DyPolDroid: Protecting Users and Organizations from Permission-Abuse Attacks in Android

Carlos E. Rubio-Medrano[1], Matthew Hill[2],
Luis M. Claramunt[3], Jaejong Baek[3], and Gail-Joon Ahn[3]

[1] Texas A&M University - Corpus Christi
carlos.rubiomedrano@tamucc.edu
[2] Independent Researcher
[3] Arizona State University
{lclaramu, jbaek7, gahn}@asu.edu

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, e.g., sending data over the Internet, within the Android Ecosystem. 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 [1], targeting users with a large variety of attacks, e.g., harvesting private data [2], making unwanted credit card charges [3], retrieving the location of users [4], etc. Whereas the root causes for such attacks have been largely explored in the literature [5], 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 [6, 7].

Also recently, *Android Enterprise* (AE) [8] 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:

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 [9].
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 configure 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 [10, 11], 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 counteracting 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` is publicly available as open-source [12].

This paper is organized as follows: Sec. 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 Sec. 3. We provide a description of a preliminary procedure we have conducted to evaluate the effectiveness of `DyPolDroid` in Sec. 4, and then discuss some future work and conclude the paper in Sec. 6. A preliminary version of this paper appeared as a poster abstract in the Proceedings of the 6th IEEE European Symposium on Security and Privacy 2021 (Euro S&P 2021) conference [13].

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 accesses 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 [14–16]. 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 [17]. 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 to access the Internet.

2.2 Android Enterprise

Android Enterprise (AE) is a device management framework that allows for organizations to remotely monitor and configure Android-run devices, e.g., automatically installing and uninstalling apps without extensive user intervention [8]. 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-Sevice* (BYOD) mode, devices may allow for two different profiles to be configured 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 configure 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 [18]. 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,

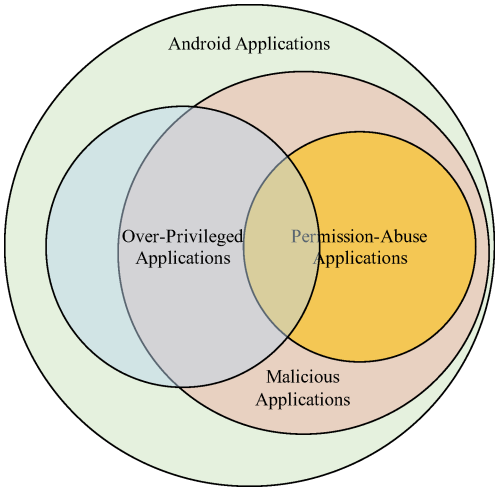

Fig. 1. Classifying Apps in Android based on Behavior. In this paper, we are interested in detecting and neutralizing PA-Apps, which are always regarded as malicious.

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 [10, 11]. 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 [19–22].

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 [3], installing tracking software [4] or collecting user data [2].

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 attack(s). 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*

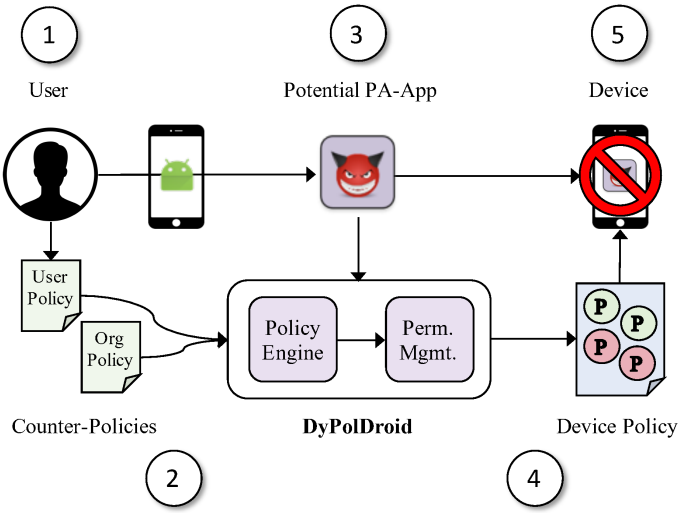

Fig. 2. How `DyPolDroid` Works: a User signs up for an Android Enterprise (1) and moves on to write Counter-Policies (2), which are later evaluated against the Attack Patterns obtained from any installed PA-Apps (3), producing a Device Policy that is then sent to the Device (4). As a result, PA-Apps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

at run-time. Following Fig. 1, detecting all potential over-privileged apps that may or may not be malicious is out of scope of this paper. Also, the detection and prevention of all other malicious Android apps that carry out attacks by means of other techniques other than the abuse of permissions, e.g., dynamic library updates [23], is also out of scope.

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

To address the problem just described, we now introduce `DyPolDroid` (Dynamic Policies in Android): a dynamic security framework graphically shown in Fig. 2, in which both users and AE-Admins can actively restrict the behavior of PA-Apps, thus preventing the occurrence of costly 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 are 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.

```
1  <Rule RuleId="Laverna_Attacks" Effect="Deny">
2   <Target>
3    <AnyOf> <AllOf> <Match Id="boolean-equal">
4     <AttributeValue>true</AttributeValue>
5      <AttributeDesignator AttributeId="Laverna"/>
6    </Match> </AllOf> </AnyOf>
7    <AnyOf> <AllOf> <Match Id="boolean-equal">
8     <AttributeValue>true</AttributeValue>
9     <AttributeDesignator AttributeId="Steal_Contacts"/>
10   </Match> </AllOf>
11    <AllOf><Match Id="boolean-equal">
12     <AttributeValue>true</AttributeValue>
13     <AttributeDesignator AttributeId="Steal_Messages"/>
14   </Match></AllOf> </AnyOf>
15  </Target>
16 </Rule>
```

Listing 1.1. A Counter-Policy for the `Laverna` PA-App.

3.1 Behavioral Patterns

Following the description started in 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 [9,24]. As an example, the gaming behavior may include the pattern: (`CAMERA`, `INTERNET`), whereas a contact-leaking attack may require an 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 attack 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 [25]. 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 1.1 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 3 presents a graphical depiction of the process just discussed: Behavioral Patterns can be leveraged to construct custom Counter-Policies, which are then

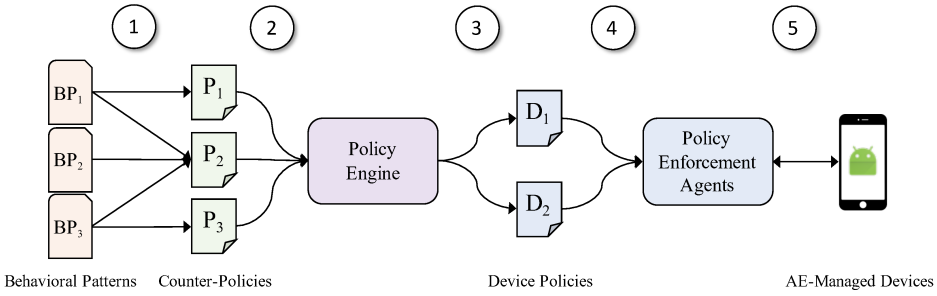

Fig. 3. From Behavioral Patterns to Device Policies: Templates describing Behavioral Patterns are leveraged by users and AE-Admins to write Counter-Policies (1), which are then fed as an input to `DyPolDroid`'s Policy Engine (2), so they can be turned into Device Policies (3). Later, Device Policies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

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 `DyPolDroid` to resolve conflicts before any resulting policies are sent to the user's device, as show in Figure 3 (2).

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 [9]. 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., 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* [26], a well-known data flow analysis technique. Initially, data flow sequences are obtained from the APK file of the PA-App by leveraging FlowDroid [27]. 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.

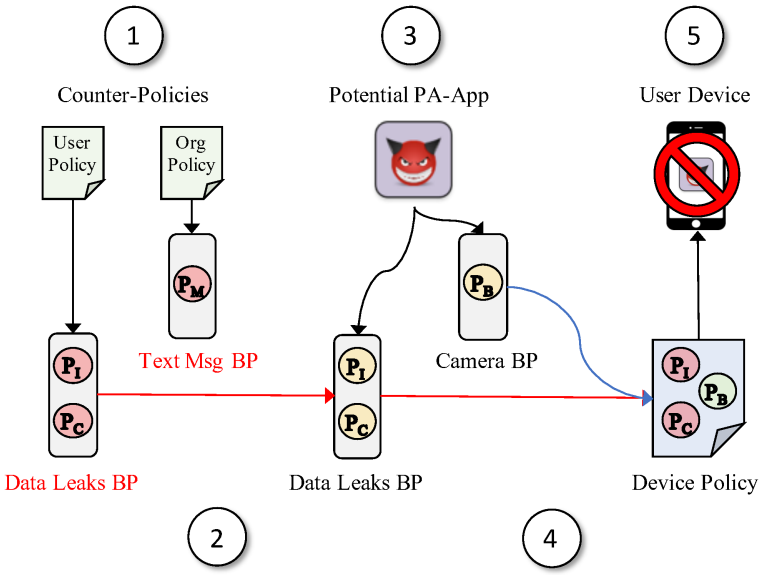

Fig. 4. Creating Device Policies in `DyPolDroid`. The set of *authorized* permissions from each Behavioral Pattern is obtained by evaluating Counter- Policies (1)(2), whereas the set of *observed* permissions is obtained via Taint Tracking analysis on potential PA-Apps (3). Later, the set of *resulting* 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).

3.4 Device Policies and Enforcement

Fig. 4 gives 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. 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. These resulting permissions are then updated within the Device Policy to allow or block their future usage. Listing 1.2 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.

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 begin to apply. 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. Finally, `DyPolDroid` uses a SHA 256 hash in conjunction with

```
{ "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 1.2. A Device Policy for the `Laverna` PA-App.

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.

4 Preliminary Evaluation

For the purposes of evaluating our approach, we have 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. The Counter-Policy shown in Listing 1.1 gives the response to the different types of attacks a users wants to defend against. In this case the two attacks are: *Steal_Contacts*, and *Steal_Messages*. Should any of the attacks be found when analyzing the application, the action taken against the used permissions will be to deny them. This change in allowed permissions is reflected in the JSON-based Device Policy shown in Listing 1.2.

In our experiments, `Laverna` was downloaded on an experimental device, and a user was allowed to select what permissions can be granted before installed such PA-App. 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. While this approach does not preemptively block the leaking of user data, once `DyPolDroid` has been performed its analysis future cases will mitigate such attacks.

5 Related Work

As described in Section 1, several 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 device policies. In this section, we compare `DyPolDroid` with previous work, describe similarities and sources of inspiration, and also clarify key differences that add up to the novelty of our approach.

VetDroid [24] 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 on this idea. `DyPolDroid` goes as step further by introducing the concept of attack patterns in Section 3.3 to identify malicious behavior in potential PA-Apps.

Kratos [5] 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 attack 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 Counter and a related Device Policy.

Slavin et al. [28] proposed a technique to automatically detect policy violations due to errors or omissions within Android applications. 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 attack 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 [29] 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 [30] leverages the *Bring Your Own Device* (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. How-

ever, since BorderPatrol uses the Xposed Module Repository [31], it requires root access to managed devices, which may introduce additional trouble [32].

Finally, Reaper [33] 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, it also leverages the Xposed framework, thus, it also requires root access to devices.

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 practice of previous approaches.

As a matter of ongoing and future work, we are currently analyzing several PA-Apps to identify Attack 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`. Finally, 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. In the past, there have been noticeable attempts to determine these, namely Axplorer [34], and PScout [35]. 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.

Acknowledgments

This work is partially supported by a grant from the National Science Foundation (NSF-SFS-1129561), a grant from the Center for Cybersecurity and Digital Forensics at Arizona State University, and by a startup funds grant from Texas A&M University – Corpus Christi.

References

1. ZDNet. (2020) Play store identified as main distribution vector for most android malware. [Online]. Available: https://www.zdnet.com/article/play-store-\identified-as-main-distribution-vector-for-most-android-malware/

2. The New York Times. (2020) The Lesson We're Learning From TikTok? It's All About Our Data. [Online]. Available: https://www.nytimes.com/2020/08/26/technology/personaltech/tiktok-data-apps.html
3. Wired. (2020) A barcode scanner app with millions of downloads goes rogue. [Online]. Available: https://www.wired.com/story/barcode-scanner-app-millions-downloads-goes-rogue/
4. Android Authority. (2020) Report: Hundreds of apps have hidden tracking software used by the government. [Online]. Available: https://www.androidauthority.com/government-tracking-apps-1145989/
5. Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. Mao, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *Proc. of the Network and Distributed System Security Symposium (NDSS) 2016*, January 2016.
6. Sunday Express. (2020) Android's biggest issue is far worse than we ever imagined, new research proves. [Online]. Available: https://www.express.co.uk/life-style/science-technology/1362551/Android-Google-Play-Store-malware-problem-researc
7. PC Magazine. (2020) Android Users Need to Manually Remove These 16 Infected Apps. [Online]. Available: https://www.pcmag.com/news/android-users-need-to-manually-remove-these-17-infected-apps
8. Google. (2021) Android Enterprise. [Online]. Available: https://www.android.com/enterprise/
9. A. Arora, S. K. Peddoju, and M. Conti, "Permpair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2020.
10. T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current android attacks," in *Proceedings of the 5th USENIX Conf. on Offensive Technologies*, ser. WOOT'11. USA: USENIX Association, 2011, p. 10.
11. R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *2017 IEEE Int. Conf. on Circuits and Systems (ICCS)*, 2017, pp. 238–244.
12. Hill, Matthew and Rubio-Medrano, Carlos E., "DyPolDroid Github Repository," 2021, https://github.com/sefcom/DyPolDroid.
13. IEEE. (2021) The 6th ieee european symposium on security and privacy. [Online]. Available: http://www.ieee-security.org/TC/EuroSP2021/
14. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conf. on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, p. 627–638.
15. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. of the Eighth Symp. on Usable Privacy and Sec.* New York, NY, USA: ACM, 2012.
16. S. Ramachandran, A. Dimitri, M. Galinium, M. Tahir, I. V. Ananth, C. H. Schunck, and M. Talamo, "Understanding and granting android permissions: A user survey," in *2017 Int. Carnahan Conf. on Security Technology (ICCST)*, 2017, pp. 1–6.
17. Google. (2021) Permissions on android. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview
18. W. Enck, "Analysis of access control enforcement in android," in *Proc. of the 25th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '20. New York, NY, USA: ACM, 2020, p. 117–118.
19. X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proc. of the 28th Annual Computer Security Applications Conf.*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, p. 31–40.

20. H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating android permission gaps with static and dynamic analysis," in *2015 IEEE Global Communications Conf. (GLOBECOM)*, 2015, pp. 1–6.
21. P. Calciati and A. Gorla, "How do apps evolve in their permission requests? a preliminary study," in *2017 IEEE/ACM 14th Int. Conf. on Mining Software Repositories (MSR)*, 2017, pp. 37–41.
22. S. Wu and J. Liu, "Overprivileged permission detection for android applications," in *ICC 2019 - 2019 IEEE Int. Conf. on Communications (ICC)*, 2019, pp. 1–6.
23. Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "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*. New York, NY, USA: ACM, 2015, p. 37–48.
24. Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conf. on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, p. 611–622.
25. OASIS Standard, "eXtensible Access Control Markup Language (XACML) Version 3.0. (2013, January 22)," 2013, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.
26. D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, p. 142–154, Feb. 2011.
27. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," ser. PLDI '14. New York, NY, USA: ACM, 2014, p. 259–269.
28. R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violations in android application code," in *Proceedings of the 38th International Conf. on Software Engineering*, ser. ICSE '16, New York, NY, USA, 2016, p. 25–36.
29. A. Dawoud and S. Bugiel, "Droidcap: Os support for capability-based permissions in android," in *Proc. of the Network and Distributed System Security Symposium (NDSS) 2019*, 01 2019.
30. O. Zungur, G. Suarez-Tangil, G. Stringhini, and M. Egele, "Borderpatrol: Securing byod using fine-grained contextual information," in *2019 49th Annual IEEE/IFIP Int, Conf. on Dependable Systems and Networks (DSN)*, 2019, pp. 460–472.
31. Drupal, "Xposed Module Repository," 2021, https://repo.xposed.info/.
32. I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, "Detecting android root exploits by learning from root providers," in *26th USENIX Security Symposium*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1129–1144. [Online]. Available: https://www.usenix.org/Conf./usenixsecurity17/technical-sessions/presentation/gasparis
33. M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "Reaper: Real-time app analysis for augmenting the android permission system," ser. CODASPY '19. New York, NY, USA: ACM, 2019, p. 37–48.
34. M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *Proceedings of the 25th USENIX Conf. on Security Symposium*, ser. SEC'16, USA, 2016, p. 1101–1118.

35. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conf. on Computer and Communications Security*, ser. CCS '12, New York, NY, USA, 2012, p. 217–228.