, s_X is the standard deviation of all feature values, n_1 and n_0 are the numbers of vishing malware and benign apps, and n is the total number of samples. The resulting correlation coefficient ranges from -1 to 1, where a value closer to 1 indicates that permission is strongly associated with vishing malware. In contrast, values closer to -1 suggest an association with benign apps.

While other correlation methods like Spearman's rank correlation or Yule's coefficient could be used, we chose the point-biserial correlation for several reasons. First, unlike Spearman's correlation, which assumes ordinal data, point-biserial correlation is specifically designed for dichotomous variables (our permission features are binary: present or absent). Second, compared to Yule's coefficient, which only considers presence/absence patterns, point-biserial correlation incorporates the relative frequencies of permissions in each class through the mean and standard deviation terms, providing a more nuanced measure of association strength. This is particularly important for distinguishing permissions that may appear in both benign and malicious apps but with different frequencies.

We excluded permissions with a p-value over 0.1, resulting in a final set of 98 permissions, consisting of 60 $\mathbf{P_{install}}$ and 38 $\mathbf{P_{runtime}}$. Appendix presents the details of those permissions.

Fig. 7 presents those 98 permissions sorted by correlation coefficients (in decreasing order). Overall, $\mathbf{P_{install}}$ permissions have higher correlation coefficients compared to $\mathbf{P_{runtime}}$ permissions. However, our ablation study results in Section 5.6 interestingly show that $\mathbf{P_{runtime}}$ permissions play a crucial role in detecting vishing malware compared to $\mathbf{P_{install}}$ permissions. This is because, while vishing malware samples in 2021 relatively used fewer $\mathbf{P_{runtime}}$ permissions, the frequency of using $\mathbf{P_{runtime}}$ permissions significantly increased in the 2022 samples. Moreover, for $\mathbf{P_{runtime}}$ permissions, the combination of several permissions appears to be more important than the significance of a single permission.

Phone-related permissions, such as `WRITE_CALL_LOG` (P1), exhibited high correlations with vishing malware as anticipated. Notably, `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` (P23) stood out among runtime features, allowing vishing malware to operate continuously in the background of a victim's device.

Permissions with the 'Normal' protection level, including `BROADCAST_STICKY` (P36) and `CHANGE_NETWORK_STATE` (P37), which do not require runtime requests (except for P23) but were detected during runtime, contributed to vishing identification.

For the benign category, permissions like `WRITE_SETTINGS` (P75), `READ_CALENDAR` (P91), and `SET_WALLPAPER` (P93), related to more common functionalities, were noted.

Unlike vishing malware, benign apps showed a low correlation with permissions commonly associated with vishing activities. These critical permissions include `READ_SMS` (P3) and `READ_CONTACTS` (P11), which are often used to access private information, and `PROCESS_OUTGOING_CALLS` (P2), essential for call-redirection deception. The distinct difference in permission usage patterns was crucial in distinguishing between vishing malware and benign apps.

Permissions such as `VIBRATE` (P55, P84) and `WRITE_EXTERNAL_STORAGE` (P26, P97) exhibited coefficient values that varied between positive and negative according to the timing of requests, highlighting the nuanced role of request timing in distinguishing between vishing and benign intents.

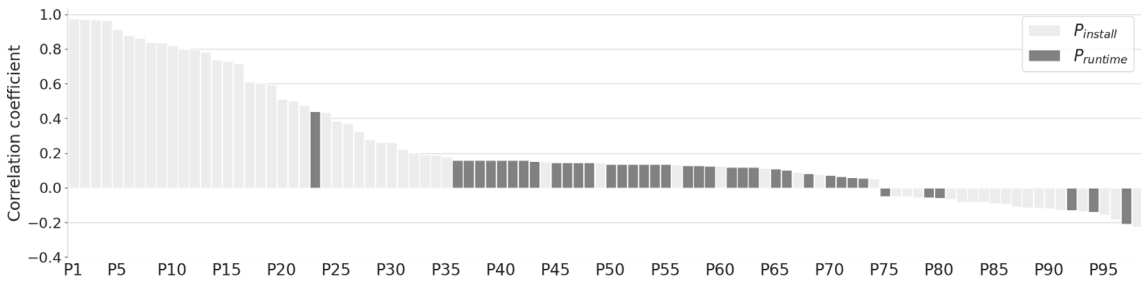


Fig. 7. 98 permissions sorted by correlation coefficients (in decreasing order). The bars in gray/black color denote that each permission was requested at the timing of installation/runtime, respectively (see the details of the permissions in Appendix).

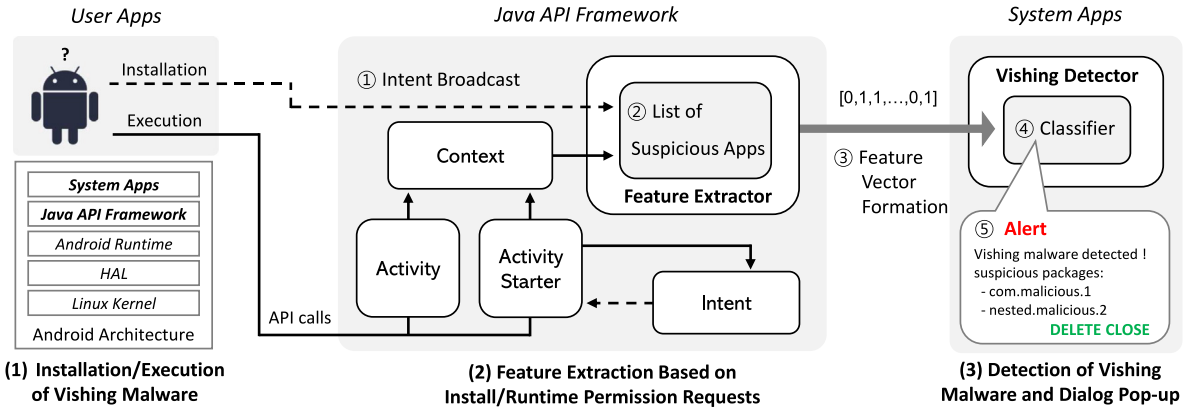


Fig. 8. Overall system architecture of `VishielDroid`.

4. Overview of `VishielDroid`

Static analysis of `AndroidManifest.xml` reveals permissions requested by an app at install time. However, this approach cannot capture runtime permissions that are dynamically requested while the app is running, which are crucial indicators of vishing behavior. While analyzing API calls like `Activity.requestPermissions()` could help detect such runtime behaviors, vishing malware commonly employs code obfuscation techniques such as packing and encryption to evade static analysis. Dynamic analysis can effectively overcome these limitations by monitoring actual permission requests during execution. Additionally, since vishing malware typically launches its attack while users are actively using their phones, our detection system must operate in real-time on the device without relying on external analysis tools or post-execution analysis.

To address these challenges, we introduce `VishielDroid`, a custom Android system engineered to detect vishing activities in real-time and alert users through system-level notifications. By tailoring Android at the operating system level, our approach ensures that `VishielDroid` effectively captures and analyzes the operations and characteristics of targeted apps, providing a robust defense against vishing threats.

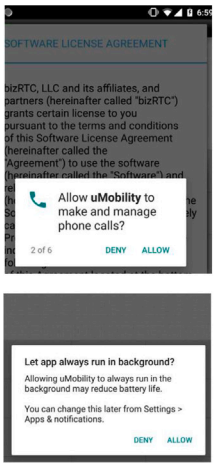
`VishielDroid` consists of two core components: `Vishing Detector` and `Feature Extractor`. As illustrated in Fig. 8, these components function as a system app and processes within the Android framework to effectively carry out the vishing malware detection process. The functions of each component are as follows.

`Vishing Detector` is implemented as a *system app* with a pre-trained classification model to assess each feature vector sent by `Feature Extractor`, determining whether it is generated by vishing malware. When identified as 'Malicious,' `Vishing Detector` alerts the victim user with a warning dialog, providing the option to remove the suspicious app.

`Feature Extractor` operates as a *system service* in the Java API Framework layer, with modifications to key Android framework classes including `Context`, `Intent`, `ActivityStarter`, and `Activity`. It monitors suspicious apps' activities and extracts install-time ($\mathbf{P_{install}}$) and runtime ($\mathbf{P_{runtime}}$) permission request patterns, tracking 60 install-time and 38 runtime permissions. For each permission, the extractor records whether a request occurred (1) or not (0), generating a binary feature vector that captures the app's permission request behavior.

Both components of VishelDroid are automatically initiated when the device is booted. The detection process begins with the installation of a new APK file on the victim's device, triggering a broadcast of an Intent with the `ACTION_PACKAGE_ADDED` action by the Android system. To efficiently manage resources, the focus is on suspiciously installed apps and analyzing their runtime permissions.

① The `Feature Extractor` checks the `installerName` from the Intent to determine if the installation source is a known app store, such as Google Play. If the app is not installed from a known app store (*e.g.*, via clicking a URL in a suspicious SMS), the `packageName` is added to the 'list of suspicious apps.' Install time permission requests are also checked here utilizing the `PackageManager`. ② The system logs runtime permission requests only if the apps on the list request permissions of interest and invoke modified APIs in the `Activity` and `ActivityStarter` classes. These modified APIs consequently invoke defined APIs inside the `Context` class, which in turn calls the APIs inside the `Feature Extractor` class. Additionally, the `ActivityStarter` class obtains information about permission requests by monitoring Intents from the `Intent` class. Through these processes, the requested permissions information, along with the context information, including the `packageName` of the requesting app, is transferred as function parameters. ③ A feature vector is generated from the gathered permissions data whenever a runtime permission request from an app monitored by VishelDroid is made. The feature vector is then sent to the `Vishing Detector`, which is implemented as a system app. ④ The `Vishing Detector` uses its machine learning-based classifier to evaluate the feature vector for vishing malware activity. ⑤ If the feature vector is classified as 'Malicious,' an alert is immediately triggered for the user, displaying the `packageName` associated with




```
1  // Inside of Activity.requestPermissions(@NonNull String[] permissions, ...)
2
3  try{
4      if (permissions.length > 0){
5          myContext = this.getApplicationContext();
6          myPackageName = this.getPackageName();
7          myContext.sendListInformation(myPackageName,
8                                        permissions, "runtime", myContext);
9      }
```

(a) Runtime request of multiple permissions including CALL_PHONE

```
1  // Inside of ActivityStarter.startActivity(..., Intent intent, ...)
2
3  String myAction = intent.getMyAction();
4
5  if ("android.settings.action.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"
6                                                          .equals(myAction)){
7      try{
8          String whoDidMakeRequest = intent.getMyData();
9          myContext.sendStringInformation(whoDidMakeRequest,
10                                         "batterybackground", myContext);
11     }
```

(b) Runtime request of permission related to battery optimization

Fig. 9. Example of runtime permission requests by vishing malware samples and VishielDroid's interception mechanism at the code level. The left side displays screenshots of permission request dialogs users encounter. The right side shows code snippets demonstrating VishielDroid's methodology for managing and responding to each permission request.

the feature vector and providing options to stop the app's execution and remove it from the device. This ensures that users are promptly notified of potential threats and empowered to take swift action to mitigate risks associated with detected malicious applications.

Fig. 9 demonstrates VishelDroid's approach to capturing runtime permission requests. For permissions with "dangerous" protection levels that require general form prompts, as shown in Fig. 9(a), the `Activity.requestPermissions()` method was modified to capture the runtime requests and convey the list of requested permissions to the `Feature Extractor` (Lines 7–8). For permissions requiring special prompts to the user by starting an activity using an Intent with `Settings.ACTION_*` actions, as shown in Fig. 9(b), a custom function (`getMyAction()`) inside the `Intent` class and modifications to `ActivityStarter.startActivity()` were added. These modifications check if the Intent action corresponds to those of interest (Line 3) and forward the information to the `Feature Extractor` through the `Context` class (Lines 9–10). A similar method is used to identify the runtime requests for `WRITE_SETTINGS` and `SYSTEM_ALERT_WINDOW` permissions.

These modifications enable `VishielDroid` to capture events when an app requests additional permissions while running, identifying vishing malware that may initially seem benign but later attempts to gain more access to the device's features or user data.

5. Evaluation

In this section, we evaluate the performance of `VishielDroid`, focusing on its detection accuracy. Our evaluation aims to provide a comprehensive understanding of `VishielDroid`'s effectiveness and efficiency in detecting vishing malware on real-world devices.

We begin by describing the experimental setup for our evaluation and then optimize the model for `VishielDroid` by considering the detection accuracy, training time, and testing time to assess its practicality for real-time detection. We compare `VishielDroid`'s performance against four state-of-the-art Android malware detection techniques: HearMeOut (Kim et al., 2022), a vishing malware detection solution; MalScan (Wu et al., 2019), a graph-based malware detection tool; Li et al. (2021), a deep learning-based malware detection tool; and Singh et al. (2024), a dynamic analysis-based detection approach using system call patterns.

To evaluate `VishielDroid`'s performance under realistic dataset configurations, we analyze how its detection accuracy varies when the number of vishing malware samples becomes extremely small compared to the number of benign samples. Additionally, we investigate `VishielDroid`'s resilience against adversarial attacks by creating vishing malware samples that maliciously add permissions similar to benign apps, aiming to evade detection. Furthermore, we conduct an ablation study to assess the individual contributions of each type of permission request ($\mathbf{P}_{\mathbf{install}}$ and $\mathbf{P}_{\mathbf{runtime}}$) to the overall effectiveness of our defense mechanism. Finally, to ensure `VishielDroid`'s practical deployability on Android devices without significantly impacting user experience or device performance, we measure the memory usage and battery consumption overheads associated with running `VishielDroid` on real devices.

5.1. Experimental setup

We executed each APK file on a Pixel 2 device equipped with `VishielDroid` using two different approaches: systematic manual execution and automated testing. For manual execution, we followed a structured interaction protocol: We interacted with each app for approximately 90 s, performing at least ten touch events to simulate typical user behavior. We systematically explored the app's interface by interacting with all visible UI elements, such as buttons, text fields, and menu items. When permission request dialogs appeared during runtime, we consistently granted them to observe the app's full permission request patterns.

For automated testing and fair comparison with dynamic analysis baselines like Singh et al. (2024), we also conducted experiments using Monkey (Android Developers, 2024a) to generate random user interactions for 2 min per app. While this automated approach enables reproducible testing, it may not trigger all permission-related behaviors that our manual testing protocol can reveal.

We used samples from 2021 and 2022 as our training set (773 benign and 383 vishing malware samples) and those from 2023 as our test set (475 benign and 230 vishing malware samples). For training and testing classifiers in `VishielDroid`, we used the last feature vector of each sample created by `Feature Extractor`. All APK files were installed via the Android Debug Bridge (ADB) tool.

5.2. Selecting the optimal classifier

To identify the most effective classifier for `VishielDroid`, we evaluated six commonly used machine learning models: XGBoost, LightGBM (LGBM), Random Forest (RF), Support Vector Machine (SVM) with a linear kernel, Logistic Regression (LR), and k-Nearest Neighbors (KNN). We excluded deep learning models due to their GPU-intensive

Table 2
Detection accuracy metrics of VishielDroid across six models.

Measure	XGBoost	LGBM	RF	SVM	LR	KNN
Precision (%)	0.00	98.11	91.79	**100.00**	**100.00**	99.57
Recall (%)	0.00	67.83	86.58	**99.57**	**99.57**	**99.57**
Accuracy (%)	66.95	89.08	95.55	**99.86**	**99.86**	99.72
F1-score (%)	0.00	80.21	88.90	**99.78**	**99.78**	99.57

Table 3
Memory usage of VishielDroid across six models.

Measure	XGBoost	LGBM	RF	SVM	LR	KNN
Memory size (KB)	4.84	8.15	8.99	**2.93**	3.34	446.61

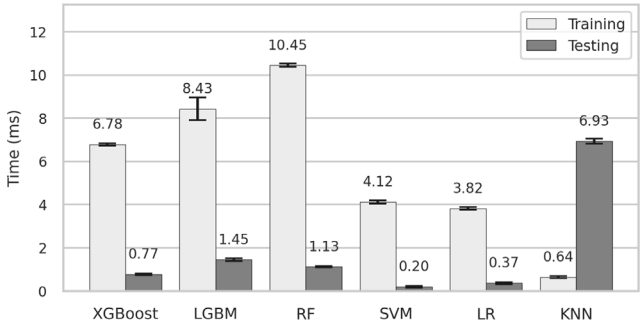


Fig. 10. Average training/testing times of VishielDroid across six models, computed over a hundred trials for each model.

operations, aiming for VishielDroid to function on low-spec devices without GPUs.

To determine the optimal hyperparameters for each model, we divided the training set into training and validation sets using a 1:1 ratio and evaluated their performance. The best-performing hyperparameters were identified for each model; notably, $k = 5$ was optimal for KNN.

We compared the performance of these optimized models, testing their effectiveness on the 2023 samples as unseen data. Table 2 summarizes the detection accuracy results for the six models. SVM and LR produced the highest F1-scores (99.78%). KNN also achieved a similarly high F1-score (99.57%). However, LGBM and XGBoost showed inadequate performance. While LGBM had a sufficient precision (98.11%), its recall (67.83%) was significantly lower. XGBoost yielded the worst results, failing to detect any vishing malware and misclassifying most as benign apps. The imbalance in the dataset (with far fewer malicious app samples than benign ones) and the significantly different characteristics of the 2023 data compared to the 2021–2022 dataset may explain the underperformance, potentially leading to overfitting in models like LGBM and XGBoost.

Following the detection accuracy evaluation, we measured training/testing times and memory usage for each model, repeating experiments 100 times for reliability. Fig. 10 illustrates average testing times with standard deviation, while Table 3 shows memory consumption in KB.

As Table 2 indicates, SVM and LR both excelled in accuracy. However, LR exhibited slightly slower testing times (0.37 ms vs. 0.20 ms for SVM) and higher memory usage (3.34 KB vs. 2.93 KB for SVM). All models, except KNN, demonstrated reasonable memory usage. Given its superior accuracy and fast testing time – crucial for real-time protection – we select SVM as VishielDroid's classifier for subsequent experiments.

When using SVM, LR, and KNN models, VishielDroid had only one false negative case. Out of the 230 vishing malware samples from 2023, a single sample was falsely identified as benign by VishielDroid. As described in Section 5.1, we executed each app for a specific duration with about ten user interactions. However, for this particular false negative case, the sample did not request any runtime permissions during our execution.

Upon further investigation, we discovered that the corresponding feature vector indicated that no runtime permission requests were made during the execution. To better understand this case, we conducted an additional experiment with more user interactions on this sample. The sample then eventually executed a nested APK installation and requested runtime permissions. In this case, VishielDroid successfully detected the malicious behavior.

This finding highlights the importance of how malware behavior is observed in dynamic analysis. It emphasizes the need for a comprehensive and adaptive approach to capturing malware behavior, taking into account factors such as the duration of execution and the level of user interaction. By simulating realistic user interactions and monitoring the app's behavior over an extended period, dynamic analysis can uncover hidden or delayed malicious activities that may not be immediately apparent during initial execution.

5.3. Comparison with state-of-the-art models

To evaluate the effectiveness of VishielDroid, we compared it with three state-of-the-art models: MalScan (Wu et al., 2019), Li et al.'s static analysis-based approach (Li et al., 2021), and Singh et al.'s dynamic analysis-based method (Singh et al., 2024). While we initially considered HearMeOut (Kim et al., 2022), the only prior tool specifically designed for vishing detection, its pattern-based API detection approach failed to detect any vishing apps in our experiments before their malicious behaviors were executed. This demonstrates the limitations of pattern-based detection compared to our ML-based approach. To our knowledge, VishielDroid is the first ML-based tool specifically targeting vishing malware.

Comparison with HearMeOut. Kim et al. (2022) introduced HearMeOut, a system service in a custom AOSP 8.1 OS that monitors the app runtime behaviors using modified APIs to detect malicious activities based on specific rules.

We installed the publicly available HearMeOut code (https://github.com/WSP-LAB/hearmeout) on a Pixel 2 device. Unlike our experiments with VishielDroid, we limited our testing to 30 vishing samples for HearMeOut. This decision was made because HearMeOut failed to detect any malicious behavior in actual vishing malware, making further testing unnecessary.

We executed these samples on the Pixel 2 device equipped with HearMeOut, following the same procedure as our experiments with VishielDroid. As expected from HearMeOut's operational design, none of the samples was detected before phone calls were made. Furthermore, we observed actual cases of call redirection in 4 samples, but none triggered HearMeOut's detection mechanism.

Our analysis revealed that HearMeOut's detection system is activated only when a phone call is initiated based on a limited set of predefined conditions. This behavior was confirmed by examining HearMeOut's source code (see Listing 1).

HearMeOut's operation can be summarized as follows: When an app attempts call redirection, the user-entered phone number is transferred to HearMeOut's detection module via `outgoingCallChanged()` and stored as the original call number. Upon initiating an actual phone call, `TelephonyConnectionService.outgoingCallcreated()` is invoked. HearMeOut modifies this function to transfer the actual call number to its detection module. It then compares these two numbers to determine if call redirection occurred, flagging it if the first number is not a substring of the latter.

```
1
2  //a. BroadcastReceiver.setResultData(String
       data){
3
4  detector.outgoingCallChanged(mPendingResult.
       mResultData,
5            ... );
6
7
8  //b. Detector.outgoingCallChanged(String
       originalNumber,
9  //            ...){
10
11 originalCallNumber = originalNumber;
12
13
14 //c. TelephonyConnectionService.
       outgoingCallCreated(...){
15
16 number = phone.getVoiceMailNumber();
17
18         ...
19
20 boolean result = detector.outgoingCallCreated(
       number);
21
22
23 //d. Detector.outgoingCallCreated(String
       finalNumber){
24
25 if (originalCallNumber != null &&
26      isPhishingCallRedirection(
       originalCallNumber
27                                        ,
       finalNumber)) {
28          ...
29
30      sender("phishing call redirection
       detected",
31           callRedirectionPackage,
       callRedirectionAppName);
32 }
```

Listing 1: Operational code of HearMeOut for detecting the call redirection attempt during outgoing calls.

Our source code analysis revealed that HearMeOut is currently implemented to only detect this specific activity. Consequently, HearMeOut cannot capture vishing malware before the phone call step, allowing vishing malware to obtain various permissions to interfere with the victim's device and potentially steal personal information.

Comparison with Static Analysis Tools. We first compare VishielDroid with two state-of-the-art static analysis solutions: MalScan (Wu et al., 2019) and Li et al.'s approach (Li et al., 2021). MalScan represents the best-performing graph-based solution according to a recent survey (Gao et al., 2024), utilizing function call graphs and network centrality of sensitive API calls (21,986 features). Li et al.'s approach employs deep learning with comprehensive feature sets, including permissions, APIs, and intent actions (379 features), known for their robustness against adversarial malware.

For a fair comparison, we used datasets that all approaches could analyze: 574 samples for training (383 benign, 191 vishing) and 577 samples for validation (385 benign, 192 vishing). To study the impact of training data size, we also created subsets with 50% and 75% of the original training data while maintaining the benign-to-vishing ratio.

The comparison results shown in Table 4 demonstrate that VishielDroid consistently outperforms both static analysis baselines. With full training data, VishielDroid achieves a 99.78% F1-score using only 98 features, significantly surpassing MalScan's 80.25% with 21,986 features and Li et al.'s 69.27% with 379 features. When reducing the training data to 50%, VishielDroid maintains its high performance with a 99.78% F1-score, while Li et al.'s approach shows considerable degradation to 64.43%. MalScan's performance remains relatively stable but lower, with F1-scores ranging from 80.19% to 80.25%. Both static analysis approaches failed to analyze 74 vishing samples due to encrypted dex files, highlighting the limitation of static analysis methods against modern vishing malware.

Table 4
Performance comparison between VishielDroid, MalScan (Wu et al., 2019), and Li et al. (2021) with varying training set sizes.

Solution	Measure	Training samples rate		
		50 (%)	75 (%)	100 (%)
VishielDroid	Precision (%)	100.00	100.00	100.00
	Recall (%)	99.57	99.57	99.57
	Accuracy (%)	99.86	99.86	99.86
	F1-score (%)	99.78	99.78	99.78
MalScan (Wu et al., 2019)	Precision (%)	98.05	98.11	98.24
	Recall (%)	67.83	67.83	67.83
	Accuracy (%)	89.06	89.08	89.11
	F1-score (%)	80.19	80.21	80.25
Li et al. (2021)	Precision (%)	66.51	73.91	75.62
	Recall (%)	67.83	67.83	67.83
	Accuracy (%)	70.59	76.03	76.72
	F1-score (%)	64.43	68.52	69.27

Table 5
Performance comparison between VishielDroid and Singh et al. (2024) with varying training set sizes.

Solution	Measure	Training samples rate		
		50 (%)	75 (%)	100 (%)
VishielDroid	Precision (%)	98.11	99.05	99.52
	Recall (%)	90.43	90.43	90.43
	Accuracy (%)	96.23	96.52	96.66
	F1-score (%)	94.12	94.55	94.76
Singh et al. (2024)	Precision (%)	93.45	93.45	93.45
	Recall (%)	68.26	68.26	68.26
	Accuracy (%)	88.00	88.00	88.00
	F1-score (%)	78.89	78.89	78.89

Comparison with Dynamic Analysis Tools. We further evaluated VishielDroid against Singh et al.'s dynamic analysis approach (Singh et al., 2024), which tracks system calls during runtime using Strace (Strace, 2024) and employs XGBoost and RF for classification with 146 features. For a fair comparison with Singh et al.'s dynamic analysis method, we also executed VishielDroid using Monkey (Android Developers, 2024a) to simulate random user interactions for 2 min.

For this comparison, we used 548 samples for training (376 benign, 172 vishing), 557 for validation (380 benign, 177 vishing), and 689 for testing (459 benign, 230 vishing). We also tested different training data sizes following the same methodology as the static analysis comparison.

Table 5 shows that VishielDroid maintains superior performance in the dynamic analysis setting, though notably lower than its performance in Table 4 (94.76% vs 99.78% F1-score). This performance difference can be attributed to the use of Monkey for automated testing instead of real human interactions — while necessary for a fair comparison with Singh et al.'s approach, automated input generation may not trigger the full range of vishing behaviors that VishielDroid is designed to detect. When the training data is reduced to 50%, VishielDroid's F1-score decreases to 94.12%, while Singh et al.'s approach maintains a constant but lower 78.89% F1-score regardless of training data size. Under these automated testing conditions, approximately 10% of vishing samples remain undetected since the simulated interactions with Monkey failed to trigger sufficient runtime permission requests ($\mathbf{P}_{\mathbf{runtime}}$) that are typically observed during actual vishing attempts.

5.4. Resilience with imbalanced dataset

To evaluate VishielDroid's resilience in handling imbalanced datasets, we adjusted the ratio between vishing and benign samples in our dataset, as summarized in Table 6.

Table 6
Impact of data imbalance on VishielDroid.

Measure	1 : 20	1 : 10	1 : 5
Precision (%)	100.00	100.00	100.00
Recall (%)	95.65	97.87	98.95
Accuracy (%)	99.80	99.81	99.82
F1-score (%)	97.78	98.92	99.47

For each case of imbalance, we included all benign samples from 2021 and 2022 and randomly selected a portion of the vishing samples from the same period according to each specified ratio for training the model. Similarly, we selected the 2023 samples in the same manner for testing. Despite the degradation in detection performance, particularly in terms of recall, as the imbalance severity increased, VishielDroid still achieved metrics above 95% even at the most severe ratio of 1:20 (trained with 773 benign apps and 38 vishing malware samples and tested with 475 benign apps and 23 vishing malware samples).

This performance indicates that VishielDroid is highly effective in detecting vishing malware even with a significantly imbalanced dataset, demonstrating its practical applicability in real-world scenarios where vishing samples are rare compared to benign applications.

5.5. Resilience against adversarial attacks

To assess the resilience of VishielDroid against potential evasion attacks that manipulate permission requests in vishing malware, we created a set of adversarial instances by modifying the permission features of 230 original vishing samples from 2023. These instances were generated by setting the values of 24 permission features to 1, which exhibited negative point-biserial coefficients (P75–P98 in Appendix), indicating that those permissions frequently appeared in benign samples. In other words, vishing malware writers could develop vishing malware samples that intentionally request permissions commonly used in benign apps, even if those permissions have no relationship with conducting vishing attacks.

When tested on these 230 modified feature vectors, VishielDroid still successfully detected 207 (90.00%) of them as vishing apps. However, this performance represents a significant drop of 9.57% from the original 99.57%, as VishielDroid failed to detect 22 vishing malware samples.

We propose two potential approaches to ensure that VishielDroid remains robust against adversarial attacks. The first approach involves incorporating adversarial instances into the training of the ML model. However, this method may lead to a decrease in the performance of the original model in detecting benign apps while also potentially triggering the creation of other types of adversarial instances, resulting in an arms race between attackers and defenders. The second approach is to use only the feature set with positive correlation coefficients, assuming that these permissions are necessary for the functionality of vishing malware. When using this method, attackers cannot generate adversarial samples by simply adding permissions. We conducted additional experiments with this reduced feature set and found that VishielDroid still maintained a high F1-score of 99.57%, only a 0.21% decrease, demonstrating its robustness even with this reduced feature set. This indicates that by considering only the features that frequently appear in vishing malware samples, we can effectively detect these samples. However, false negatives may increase if we consider more benign apps that use phone call-related functions. Therefore, the optimal feature set may vary depending on the observed benign apps and vishing malware samples. We surmise that determining the optimal feature set based on experimental results may lead to overfitting issues.

Table 7
Ablation study results.

Measure	$\mathbf{P}_{\mathbf{install}}$	$\mathbf{P}_{\mathbf{runtime}}$	$\mathbf{P}_{\mathbf{install}} + \mathbf{P}_{\mathbf{runtime}}$
Precision (%)	100.00	99.13	100.00
Recall (%)	90.43	99.57	99.57
Accuracy (%)	96.88	99.57	99.86
F1-score (%)	94.98	99.35	99.78

Table 8
Memory usage of VishielDroid.

Memory	w/o VishielDroid		w/ VishielDroid		
(MB)	Total	System	Total	System	App
Benign	1,346.26	138.70	1,413.76	140.82	36.28
Vishing	1,444.09	187.30	1,481.63	188.92	37.25
Average	1,395.18	163.00	1,447.70	164.87	36.77

5.6. Ablation study

We conducted an ablation study to evaluate the individual contribution of each type of permission request ($\mathbf{P}_{\mathbf{install}}$ and $\mathbf{P}_{\mathbf{runtime}}$) to the effectiveness of our defense mechanism. We retrained the optimized SVM models with each type of feature set separately and then combined them.

Table 7 presents the test results of these retrained models. The combination of $\mathbf{P}_{\mathbf{install}}$ and $\mathbf{P}_{\mathbf{runtime}}$ yielded the highest F1-score performance at 99.78%. Surprisingly, using only $\mathbf{P}_{\mathbf{runtime}}$ achieved a nearly identical F1-score of 99.35%, despite exhibiting lower correlation coefficients compared to $\mathbf{P}_{\mathbf{install}}$ in Fig. 7. In contrast, using only $\mathbf{P}_{\mathbf{install}}$ resulted in a high F1-score of 94.98%, but its performance was somewhat limited compared to $\mathbf{P}_{\mathbf{runtime}}$, failing to detect some vishing samples.

This study reveals that $\mathbf{P}_{\mathbf{runtime}}$ plays a critical role in distinguishing between benign and vishing samples, suggesting that runtime behavior is a key indicator of malicious activity. The effectiveness of $\mathbf{P}_{\mathbf{runtime}}$ underscores the importance of monitoring apps' behavior during execution for reliable malware detection. However, the combination of $\mathbf{P}_{\mathbf{install}}$ and $\mathbf{P}_{\mathbf{runtime}}$ provides the most robust and effective detection of vishing malware.

5.7. System overhead of VishielDroid

To assess the impact of VishielDroid on actual user experience, we implemented VishielDroid on an AOSP 8.1 system. Specifically, Feature Extractor was implemented as a service, while Vishing Detector functioned as a system app. This integrated solution was deployed on a Pixel 2 Android device, and its system resource usage (memory and battery consumption) was compared against a vanilla custom OS without VishielDroid. This comparison aimed to understand how VishielDroid influences device performance in real-world scenarios, ensuring it does not significantly hinder user experience. The skl2onnx library was used to convert and integrate an optimized SVM model into the Vishing Detector, placing it in its res/raw directory.

We selected 30 benign and 30 vishing samples from the 2023 dataset and executed them as outlined in Section 5.1, extending the execution to 2 min and disregarding alert dialogs during each vishing sample's execution. Memory usage and battery consumption were measured during each app's installation and execution using Android's dumpsys tool, with measurements taken every 10 s. The usage or consumption by the entire device (Total), including Feature Extractor (System) and Vishing Detector (App), was specifically tracked.

Memory Usage: On average, Feature Extractor accounted for an increase of 1.87 MB, representing 0.13% of the total memory usage

Table 9
Battery consumption of VishielDroid.

Battery	w/o VishielDroid		w/ VishielDroid		
(mAh)	Total	System	Total	System	App
Benign	11.79	1.70	11.88	1.76	0.01
Vishing	11.91	1.90	12.20	1.99	0.02
Average	11.85	1.80	12.04	1.87	0.01

without VishielDroid. Vishing Detector introduced additional memory usage of 36.77 MB, constituting 2.54% of the total device memory usage with VishielDroid (see Table 8).

Battery Consumption: On average, Feature Extractor accounted for an increase of 0.08 mAh, representing 0.59% of the total battery consumption without VishielDroid. The Vishing Detector displayed a discernible difference in consumption between benign and vishing samples, consuming on average 0.01 mAh, which is only 0.08% of the total consumption with VishielDroid (see Table 9).

6. Applicability of VishielDroid to Android 12

To verify VishielDroid's compatibility with newer Android versions, we implemented it on a Google Pixel 3XL device running Android 12 (API level 31). The porting process from Android 8.1 (SDK level 27) to Android 12 proved to be straightforward, requiring only a minor modification to relocate our permission monitoring code from Activity.startActivity() to Activity.executeRequest(). We easily identified this modification point by analyzing Logcat Android Developers (2024b) outputs during sample execution.

We evaluated VishielDroid on Android 12 using our original dataset, excluding 17 benign samples that failed installation or execution due to version compatibility issues. The experiments showed that permission-related behaviors remain consistent across versions. For example, even deprecated permissions like USE_FINGERPRINT (SDK 28) in benign apps and PROCESS_OUTGOING_CALLS (SDK 29) in vishing samples were still actively used, demonstrating that VishielDroid's permission monitoring logic remains valid across Android versions.

The successful porting of VishielDroid to Android 12 with minimal code changes suggests that our approach can be readily adapted to different Android versions, requiring only version-specific adjustments to the permission monitoring implementation points.

7. Discussion

Satisfying the Key Requirements. We discuss how VishielDroid meets the requirements mentioned in Section 2.2. First, unlike HearMeOut (Kim et al., 2022), which focuses on call redirection, VishielDroid is designed to detect suspicious activities of apps before any attack occurs, during installation (Phase 1) or at the time of runtime permission requests (Phase 2). Second, as shown in Section 5.2, we showed that by training with samples from 2021 and 2022, VishielDroid could almost perfectly detect *unseen* samples from 2023. Third, as shown in Section 5.4, VishielDroid is designed using the unavoidable permission requests to perform voice phishing activities, successfully addressing the imbalanced nature of vishing malware datasets and maintaining high accuracy despite fewer malicious samples. Fourth, as shown in Section 5.7, VishielDroid operates successfully on Pixel 2 with marginal memory and battery consumption overhead. Finally, VishielDroid is designed to be resilient against user-level vishing malware because it is integrated as a system service and system app within the Android OS.

Generic Malware Detection. Although VishielDroid was originally designed to detect vishing malware, its foundational principles can be extended to detect generic Android malware using runtime permissions. Permission requests are essential for all Android apps and provide valuable insights into potential malicious behavior. Malware inevitably requires permissions to access sensitive data or utilize specific functions. VishielDroid can leverage both install-time and runtime permissions to detect these malicious activities effectively. By analyzing how runtime permissions are used in malware beyond vishing, we aim to identify commonalities and differences between vishing malware and other malware types in their permission requests. This analysis will be a part of our future work, as it will help us expand the applicability of VishielDroid to a broader range of Android malware and enhance its overall effectiveness in protecting users from various mobile threats.

Detection of Colluding Apps. While VishielDroid currently analyzes permission request patterns on a per-app basis, colluding apps that collaborate to evade detection could pose a challenge. For example, in a colluding attack scenario, one app might request call-related permissions while another requests audio recording permissions, with their combined capabilities enabling vishing attacks. To address this potential challenge, one possible approach would be to extend VishielDroid with a time window-based analysis that examines permission requests from multiple apps collectively. Since permissions are typically requested only once when granted, this could help identify potential colluding apps by grouping those that request complementary vishing-related permissions and monitoring their correlated API usage patterns (*e.g.*, getDefaultDialerPackage(), getLine1Number()). Such an extension might involve adapting VishielDroid module to aggregate permission requests across apps within configurable time windows, potentially enabling the detection of coordinated malicious behaviors.

8. Limitations

Challenges in Dynamic Analysis. Although VishielDroid's approach to combating vishing malware through runtime permissions analysis has shown promising results, it faces challenges due to its reliance on dynamic analysis. The permission information collected during experiments may vary depending on how the app is executed, potentially leading to inconsistencies and affecting malware detection accuracy. To address this issue, our future work will focus on developing a strategy that combines static and dynamic analysis techniques to handle code obfuscation methods, such as packing and encryption, employed by vishing malware. Specifically, we consider a two-step process to enhance the detection of runtime permissions used by vishing malware. First, we dynamically execute the app to obtain the unpacked, decrypted code. Once the code is obtained, we dump it for further analysis. Second, we employ static analysis to examine the dumped code, focusing on identifying the permissions requested and used by the app. By adopting this hybrid approach, which combines dynamic execution and static analysis, we aim to establish a more consistent and reliable method for detecting runtime permissions utilized by vishing malware.

Android OS Code Modification. VishielDroid's system-level implementation offers robust protection against vishing malware on Android through real-time app behavior monitoring while maintaining resilience against anti-defense attacks. However, the need for Android OS code modification presents a significant challenge to VishielDroid's widespread adoption. To overcome this hurdle and reach end-users, collaboration with major Android device manufacturers like Samsung and Google is crucial. Integration into the stock Android OS would facilitate wider distribution and resolve the benign app compatibility issues encountered during our dataset collection.

Absence of User Study. Although our work primarily focuses on detection, the lack of a user study in our evaluation process limits the assessment of VishielDroid's effectiveness in detecting vishing attacks in real-world scenarios. In practice, users may overlook detection alerts amidst numerous runtime permission requests. To address this

issue, we should consider conducting user studies to gather insights and guide improvements in VishielDroid's warning design. However, evaluating the effectiveness of warning designs is considered beyond the scope of this paper.

9. Related work

Static Android Malware Detection. Numerous methods employing static and dynamic analysis have been developed for detecting Android malware (Li et al., 2021; Onwuzurike et al., 2019; Kouliaridis et al., 2020; Li et al., 2018; Ding et al., 2023; Cai et al., 2018; Bhat et al., 2023; Alzaylaee et al., 2020; Wu et al., 2019). Static analysis inspects an app's code without executing it, identifying malicious behaviors such as malware patterns or suspicious permissions. This non-intrusive approach detects potential threats without risking devices. Wu et al. (2019) introduced MalScan, which constructs control flow graphs (CFGs) and integrates network centrality metrics, focusing on sensitive API occurrences in the execution flow. These models emphasize different dimensions of API usage: Onwuzurike et al.'s MaMaDroid (Onwuzurike et al., 2019) generalizes patterns through abstraction, while MalScan prioritizes structural significance. Li et al. (2021) introduced a deep learning model leveraging permissions, API usage, and intents to detect complex patterns undetectable through traditional methods. In contrast, VishielDroid focuses on runtime permissions introduced in Android 6.0 (API level 23) (Android Developers, 2024c), offering granular, real-time user decision analysis. This aligns VishielDroid with the latest Android security model, providing a more comprehensive evaluation of user–permission interactions.

Dynamic Android Malware Detection. Static analysis struggles with detecting sophisticated malware reliant on runtime behaviors, such as dynamically loaded code and reflective API calls. To overcome these limitations, dynamic analysis emerged, monitoring app behavior during execution to capture real-time activities like API calls, system calls, and runtime permissions. Yang et al. (2017) dynamically gathered inter-component communication and method invocation features in emulated environments, while Alzaylaee et al.'s DL-Droid (Alzaylaee et al., 2020) combined static and dynamic data, including Intent actions and real-device API calls. Similarly, Bhat et al. (2023) focused on system calls and Android objects in emulated settings. Singh et al. (2024) applied interpretable machine learning (XAI) techniques to explain anomalous behavior detected through system call-based dynamic analysis. Despite advances in dynamic analysis, existing research predominantly targets general malware categories like spyware, trojans, and ransomware. Vishing malware, with its unique reliance on runtime permissions, often bypasses static and dynamic methods. VishielDroid addresses this challenge by specifically optimizing detection for vishing malware, combining runtime and installation-time permission analysis.

Phishing Detection in Telephony. Phishing attacks in telephony have garnered increasing attention, with studies emphasizing behavioral analysis of call scams (Song et al., 2014; Kim et al., 2022; Tu et al., 2019; Choi et al., 2017; Gupta et al., 2015; Pandit et al., 2023; Oh et al., 2023). Oh et al. (2023) developed a SIM box fraud prevention system leveraging fingerprint-based access policies. Pandit et al. (2023) proposed an NLP-based assistant to block spoofed robocalls. Focusing on vishing malware, Kim et al. (2022) examined phishing functionalities used to deceive victims during calls. They introduced HearMeOut, which detects call-based vishing activities but leaves a gap for pre-call exploits of privacy or device control. In contrast, VishielDroid bridges this gap by preemptively detecting and blocking vishing activities through a combination of install-time and runtime permission analysis.

10. Conclusion

In this paper, we present VishielDroid, a lightweight vishing malware detection system designed specifically for Android devices. Our approach distinguishes between permissions granted at install time and those requested during runtime, providing a more comprehensive view of an app's behavior. We have modified the Android API framework to capture app runtime permission (Android Developers, 2024c) requests in real-time.

VishielDroid outperforms state-of-the-art models across various testing scenarios. We demonstrated its effectiveness in controlled and randomized environments through systematic manual testing and automated evaluation. While manual testing with structured UI interactions revealed the full range of vishing behaviors, VishielDroid maintained strong performance even under automated testing conditions, showing its robustness in real-world scenarios. The system has demonstrated exceptional resilience in challenging conditions: maintaining a high F1-score of 97.78% under severe class imbalance (20:1 benign-to-vishing ratio), achieving 99.57% F1-score with a reduced feature set, and performing consistently well across different Android versions from 8.1 to 12 with minimal modifications required. These results establish VishielDroid as a practical and adaptable solution for detecting vishing malware across the Android ecosystem.

CRediT authorship contribution statement

Chanjong Lee: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Bedeuro Kim:** Writing – original draft, Visualization, Supervision. **Hyoungshick Kim:** Writing – original draft, Writing – review & editing, Supervision, Resources, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This work was supported by the IITP grants (2022-0-01199, 2022-0-00495, RS-2024-00459638, RS-2024-00451909, and 2018-0-00532 (30%)) from the Korea government.

Appendix. Permissions used in VishielDroid

Table A.10 shows the list of permissions used by VishielDroid to identify vishing malware on Android devices. Each entry in the table includes:

- Index: A unique identifier for the permission.
- Permission name: A descriptive name for the permission.
- Correlation: The biserial correlation coefficient, indicating the statistical association between the permission and vishing malware (1) versus benign apps (0). Higher values suggest a stronger association with vishing malware.
- Request Type: Whether the permission is requested during app installation ($\mathbf{P_{install}}$) or when the app is running ($\mathbf{P_{runtime}}$).
- Protection Level: The Android-defined protection level for the permission, determining how sensitive it is and what restrictions apply to its use.

Table A.10
List of the permissions used in VishielDroid.

Index	Permission name	Correlation	Request type	Protection level
P1	WRITE_CALL_LOG	0.975	$\mathbf{P}_{\text{install}}$	Dangerous
P2	PROCESS_OUTGOING_CALLS	0.971	$\mathbf{P}_{\text{install}}$	Dangerous
P3	READ_SMS	0.969	$\mathbf{P}_{\text{install}}$	Dangerous
P4	READ_CALL_LOG	0.965	$\mathbf{P}_{\text{install}}$	Dangerous
P5	WRITE_CONTACTS	0.913	$\mathbf{P}_{\text{install}}$	Dangerous
P6	CALL_PHONE	0.878	$\mathbf{P}_{\text{install}}$	Dangerous
P7	DISABLE_KEYGUARD	0.862	$\mathbf{P}_{\text{install}}$	Normal
P8	REORDER_TASKS	0.838	$\mathbf{P}_{\text{install}}$	Normal
P9	CHANGE_WIFI_STATE	0.835	$\mathbf{P}_{\text{install}}$	Normal
P10	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	0.820	$\mathbf{P}_{\text{install}}$	Normal
P11	READ_CONTACTS	0.800	$\mathbf{P}_{\text{install}}$	Dangerous
P12	GET_TASKS	0.795	$\mathbf{P}_{\text{install}}$	Others
P13	SYSTEM_ALERT_WINDOW	0.784	$\mathbf{P}_{\text{install}}$	Others
P14	RECORD_AUDIO	0.735	$\mathbf{P}_{\text{install}}$	Dangerous
P15	READ_PHONE_STATE	0.731	$\mathbf{P}_{\text{install}}$	Dangerous
P16	MODIFY_AUDIO_SETTINGS	0.716	$\mathbf{P}_{\text{install}}$	Normal
P17	ACCESS_FINE_LOCATION	0.611	$\mathbf{P}_{\text{install}}$	Dangerous
P18	READ_PHONE_NUMBERS	0.603	$\mathbf{P}_{\text{install}}$	Dangerous
P19	ANSWER_PHONE_CALLS	0.594	$\mathbf{P}_{\text{install}}$	Dangerous
P20	CAMERA	0.511	$\mathbf{P}_{\text{install}}$	Dangerous
P21	RECEIVE_SMS	0.501	$\mathbf{P}_{\text{install}}$	Dangerous
P22	BROADCAST_STICKY	0.476	$\mathbf{P}_{\text{install}}$	Normal
P23	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	0.440	$\mathbf{P}_{\text{runtime}}$	Normal
P24	ACCESS_WIFI_STATE	0.434	$\mathbf{P}_{\text{install}}$	Normal
P25	REQUEST_INSTALL_PACKAGES	0.385	$\mathbf{P}_{\text{install}}$	Others
P26	WRITE_EXTERNAL_STORAGE	0.371	$\mathbf{P}_{\text{install}}$	Dangerous
P27	READ_EXTERNAL_STORAGE	0.325	$\mathbf{P}_{\text{install}}$	Dangerous
P28	RECEIVE_BOOT_COMPLETED	0.279	$\mathbf{P}_{\text{install}}$	Normal
P29	RECEIVE_MMS	0.262	$\mathbf{P}_{\text{install}}$	Dangerous
P30	REQUEST_DELETE_PACKAGES	0.262	$\mathbf{P}_{\text{install}}$	Normal
P31	MOUNT_UNMOUNT_FILESYSTEMS	0.221	$\mathbf{P}_{\text{install}}$	Others
P32	ACCESS_LOCATION_EXTRA_COMMANDS	0.202	$\mathbf{P}_{\text{install}}$	Normal
P33	ACCESS_COARSE_LOCATION	0.191	$\mathbf{P}_{\text{install}}$	Dangerous
P34	RESTART_PACKAGES	0.188	$\mathbf{P}_{\text{install}}$	Others
P35	BLUETOOTH_ADMIN	0.177	$\mathbf{P}_{\text{install}}$	Normal
P36	BROADCAST_STICKY	0.157	$\mathbf{P}_{\text{runtime}}$	Normal
P37	CHANGE_NETWORK_STATE	0.157	$\mathbf{P}_{\text{runtime}}$	Normal
P38	CHANGE_WIFI_STATE	0.157	$\mathbf{P}_{\text{runtime}}$	Normal
P39	GET_TASKS	0.157	$\mathbf{P}_{\text{runtime}}$	Others
P40	READ_SMS	0.157	$\mathbf{P}_{\text{runtime}}$	Dangerous
P41	RECEIVE_SMS	0.157	$\mathbf{P}_{\text{runtime}}$	Dangerous
P42	REQUEST_DELETE_PACKAGES	0.157	$\mathbf{P}_{\text{runtime}}$	Normal
P43	DISABLE_KEYGUARD	0.157	$\mathbf{P}_{\text{runtime}}$	Normal
P44	EXPAND_STATUS_BAR	0.148	$\mathbf{P}_{\text{install}}$	Normal
P45	MODIFY_AUDIO_SETTINGS	0.147	$\mathbf{P}_{\text{runtime}}$	Normal
P46	PROCESS_OUTGOING_CALLS	0.147	$\mathbf{P}_{\text{runtime}}$	Dangerous
P47	WAKE_LOCK	0.147	$\mathbf{P}_{\text{runtime}}$	Normal
P48	WRITE_CALL_LOG	0.147	$\mathbf{P}_{\text{runtime}}$	Dangerous
P49	WAKE_LOCK	0.145	$\mathbf{P}_{\text{install}}$	Normal
P50	ACCESS_WIFI_STATE	0.137	$\mathbf{P}_{\text{runtime}}$	Normal
P51	BLUETOOTH	0.137	$\mathbf{P}_{\text{runtime}}$	Normal
P52	BLUETOOTH_ADMIN	0.137	$\mathbf{P}_{\text{runtime}}$	Normal
P53	READ_CALL_LOG	0.137	$\mathbf{P}_{\text{runtime}}$	Dangerous
P54	READ_PHONE_NUMBERS	0.137	$\mathbf{P}_{\text{runtime}}$	Dangerous
P55	VIBRATE	0.137	$\mathbf{P}_{\text{runtime}}$	Normal
P56	READ_LOGS	0.134	$\mathbf{P}_{\text{install}}$	Others
P57	ACCESS_NETWORK_STATE	0.128	$\mathbf{P}_{\text{runtime}}$	Normal
P58	INTERNET	0.128	$\mathbf{P}_{\text{runtime}}$	Normal
P59	REQUEST_INSTALL_PACKAGES	0.126	$\mathbf{P}_{\text{runtime}}$	Others
P60	BLUETOOTH	0.124	$\mathbf{P}_{\text{install}}$	Normal
P61	ACCESS_LOCATION_EXTRA_COMMANDS	0.119	$\mathbf{P}_{\text{runtime}}$	Normal
P62	EXPAND_STATUS_BAR	0.119	$\mathbf{P}_{\text{runtime}}$	Normal
P63	MODIFY_PHONE_STATE	0.112	$\mathbf{P}_{\text{runtime}}$	Others
P64	MODIFY_PHONE_STATE	0.112	$\mathbf{P}_{\text{install}}$	Others
P65	ANSWER_PHONE_CALLS	0.109	$\mathbf{P}_{\text{runtime}}$	Dangerous
P66	RECEIVE_BOOT_COMPLETED	0.103	$\mathbf{P}_{\text{runtime}}$	Normal
P67	CHANGE_NETWORK_STATE	0.088	$\mathbf{P}_{\text{install}}$	Normal
P68	WRITE_CONTACTS	0.084	$\mathbf{P}_{\text{runtime}}$	Dangerous
P69	ACCESS_NETWORK_STATE	0.075	$\mathbf{P}_{\text{install}}$	Normal
P70	MOUNT_UNMOUNT_FILESYSTEMS	0.072	$\mathbf{P}_{\text{runtime}}$	Others
P71	SYSTEM_ALERT_WINDOW	0.067	$\mathbf{P}_{\text{runtime}}$	Others
P72	RECEIVE_MMS	0.059	$\mathbf{P}_{\text{runtime}}$	Dangerous
P73	CALL_PHONE	0.056	$\mathbf{P}_{\text{runtime}}$	Dangerous
P74	INTERNET	0.051	$\mathbf{P}_{\text{install}}$	Normal
P75	WRITE_SETTINGS	−0.051	$\mathbf{P}_{\text{runtime}}$	Others

(*continued on next page*)

Table A.10 (*continued*).

Index	Permission name	Correlation	Request type	Protection level
P76	MANAGE_OWN_CALLS	−0.051	$\mathbf{P}_{\mathbf{install}}$	Normal
P77	BIND_APPWIDGET	−0.051	$\mathbf{P}_{\mathbf{install}}$	Others
P78	GET_PACKAGE_SIZE	−0.058	$\mathbf{P}_{\mathbf{install}}$	Normal
P79	CAMERA	−0.058	$\mathbf{P}_{\mathbf{runtime}}$	Dangerous
P80	ACCESS_COARSE_LOCATION	−0.060	$\mathbf{P}_{\mathbf{runtime}}$	Dangerous
P81	SET_WALLPAPER_HINTS	−0.066	$\mathbf{P}_{\mathbf{install}}$	Normal
P82	KILL_BACKGROUND_PROCESSES	−0.083	$\mathbf{P}_{\mathbf{install}}$	Normal
P83	READ_SYNC_STATS	−0.083	$\mathbf{P}_{\mathbf{install}}$	Normal
P84	VIBRATE	−0.085	$\mathbf{P}_{\mathbf{install}}$	Normal
P85	ACCESS_NOTIFICATION_POLICY	−0.091	$\mathbf{P}_{\mathbf{install}}$	Normal
P86	CHANGE_WIFI_MULTICAST_STATE	−0.093	$\mathbf{P}_{\mathbf{install}}$	Normal
P87	READ_SYNC_SETTINGS	−0.111	$\mathbf{P}_{\mathbf{install}}$	Normal
P88	NFC	−0.116	$\mathbf{P}_{\mathbf{install}}$	Normal
P89	WRITE_SYNC_SETTINGS	−0.117	$\mathbf{P}_{\mathbf{install}}$	Normal
P90	WRITE_CALENDAR	−0.121	$\mathbf{P}_{\mathbf{install}}$	Dangerous
P91	READ_CALENDAR	−0.126	$\mathbf{P}_{\mathbf{install}}$	Dangerous
P92	READ_EXTERNAL_STORAGE	−0.131	$\mathbf{P}_{\mathbf{runtime}}$	Dangerous
P93	SET_WALLPAPER	−0.137	$\mathbf{P}_{\mathbf{install}}$	Normal
P94	ACCESS_FINE_LOCATION	−0.139	$\mathbf{P}_{\mathbf{runtime}}$	Dangerous
P95	WRITE_SETTINGS	−0.158	$\mathbf{P}_{\mathbf{install}}$	Others
P96	USE_FINGERPRINT	−0.183	$\mathbf{P}_{\mathbf{install}}$	Normal
P97	WRITE_EXTERNAL_STORAGE	−0.210	$\mathbf{P}_{\mathbf{install}}$	Dangerous
P98	GET_ACCOUNTS	−0.226	$\mathbf{P}_{\mathbf{install}}$	Dangerous

Data availability

The code and file for reproducing our evaluation process is partly provided via anonymous github we denoted in our paper, along with the hash values representing the data samples used in our work.

References

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2020. DL-droid: Deep learning based android malware detection using real devices. Comput. Secur. 89, 1–11.

Androguard, 2024. Available online at: https://androguard.readthedocs.io/en/latest/.

Android Developers, 2024a. Available online at: https://developer.android.com/studio/test/other-testing-tools/monkey.

Android Developers, 2024b. Available online at: https://developer.android.com/tools/logcat.

Android Developers, 2024c. Android 6.0 changes. Available online at: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.

Anon, 2023a. Frida. Available online at: https://www.frida.re.

Anon, 2023b. Korea launches high-accuracy AI-based voice analysis model to tackle voice scams. Korea JoongAng Daily. Available online at: https://koreajoongangdaily.joins.com/2023/07/07/national/socialAffairs/voice-analysis-AI-voice-scam/20230707095705147.html.

Bai, Y., Xing, Z., Ma, D., Li, X., Feng, Z., 2021. Comparative analysis of feature representations and machine learning methods in android family classification. Comput. Netw. 184, 1–11.

Bernama, 2022. Macau Scam's new tactic: A police notice to probe Macau Scam. Free Malaysia Today. Available online at: https://www.freemalaysiatoday.com/category/nation/2022/06/13/macau-scams-new-tactic-a-police-notice-to-probe-macau-scam/.

Bhat, P., Behal, S., Dutta, K., 2023. A system call-based android malware detection approach with homogeneous & heterogeneous ensemble machine learning. Comput. Secur. 130, 1–20.

Cai, H., Meng, N., Ryder, B., Yao, D., 2018. Droidcat: Effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 14 (6), 1455–1470.

Choi, K., Lee, J., Chun, Y., 2017. Voice phishing fraud and its modus operandi. Secur. J. 30, 454–466.

Ding, Y., Zhang, X., Hu, J., Xu, W., 2023. Android malware detection method based on bytecode image. J. Ambient. Intell. Humaniz. Comput. 14 (5), 6401–6410.

Gao, C., Huang, G., Li, H., Wu, B., Wu, Y., Yuan, W., 2024. A comprehensive study of learning-based android malware detectors under challenging environments. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering.

Gupta, P., Srinivasan, B., Balasubramaniyan, V., Ahamad, M., 2015. Phoneypot: Data-driven understanding of telephony threats. In: Proceedings of the 22th Annual Network and Distributed System Security Symposium.

Hao, L., 2023. 2,349 Chinese suspects of telecom scam handed over to China, marking the largest single transfer since launch of crackdown campaign. Global Times. Available online at: https://www.globaltimes.cn/page/202310/1299934.shtml.

Kim, J., Kim, J., Wi, S., Kim, Y., Son, S., 2022. HearMeOut: Detecting voice phishing activities in android. In: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services.

Kornbrot, D., 2014. Point biserial correlation. Wiley StatsRef: Stat. Ref. Online.

Kouliaridis, V., Kambourakis, G., Peng, T., 2020. Feature importance in android malware detection. In: Proceedings of the 19th International Conference on Trust, Security and Privacy in Computing and Communications.

LaMont, L., 2024. The true cost of spam and scam calls in America. Truecaller. Available online at: https://www.truecaller.com/blog/insights/the-true-cost-of-spam-and-scam-calls-in-america.

Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., Ye, H., 2018. Significant permission identification for machine-learning-based android malware detection. IEEE Trans. Ind. Informatics 14 (7), 3216–3225.

Li, H., Zhou, S., Yuan, W., Luo, X., Gao, C., Chen, S., 2021. Robust android malware detection against adversarial example attacks. In: Proceedings of the 30th International Conference on World Wide Web.

Liu, E., Rao, S., Havron, S., Ho, G., Savage, S., Voelker, G.M., McCoy, D., 2023. No privacy among spies: Assessing the functionality and insecurity of consumer android spyware apps. In: Proceedings of the 24th International Symposium on Privacy Enhancing Technologies.

Maggi, F., 2010. Are the con artists back? A preliminary analysis of modern phone frauds. In: Proceedings of the 10th IEEE International Conference on Computer and Information Technology.

Mayfield, J., 2023. New FTC data show consumers reported losing nearly $8.8 billion to scams in 2022. Federal Trade Commission. Available online at: https://www.ftc.gov/news-events/news/press-releases/2023/02/new-ftc-data-show-consumers-reported-losing-nearly-88-billion-scams-2022.

Nahapetyan, A., Prasad, S., Childs, K., Oest, A., Ladwig, Y., Kapravelos, A., Reaves, B., 2024. On SMS phishing tactics and infrastructure. In: Proceedings of the 45th IEEE Symposium on Security and Privacy.

Naqvi, B., Perova, K., Farooq, A., Makhdoom, I., Oyedeji, S., Porras, J., 2023. Mitigation strategies against the phishing attacks: A systematic literature review. Comput. Secur. 132, 1–25.

Oh, B., Ahn, J., Bae, S., Son, M., Lee, Y., Kang, M., Kim, Y., 2023. Preventing SIM box fraud using device model fingerprinting. In: Proceedings of the 30th Annual Network and Distributed System Security Symposium.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G., 2019. Mamadroid: Detecting android malware by building Markov chains of behavioral models (Extended version). ACM Trans. Priv. Secur. 22 (2), 1–34.

Pandit, S., Perdisci, R., Ahamad, M., Gupta, P., 2018. Towards measuring the effectiveness of telephony blacklists. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium.

Pandit, S., Sarker, K., Perdisci, R., Ahamad, M., Yang, D., 2023. Combating robocalls with phone virtual assistant mediated interaction. In: Proceedings of the 32th USENIX Security Symposium.

Roy, S., DeLoach, J., Li, Y., Herndon, N., Caragea, D., Ou, X., Ranganath, V.P., Li, H., Guevara, N., 2015. Experimental study with real-world data for android app security analysis using machine learning. In: Proceedings of the 31st Annual Computer Security Applications Conference.

Ryall, J., 2021. Overseas Chinese in Japan warned on phone scams demanding bank transfers. South China Morning Post. Available online at: https://www.scmp.com/week-asia/lifestyle-culture/article/3116336/overseas-chinese-japan-warned-phone-scams-demanding.

Sahin, M., Francillon, A., Gupta, P., Ahamad, M., 2017. Sok: Fraud in telephony networks. In: Proceedings of the 2nd European Symposium on Security and Privacy.

Sebastián, S., Caballero, J., 2020. AVCLASS2: Massive malware tag extraction from AV labels. In: Proceedings of the 36th Annual Computer Security Applications Conference.

Shuang, L., 2023. Shanghai police warn of FaceTime scams. China Daily. Available online at: https://global.chinadaily.com.cn/a/202307/09/WS64aaa018a310bf8a75d6e12b.html.

Singh, A., Tanha, M., Girdhar, Y., Hunter, A., 2024. Interpretable android malware detection based on dynamic analysis. In: Proceedings of the 10th International Conference on Information Systems Security and Privacy.

Song, J., Kim, H., Gkelias, A., 2014. Ivisher: Real-time detection of caller ID spoofing. ETRI J. 36 (5), 865–875.

Strace, 2024. Available online at: https://strace.io/.

Tu, H., Doupé, A., Zhao, Z., Ahn, G.-J., 2019. Users really do answer telephone scams. In: Proceedings of the 28th USENIX Security Symposium.

VirusTotal, 2023. Available online at: https://www.virustotal.com.

Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., Jin, H., 2019. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering.

Yang, L., Guo, W., Hao, Q., Ciptadi, A., Ahmadzadeh, A., Xing, X., Wang, G., 2021. CADE: Detecting and explaining concept drift samples for security applications. In: Proceedings of the 30th USENIX Security Symposium.

Yang, M., Wang, S., Ling, Z., Liu, Y., Ni, Z., 2017. Detection of malicious behavior in android apps through API calls and permission uses analysis. Concurr. Computation: Pr. Exp. 29 (19), 1–13.

Chanjong Lee is an M.S. student in the Department of Science and Engineering, Sungkyunkwan University. He received a B.Ec. degree from the Department of Economics and a B.S. degree from the Department of Computer Science and Engineering. He is studying computer security supervised under Hyoungshick Kim. His research interest is focused on Android security.

Bedeuro Kim is a Ph.D. student in the Department of Computer Science and Engineering, Sungkyunkwan University. He received an M.S. degree from the Department of Computer Science and Engineering, Sungkyunkwan University. His current research interest is focused on usable security and security engineering.

Hyoungshick Kim is an associate professor in the Department of Computer Science and Engineering, College of Software, Sungkyunkwan University. He received a BS degree from the Department of Information Engineering at Sungkyunkwan University, an MS from the Department of Computer Science at KAIST, and a Ph.D. from the Computer Laboratory at the University of Cambridge in 1999, 2001, and 2012, respectively. After completing his Ph.D., he worked as a post-doctoral fellow in the Department of Electrical and Computer Engineering at the University of British Columbia. He previously worked for Samsung Electronics as a senior engineer from 2004 to 2008. He also worked as a distinguished visiting researcher at CSIRO Data61 from 2019 to 2020. His current research interests include usable security, vulnerability analysis, and data-driven security. He enjoys finding security issues in new systems, particularly recent AI systems and applications, and has been deeply engaged in identifying real security problems in these areas. His work aims to uncover and address the practical security challenges posed by emerging technologies.