Information Systems Frontiers
https://doi.org/10.1007/s10796-022-10328-8

DyPolDroid: Protecting Against Permission-Abuse Attacks in Android

Carlos E. Rubio-Medrano[1] · Pradeep Kumar Duraisamy Soundrapandian[3] · Matthew Hill[4] · Luis Claramunt[2] · Jaejong Baek[2] · Geetha S[3] · Gail-Joon Ahn[2]

Accepted: 12 August 2022

Abstract
Android applications are extremely popular, as they are widely used for banking, social media, e-commerce, etc. Such applications typically leverage a series of *Permissions*, which serve as a convenient abstraction for mediating access to security-sensitive functionality within the Android Ecosystem, e.g., sending data over the Internet. However, several *malicious* applications have recently deployed attacks such as data leaks and spurious credit card charges by *abusing* the Permissions granted initially to them by unaware users in good faith. To alleviate this pressing concern, we present `DyPolDroid`, a dynamic and semi-automated security framework that builds upon Android Enterprise, a device-management framework for organizations, to allow for users and administrators to design and enforce so-called *Counter-Policies*, a convenient user-friendly abstraction to restrict the sets of Permissions granted to potential malicious applications, thus effectively protecting against serious attacks without requiring advanced security and technical expertise. Additionally, as a part of our experimental procedures, we introduce `Laverna`, a fully operational application that uses permissions to provide benign functionality at the same time it also abuses them for malicious purposes. To fully support the reproducibility of our results, and to encourage future work, the source code of both `DyPolDroid` and `Laverna` is publicly available as open-source.

Keywords Permission-abuse attacks · Access control · Android enterprise

1 Introduction

In recent years there has been an increase in the number of malicious applications in the Android Ecosystem (ZDNet, 2020) targeting users with a large variety of attacks, e.g., harvesting private data (The New York Times, 2020), making unwanted credit card charges (Wired, 2020), retrieving the location of users (Android Authority, 2020), etc. Whereas the root causes for such attacks have been largely explored in the literature (Shao et al., 2016), an increasing number of applications look to use and abuse the permissions granted legitimately by users to carry out attacks. These so-called *Permission-Abusing Applications* (PA-Apps) initially pose as *benign* and request users to grant a seemingly normal set of permissions to deliver some *harmless* functionality, e.g., sorting out contact information. However, they later *abuse* the granted permissions to facilitate attacks, e.g., leaking the user's contacts to a remote server via the Internet (Sunday Express, 2020; PC Magazine, 2020).

Also recently, *Android Enterprise* (AE) (Google, 2021a) has emerged as a convenient framework for monitoring and configuring Android devices in a remote fashion, e.g., automatically installing and uninstalling apps and services. These features allow for AE administrators, *AE-Admins* for short, to manage and enforce security policies protecting users and organizations from costly attacks, e.g., by automatically removing *previously-known* malicious apps from devices at once. In such a context, AE-Admins may also want to prevent the deployment of attacks carried out by PA-Apps that are *unknown* beforehand, and may be downloaded and installed on devices by users at any moment of time. However, solving such a problem involves the following challenges:

Matthew Hill is an independent researcher.

✉ Carlos E. Rubio-Medrano
carlos.rubiomedrano@tamucc.edu

[1] Texas A&M University - Corpus Christi, Corpus Christi, TX, USA

[2] Arizona State University, Tempe, AZ, USA

[3] VIT, Chennai, India

[4] Tempe, AZ, USA

1. *Detection.* How to detect *previously-unknown* PA-Apps running on devices?
2. *Prevention.* How to efficiently prevent PA-Apps from carrying out attacks?.
3. *Administration.* How to help AE-Admins to deploy protections against PA-Apps to several different devices in an straightforward and efficient way?
4. *Flexibility.* How to keep protections against PA-Apps up-to-date with respect to changes in the configuration of devices, i.e., the installation of new apps?.
5. *Adoption.* How to protect users from PA-Apps without requiring security expertise and/or modifications to either devices, the OS, or PA-Apps?.

To address these challenges, this paper presents `DyPolDroid` (Dynamic Policies in Android), a dynamic, semi-automated security framework for effectively detecting and neutralizing PA-Apps by means of the following:

1. *Detection.* `DyPolDroid` starts by identifying a series of *Behavioral Patterns*: pairs of Permissions that, if used in combination inside the code of a potential PA-App, may facilitate a successful attack, e.g., combining the `Internet` and `Read-Contacts` permissions to perform a data leak (Arora et al., 2020).
2. *Prevention.* Then, `DyPolDroid` allows for users and AE-Admins to easily write *Counter-Policies* restricting the occurrence of Behavioral Patterns within Android apps. Later, such Counter-Policies are evaluated and translated into *Device Policies*: lists of permissions that are allowed or denied for each potential PA-App, and are sent for enforcement on devices via the AE.
3. *Administration.* Also, `DyPolDroid` allows for AE-Admins to easily con and deploy default security Counter and Device Policies restricting the permissions patterns that may be abused by potential PA-Apps, thus effectively preventing them from carrying out attacks on AE-managed devices.
4. *Flexibility.* In addition, up-to-date information on the specific configuration of each device can be also retrieved by means of the AE, and later leveraged to create custom Counter-Policies that can not only account for previously-unknown, newly-installed PA-Apps, but may also enforce other relevant organizational policies, e.g., restricting gaming apps during office hours.
5. *Adoption.* Finally, `DyPolDroid` requires no manual, user-made configurations of devices, nor it requires modifications to the device OS, the supporting hardware, nor modifications to the code of potential PA-Apps, as required by other approaches in the literature (Vidas et al., 2011; Zachariah et al., 2017), which greatly increases its suitability and convenience for being successfully deployed in practice.

Overall, this paper makes the following contributions:

1. We present a description of PA-Apps, including their relationship with other types of malicious apps for Android that have been studied in the literature.
2. We introduce `DyPolDroid`, which provides an effective solution for counter-acting PA-Apps at the same time it offers an convenient degree of automation that requires no advanced security expertise from either users or AE-Admins.
3. As a part of our experimental procedure, we also introduce `Laverna`, a fully operational PA-App, which uses permissions to provide benign functionality, e.g., send automated text messages to phone contacts, at the same time it also abuses them for malicious purposes, e.g., leaking the name and phone of all contacts to a remote server over the Internet.
4. Finally, to support the reproducibility of our experimental results, and to encourage future work based on our reported findings, the source code of both `DyPolDroid` and `Laverna` are publicly available as open-source (Hill & Rubio-Medrano, 2021).

This paper is organized as follows: Section 2 presents some background on the technologies later explored in the paper, and provides a concise definition of the problem that is then later addressed in Section 3. We provide a description of a preliminary procedure we have conducted to evaluate the effectiveness of `DyPolDroid` in Section 4. We review the related work in Section 5 and then discuss some future work and conclude the paper in Section 6. A preliminary version of this paper appeared in the Proceedings of the International Conference on Secure Knowledge in the Artificial Intelligence Era (SKM) 2021 (Rubio-Medrano et al., 2020a), and as a poster abstract in the Proceedings of the 6th IEEE European Symposium on Security and Privacy 2021 (Euro S&P 2021) conference (Rubio-Medrano et al., 2020b).

2 Background and Problem Statement

2.1 Android Permissions

In the Android Ecosystem, apps must request and obtain so-called *Permissions*, which serve as convenient abstractions for mediating access to the resources of the host device, e.g., sending data over the Internet, turning the camera on and off, sending SMS texts and calls, etc. Android Permissions have been extensively studied in the literature, and have seen a number of changes over the years (Felt et al., 2011; Felt et al., 2012a; Ramachandran et al., 2017). Historically, there are two major recognizable eras: the *all-or-nothing* era, and the *run-time* era. Prior to Android 6.0, all permissions

requested by an app needed to be granted by users at installation time; users were presented with a list of permissions to accept or deny once the app have been downloaded but before installation could begin. If users would choose to deny the requested permissions, the installation of the app would fail. With the release of Android 6.0, the permission model was modified such that apps needed to request access to a permission the first time that they wanted to use it, which allowed for a more fine-grained approach in which users would accept or reject each permission individually (Google, 2021b). Finally, once a permission is granted to an app, it can be used repeatedly by the instructions of the app's code to access the functionality *guarded* by it, e.g., using the `Internet` permission.

Android system permissions are divided into *normal*, *signature* and *dangerous* permissions (Android Developers Reference, 2022). Android apps declare the requested permissions in the apps Manifest.xml. Android by default allows normal permissions requested in the Manifest.xml, such as giving apps access to the internet. The inherent idea behind granting normal permissions implies that apps granted with normal permissions shouldn't pose a privacy threat. Similarly, the signature permissions are granted by default, if the requesting application is signed with the same certificate as the application that declared the permission. Android provides 46 normal permissions and 49 signature permissions that can be used by an application.

On the other hand, the dangerous permissions require explicit permission approval from the device user. The dangerous permissions include accessing sensitive information like the device location, contacts, and more. Android provides 41 dangerous permissions that can be used by an application, which may open the door for attacks.

2.2 Android Enterprise

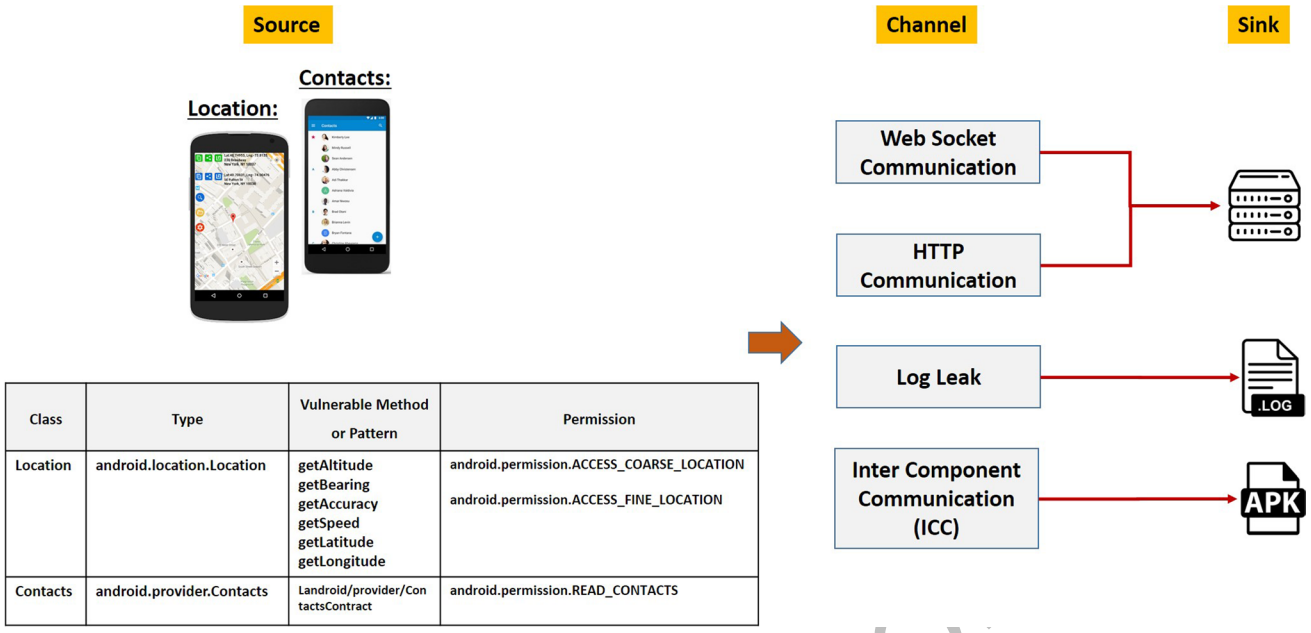
Android Enterprise (AE) is a device management framework that allows for organizations to remotely monitor and con Android-run devices, e.g., automatically installing and uninstalling apps without extensive user intervention (Google, 2021a). In addition, for security purposes, AE leverages the permission model described before to dynamically update, e.g., grant or deny, the permissions requested by individual apps, thus allowing for AE administrators to remotely allow or restrict the functionality of the apps installed on a managed device at will. Devices can be remotely managed in two different modes: in the *Fully-Managed* mode, devices may have their configurations set remotely by an AE administrator, leaving little room for users to change the settings of the device. Alternatively, in the *Bring-Your-Own-Device* (BYOD) mode, devices may allow for two different profiles to be cond and co-exist inside a device: a *work* profile fully controlled by an AE as described before, and a *user* profile that can be left for users to con at will, e.g., downloading and installing apps at will.

In addition, leveraging the features provided by AE, administrators can also obtain real-time device configuration data, which may allow them to dynamically send and install, a.k.a., *push*, customized, app-specific permissions on the device depending on the current configuration and any other related context information. This introduced a convenient approach for remote security management that removes the need of instrumenting the device itself, the device OS, the code of apps (APK files), or any other supporting API, as required by previous approaches in the literature (Enck, 2020). However, this approach for remotely updating permissions may be in fact limited by the network bandwidth available to the device at a given moment of time, which may affect the deployment of immediately needed changes, e.g., denying permissions to a potentially malicious app that has been just detected by AE as installed in the managed device. Also, AE is currently available to devices running versions of Android greater than 5.0.*, and the BYOD mode discussed before is only available to versions of Android running an API level 23 to 29. For the purposes of this paper, we will assume the devices implementing our approach are managed by an existing AE, follow the Android version features just mentioned, and implement either the fully-managed mode or the BYOD mode with a work profile as discussed before.

2.3 The Behavior of Android Applications

For the purposes of this paper, we define *Application Behavior*, or simply *Behavior* for short, as any functionality depicted by an app when executed. Examples include, but are not limited to: gaming, social networking, picture-taking, etc., Conversely, an *Attack* is a well-recognized and highly-undesirable behavior, which may have a negative effect on the user and/or the device. Illustrative examples may include the violation of user privacy via leaking of user contacts, or a financial affectation via unwanted texts or calls.

Having said this, an app is said to be *Benign* if it strictly provides the behavior expected by the user, as stated either by means of a formal or informal documentations and/or descriptions, without causing any affectation to the user or the device. In contrast, a *Malicious* app attempts to subvert the normal, intended use of the expected behavior in an attempt to cause an unwanted affectation either to the user or the device itself (Vidas et al., 2011; Zachariah et al., 2017). In addition, an *Over-Privileged* app requests more permissions than the ones needed to provide its expected benign behavior, and can either neglect such *extra* permissions, thus staying as a benign app, or can actively use them in a malicious way (Wei et al., 2012; Wang et al., 2015; Calciati & Gorla, 2017; Wu & Liu, 2019).



Class	Type	Vulnerable Method or Pattern	Permission
Location	android.location.Location	getAltitude getBearing getAccuracy getSpeed getLatitude getLongitude	android.permission.ACCESS_COARSE_LOCATION android.permission.ACCESS_FINE_LOCATION
Contacts	android.provider.Contacts	Landroid/provider/ContactsContract	android.permission.READ_CONTACTS

Fig. 1 PA-Apps Abusing Android Permissions to Leak User's Contact Information

Finally, a *Permission-Abusing* app (PA-App) is a seemingly benign app that is also secretly malicious: its formal or informal usage documentation states that it uses permissions in an expected, harm-free way, e.g., for sending messages to contacts via the Internet, but it may also use them in a malicious, unwanted, and potentially user-harming way as well, e.g., for leaking contacts data to a remote server (Wired, 2020), installing tracking software (Android Authority, 2020) or collecting user data (The New York Times, 2020).

2.4 Problem Statement

For the purposes of this paper, we assert that apps that request access to permissions and knowingly misuse them are malicious, i.e., they are PA-Apps, as such permissions may allow for them to successfully carry out attacks. Therefore, *we aim to detect all potential apps installed on devices that may be PA-Apps, and we also aim to prevent them from successfully exploiting any granted permissions at run-time.* Figure 1 illustrates a series of PA-Apps in which a set of vulnerable, i.e., *dangerous* permissions are abused to leak the users' private information like location and contacts through web sockets, log leaks, and Inter Component Communication (ICC). In Section 4.1 we present the results of an empirical study we conducted to show that PA-Apps are available and can be downloaded by users in a major application distribution venue.

Finally, for the sake of clarity, detecting all potential over-privileged apps that may or may not be malicious is out of the scope of this paper, as shown in Fig. 2. Also, the detection of other malicious Android apps that carry out attacks by means of other techniques other than the abuse of permissions, e.g., dynamic library updates (Zhauniarovich et al., 2015), is also out of scope.

3 Our Approach: Dynamic Permission Updates for Potential PA-Apps via the AE

To address the problem just described, we now introduce `DyPolDroid` (Dynamic Policies in Android): a dynamic security framework shown in Fig. 3, in which both users and AE-Admins can actively restrict the behavior of PA-Apps, thus preventing the occurrence of attacks in Android devices.

We start in Section 3.1 by introducing the concept of *Behavioral Patterns*: pairs of permissions which, if used together within an app's code, may facilitate permission-abusing attacks. Then, we move on to describe in Section 3.2 how users and AE-Admins can write so-called *Counter-Policies* for restricting Behavioral Patterns in Android apps. As it is further described in Section 3.3, such patterns are in turn discovered by analyzing the data flow of potential PA-Apps installed on a device, and represent a key component for ultimately producing so-called *Device Policies*, which, as it will be shown in Section 3.4, are subsequently enforced by leveraging the dynamic permission updates provided by the AE.

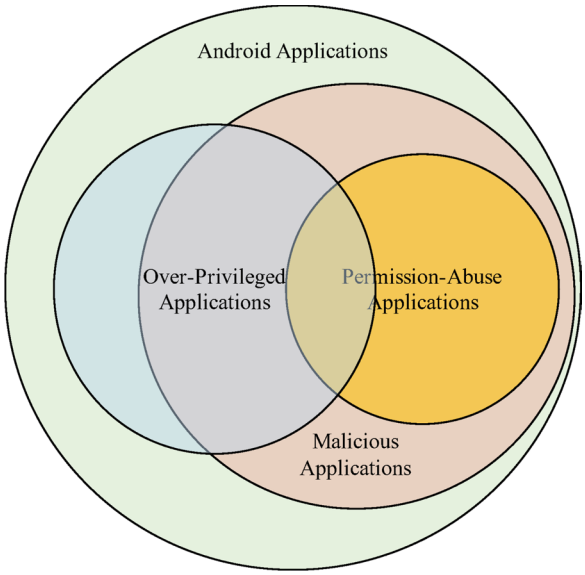


Fig. 2 Classifying Apps in Android based on Behavior. In this paper, we are interested in detecting and neutralizing PA-Apps, which are always regarded as malicious

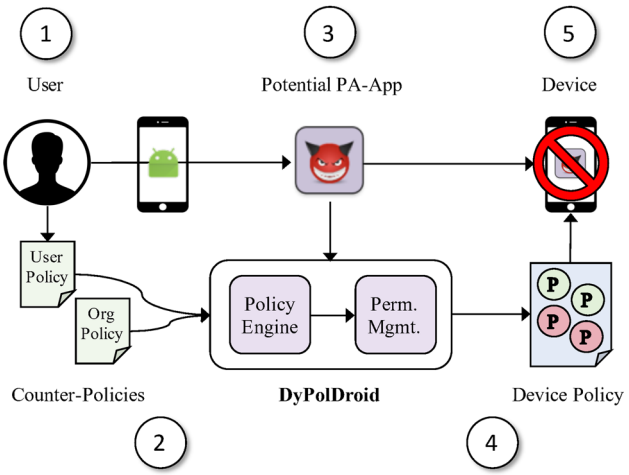


Fig. 3 How DyPolDroid Works: a User signs up for an Android Enterprise (1) and moves on to write Counter-Policies (2), which are later evaluated against the Behavioral Patterns obtained from any installed PA-Apps (3), producing a Device Policy that is then sent to the Device (4). As a result, PA-Apps have their permissions blocked (5)

3.1 Behavioral Patterns

Following Section 2.1, we define a *Behavioral Pattern* as a sequence of permissions required by apps to execute either a benign behavior or an attack (Zhang et al., 2013; Arora et al., 2020). As an example, the gaming behavior may include the pattern: (CAMERA, INTERNET), whereas a contact-leaking attack may require a pattern such as (READ_CONTACTS, INTERNET).

Android apps, including PA-Apps, may in turn depict different behavioral patterns, and there may be an overlap between the permissions exhibited in benign and Behavioral Patterns, e.g., the INTERNET permission being simultaneously used for sending messages (benign) and leaking private data (attack) as just discussed.

3.2 Writing Counter-Policies

Initially, Counter-Policies are written using a series of *templates* depicting a subset of XACML, the *de facto* language for authorization and access control (OASIS Standard, 2013). Users and AE-Admins are then able to protect their device by specifying a variety of rules including features like: which applications can be installed, the default permission policy of any newly installed application, and what potential attacks the user would like to defend against. More interestingly, rules may also include what Behavioral Patterns may be allowed for Android apps that are installed on the device in the future. As an example, Listing 1 shows an excerpt of a Counter-Policy for Laverna, a self-developed PA-App that will be featured in Section 4. Two Behavioral Patterns, namely, *Steal_Contacts* and *Steal_Messages*, which correspond to the namesake attacks, are specified in lines 7-10 and 11-14. Figure 4 presents a graphical depiction of the process just discussed: Behavioral Patterns can be leveraged to construct custom Counter-Policies, which are then subsequently processed by DyPolDroid to create Device Policies. In addition, Counter-Policies leverage the conflict resolution features provided by XACML for the case when multiple policies are applied to the same device, allowing for conflicts to be resolved before policies are sent to the user's device, as shown in Fig. 4 (2).

Counter-Policies are further subdivided into two different hierarchical levels. First, *Level 1* Counter-Policies are intended to restrict the behavioral patterns depicted by potential PA-Apps. Listing 2 shows an example of two Level 1 Counter-Policies using a simplified, reader-friendly notation, which are then used to define two Behavioral Patterns: *Steal_Contacts* and *Gaming*, which are in turn depicted by different pairings of the READ_PERMISSION, INTERNET, and CAMERA permissions. In addition, an additional *Level 2* of Counter-Policies is also introduced by leveraging *Attribute-based Access Control* (ABAC) (Chung et al., 2019), which, besides being the underlying theoretical model behind XACML, has also been found to be convenient for specifying rich and flexible policies in the context of other emerging technologies such as *Augmented Reality* (Rubio-Medrano et al., 2019). This way, using these two levels of abstraction, our behavioral patterns may be defined separately as a part of

```
<Rule RuleId="Laverna_Attacks" Effect="Deny">
 <Target>
  <AnyOf> <AllOf> <Match Id="boolean-equal">
   <AttributeValue>true</AttributeValue>
    <AttributeDesignator AttributeId="Laverna"/>
  </Match> </AllOf> </AnyOf>
  <AnyOf> <AllOf> <Match Id="boolean-equal">
   <AttributeValue>true</AttributeValue>
   <AttributeDesignator AttributeId="Steal_Contacts"/>
  </Match> </AllOf>
   <AllOf><Match Id="boolean-equal">
    <AttributeValue>true</AttributeValue>
    <AttributeDesignator AttributeId="Steal_Messages"/>
  </Match></AllOf> </AnyOf>
 </Target>
</Rule>
```

Listing 1 A Counter-Policy for the **Laverna** PA-App

Level 1, and then, AE-Admins and end-users may leverage Level 2 then prepare their own restrictions using ABAC as a part of Level 2.

As an example, Listing 3 shows three sample Level 2 Counter-Policies, which, besides referring to the Behavioral Patterns defined in Listing 2, also introduce three different attributes: `A_1`, which denotes a convenient time abstraction for events happening all the time; `A_2`, which denotes a set of categories for different Android apps, e.g., *Gaming*; and `A_3`, which denotes a time range abstraction for the standard working hours in the United States. Following this notation, the `L_2_P_1` Level 2 Counter-Policy restricts the occurrence of the Behavioral Pattern labeled as `BP_1` (*Steal_Contacts*) at all times. Conversely, the `L_2_P_2` and `L_2_P_2` Counter-Policies allow for the `BP_2` (*Gaming*) Behavioral Pattern to occur during hours outside regular working hours only.

As an example, Listing 3 shows three sample Level 2 Counter-Policies, which, besides referring to the Behavioral Patterns defined in Listing 2, also introduce three different attributes: `A_1`, which denotes a convenient time abstraction for events happening all the time; `A_2`, which denotes a set of categories for different Android apps, e.g., *Gaming*; and `A_3`, which denotes a time range abstraction for the standard working hours in the United States. Following this notation, the `L_2_P_1` Level 2 Counter-Policy restricts the occurrence of the Behavioral Pattern labeled as `BP_1` (*Steal_Contacts*) at all times. Conversely, the `L_2_P_2` and `L_2_P_2` Counter-Policies allow for the `BP_2` (*Gaming*) Behavioral Pattern to occur during hours outside regular working hours only.

In our implementation, which is further discussed in Section 3.5, both Level 1 and Level 2 Counter-Policies can be evaluated *online*, e.g., while a potential PA-App is being analyzed by means of the procedure defined in Section 3.3, or can also

```
L_1_P_1: [BP_1 (Steal_Contacts): (READ_CONTACTS, INTERNET)];
L_1_P_2: [BP_2 (Gaming): (CAMERA, INTERNET)];
```

Listing 2 Sample Level 1 Counter-Policies

```
1 L_2_P_1: [BP_1, A_1: <Anytime, True> -> Deny];
2 L_2_P_2: [BP_2, A_2: <Category, Games>, A_3: <Time, 9-5> -> Deny];
3 L_2_P_3: [BP_2, A_2: <Category, Games>, A_3: <Time, 9-5> -> Allow];
```

Listing 3 Sample Level 2 Counter-Policies

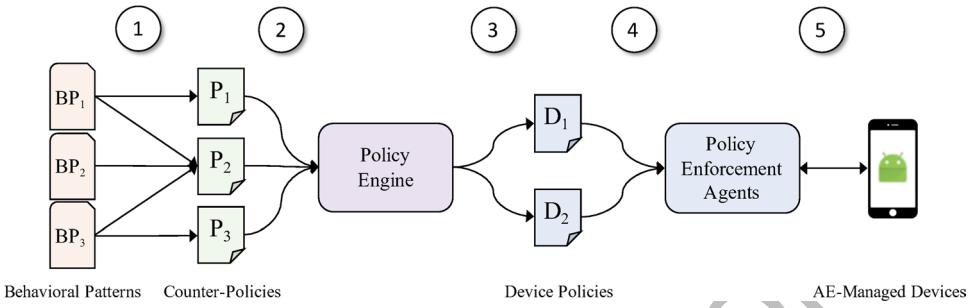


Fig. 4 From Behavioral Patterns to Device Policies: Templates describing Behavioral Patterns are leveraged by users and AE-Admins to write Counter-Policies (1), which are then fed as an input to DyPolDroid's Policy Engine (2), so they can be turned into Device Policies (3). Later, Device Policies are handled by a Policy Enforcement Agent (4), which also retrieves up-to-date device configuration data from the Device (5)

be preemptively evaluated following an *offline* mode, in which relevant policies are retrieved and evaluated in advance, and the evaluation results are stored for faster future processing. In addition, runtime values for the attributes like the ones featured on Listing 3 are collected from a variety of online sources and forwarded to the Dynamic Policy Engine featured as a part of our architectural description presented in Section 3.5.3.

3.3 Discovering Behavioral Patterns

Our Behavioral Patterns are inspired by a set of predetermined attack vectors that were found to be common place across a number of known malicious apps (Arora et al., 2020). Those vectors can be represented as a sequence of instructions mapping data from a *source* instruction to a *sink* instruction within the app's code. Normally, both source and sink instructions will include a function call to an Android Class Function (ACF) performing a *sensitive* functionality operation, which will be in turn *guarded* by a given Android Permission. For example, the Behavioral Pattern: (READ_CONTACTS, INTERNET), may be depicted within a PA-App code as a sequence of instructions depicting the flow of sensitive data, e.g., the user's contact information, in which the first instruction extracts the contacts (source) and the last one sends them to a remote server via the Internet (sink).

To detect the occurrence of Behavioral Patterns within potential PA-Apps, DyPolDroid leverages *Taint Tracking* (Zhu et al., 2011), a well-known data flow analysis technique. Initially, data flow sequences are obtained from the APK file of the PA-App. Then, for each sequence, its source and sink instructions are cross-referenced against a list containing a series of *mappings* between ACFs and the Permissions such ACFs require for successful execution, as mentioned before. If the permissions mapped to both the sink and source instructions are found to depict a Behavioral Pattern *P*, then the permissions included in *P* are returned as a result for further processing, as detailed next.

Following the definition described earlier in this section, both source and sink instruction may in turn require specific permission to fulfill their intended purpose. As an example, a source instruction reading contacts may require the aforementioned READ_CONTACTS whereas a sink instruction sending data out through the internet may in turn require the INTERNET permission).

3.4 Generating and Enforcing Device Policies

Figure 5 and Algorithm 1 provides an overview of how Device Policies are created. First, the set of *authorized* permissions is calculated by evaluating the Counter-Policies that may be relevant under the current context, e.g., the AE, the organization, the user, the device, etc. (Algorithm 1, Line 5). Second, the set of *observed* permissions, as depicted by the code of a potential PA-App, is obtained by means of the procedure described in the previous Section. Third, the set of *resulting* permissions is obtained by intersecting the sets of authorized and observed permissions (Algorithm 1, Line 8). These resulting permissions are then updated within the

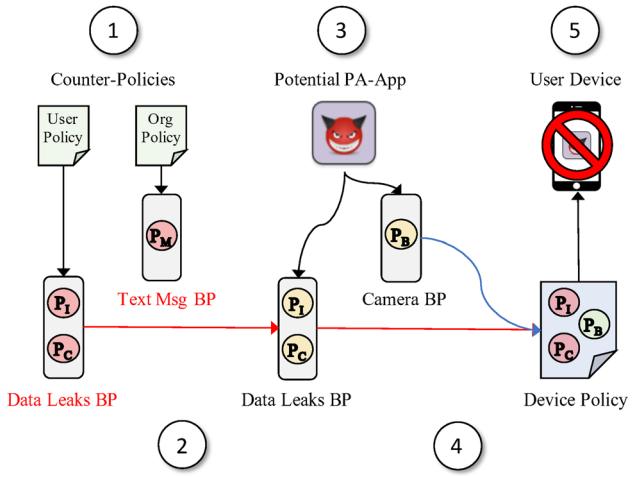


Fig. 5 Creating Device Policies in DyPolDroid. The set of *authorized* permissions from each Behavioral Pattern is obtained by evaluating Counter-Policies (1)(2), whereas the set of *observed* permissions is obtained via Taint Tracking analysis on potential PA-Apps (3). Later, the set of *resulting* permissions is calculated by comparing the denied and the requested permissions, and it is later encoded as a Device Policy (4), which is set out to the Device for enforcement via the AE (5)

Device Policy to allow or block their future usage (Algorithm 1, Line 11). Listing 10 shows a sample Device Policy that blocks the READ_CONTACTS (lines 6-7) and READ_SMS (lines 8-9) permissions for the Laverna PA-App that will be discussed in Section 4.

Listing 4 shows an example of two different Device Policies and three Behavioral Patterns, which are in turn composed of a set of permissions that have been observed within a potential PA-App. Taking Listings 2 and 3 as a reference, the DP_1 Device Policy (lines 3-5) denies the READ_CONTACTS, CAMERA, and INTERNET permissions, at the same time it also authorizes the MESSAGING permission to the Android app identified by App_1 when the three attributes labeled as A_1, A_2, and A_3 are evaluated to True. Conversely, the DP_2 Device Policy (lines 6-7) denies the READ_CONTACTS permission while authorizing the CAMERA, INTERNET, and MESSAGING permissions to the same app when the A_1 and the A_2 and the A_3 attributes evaluate to True and False respectively.

Finally, once a newly-generated Device Policy is received by the AE, it is forwarded to the device following the procedures described in Section 2.2. Once received, the policy will immediately come into effect. If there are any conflicts between the user's device and the new-applied policy, e.g., an installed application is not allowed by the policy, the device manager will freeze the profile until the device is compliant with the policy, e.g., forcing the user to manually uninstall the offending PA-App. DyPolDroid uses a SHA 256 hash in conjunction with the application package to ensure that if different versions of the same potential PA-App are installed, only matching apps have the appropriate actions taken against them. This is important when there are multiple versions of the same app installed on devices for different users, e.g. v1.1.33 and v1.1.34.

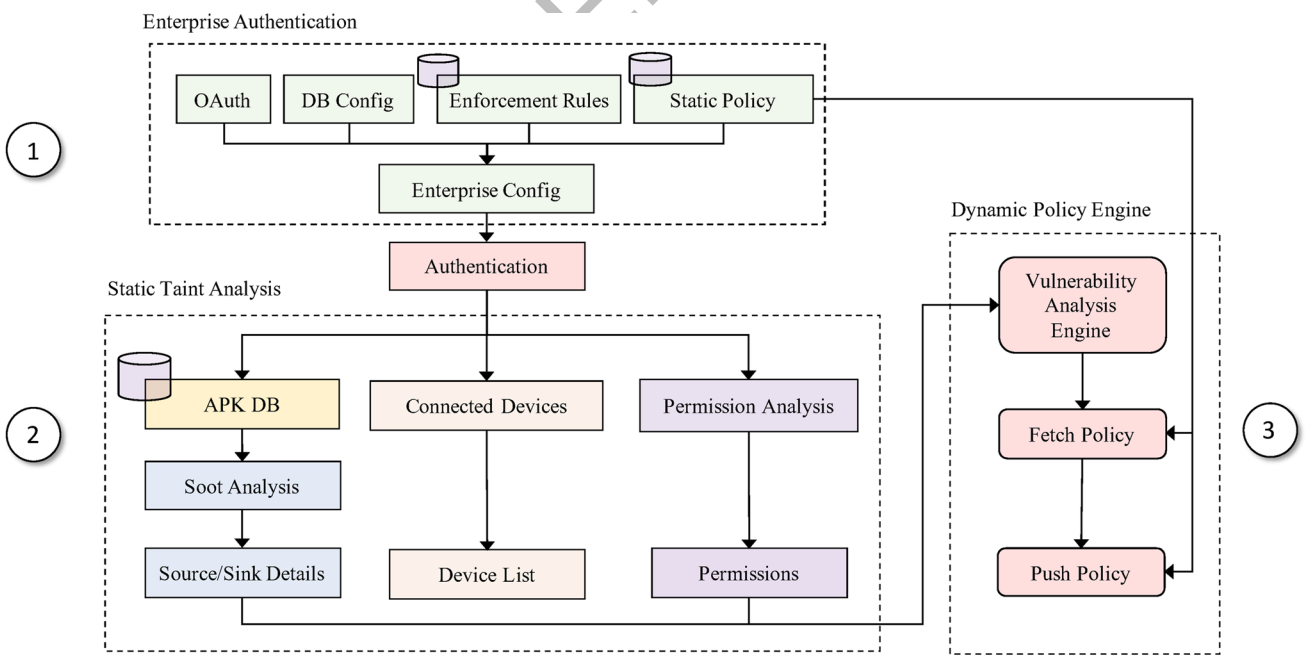


Fig. 6 The DyPolDroid Architecture. The Enterprise Authentication Module is in charge of retrieving information from client devices leveraging the AE configuration (1). The Static Taint Analysis module (2) parses the APK files of potential PA-Apps and maps pairs of sink and source instructions into Android Permissions (2). Finally, the Dynamic Policy Engine is in charge of fetching and updating counter and device policies into client devices (3)

Algorithm 1 Generating Device Policies from Counter Policies

Require: A Set of Counter Policies C, A Set of Permissions O_{perms}
Ensure: A Device Policy D
1: $N \leftarrow size(C)$
2: $res_{perms} \leftarrow 0$
3: **while** $N \neq 0$ **do**
4: $c \leftarrow next(C)$
5: $auth_{perms} \leftarrow evaluate(c)$ ▷ Evaluating policy as discussed in Sec. 3.2
6: $S \leftarrow size(auth_{perms})$
7: **if** $S \neq 0$ **then**
8: $res_{perms} \leftarrow res_{perms} \cup (auth_{perms} \cap O_{perms})$
9: **end if**
10: **end while**
11: $D \leftarrow encode(res_{perms})$ ▷ Generates encoding as shown in Listing 4

```
1 BP_1: (READ_CONTACTS, INTERNET);
2 BP_2: (CAMERA, INTERNET);
3 BP_3: (MESSAGING);
4 DP_1 (A_1, A_2, A_3):
5    [App_1:(Deny: READ_CONTACTS, CAMERA, INTERNET, Allow: MESSAGING)];
6 DP_2 (A_1, A_2, !A_3):
7    [App_1:(Deny: READ_CONTACTS, Allow: CAMERA, INTERNET, MESSAGING)];
```

Listing 4 Sample Device Policies

3.5 Implementation and Architectural Details

In order to efficiently implement the features discussed in this section, DyPolDroid has been developed as a server-side solution consisting of three major modules, which are graphically featured in Fig. 6: an Enterprise Authentication Module (1), a Static Taint Analysis Module (2), and a Dynamic Policy Engine Module (3). In the rest of this subsection, we provide extended details, including same code snippets, on each of these modules.

3.5.1 The Enterprise Authentication Module

For the purposes of authentication, DyPolDroid uses the OAuth2 (IETF OAuth Working Group, 2022) industry-standard protocol, which allows for obtaining access to information authorized by the end-user via a set of AE-issued credentials. Listing 5 shows a sample OAuth2 configuration, where every enterprise account has a unique client_ID (line 2), project_id (line 3), client_secret (line 7) and redirect_uris (line 8). Redirect URIs are the

callback used by the authorization server after a user successfully authorizes an application (Line 8). The Enforcement rules initialize the default enterprise configuration for the Android enterprise. The enterprise configuration contains information of the devices and apps databases. Each application has different settings like default permission, policy enforcement (i.e. installation time, for every access etc). Finally, DyPolDroid checks if there exists a rule for the application matching the given package name, and adds the permission to the list of application permissions if it does not already exist in the apps database. By default, any application matching the package name, which happens to be violating the given policy, is removed.

3.5.2 The Static Taint Analysis Module

The static taint analysis module fetches the list of devices connected to the given AE account and the associated APK's and the policy configuration. The extracted details are then updated to the database. The downloaded APK's are statically analyzed using the well-known Soot Framework (Vallée-Rai et al., 1999; Paderborn University, 2022), which is leveraged to provide source and sink components relations and their permissions as an XML output file. This XML file is then analyzed to check for any overlap between the sources and

sinks and the mapping of Android permissions to API calls. Listing 6 is the core function that extracts all the permissions that are required directly/indirectly by the

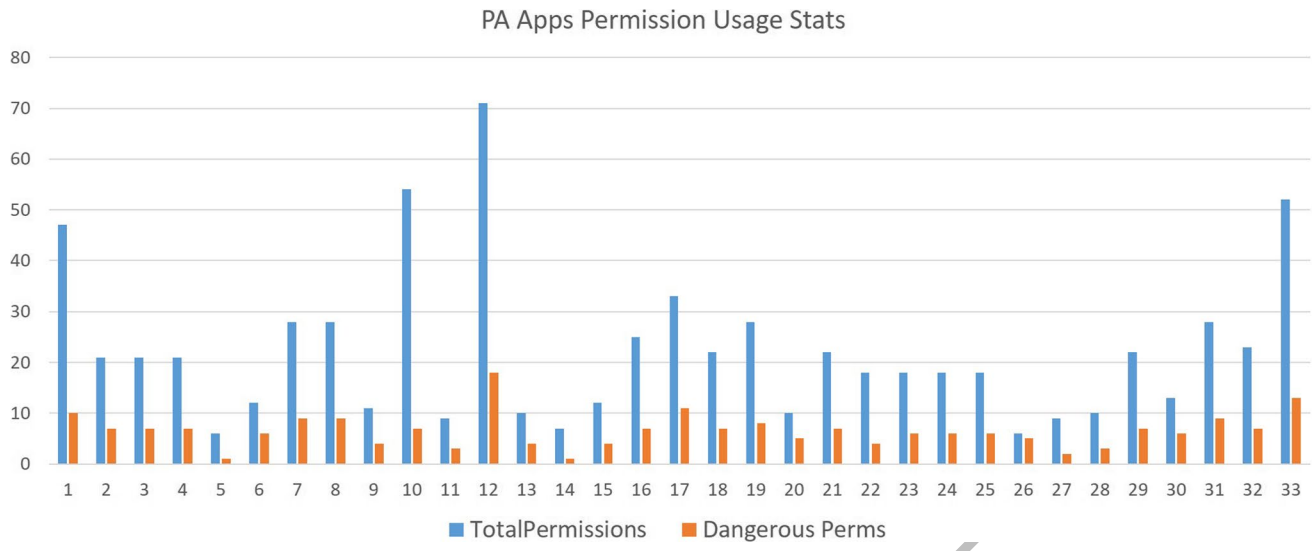


Fig. 7 PA apps dangerous permission usage stats

potential PA-App. Lines (4-7) extract the package name, return type, function name and the function arguments for every sink statement. Later, line 10 appends all the permissions to a list data structure. Details on such mapping are provided in the next section.

3.5.3 The Dynamic Policy Engine Module

As described in Section 3.3, a pre-calculated mapping between APK instructions and Android permissions is used to obtain the list of permissions associated with a given sink instruction. In the current implementation of `DyPolDroid`, such mapping was obtained by combining the mappings provided by previous projects in the literature, namely, Axplorer (Backes et al., 2016) and PScout (Au et al., 2012). In such regard, Listing 7 checks for a list of *vulnerabilities* based on the aforementioned mapping list. Lines (7-15) iterate through the permission list to verify for *vulnerable* calls, and upon detection the corresponding Android API permission is mapped to a list. Finally, the identified vulnerability is published back to the AE account using the function `PushPolicy(policyname, modified_policy)`, as shown in Listing 8). Line 2 publishes the policy that handles the vulnerability back to the enterprise server using the Android Management API. Thus all of the devices that use the policy will be automatically updated with the newly modified one. Lines (5-9) format the modified policy and update it in the local file system.


```
1 {"web":{
2 "client_id":"---",
3 "project_id":"---",
4 "auth_uri":"https://accounts.google.com/o/oauth2/auth",
5 "token_uri":"https://oauth2.googleapis.com/token",
6 "auth_provider_x509_cert_url":"https://www.googleapis.com/oauth2/v1/
      certs",
7 "client_secret":"--",
8 "redirect_uris":["https://example.com"]}}
```

Listing 5 Sample OAuth2 Configuration File displaying Credentials

```
func getPermissionList(apkData apk.DataFlowResult) []mapping.
    Permission {
    var permissionList []mapping.Permission
    for _, apk := range apkData.Results {
        pkgName := getPackage(apk.Sink.Statement)
        rtnType := getReturnType(apk.Sink.Statement)
        funcName := getFunctionName(apk.Sink.Statement)
        argv := getArguments(apk.Sink.Statement)

        addPermission := mapping.GeneratePermission("PERMISSION",
            pkgName, rtnType, funcName, argv)
        permissionList = append(permissionList, addPermission)
    }
    return permissionList
}
```

Listing 6 Extracting Android Permissions from a given APK File


```
func GetPotenialVulnerabilities(apkData apk.DataFlowResult,
    apiData mapping.PermissionMapping) bool {

    permissionList := getPermissionList(apkData)
    var vulnerableCalls []mapping.Permission

    for _, apk := range permissionList {
        for _, api := range apiData.Permission.Permission {
            res := equals(apk, api)
            if res {
                fmt.Println("Permission:", api.PermissionName)
                vulnerableCalls = append(vulnerableCalls, api)
            }}}
    return len(vulnerableCalls) != 0;}
```

Listing 7 Mapping Android Permissions to Call Vulnerabilities

```
1 func PushPolicy(pol *androidmanagement.Policy, am *androidmanagement
      .Service) {
2     pol, err := am.Enterprises.Policies.Patch(pol.Name, pol).Do()
3     handelError(err)
4     js, err := json.MarshalIndent(pol, "", " ")
5     handelError(err)
6     err = ioutil.WriteFile("test.json", js, 0644)
7     handelError(err)}
```

Listing 8 Updating and Publishing Device Policies

3.5.4 Device Policy Definition Language

Finally, Listing 9 provides a JSON snippet of the language that is used to create Device Policies. The `packageName` attribute (line 1) specifies the target PA-App. Attribute `onMalicious` (line 3) defines the action to be taken if a PA-App is identified; whereas attribute `permissionGrants` (lines 7-11) defines the actions to be taken for every permission declared in the AndroidManifest.xml file of an offending PA-App.

4 Evaluation

In order to explore the existence of PA-Apps in the *wild*, e.g., publicly available for users to download and install, in Section 4.1 we present the results of an empirical study in which we randomly collected and analyzed a series of Android applications from Google Playstore. Next, in Section 4.2 we analyzed each of the discovered PA-Apps in our study by using our `DyPolDroid` tool, in an effort to restrict the *dangerous* Android permissions that may allow for attacks to take place.


```
1  "packageName": "com.some.android.package",
2  "defaultPermissionPolicy": "DENY|GRANT|PROMPT",
3  "onMalicious": "BLOCK_PERMISSION|BLOCK_APP|IGNORE",
4      "installType": "REQUIRED|ALLOW|BLOCK|FORCED",
5      "defaultPermissionPolicy": "DENY|GRANT|PROMPT",
6      "onMalicious": "BLOCK_PERMISSION|BLOCK_APP|IGNORE",
7      "permissionGrants": [
8          {
9              "permission": "android.permission.PERMISSION_NAME",
10             "policy": "DENY|GRANT|PROMPT"
11         }
12     ]
```

Listing 9 Policy Definition Language

Table 1 PA-Apps as found on Google Playstore

SNo.	Hash of APK	No. of Installs	Updated on
1	3e00cf26d79e4f25327c176d298d00fe	100000000	Mar 8 2022
2	f96f7b7f784fefccd1ddc6dece92daf9	5000000	Feb 22 2021
3	ee91ddf83673a983c106856432062a94	5000000	Apr 28 2022
4	606595f7c2bca435b6a098021ab2b823	100000	Sep 16 2020
5	0089b3242909dbf1b9c7c36164fff708	100000	Feb 17 2019
6	0d26533ac02974832a881b02bdd7e7cd	100000	Jun 3 2022
7	e2b5aad9127d5b34be165b7820b07599	10000	Mar 31 2020
8	1c1356984bfb2d23edd9f4826b3dbfe2	10000	Apr 30 2020
9	3b738a6c94f526ec8314e20b5adb94df	10000	May 16 2022
10	9630bffab41182c851d465aaa2a1e684	10000	Oct 29 2020
11	14b598607142a065c79a26de5e0019b7	10000	Feb 20 2019
12	6347b633eb1c8967db9afe55233b5db9	5000	Jun 8 2022
13	4007e7c0638ab37110746c16d8bb95e4	5000	Mar 17 2020
14	11042e26167328fca5ac77083c45b874	1000	Jan 28 2020
15	8cb0304450d0a15ecd034827b964c775	1000	Mar 5 2019
16	3e38e5cd41ccc81a66a0761a613896aa	1000	Jun 7 2019
17	eaea166cc6a08d6d5fc44dba59416c19	1000	Apr 23 2020
18	60e867366a6e6be58fa5e55bd747153f	1000	May 4 2021
19	a63339585410c122c2cd742a1d29e0f4	1000	02-Aug-18
20	b1a1f9edc4861f98c3704e0fcab7f751	500	Mar 31 2020
21	4c1252c14daf168b73a966acc32796c3	500	Oct 31 2020
22	15b2b7d4be30c0617e346903e8ca8d68	100	Oct 14 2021
23	c45cede410d96ba6112883265b65d3e4	100	Mar 31 2020
24	4ad37e3a6ee5b8b8a7056ef03a647ab6	100	Nov 4 2020
25	34498dfb91e484ec156d62c9802b8663	100	Nov 12 2020
26	f9034531cbab9217c1ecf0a4ba3dc86d	100	Sep 5 2020
27	4f66545f84675cc7d14a73c99125185c	50	May 16 2022
28	9015402bca5f423b2222305418804dd2	10	Feb 26 2020
29	7b557ff760f8ff88ecb32c805af03b6c	10	Jan 30 2021
30	478f3b4271d93cf68a111f74a95ca145	10	Mar 8 2021
31	2be391b0be83cc5abd59372065059390	10	Feb 14 2021
32	10b1b6a92945a2d2eeeced1b5267de39	1	Jun 14 2018
33	e1f6c0919617c823286274e6025484b4	1	Aug 2 2018

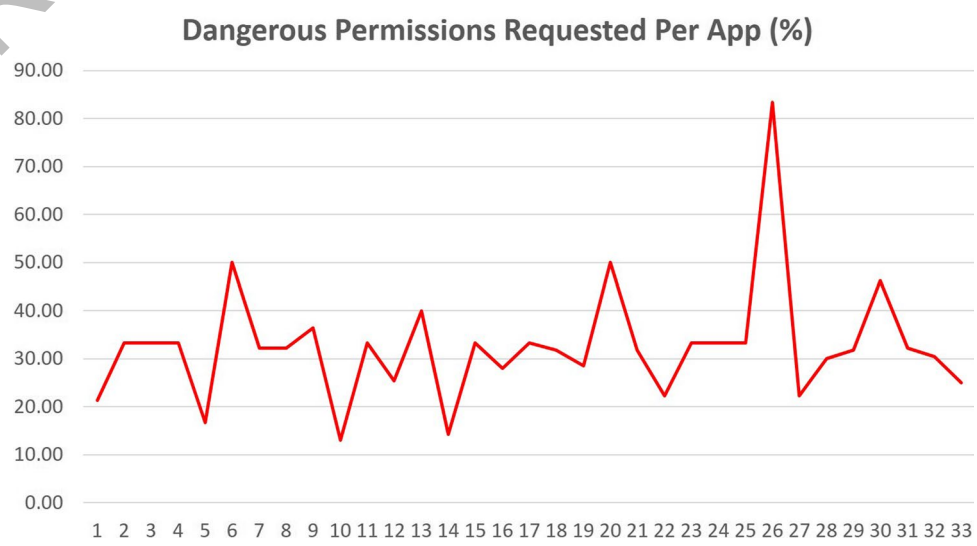


Fig. 8 Per App Dangerous permission percentage

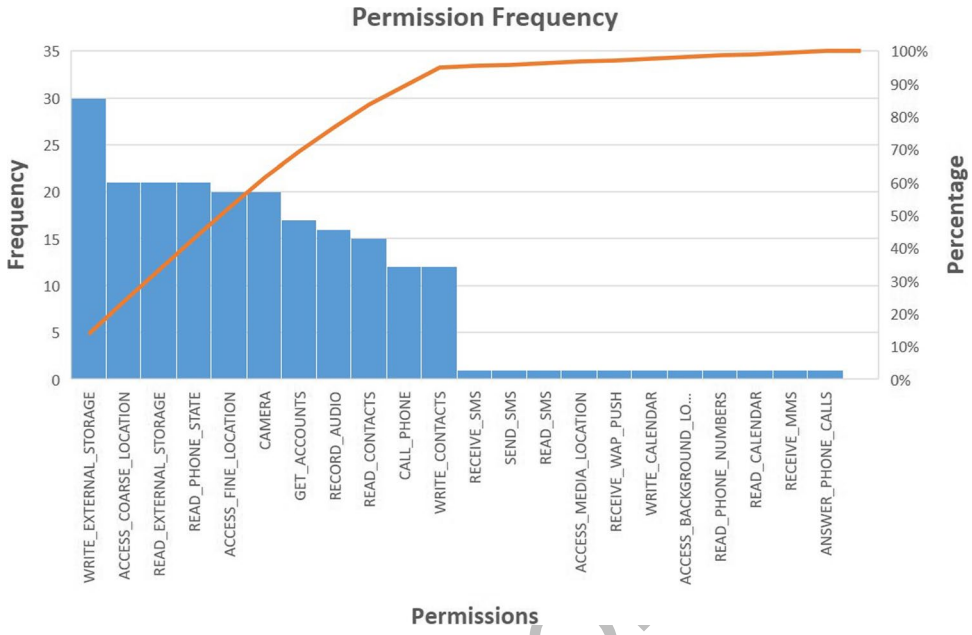


Fig. 9 Pareto chart displaying the frequency of permissions used by PA apps

Finally, in Section 4.3, we report our experimental results on `Laverna`: a self-developed PA-App that requests several permissions for benign functioning, e.g., getting full access to the user's contacts, at the same time it also exploits such permissions to leak data to a remote server.

4.1 An Empirical Study on PA-Apps

For our study, we first created a dataset consisting of 7978 randomly-selected applications from Google Playstore. Our analysis then started by defining a PA definition file, which specifies various source and sink patterns that are the characteristics of a potential PA-App, a.k.a., Behavioral Patterns (Section 3.1). The PA definition file was then fed into the FlowDroid data flow analysis tool (Steven Arzt, 2022) to statically analyze each app in our dataset. As a result, we found that 33 out of 7978 apps depict the presence of at least one PA pattern. Table 1 shows the 33 potential PA-Apps. For privacy reasons, we elude the name of the potential PA-Apps and use the hash of their corresponding APK file as an identifier instead. Whereas the percentage of identified PA-Apps with respect to the total number of apps under analysis (i.e. 0.41%) is low, we found that a significant number of users may be using such potential PA-Apps (e.g., the *No.of.Installs* column), thus showing the number of possible users whose private data can be breached. Similarly, from the third column (Updated on) we can observe the existence of PA-Apps even recent updates.

Figure 7 shows the number of *dangerous* permissions used per application. Also, Fig. 8 shows the percentage of dangerous permissions used by each application to respect to the total of identified dangerous permissions. As an example, from Fig. 8 it can be seen that the potential PA-App identified by number 26 is using 83.3% of the dangerous permissions.

Figure 9 shows the frequency of permissions used by the potential PA-Apps. Here, we found that

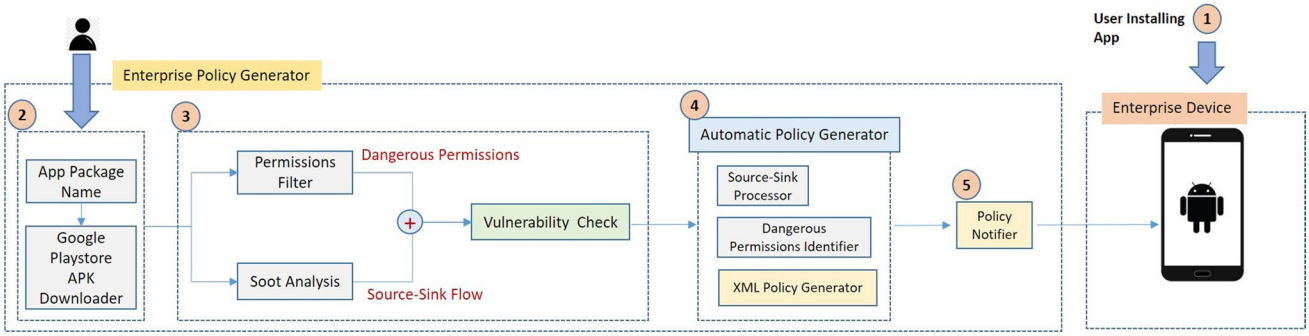


Fig. 10 Real-time Evaluation of PA-Apps using DyPolDroid

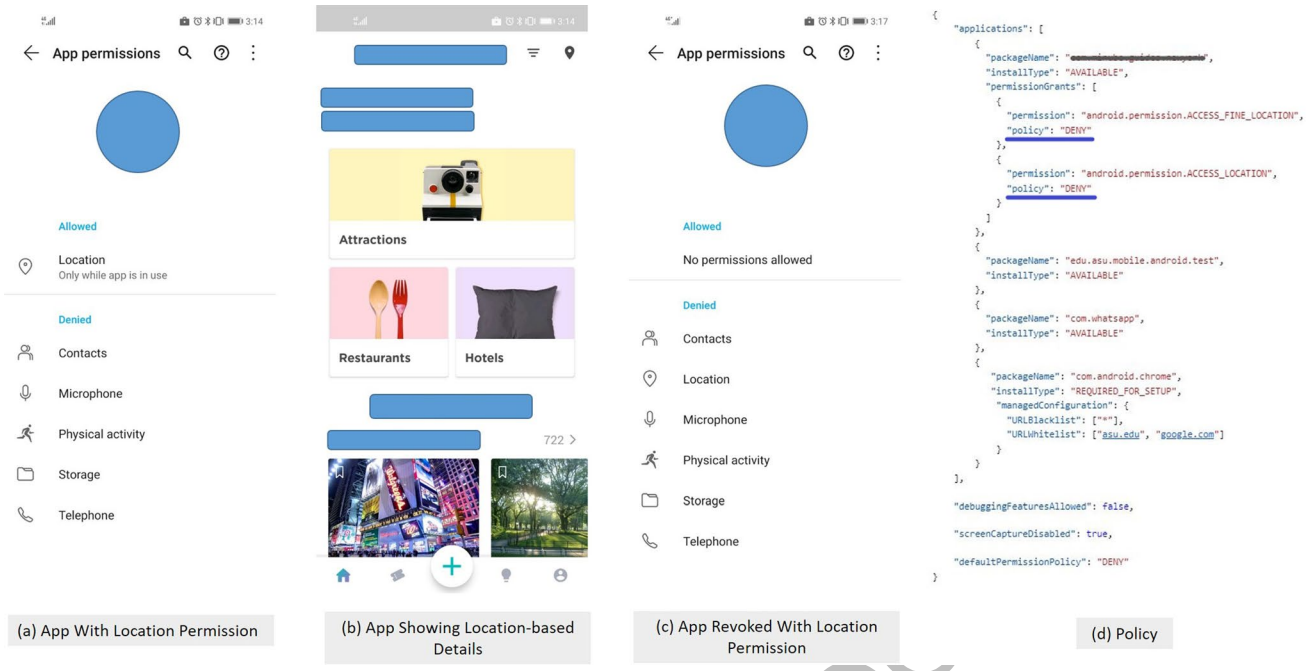


Fig. 11 An Experiment with DyPolDroid's Policy Generator. (The PA-App details are hidden for privacy purposes.)

WRITE_EXTERNAL_STORAGE is one of the most frequently used permissions. In addition, ACCESS_COARSE_LOCATION, READ_EXTERNAL_STORAGE, READ_PHONE_STORAGE, ACCESS_FINE_LOCATION, and CAMERA are the five permissions that are used frequently (i.e. more than 50%).

Based on these results, we conclude that potential PA-Apps can be found in the wild and can be potentially downloaded and installed by users, which typically grant all permissions to applications without understanding its consequences (Felt et al., 2012b), thus increasing the chances PA-Apps eventually become harmful. Therefore, AE-Admins need to be prepared for potential PA-Apps when preparing AE deployments.

4.2 Real-time Evaluation on Potential PA-Apps

Following the description presented in Section 2.2, for our experimental procedure we resorted to the COSU (pronounced *kiosk*) mode provided by the AE, which allows for AE-Admins to manually select, i.e., *authorize*, the Android applications that can be eventually installed on devices thus effectively restricting the installation of any other *unauthorized* applications. Also, we resorted to a scenario where the user's personal device is also used for work. Following Section 2.2, such a scenario, called *Bring-Your-Own-Device* (BYOD), allows for a containerized work/life separation where organization-owned devices can be used for personal matters as well. Moreover, our experimental procedure was focused mainly on the Automated Policy Generator and Policy Enforcement modules implemented by DyPolDroid. With that in mind, we manually fed the name of each potential PA-App to the DyPolDroid's Static Analysis Module, which in turn fetched the application directly from Google Play. [1].

Overall, as it is graphically depicted in Fig. 10, our experiments for each potential PA-App identified in Section 4.1 consisted of the following steps:

1. We first installed each potential PA-App on the user BYOD profile of an experimental device.
2. The application details (i.e., the package name) of the PA-App were entered manually to the Policy Generator module. The Playstore Downloader sub-module then fetched the PA-App from Google Play.
3. From the corresponding APK we extracted the Permissions and the Source-to-Sink flow as follows: first, the permissions filter extracted the list of dangerous permissions that were used by the given application. Next, the APK was parsed for some of the known vulnerable patterns, following the approach described in Section 3.3. Finally, The application's matched pattern(s) and its defined permissions were mapped to form a tuple, {(dangerous_permissions, vulnerable_method)}. For example, an application with dangerous permission ACCESS_FINE_LOCATION, and exposing the location data through some public channel, was represented as {(ACCESS_FINE_LOCATION, getLatitude)}.

[1] We assumed that apps are installed only from Google Play and not from any other stores

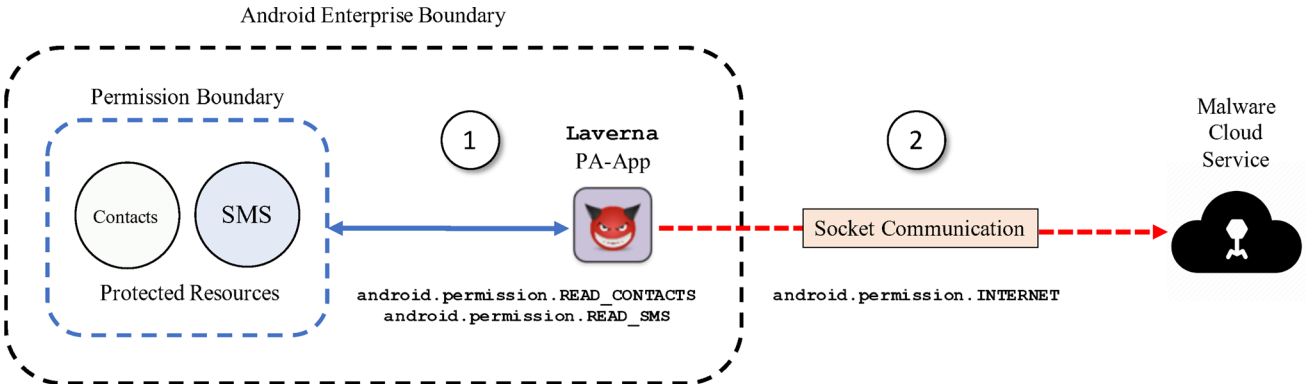


Fig. 12 The Laverna PA-App (Installed within an AE boundary). First, Laverna will obtain the READ CONTACTS and READ SMS permissions to obtain the corresponding data, thus legitimately crossing the permission boundary as defined by the AE (1). However, it will later leak such information to a remote malware cloud served by abusing the INTERNET permission, thus effectively breaching the AE boundary (2)

4. The Policy Generator module created the device policy code that blocks the dangerous permissions for the potential PA-App following the procedure detailed in Algorithm 1.
5. The generated Device Policy file was then uploaded to the testing device using the AE. Following the example shown before, the device in turn applied the updated policy by revoking the dangerous permission `ACCESS_FINE_LOCATION` for the corresponding PA-App.

Using this procedure, we revoked all the permissions included in Behavioral Patterns that may have allowed the potential PA-Apps to carry out permission-abusing attacks. As an example, Fig. 11 showcases an experiment on one of the real-world PA-Apps from Google Play as given in Table 1. Figure 11(a), (b) shows the PA-App exhibiting location details, and Fig. 11c shows the same PA-App after applying the generated Device Policy shown in Fig. 11d.

4.3 `Laverna`: A Self-Developed PA-App

In a further attempt to evaluate the effectiveness of our approach, we developed `Laverna`: a *proof-of-concept* PA-App that requests several permissions for benign functioning: getting full access to the user's contacts, real time location, and SMS so it can serve as a messaging application. However, it also silently exploits the granted permissions to collect and leak data to a remote server when the user is messaging another user. The leaked data includes the contact's full name and phone number and the messages sent, including who the sender and receiver are.

In our experiments, `Laverna` was downloaded on an experimental device, and a simulated user was allowed to select what permissions can be granted before installing such PA-App, following the scheme featured on Fig. 12. Also, the simulated user was allowed to specify the Counter-Policy shown in Listing 1, which gives the response to

```
{ "defaultPermissionPolicy": "PROMPT",
  "applications": [{
    "packageName": "com.example.laverna",
    "installType": "REQUIRED_FOR_SETUP",
    "permissionGrants": [
      { "permission": "android.permission.READ_CONTACTS",
        "policy": "BLOCK"},
      { "permission": "android.permission.READ_SMS",
        "policy": "BLOCK"}]}
```

Listing 10 Policy Definition Language

the different types of attacks a user wants to defend against. In this case the two attacks

are: *Steal_Contacts*, and *Steal_Messages*. Since the two attacks can be found when analyzing the code of `Laverna`, the permissions involved on each of the attacks are automatically denied. Such an action is reflected in the JSON-based Device Policy shown in Listing 10, which is based on the language snippet presented in Listing 9. In such listing, `Laverna` requests for permissions like `READ_CONTACTS`, `READ_SMS`, `INTERNET`, and the defined enterprise policy blocks the PA-App request for the {`READ_CONTACTS`, and `READ_SMS`} permissions. Our tests show that `DyPolDroid` was able to block this application from collecting the user's data and sending it off the device. Since a subset of the permissions requested by `Laverna` were found to be malicious, the default policy was overridden to block them on the device.

5 Related Work

As described in Section 1, different approaches in the literature have addressed the problem of malicious applications in Android. In such regard, `DyPolDroid` is not the first attempt at increasing the security of mobile devices, nor the first to propose fine-grained policies. In this section, we compare `DyPolDroid` with previous work, describe similarities and sources of inspiration, and clarify key differences that add up to the novelty of our approach.

Bartel et al. (Bartel et al., 2012) introduced an approach for securing Android apps by comparing the set of permissions they request to users against the set of permissions that are actually used within the application's code, subsequently blocking over-privileged applications. `DyPolDroid` shares a similar approach for identifying potential PA-Apps by inspecting the set of permissions that are leveraged within the app's code. However, instead of targeting over-privileged apps, our approach strives to identify and block the abuse of permissions that are also leveraged for providing the normal behavior of the app, as detailed in Section 2.3.

VetDroid (Zhang et al., 2013) was intended to discover and vet undesirable behaviors in Android applications, by analyzing how permissions are used to access (sensitive) system resources, and how these resources are further utilized by the application, allowing for security analysts to easily examine the internal sensitive behaviors of an app. Our description of PA-Apps, presented in Section 2, is inspired by this idea. `DyPolDroid` goes as step further by introducing the concept of behavioral patterns in Section 3.3 to identify malicious behavior in potential PA-Apps.

Kratos (Shao et al., 2016) is a vendor independent tool for detecting errors in Android security enforcement. It allows for potential permission misuse to be more easily located by creating a call graph of the Android system image, and marking each entry-point to the graph. The nodes in the graph are annotated with security relevant information. The taint analysis depicted by `DyPolDroid`, which is described in Section 3.3, follows a similar approach. However, we aim to detect well-defined Behavioral Patterns on the sequences of method calls exhibited by potential PA-Apps. If a pattern is detected, it may be then subsequently restricted by means of a Device Policy.

Slavin et al. (Slavin et al., 2016) proposed a technique to automatically detect policy violations due to errors or omissions within Android apps. They were able to classify these violations into two categories: strong and weak violations. The former is when an application fails to state the data collection purpose, while the latter is when the application vaguely describes its data collection process. `DyPolDroid` depicts a similar approach in which potentially malicious PA-Apps are identified by the Behavioral Patterns they depict within their code. However, the restriction of such PA-Apps may not only depend on their successful identification, but also on the Counter and Device policies as illustrated in Section 3.4.

DroidCap (Dawoud & Bugiel, 2019) introduced OS-level support for so-called *capability-based* permissions in Android, which provided further separation of privileges within an application by modifying the Android Zygote and IPC. Whereas this technique may able to provide a fine-grained, more specific approach for defeating malicious apps, it still requires modifications to the Android OS itself, which can be a considerable barrier for its adoption in practice. In contrast, since `DyPolDroid` relies on the remote configuration features of the AE, it requires no modification to the OS of the managed devices.

BorderPatrol (Zungur et al., 2019) leverages the BYOD paradigm, similar to the *work* profile discussed in Section 2.2. It protects devices by creating a customized Mobile Device Manager that leverages fine-grained contextual information, thus providing a more fine-grained approach than the AE. However, since BorderPatrol uses the Xposed Module Repository (Drupal, 2021), it requires root access to managed devices, which may significantly complicate maintenance and usability (Gasparis et al., 2017).

Reaper (Diamantaris et al., 2019) provides real-time analysis of Android apps, in an effort to augment and complement the Android Permission System, thus potentially counteracting ongoing attacks. As with `DyPolDroid`, Reaper leverages dynamic analysis of Android APK files to detect permission abuse, and also uses stack trace info of the running process for further processing. However, since it uses the Xposed framework, which requires root access to devices.

More recently, the HamDroid (Seraj et al., 2022) aims to detect fake anti-malware based on the permissions with the dataset of harmful and benign apps, using customized a multi-layer perception (MLP) neural network. They extract permission information from manifest files prior to the installation of the anti-malware. In addition,

MLdroid (Mahindru & Sangal, 2021) supports detecting unknown malware using feature selection approaches, API calls, and app ratings to detect the violation of permission with meaningful accuracy.

Finally, GDdroid (Gao et al., 2021) leverages graph neural networks to map apps and APIs into a graph, converting the app classification into a node classification task. For modeling the relevance among APIs, they present an embedding-based method to mine API behavior patterns. Unlike these machine learning approaches, `DyPolDroid` extracts malicious behavioral patterns from PA-Apps and leverages the dynamic permission updates provided by the AE.

6 Conclusions and Future Work

PA-Apps are still an ongoing problem for Android Ecosystems. In such regard, `DyPolDroid` offers an effective and convenient solution that requires no root access to user's devices nor any modifications to the code of PA-Apps: two constraints that have limited the deployment in the practice of previous approaches.

As a matter of ongoing and future work, we are currently analyzing several PA-Apps to identify Behavioral Patterns and potential templates for Counter-Policies that can effectively defeat them. We plan to use this insight to conduct a study in which users sign up for an experimental Android Enterprise. Then, we aim to collect data on how the devices are used, and verify whether `DyPolDroid` was able to accurately detect when permissions were improperly abused. Also, we will collect data regarding the level of user satisfaction with respect to the restrictions observed in the functionality of potential PA-Apps as a result of using `DyPolDroid`.

In addition, we must notice that the Android Open Source Project does not maintain a complete mapping of the public permission functions, which is required by our analysis described in Section 3.3. As described in Section 3.5.2, there have been noticeable attempts in the past to determine these, namely Axplorer (Backes et al., 2016), and PScout (Au et al., 2012). However, at the moment of publication of this paper, the aforementioned approaches were no longer up-to-date with newer versions of Android. Therefore, we plan to further work on this issue, as should more up-to-date mappings become available in the future, the accuracy of `DyPolDroid` will likely increase.

Finally, one common issue with dynamic permission systems such as `DyPolDroid` is *fine tuning*, e.g., how aggressive they are: a tool that is too aggressive will block more than required, degrading the user's experience, whereas a tool that is too lenient will not block enough, thus the user is still vulnerable to attacks. While `DyPolDroid` attempts to walk the line between the two, it does err on the side of leniency to preserve application functionality for the user. Future work may then focus on providing a balance between security guarantees and usability, such that users can be effectively protected against Permission-Abusing attacks at the same time they can enjoy the many different apps that are constantly developed and released for Android Enterprises.

Acknowledgements This work is partially supported by a grant from the Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for CyberThreat Intelligence), a grant from the Center for Cybersecurity and Digital Forensics (CDF) at Arizona State University, and a startup funds grant from Texas A&M University - Corpus Christi.

Declarations

Conflict of Interests All authors declare that they have no conflicts of interest.

References

Android Authority. (2020). Report: Hundreds of apps have hidden tracking software used by the government. https://www.androidauthority.com/government-tracking-apps-1145989/. Accessed 14 Sept 2022.

Android Developers Reference. (2022). Manifest.permission. https://developer.android.com/reference/android/Manifest.permission. Accessed 14 Sept 2022.

Arora, A., Peddoju, S. K., & Conti, M. (2020). Permpair: Android malware detection using permission pairs. *IEEE Transactions on Information Forensics and Security, 15*, 1968–1982.

Au, K.W.Y., Zhou, Y.F., Huang, Z., et al. (2012). Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conf. on computer and communications security, New York, NY, USA, CCS '12, pp. 217–228*.

Backes, M., Bugiel, S., Derr, E., et al. (2016). On demystifying the android application framework: Re-visiting android permission specification analysis. In *Proceedings of the 25th USENIX Conf. on security symposium, USA, SEC'16, pp. 1101–1118*.

Bartel, A., Klein, J., Le Traon, Y., et al. (2012). Automatically securing permission-based software by reducing the attack surface: an application to android. In *2012 Proc. of the 27th IEEE/ACM international Conf. on automated software engineering* (pp. 274–277).

Calciati, P., & Gorla, A. (2017). How do apps evolve in their permission requests? A preliminary study. In *2017 IEEE/ACM 14th Int. Conf. on mining software repositories (MSR)* (pp. 37–41).

Chung, F.D., Kuhn, D., et al. (2019). Guide to attribute based access control (abac) definition and considerations. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=927500. Accessed 14 Sept 2022.

Dawoud, A., & Bugiel, S. (2019). Droidcap: Os support for capability-based permissions in android. In *Proc. of the network and distributed system security symposium (NDSS) 2019*.

Diamantaris, M., Papadopoulos, E. P., Markatos, E. P., et al. (2019). Reaper: Real-time app analysis for augmenting the android permission system, ACM New York NY, USA, CODASPY '19, pp. 37–48.

Drupal. (2021). Xposed module repository. https://repo.xposed.info/. Accessed 14 Sept 2022.

Enck, W. (2020). Analysis of access control enforcement in android. In Proc. of the 25th ACM symposium on access control models and technologies. ACM, New York, NY, USA, SACMAT '20, pp. 117–118.

Felt, A.P., Chin, E., Hanna, S., et al. (2011). Android permissions demystified. In Proceedings of the 18th ACM Conf. on computer and communications security. ACM, New York, NY, USA, CCS '11, pp. 627–638.

Felt, A.P., Ha, E., Egelman, S., et al. (2012a). Android permissions: User attention, comprehension, and behavior. In Proc. of the Eighth Symp. on usable privacy and Sec. ACM, New York, NY, USA.

Felt, A.P., Ha, E., Egelman, S., et al. (2012b). Android permissions: user attention, comprehension, and behavior. In Proceedings of the eighth symposium on usable privacy and security, pp. 1–14.
Gao, H., Cheng, S., & Zhang, W. (2021). Gdroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, *106–102*, 264.
Gasparis, I., Qian, Z., Song, C., et al. (2017). Detecting android root exploits by learning from root providers. In *26th USENIX security symposium* (pp. 1129–1144). USENIX Association.
Google. (2021a). Android enterprise. https://www.android.com/enterprise/. Accessed 14 Sept 2022.
Google. (2021b). Permissions on android. https://developer.android.com/guide/topics/permissions/overview. Accessed 14 Sept 2022.
Hill, M., & Rubio-Medrano, C. E. (2021). DyPolDroid Github Repository. https://github.com/sefcom/DyPolDroid. Accessed 14 Sept 2022.
IETF OAuth Working Group. (2022). OAuth 2.0. https://oauth.net/2/. Accessed 14 Sept 2022.
Mahindru, A., & Sangal, A. (2021). Mldroid—framework for android malware detection using machine learning techniques. *Neural Computing and Applications*, *33*(10), 5183–5240.
OASIS Standard. (2013). eXtensible Access control markup language (XACML) Version 3.0. (2013, January 22). http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. Accessed 14 Sept 2022.
Paderborn University. (2022). Soot - A framework for analyzing and transforming Java and Android applications. http://soot-oss.github.io/soot/. Accessed 14 Sept 2022.
PC Magazine. (2020). Android users need to manually remove these 16 infected apps. https://www.pcmag.com/news/android-users-need-to-manually-remove-these-17-infected-apps. Accessed 14 Sept 2022.
Ramachandran, S., Dimitri, A., Galinium, M., et al. (2017). Understanding and granting android permissions: A user survey. In *2017 Int. Carnahan Conf. on security technology (ICCST)* (pp. 1–6).
Rubio-Medrano, C.E., Jogani, S., Leitner, M., et al. (2019). Effectively enforcing authorization constraints for emerging space-sensitive technologies. In *Proc. of the 24th ACM symposium on access control models and technologies. ACM, New York, NY, USA, SACMAT '19, pp. 195–206*.
Rubio-Medrano, C.E., Hill, M., Claramunt, L., et al. (2020a). DyPolDroid: Protecting Users and Organizations Against Permission-Abuse Attacks in Android. In *Proceedings of the international conference on secure knowledge management in the artificial intelligence Era. Springer*.
Rubio-Medrano, C.E., Hill, M., Claramunt, L., et al. (2020b). Poster: DyPolDroid: protecting users and organizations against permission-abuse attacks in android. In *Proceedings of the 6th IEEE european symposium on security and privacy. IEEE*.
Seraj, S., Khodambashi, S., Pavlidis, M., et al. (2022). Hamdroid: permission-based harmful android anti-malware detection using neural networks. Neural Computing and Applications, pp. 1–10.
Shao, Y., Ott, J., Chen, Q. A., et al. (2016). Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Proc. of the network and distributed system security symposium (NDSS) 2016*.
Slavin, R., Wang, X., Hosseini, M. B., et al. (2016). Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th international Conf. on software engineering, New York, NY, USA, ICSE '16, pp. 25–36*.
Steven Arzt. (2022). FlowDroid data flow analysis tool. https://github.com/secure-software-engineering/FlowDroid/. Accessed 14 Sept 2022.
Sunday Express. (2020). Android's biggest issue is far worse than we ever imagined, new research proves. https://www.express.co.uk/life-style/science-technology/1362551/Android-Google-Play-Store-malware-problem-researc. Accessed 14 Sept 2022.
The New York Times. (2020). The lesson we're learning from TikTok? It's all about our data. https://www.nytimes.com/2020/08/26/technology/personaltech/tiktok-data-apps.html. Accessed 14 Sept 2022.
Vallée-Rai, R., Co, P., Gagnon, E., et al. (1999). Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for advanced studies on collaborative research. IBM Press, CASCON '99, pp. 13*.
Vidas, T., Votipka, D., & Christin, N. (2011). All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conf. on offensive technologies. USENIX Association, USA, WOOT'11, pp. 10*.
Wang, H., Guo, Y., Tang, Z., et al. (2015). Reevaluating android permission gaps with static and dynamic analysis. In *2015 IEEE global communications Conf. (GLOBECOM)* (pp. 1–6).
Wei, X., Gomez, L., Neamtiu, I., et al. (2012). Permission evolution in the android ecosystem. In *Proc. of the 28th annual computer security applications Conf. ACM, New York, NY, USA, ACSAC '12, pp. 31–40*.
Wired. (2020). A barcode scanner app with millions of downloads goes rogue. https://www.wired.com/story/barcode-scanner-app-millions-downloads-goes-rogue/. Accessed 14 Sept 2022.
Wu, S., & Liu, J. (2019). Overprivileged permission detection for android applications. In *ICC 2019 - 2019 IEEE Int. Conf. on communications (ICC)* (pp. 1–6).
Zachariah, R., Akash, K., Yousef, M. S., et al. (2017). Android malware detection a survey. In *2017 IEEE Int. Conf. on circuits and systems (ICCS)* (pp. 238–244).
ZDNet. (2020). Play Store identified as main distribution vector for most Android malware. https://www.zdnet.com/article/play-store-identified-as-main-distribution-vector-for-most-android-malware/. Accessed 14 Sept 2022.
Zhang, Y., Yang, M., Xu, B., et al. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conf. on computer and communications security. ACM, New York, NY, USA, CCS '13, pp. 611–622*.
Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., et al. (2015). Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proc. of the 5th ACM conf. on data and application security and privacy. ACM, New York, NY, USA, pp. 37–48*.
Zhu, D.Y., Jung, J., Song, D., et al. (2011). Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Operating Systems Review*, *45*(1), 142–154.
Zungur, O., Suarez-Tangil, G., Stringhini, G., et al. (2019). Borderpatrol: Securing byod using fine-grained contextual information. In *2019 49th Annual IEEE/IFIP Int, Conf. on dependable systems and networks (DSN)* (pp. 460–472).

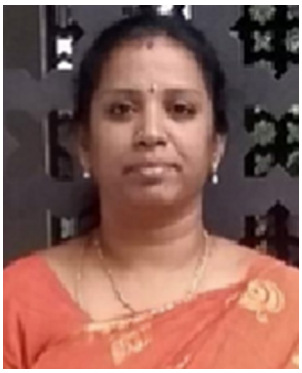
Carlos E. Rubio-Medrano PhD, is an Assistant Professor of Computer Science at Texas A&M University – Corpus Christi (TAMU-CC), where he leads the Cybersecurity Research and Innovation Laboratory (CSRIL). He received a PhD in Computer Science from Arizona State University in 2016. His research interests are focused on the specification, evaluation, verification, and validation of cybersecurity properties for mission-critical software.

Rubio-Medrano has authored more than 25 research publications in top cybersecurity venues such as ACM Computer and Communications Security (CCS), USENIX Security, and IEEE Security and Privacy (S&P). Also, Dr. Rubio-Medrano has served as an instructor of courses such as Information Assurance, Computer Networks, Digital Forensics, etc. Dr. Rubio-Medrano has organized several hackathon competitions for high-school and college students, and has also excelled on mentoring and developing undergraduate and graduate college students for academic research. An extended summary of Dr. Rubio-Medrano's experience can be found at https://carlosrubiomedrano.com/.

Pradeep Kumar Duraisamy Soundrapandian is currently pursuing his PhD with the School of Computer Science and Engineering, Vellore Institute of Technology, Chennai Campus, India. He received the B.E. degree in Computer Science and Engineering from Madras University, his M.S.(By Research) degree in from College of Engineering, Guindy, Anna University, Chennai, India. His research interests include language-based security, type system, cybersecurity attacks on mobile applications, and secure coding best practices. He has more than 12 years of IT research and software development experience.

Luis Claramunt M.S., is starting his professional career as an Information Security Engineer at Intel Corporation in the Crypto Management Department. He commenced this position after concluding his Master's Degree in Computer Science at Arizona State University in 2022. He began his vocation in cybersecurity as a research assistant at ASU as an affiliate of the Laboratory of Security Engineering for Future Computing (SEFCOM), where he collaborated on three research projects focused on topics such as Authorization, Access Control, Machine Learning, and Mobile Augmented Reality.

Jaejong Baek PhD, is an Assistant Research Scientist at Arizona State University. He is currently affiliated with the Center for Cybersecurity and Trusted Foundations (CTF), as well as with the Laboratory of Secure Engineering for Future Computing (SEFCOM), His research focuses on cybersecurity and privacy, and his work spans the areas of telecom and mobile security analysis techniques, zero-trust model, access mechanism, digital forensics, and blockchain framework. He worked as an adjunct professor at Howon University teaching network security and cyber warfare. He has a background in cyberwarfare and digital forensic expertise in the Navy. He received a doctorate from Yonsei University in South Korea in 2011.



Geetha S is currently a Professor and the Associate Dean, Research with the School of Computer Science and Engineering, Vellore Institute of Technology, Chennai Campus, India. She received the B.E. degree in Computer Science and Engineering from Madurai Kamaraj University, India, in 2000, and the M.E. degree in Computer Science and Engineering and the Ph.D. degree from Anna University, Chennai, India, in 2004 and 2011, respectively. Her research interests include steganography, steganalysis, multimedia security, intrusion detection systems, machine learning paradigms, computer vision and information forensics. She has more than 20 years of rich teaching and research experience. She has published more than 100 papers in reputed international conferences and refereed journals like IEEE, Springer, Elsevier publishers. She joins the Review Committee and the Editorial Advisory Board of reputed journals. She has given many expert lectures, keynote addresses at international and national conferences. She has organized many workshops, conferences, and FDPs. She was a recipient of the University Rank and Academic Topper Award in her B.E. and M.E. degrees, in 2000 and 2004, respectively. She was also the proud recipient of the ASDF Best Academic Researcher Award 2013, the ASDF Best Professor Award 2014, the Research Award in 2016, and the High Performer Award 2016, Best Women Researcher Award 2021.

Gail-Joon Ahn PhD, CISSP, is a Full Professor of Computer Science and Engineering in the School of Computing and Augmented Intelligence and the Senior Advisor of the Center for Cybersecurity and Trusted Foundations at Arizona State University. His research foci include security analytics and big data-driven security intelligence, vulnerability and risk management, access control and security architecture for distributed systems, identity and privacy management, cybercrime analysis, security-enhanced computing platforms, and formal models for computer security devices. Ahn is the author of more than 150 refereed research papers. Ahn is currently the information director of the Association for Computing Machinery's Special Interest Group on Security, Audit, and Control (SIGSAC) and he is a recipient of the U.S. Department of Energy Early Career Principal Investigator Award, Educator of the Year Award from the Federal Information Systems Security Educators' Association and Best Researcher Award from CIDSE. Also, he serves as associate editor-in-chief of IEEE Transactions on Dependable and Secure Computing, associate editor of ACM Transactions on Information and Systems Security, and as an editorial board member of Computers & Security. An extended summary of Dr. Gail-Joon Ahn's experience can be found at https://sefcom.asu.edu/.