Permissions Plugins as Android Apps

Nisarg Raval
Duke University

Ali Razeen
University of British Columbia

Ashwin Machanavajjhala
Duke University

Landon P. Cox
Microsoft Research

Andrew Warfield
University of British Columbia

ABSTRACT

The permissions framework for Android is frustratingly inflexible. Once granted a permission, Android will always allow an app to access the resource until the user manually revokes the app's permission. Prior work has proposed extensible plugin frameworks, but they have struggled to support flexible authorization and isolate apps and plugins from each other. In this paper, we propose Dalf, a framework for extensible permissions plugins that provides both flexibility and isolation. The insight underlying Dalf is that permissions plugins should be treated as apps themselves. This approach allows plugins to maintain state and access system resources such as a device's location while being restricted by Android's process-isolation mechanisms. Experiments with microbenchmarks and case studies with real third-party apps show promising results: plugins are easy to develop and impose acceptable overhead for most resources.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**.

KEYWORDS

flexible permissions, android, plugins

ACM Reference Format:
Nisarg Raval, Ali Razeen, Ashwin Machanavajjhala, Landon P. Cox, and Andrew Warfield. 2019. Permissions Plugins as Android Apps. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307334.3326095

1 INTRODUCTION

Android's permissions framework mediates apps' access to system-wide resources, such as location, the camera, and SMS messages. In the earliest versions of Android, users granted apps permissions only at install-time, though beginning with Android 6.0 users could give apps permissions at runtime as well [6]. But regardless of when a user gives an app a permission, Android's permissions framework remains frustratingly inflexible. Once an app acquires a permission for a resource, Android will allow the app to access that resource until the permission is manually revoked. As a result, Android offers no way to use a device's context to evaluate requests (e.g., allow camera access, but not at work) or to modify a resource's fidelity (e.g., when at home, coarsen location information). As demonstrated by the popularity of XPrivacy [30], a third-party open-source modification to Android permissions which has over a million downloads, users desire more control over how apps access resources.

Numerous projects have proposed permissions frameworks for Android that allow third-party plugins to make nuanced authorization decisions based on contextual cues, such as location and time-of-day [30, 38, 41, 52, 54, 55]. Unfortunately, these frameworks have been forced to make difficult trade-offs between flexibility and isolation. Flexibility empowers plugins in two ways. First, it gives plugins the ability to make authorization decisions based on a variety of inputs, including accumulated session state and decision-time sensor readings. Second, flexibility allows plugins to modify the data they return to apps, such as redacting images or applying differentially-private noise to locations.

Isolation ensures that interactions between apps and plugins are restricted to the narrow resource-request interface, and that apps and plugins cannot otherwise circumvent, corrupt, or spy on each other. Many permissions frameworks isolate plugins by processing requests in the OS [38, 52, 54, 55]. However, safely executing plugins in the OS requires developers to write them in restrictive domain-specific languages that hinder flexibility. Other frameworks support writing plugins in a high-level language and directly execute them in an app's address space [30]. This approach sacrifices isolation for flexibility since it cannot prevent a buggy or malicious plugin from harming an app or the app from circumventing the plugin.

In this paper we present Dalf, a permissions framework for Android that provides the flexiblity of in-address-space plugins without sacrificing isolation. With Dalf, users may install plugins to mediate resource requests made by apps. Each time an app tries to access a resource, Dalf will pass the request to the plugin mediating that app-resource combination. The plugin may allow or deny the request, or modify the data returned to the app. Plugins may also maintain state and access other system resources such as the device's location.

The primary challenge for Dalf is ensuring that apps and permissions plugins are properly isolated from each other, and Dalf solves this problem by treating plugins as stand-alone apps. This approach allows Dalf to re-use Android's existing isolation mechanisms, such as running plugins in separate processes under unique UIDs and restricting app-plugin communication to Binder inter-process communication (IPC). Thus, apps and plugins can only interact through the resource-request interface, which limits the harm that a buggy or malicious plugin can do. Users can further limit the trust they place in plugins by managing the permissions granted to the plugins themselves. For example, to prevent a plugin from directly leaking information, a user can deny the plugin permission to access the Internet. Nonetheless, because plugins can interpose on apps' resource requests, a malicious plugin could launch denial of service attacks or feed apps false data. In Section 4.4, we discuss the kinds of damage that a malicious plugin could cause and some possible mitigation strategies.

Our prototype implementation of Dalf protects the following resources: a device's location; a user's contacts and calendar entries; image frames from the camera; and files stored on the external storage partition. Other resources, such as motion data, could also be supported with our proposed techniques. An evaluation of our Dalf prototype with microbenchmarks and a case study of unmodified Android apps demonstrate that Dalf is practical. Dalf imposes up to 3.5x slowdown when mediating access to location, contacts, and camera data, and an 80x slowdown when mediating accesses to external storage. In Section 8, we discuss several ways of reducing the external storage overhead. The source code of our prototype is available at https://github.com/dalfdroid/dalf.

This paper makes following contributions:

- We propose treating permissions plugins as Android apps as a way to provide both flexiblity and isolation. By using Android's process-based isolation mechanisms and Binder IPC, Dalf supports a nearly limitless number of authorization decisions, including context-based data modification.
- Experiments with our prototype demonstrates that our framework is flexible enough to implement prior work as Dalf plugins. More specifically, we implemented (i) a plugin to protect a user's location privacy using geo-indistinguishability [33], and (ii) a plugin to protect visual secrets using PrivateEye [51].

The rest of the paper is organized as follows: we provide necessary background on the Android OS in Section 2; we describe prior work in Section 3; we discuss their limitations in the context of our design principles and also provide an overview of Dalf in Section 4; we detail the implementation of our prototype in Section 5 and describe the plugins we developed in Section 6. Finally, we evaluate Dalf in Section 7, discuss its limitations in Section 8, and conclude in Section 9.

2 BACKGROUND

In this section, we provide background on the portions of Android salient to this paper. Android is a Linux-based OS that is bundled with a set of trusted software components collectively known as the Android framework. The framework implements all of the necessary tooling and API required by apps. It is also responsible for several important tasks such as setting up a runtime environment for each app; ensuring the usability of the device by terminating misbehaving apps that use too much memory or are unresponsive to user inputs; and mediating apps' accesses to sensitive resources on the device, e.g., the camera or GPS location.

2.1 Android permissions

The framework assigns unique permissions labels to each system resource and requires apps to declare the resources they need. For example, developers must specify the `CAMERA` label in a manifest file bundled with the app if it makes use of the camera. Android makes a distinction between *normal* and *dangerous* permissions. The resources in the former class, such as Internet access, are considered to pose a low risk to user privacy and are granted automatically [6]. The latter are considered high-risk, sensitive resources. Prior to Android 6.0, a user must grant all the dangerous permissions an app needs during install time. Otherwise, the app is not installed.

On Android 6.0 and later, apps do not receive dangerous permissions automatically. Before an app uses sensitive resources, it is required to check if it has the necessary permissions. If it does not, it may request them from the user who is then presented with a dialog that indicates the permission the app needs. If the user grants the request, the app may access the resource for the rest of its lifetime or until the user revokes it later. If permissions are denied, the app may either provide a degraded service or refuse to provide it at all until permissions are granted. The permissions granted to an app are recorded in a database managed by the `PackageManagerService`, a trusted service that is a part of the framework.

Android enforces most permissions using IPC. It only allows certain trusted components in the framework to have unmediated accesses to the sensitive resources. All other apps have to request for them via IPC calls to the framework. When the trusted components receive a request for a resource, they use the identity of the requesting app in conjunction with the permissions database managed by `PackageManagerService` to determine if the request should be allowed. If so, the framework sends the resource or a reference to the resource in a return IPC call to the app. This approach enables the framework to independently verify if an app has the necessary permissions to access different resources.

2.2 Binder IPC

IPC in Android is done with *Binder*, a core system feature, and with the notion of *binder objects*. The methods of a binder object instance may be invoked by remote processes even if they did not originally create the object. The only requirement is that they must first receive a valid reference to the object from a process that possesses such a reference. A binder object's methods are assigned unique identifiers and are either marked as *one-way* or *blocking*. As their names suggest, a one-way method returns immediately while a blocking method blocks the caller until the callee returns from the method. The callee is always the process that created the binder object while the caller is the process with the object reference.

When a binder object's method is called, if the caller and the callee are different processes, a *binder transaction* is initiated using the binder driver in the Linux kernel. The driver copies the data of the IPC message from the caller's address space to the callee's. In blocking transactions, it copies the return value from the callee to the caller at the end of the method. If the caller and callee are the same process, the method calls take place within the process and the binder driver is not involved.

The method arguments and the return data between the caller and callee are transferred using instances of the `Parcel` class, an Android abstraction over byte arrays. The caller will serialize a method's identifier and arguments into a parcel object before invoking the binder object method and passing the parcel. The callee will read the method identifier from the received parcel, deserialize all the arguments for that method from the parcel, and make a direct call to the method with the deserialized arguments. If the invoked method has a return value, the callee sends it to the caller in a separate reply parcel.

2.3 Launching apps

Android uses a unique approach of launching new apps. When the OS boots, the framework starts a special *zygote* process. Zygote loads the application runtime environment in its address space and then listens for requests to launch new apps. When one is received, it uses the `fork()` syscall to create a new process. The parent process resumes being zygote while the child process initializes the runtime, relinquishes system capabilities not required by the app, and starts executing the app code. In other words, Android does not use the traditional `exec()` syscall to execute an application binary. This technique reduces both app startup time and memory usage because an app process starts with the runtime already loaded and will share unmodified regions of its address space, such as those that contain code, with the zygote process.

3 RELATED WORK

Improving the permissions model on Android is an active area of work both in the academic and open-source communities. Since the default permissions model is binary (allow or deny), the aim has generally been to provide flexible permissions. Some of them also focus on improving usability so that users may enjoy privacy protections with minimal manual effort. We describe prior work in detail in this section. We summarize their limitations relative to Dalf in Table 1 and elaborate further in Section 4.2.

Flexible permissions: A popular approach of providing flexible permissions has been to take into account the context in which a resource request is made and to return custom data in response to a request, instead of merely allowing or denying it. CoDRA [52], ipShield [38], CRePe [41], and SemaDroid [55] allow users to specify policies to control apps' accesses to resources on the device based on external context (location, time of day, etc.). ipShield and SemaDroid focus on protecting sensor data while CoDRA and CRePe allow control over a wider set of resources, such as access to WiFi.

Pegasus [39] and SweetDroid [40] use code contexts to capture the code paths of apps' sensitive resource requests. This may be used, for example, to prevent an audio recording app from accessing the microphone when it is done by an advertisement library bundled within the app but not when the user explicitly records audio. INSPIRED [43] uses the UI elements shown on the screen and the relationships between them as the context of a resource request. It checks if an app's UI is in agreement with the intent of the permission requested e.g., an app that requests the `SEND_SMS` permission should *look* like a messaging app.

Xposed [29] is an open-source framework for Android that enables interposition on general portions of the Android API. It allows users to execute custom plugins within the address space of each app and hook well-known API methods. This enables plugins to have strong controls over an app's actions. XPrivacyLua [31] builds on top of Xposed to provide a flexible permissions system with a set of default actions such as faking location data. It also lets users write customs policies using Lua.

BinderFilter [54] is a kernel-level firewall for Binder. It allows users to use context-aware policies to filter and modify IPC messages. These policies are executed directly in the kernel. ASF [34] proposes an architecture for security experts to develop security modules that control the data apps receive when they request sensitive resources.

Flexible permissions on an unmodified OS: In contrast to earlier work and Dalf, there have been attempts to provide flexible permissions without modifying Android. Dr. Android [46] provides Mr. Hide, a trusted service app, and rewrites the bytecode of Android apps to call Mr. Hide to access sensitive resources instead of invoking Android APIs. Mr. Hide arbitrates resource requests according to custom policies. Boxify [35] and NJAS [37] go one step further and do not rely on bytecode rewriting. Instead, they propose manager apps, which are regular Android apps, that have the ability to execute third-party apps within specially crafted sandbox processes. A sandbox will enforce user-specified policies by intercepting attempts by the target app to initiate syscalls or calls to Android APIs.

Boxify and NJAS are attractive because of their ability to work on an unmodified OS. However, they are fundamentally dependent on the existence of a general mechanism on Android for a third-party app X to be able to execute another third-party app Y in X's process. While Boxify and NJAS use such mechanisms for sandboxing, a malicious app may use them to exfiltrate information from legitimate apps. Consider a malicious app disguised as a home screen app that lets users click on app icons and launch them. The malware may run each app in a separate process with hooks to exfiltrate all sensitive user and app data.

Due to such security concerns, these mechanisms may be removed in future versions of the OS. For instance, Boxify relies on APIs that are now either hidden or are in the process of being hidden[1] and Android is restricting apps from using hidden APIs [17]. NJAS uses a combination of `ptrace` and the `Context.createPackageContext()` API in its sandbox process to load, execute, and interpose on code belonging to the target app. Although this technique works at the moment, albeit with a significant amount of engineering, it may be restricted in the future. In sum, providing flexible permissions without modifying the OS is not necessarily sustainable.

Protections against specific attacks: PrivateEye and WaveOff protect visual secrets from being leaked in images and videos captured by smartphone devices [51]. The authors modified the camera subsystem in Android so that only specially marked regions of the physical environment are made visible to apps using the camera; all other regions are blocked and appear black in the frame.

Usability improvements: SmarPer [49] and Wijsekera et al. [53] proposed the use of machine learning techniques to automatically respond to permission requests from apps and involve the user only when absolutely necessary. ipShield [38] lets users rate how concerned they are about an app's ability to perform inference attacks, which is the use of data from multiple sources to infer sensitive information e.g., using the accelerometer and gyroscope together to

[1]In particular, `ActivityThread.getApplicationThread()` and `ApplicationThread.bindApplication()`. The most recent implementations of these methods reflect their hidden status [1, 12, 28].

	Limits trust in permissions plugins	**Prevents circumvention of plugins**	**Flexible plugins**
CoDRA [52], Pegasus [39], SweetDroid [40], SemaDroid [55], ipShield [38], BinderFilter [54], CRePe [41], Dr. Android [46], Boxify [35], NJAS [37]	✓	✓	✗
ASF [34]	✗	✓	✓
Xposed [29], XPrivacyLua [31]	✗	✗	✓
Dalf	✔	✔	✔

Table 1: A summary of how prior work compares to Dalf. We discuss prior work and provide detailed comparison with Dalf in Section 3 and Section 4.2, respectively.

infer keystrokes [48]. Based on their ratings, ipShield automatically recommends the policies that should be applied to the app.

Others: TISSA [56] is an early work that studied the feasibility of filtering data before delivering it to apps. MockDroid [36] is a version of Android that lets users send mock data to apps when they request for resources. SAINT [50] allows an app to provide policies to regulate how other apps may interact with it. Independent of our work, Android is experimenting with a plugin architecture to modify the appearance of the system UI [24]. In this paper, we propose plugins for flexible permissions.

4 OVERVIEW

In this section, we present the design of Dalf, an extensible permissions framework for Android, and the principles that guided it. In the rest of this paper, our discussions take place in the context of the permissions model used in Android 6.0 and onwards.

4.1 Trust model

The TCB (Trusted Computing Base) in this work is the Android OS, which includes the Linux kernel, the Android framework, and the device drivers. As Dalf enables users to install plugins to interpose on apps' access to sensitive resources, we expect users to trust the plugins they install. However, Dalf does not include plugins in its TCB and employs a variety of safeguards to protect users from malicious plugins. We detail these safeguards and malicious plugins in Section 4.3 and Section 4.4, respectively.

4.2 Design principles

Limit trust in permissions plugins: We believe that a prerequisite for the practical adoption of an extensible permissions architecture is the ability to limit the trust in the permissions plugins. In other words, the damage that a misbehaving (buggy or malicious) plugin can cause should be limited.

As the majority of prior work [34, 38–41, 52, 54, 55] execute their permissions plugins directly in the OS, the plugins run with elevated privileges. In these systems, a misbehaving plugin could in theory cause a lot of damage. As shown in Table 1 however, with the exception of ASF [34], many of them do limit the trust extended to plugins. This is a direct consequence of those systems requiring plugins to be written in custom domain-specific languages (DSLs) with restricted capabilities. The limitations of the DSLs are directly responsible for limiting the damage a misbehaving plugin can perform. Unfortunately, this also means that increasing the capabilities of the DSLs increases the damage potential of misbehaving plugins.

While Dr. Android [46], Boxify [35], and NJAS [37], do not modify the OS, they each have a trusted component that executes permissions policies and mediates resource requests made by a sandboxed app. The current implementations of these systems support relatively simple policies. If they were to support more complex policies, the potential damage of a misbehaving policy would also increase since the trusted components are more privileged than the sandboxed app.

Xposed [29], XPrivacyLua [31], SmarPer [49] run the plugins in the address space of each app process. In this approach, misbehaving plugins may exfiltrate confidential information from the app. We demonstrate this by developing a custom Xposed plugin that perturbs location data when the app receives it from Android framework via IPC. It hooks `Location.CREATOR.createFromParcel()`, an internal Android method that deserializes the location data structure received in a parcel, to obfuscate the location after deserialization completes. It also hooks the constructor of the `android.text-.TextView` class to exfiltrate the content of all text elements in the UI. At the moment, the plugin prints them to `logcat`, the system logger in Android. However, it can be easily modified to send them to a remote server over the Internet. We tested this plugin with Telegram [25], a secure messaging app, by manually sharing the current location with a Telegram contact. As expected, the shared location was perturbed. However, the plugin was able to access the chat messages since they are rendered on the UI.

Prevent circumvention of permissions plugins: It should not be possible for apps to circumvent the plugins applied on them. This requires the plugins to run their arbitration logic *before* apps receive the requested resource. Consider instead Xposed-based frameworks. Since an Xposed plugin's hooks run in the same address space as the app, they execute only after the data is received in the app process. Hence, it is possible for apps that are aware of these plugins to change their process environment and access resources before the plugins modify them. To demonstrate this, we added about 100 lines of code to the free and open-source version of Telegram [26].

We first added a new method to manually deserialize the location from a parcel. Next, we used YAHFA [32] to modify the pointer of the `Location.CREATOR.createFromParcel()` method to point to the new method during runtime. We used this modified version of

Telegram with XPrivacyLua, reran the test described earlier, and found that the shared location was obfuscated. However, recall from Section 2.2 that in binder IPC, methods run *after* the deserialization of objects from the parcel. The new method we added is invoked during the deserialization step. When it returns, the location object it creates is eventually perturbed by XPrivacyLua. However, it is too late by this time: the app has already observed the true coordinates during deserialization in our method. Thus, XPrivacyLua was only providing an illusion of safety to the user in this test. We also used our custom malicious Xposed plugin and found it to be ineffective as well. Note its doubly harmful effects in this test: it did not provide any protections *and* it was exfiltrating chat messages.

Flexible permissions plugins: We believe developers should be able to create plugins without unnecessary restrictions to aid the traction of an extensible permissions framework . SemaDroid [55], ipShield [38], and Pegasus [39], among others, use domain-specific languages (DSLs) to write permissions plugins and are limited. For example, they may not be able to maintain program state, run arbitrary computations, or access other resources such as the Internet. These limit the types of plugins that can be developed. Consider a trusted privacy watchdog organization (e.g., the EFF [14]). They may wish to create a plugin that tracks if certain apps make network requests to suspicious IP ranges. The plugin itself may need to periodically connect to the Internet to update the IP ranges. Such plugins are not possible in prior work that rely on restrictive DSLs. Dr. Android, Boxify, and NJAS, do not allow third-party developers to create custom permissions policies. By default, they support certain predefined but configurable policies (e.g., "prevent the sending of text messages to particular phone numbers").

Allow support of all resource types: A permissions architecture's design should not be limited to certain resource types. In contrast, consider prior work: SemaDroid and ipShield focus on sensor data such as the accelerometer; BinderFilter [54] does not support interposing accesses to files and does not appear to support perturbing the frames in a camera stream; ASF does not support the camera stream either and although it supports interposing accesses to files, it does so by rewriting the app, which may be defeated if apps use direct syscalls (e.g., `open()`) in native code.

4.3 DALF

Dalf is an extensible permissions architecture for Android that complements its existing permissions model. In Figure 1, we illustrate a high-level overview of Dalf. It enables users to apply plugins on a per app per resource basis. If an app accesses a permitted resource and there is a plugin applied for that resource, Dalf invokes the plugin using binder IPC, waits for a reply from the plugin, and then responds to the app's request based on the reply. It provides the plugin the identity of the app and an unmodified version of the data. The plugin may allow the request by returning the input data, deny the request by completely redacting the data, or return some modified version of the data. In Dalf, plugins are essentially apps that implement the Dalf plugin API described in Section 5. That aside, they have the same capabilities as regular apps. For example, they may have a UI or request permissions from the user to access additional sensitive resources to determine the current context of the device (e.g., time of day, current location, etc.).

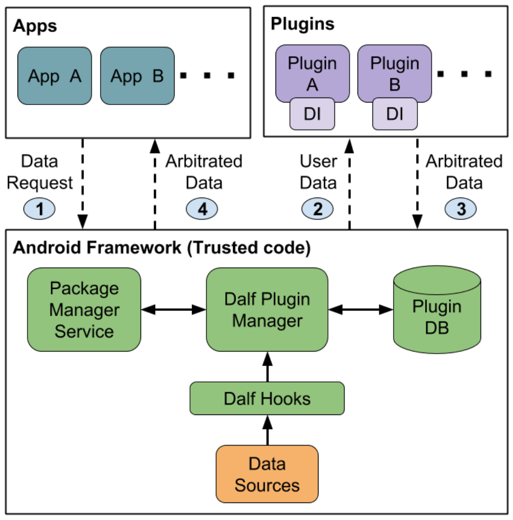


Figure 1: A high-level overview of Dalf's design. The green boxes within the Android Framework represent the components we added or modified; the numbers next to each line indicate the order of data flow between apps and plugins; the `DI` next to each plugin are the *data interposers*, which plugins must implement to arbitrate resources; and both the regular apps and plugins run as separate processes and are not inherently trusted by Dalf.

In practice, we envision the presence of a plugin store similar to app stores. We expect privacy experts to develop and publish different kinds of plugins on the plugin store, and expect users to install the plugins that suit them. We believe that the plugins' reputation and trustworthiness would depend on the user ratings of each plugin, the reputation of the entities making them, and whether or not the source code for those plugins are available.

Restrictions: We intentionally placed two restrictions on our design. First, a plugin's accesses to resources are not themselves mediated by other plugins. Second, users may only apply a single plugin for each resource type accessed by an app. Plugins may not be composed for an app-resource pair. From an implementation perspective, it is straightforward to remove both of these restrictions. However, doing so makes it difficult to reason about the protections ultimately applied to apps. We leave a feasibility study of loosening these restrictions to future work.

Satisfying the design goals: Our approach of having plugins as apps meets the first three goals stated above. First, they are subject to the same isolation mechanisms Android places on regular apps. They do not run with elevated privileges in the OS nor do they have access to apps' address spaces. Second, because plugins run as separate processes and are invoked before resources are delivered, apps only receive arbitrated data and cannot circumvent plugins. Third, aside from having to implement the Dalf plugin API, developers may use the same tools they use to develop Android apps to create plugins and have access to the same capabilities. Finally, Dalf's general design supports all resource types. We demonstrate this with our prototype in Section 5.

4.4 Malicious plugins

In this section, we discuss what might happen if users inadvertently install malicious plugins while using Dalf and potential mitigation measures. As plugins are apps, they are subject to Android's application sandbox which isolates apps from each other and from the OS. Thus, a malicious plugin cannot directly interfere with other apps or the OS. However, for each app-resource combination, a malicious plugin (a) has access to the raw sensitive data, (b) may track apps' access patterns to the resource, and (c) send apps garbage data when they access the resource. A malicious plugin can also access other resources on the device if granted by the user.

Instead of providing plugins the raw data, Dalf can provide *opaque references* [45] to require plugins to perform their manipulation tasks without directly accessing the data. For instance, the GeoInd plugin described in Section 6.1 uses differential privacy to perturb the user's location. It can function correctly with an opaque reference to the location since its perturbation method is independent of the user's *true* location values. It is difficult to prevent a plugin from tracking access patterns because the plugin must be invoked for each resource request by an app. However, we can prevent plugins from sharing such data with the outside world, e.g., by denying the `INTERNET` permission. To prevent plugins from sending garbage data, Dalf can perform integrity checks on the output of plugins before sending the perturbed data to apps. That said, it may be difficult to distinguish valid perturbed data from garbage data.

Since plugins are apps, we may borrow the measures that are employed to reduce the chances users install malicious apps to also reduce the probability that malicious plugins are installed in the first place. We can establish a vetting mechanism for plugins or plugin developers, similar to the vetting mechanisms of regular apps, e.g., Google Play Protect [18]. Observe that the scope of a plugin is much more narrow compared to a regular app: a plugin's focus is on resource mediation while apps are far more richer in functionality. Hence, we argue that in most cases vetting a plugin would be easier than vetting a regular app. For instance, while a normal app may use a lot of external libraries (e.g., for ads), which are a prime source of malicious/buggy behavior [44], we can ensure plugins only use trusted libraries (e.g., OpenCV for computer vision tasks). Alternatively, the vetting process may be stricter for plugin developers and require them to submit the source code of their plugins for malware analysis.

Finally, similar to open-source privacy plugins for web browsers [2, 16, 20, 22, 27], we believe useful Dalf plugins will gain popularity and trustworthiness by being open-source. We also envision a repository where developers are encouraged to publish open-source plugins (such as F-Droid [15]). Although this does not prevent a malicious open-source plugin from existing, it decreases the opportunity for plugin developers to hide malware in their code.

Note that our plugins as apps design allows users to uninstall a plugin as soon as they identify it as malicious. This is akin to uninstalling a malicious app. While it does not undo the damage done (e.g., the leaking of resource data), it prevents further damage.

5 IMPLEMENTATION

We implemented a prototype of Dalf for Android 8.1 (AOSP branch `android-8.1.0_r1` [10]). We modified the Android framework to implement a plugin API, allow users to install plugins, and to provide a companion settings app that lets users select the plugins to apply for each app-resource combination.

5.1 Plugins

Each plugin must meet the following requirements. First, it must be able to run in the background using Android's `Service` class. Next, it has to declare the resources it arbitrates in a plugin manifest file and for each of them, define a corresponding *interposer* (explained below). Finally, it must implement Dalf's `PluginService` class and override the appropriate `get()` methods in the class to return the interposers. Aside from these, plugins have access to the same capabilities as regular apps e.g., use native code, provide a UI, etc.

When the user installs a plugin, the `PackageManagerService` reads its manifest and populates a new plugin database. This database is used by the settings app to list the available plugins to the user and to record the plugins applied for each app-resource combination. During runtime, the Android framework starts a plugin's background service on demand, when there is a need to arbitrate on a resource for an app. It retrieves the interposer corresponding to that resource from the plugin, invokes its methods using binder IPC, and then responds to the app based on the plugin's reply. Our prototype only starts a single instance of each plugin and shares it across all the apps that require it.

5.2 Interposers

Interposer	Data type	Arbitrates
Location	Plain object	Device location.
Contacts	Content provider	The user's contacts database.
Calendar	Content provider	The user's calendar database.
Camera	Streaming	Frames from a camera stream.
External storage	Kernel-provided	External storage file access.

Table 2: The resource interposers currently supported by the Dalf prototype.

In Dalf, an *interposer* is a binder object represented as a Java class with several abstract callback methods. These methods must be implemented by the plugin and are invoked by the framework to interpose on a resource. In general, the framework passes the app's name and the resource data to the interposer and in return, the interposer must respond with how the app's resource access should be handled. As the details of an interposer differ based on its data type, we discuss them separately below.

5.2.1 Plain objects. It is straightforward to interpose on resources that are simple class objects. For example, location data in Android are represented as instances of the `Location` Java class. It uses field variables to represent attributes such as latitude and longitude. Our modifications to the Android framework to handle plain objects are summarized as follows:

During serialization, Dalf tracks whether a parcel contains plain objects that represent resources to be interposed on and if so, their positions within the parcel. Before the parcel is sent in a transaction to an app, the framework loops through each object and invokes the callback of the appropriate plugin interposer using binder IPC. In doing so, the framework provides the plugin the app's name and a copy of the object. In response, the interposer must return an object instance of the same type but it has the freedom of changing the values of the object's fields. Once all interposers are invoked, the framework creates a copy of the original parcel but replaces the objects that were interposed on with the objects received from the plugins. Finally, the framework resumes the initial binder transaction to the app but sends the new parcel instead of the original parcel.

5.2.2 Content providers. A class of resources in Android, such as contacts, calendar, and call logs, are accessed using the *content provider* design pattern [4]. Apps need to query the framework[2] and provide (i) a URI that identifies the data type, (ii) the specific data columns the app needs, e.g., display name and phone numbers of contacts, and (iii) other arguments that may affect the results of the query, such as conditional operators. A query's results are tabular in nature, much like the results of a regular database query.

The framework maintains a different content provider for each resource URI. When a content provider receives a query it is responsible for, it executes the query and instantiates a `CursorWindow` object, thereby allocating a chunk of memory. It writes the query results into the allocated memory and sends a `Cursor` object back to the app so that it may read the results. Internally, the cursor uses a read-only `CursorWindow` handle to the memory allocated by the content provider.

Similar to plain objects, we support such resources by keeping track of whether a parcel contains `CursorWindow` handles that were created because the app queried for data interposed on by a plugin. If so, the framework invokes the corresponding interposer and provides the app's name and the `CursorWindow` handle. In response, the interposer must return a `CursorWindow` handle to return to the app. If a plugin needs to customize the data returned, the interposer must create a new instance of the `CursorWindow` object, fill the necessary fields, and return the handle.

5.2.3 Streaming data. A typical Android device has several streaming data sources such as the accelerometer, gyroscope, camera, etc. The framework delivers data from these sources to apps using different implementations of the producer-consumer pattern. We support them in DALF by allowing plugins to relay the data flow between the producer at the source and the consumer at the app. We discuss how our prototype supports the camera stream below.

An app has to send a capture request to `libcameraservice`, a framework component, to access the camera. At a high-level, a request contains camera configuration parameters and the surface that will receive the camera output (i.e., the rendering target). Internally, a surface is a `BufferQueue` object with references to a producer-consumer binder object pair [9] and a capture request contains the surface's producer object.

When `libcameraservice` receives a request, it verifies that the app has permissions to access the camera, creates a new camera stream, configures it according to the parameters of the request, and registers the surface's producer object with the stream. Subsequently, when the stream renders new frames, it uses the producer object to deliver the frame handles to the app. It leaves it to the app to use the corresponding consumer object to read the frames.

In our prototype, when `libcameraservice` configures a camera stream for an app, it checks if the app's camera access is arbitrated by a plugin. If so, it invokes the `shouldInterpose()` method on the camera interposer and passes the app's name and surface producer object. This enables the plugin to decide whether or not to interpose on that particular stream. If it does not, then the stream is configured normally and frames are rendered directly to the app's surface.

Otherwise, the plugin creates a new surface of its own and sends that surface's producer object back to `libcameraservice` which will, in turn, register the producer received from the plugin in the camera stream. Now, whenever the camera stream renders a new frame, the plugin's camera interposer's `onFrameAvailable()` callback will be invoked with a pointer to the raw frame data to let the interposer modify the frame. Once the method returns, the frame is copied and a handle to the copy is sent to the app (using the app surface's producer object received in the earlier invocation to `shouldInterpose()`).

Our prototype introduces a new `InterposableSurface` object to perform the heavy lifting necessary for the above tasks and to simplify the development of a camera plugin. Note that frames must be copied to prevent data leakage. We observed that the camera stream reuses the memory allocated for past frames to render a new frame. If instead of performing a copy, we pass the original handle received from the camera stream to the app, it may be able to read new frames directly and circumvent the plugin.

Our prototype does not yet support other sources of streaming data, such as the microphone and sensors. Our initial investigations suggest that an approach similar to the above will work as they use different implementations of a producer-consumer model (e.g., `BitTube` [11] in the case of sensors).

5.2.4 Kernel-provided resources. Certain resources, such as files and the network, are provided by the OS kernel. The Android framework is only involved in granting an app the capabilities to request those resources. Once granted, apps may access them using the appropriate Linux syscalls. In the following, we detail how our prototype uses `ptrace`, a syscall tracer in Linux, to interpose on accesses to files on the external storage partition. In Android devices, this partition typically has a large capacity and is used for mass storage. Our approach is similar to that of MBOX [47].

In general, `ptrace` allows a *tracer* process to trace the syscalls made by another *tracee* process. When the tracee calls a syscall, the kernel traps to the tracer twice: once before the kernel executes the syscall (`syscall-enter-stop`), and once afterwards (`syscall-exit-stop`). The former allows the tracer to inspect and modify the input arguments to the syscall and the latter allows the same for the syscall's return values.

As explained in Section 2.3, Android uses the zygote process to launch new apps. We tweaked it so as to be able to use `ptrace`. As usual, zygote performs a `fork()` to start a child process to run the app. We denote this child C_A. If the user applied a plugin on the app to arbitrate accesses to external storage, zygote performs a second `fork()` and the second child, C_T, is designated as C_A's tracer.

[2]See: `ContentProvider.query()`

We synchronize C_A and C_T using shared semaphores so that the tracer may set various ptrace options before C_A starts executing app code. For example, since Android apps are inherently multi-threaded, it sets the PTRACE_O_TRACECLONE option to monitor the syscalls made by all threads in C_A.

When C_T traps because of syscall-enter-stop, it identifies the syscall being called. If it is one that is used to open files, it reads the filename argument to check whether it resides in external storage. If so, it invokes the storage interposer of the plugin and sends the app's name and the filename. The interposer must now return a path to the file that should be opened. It may allow the access, deny it (by returning an empty string), or redirect it to a different file by returning a new path. In the last two cases, the tracer copies the returned string into an unused portion of the app thread's stack (a popular thread-safe trick of placing new data in the app [21]), changes the filename pointer argument to point to the new string, and then resumes the syscall.

Many of the ptrace operations, such as identifying the syscall being called, are architecture-dependent. Our prototype currently supports aarch64 (ARMv8-A, 64-bit mode), the architecture used in modern Android devices, and only checks for calls to openat(), the syscall used to open files in aarch64. We leave the support of other architectures to future work. Even though we focus on interposing on the external storage in this paper, this ptrace-based approach can also be used to interpose on other resources accessed through syscalls, such as network requests.

6 PERMISSIONS PLUGINS

We demonstrate the ability of Dalf to address the diverse privacy requirements of users. In this section, we describe four different motivating scenarios and discuss Dalf plugins to address them.

6.1 Location plugin

Use case: *Alice has an app on her smartphone to discover nearby points of interests (POI) such as restaurants, bars or coffee shops. It sends her location to the cloud to retrieve the POI near her location. As Alice is afraid the app might track her movements, she often avoids using the app even though she likes it.*

To address such scenarios, we developed GeoInd, a plugin that perturbs a user's location data with geo-indistinguishability [33], a differentially-private location obfuscation technique. It adds a carefully chosen amount of noise to the user's location so that with high probability, an adversary cannot infer a user's true location even after observing the noisy location. The amount of noise added depends on r, a user-specified parameter that represents the radius within which the user wants her privacy protected. Increasing r adds more noise to the location and improves privacy but reduces the accuracy of the POI results. Alice may install this plugin and choose r as desired to receive reasonably accurate results while protecting her true location.

6.2 Contacts plugin

Use case: *Bob uses a social networking app and it allows him to find his friends on the network by giving it access to his address book on the phone. Although he likes this feature, he does not want to share* all *of his personal contacts.*

We developed ContactsGuard, a plugin that uses the contacts interposer to run user-specified policies each time an app queries the contacts database. The plugin inspects the results and removes or perturbs entries based on the policies. Bob may install this plugin and set a policy to filter out contacts he does not want to share with the social networking app. Our prototype implementation of ContactsGuard supports two policies: (i) filtering out contacts based on their phone numbers' area code, and (ii) hide email addresses.

6.3 Camera plugin

Use case: *Charlie installed a translation app on his phone when he traveled overseas because it simplified translating foreign language text into his native language. He only had to point his phone camera at the text and the app would translate it. He wants to use it back home, too, to translate foreign language documents. However, he is worried that he might inadvertently share images of sensitive information lying around in his house, such as bank statements and family pictures.*

As discussed in Section 3, PrivateEye [51] addresses the above problem with a default-deny approach. Users have to specifically identify the portions of the physical environment considered public using special markers. PrivateEye then uses computer vision techniques to identify and disclose only the portions of the image frame that lie within the markers. As PrivateEye is implemented by modifying the camera subsystem, it affects all camera apps and does not consider user context.

Using Dalf, we implemented GeoPrivateEye, a location-aware PrivateEye plugin. It allows users to specify the locations where they are concerned about inadvertently sharing sensitive information and only enables PrivateEye in those locations. Charlie may install this plugin, add his home as a sensitive location, and apply it to the translation app. Consequently, when he launches the app at home, the plugin blocks everything by default. He will have to place markers around the documents he wants to translate. However, once he leaves the house, the plugin will not perturb the image the app receives. GeoPrivateEye demonstrates how a plugin in Dalf may access data from one resource (location) to arbitrate another resource (camera).

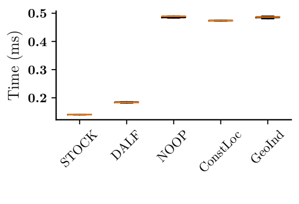
6.4 Storage plugin

Use case: *Eve stores photos and sensitive documents on the external storage partition of her phone. She wants to use an image-sharing app to share her photos with friends and family. If she uses one, she has to grant it the permission to read files on the external storage since her photos are stored there. However, in doing so, she will also grant it the permission to access her sensitive documents. Hence, she is afraid of using such an app.*

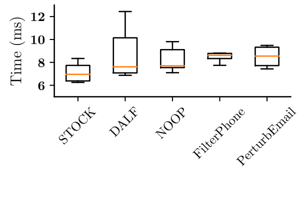
We developed a FileGuard plugin that implements a storage interposer to handle these scenarios. Similar to ContactsGuard, it allows user-specified policies to dictate whether accesses to files on the external storage should be allowed. It currently supports the following policy types: (i) allow access to whitelisted files, (ii) deny access to blacklisted files, and (iii) deny access to photos based on where they were taken. To support the last policy, it relies on the location data stored in the photo's exif metadata. Eve may apply the FileGuard plugin to an image-sharing app and whitelist just

Resource	Configuration	Description
	Stock	Unmodified Android 8.1 (AOSP branch `android-8.1.0_r1` [10])
	Dalf	Android 8.1 instrumented with our permissions plugin framework.
Location	LocNoOp	Dalf + location plugin that does not perform any operation.
	ConstLoc	Dalf + location plugin that replaces the original location with a constant location.
	GeoInd	Dalf + GeoInd plugin.
Contacts	ContactsNoOp	Dalf + contacts plugin that does not perform any operation.
	FilterPhone	Dalf + ContactsGuard plugin that filters out private contacts.
	PerturbEmail	Dalf + ContactsGuard plugin that perturbs emails.
Camera	CameraNoOp	Dalf + camera plugin that does not perform any operation.
	Blur	Dalf + camera plugin that blurs camera frames.
	GeoPrivateEye	Dalf + GeoPrivateEye plugin.
Storage	StorageNoOp	Dalf + storage plugin that does not perform any operation.
	Whitelist	Dalf + FileGuard plugin with a whitelisting policy.
	ImageGuard	Dalf + FileGuard plugin with location-based access control to images.

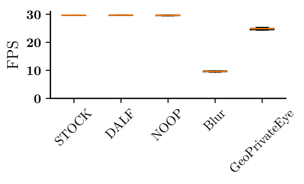
Table 3: Configurations used in evaluating Dalf.



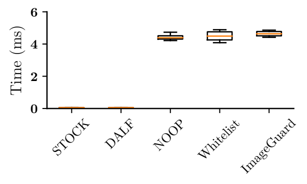
(a) Location retrieval time



(b) Contacts load time



(c) Camera preview FPS



(d) File read time

Figure 2: The performance slowdown in Dalf when accessing various resources. The NoOp in (a), (b), (c), and (d) represent LocNoOp, ContactsNoOp, CameraNoOp, and StorageNoOp, respectively.

the photos directory on her external storage, thereby preventing the app from accessing anything else.

7 EVALUATION

To evaluate Dalf, we ask two questions: (i) Is its design practical? (ii) How do plugins behave with real world Android apps? To answer the first question, we measured the different aspects of overhead of our prototype. We answer the second question by manually running real-world apps with our plugins and performing a qualitative investigation. In this section, we report our findings.

7.1 Experimental methodology

In order to perform the overhead experiments, we used two different variants of Android: Stock, an unmodified version of Android 8.1, and Dalf, which is Android 8.1 with our permissions framework implemented. We used Stock as the baseline and ran different workloads on Dalf using four microbenchmark apps, one for each resource type. We used each app with a plugin that interposes on the corresponding resource type. The apps and their workloads are detailed below:

- LocFinder retrieves the user's location repeatedly a 100,000 times using the `LocationManager.getLastKnownLocation()` API.
- ContactsLoader retrieves the display name, phone numbers, and emails, of the contacts saved on the phone using the `ContentProvider.query()` API. We pre-populated the phone with 100 contact entries.
- CameraPreview displays the frames from a camera stream (i.e., a camera preview) for 30 seconds using the `camera2` API [3].
- FileReader reads all the files in a specified directory located on the phone's external storage. In our workload, the directory has 1000 files, each of size 1 KB.

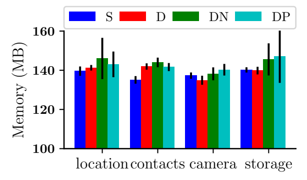
In Table 3, we list the complete set of experimental configurations we used. Note that we ran all our experiments on a Pixel 2 XL smartphone, which has an octa-core CPU and 4 GB of RAM. We used Python scripts that control the phone with ADB (Android Debug Bridge) to run the experiments systematically.

7.2 Performance slowdown

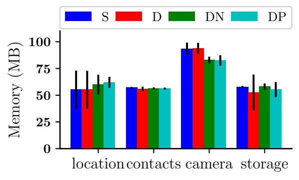
We instrumented our microbenchmark apps to compute the time elapsed between requesting for a resource and receiving it. We use this to measure the slowdown plugins impose from an app's perspective. For location, contacts and storage, this is given in milliseconds. For the camera, we measured the frames per seconds (FPS) of the preview. Our results are illustrated in Figure 2.

As the graphs illustrate, Dalf's impact on apps when plugins are not applied is negligible. When plugins are applied, there is a visible slowdown on the time taken for apps to receive resources. In location and contacts, the slowdown is 3.5x and 1.2x, respectively. We attribute this to the additional IPCs in Dalf from the Android framework to the plugins. However, in absolute terms, the slowdowns are less 1 ms.

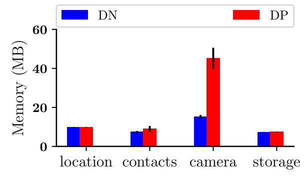
In the case of camera, Blur and GeoPrivateEye deliver the preview frames at a median rate of 10 FPS and 15 FPS, respectively. GeoPrivateEye is faster than Blur even though it employs complex detection algorithms because unlike Blur, it resizes the input frames (from 4032×3024 to 400×240) before performing its detection. Note that GeoPrivateEye does not achieve the frame rate reported in the original paper (20 FPS) because we did not employ



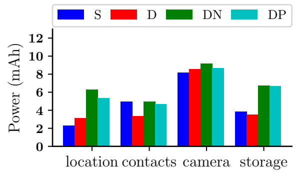
(a) System memory usage



(b) App memory usage



(c) Plugin memory usage



(d) Battery usage

Figure 3: The memory overhead and battery usage in Dalf under different workloads. The configuration labels are as follows: S=Stock, D=Dalf, DN=Dalf_NoOpPlugin and DP=Dalf_Plugin. In the DP configurations, we used GeoInd, FilterPhone, GeoPrivateEye and ImageGuard while accessing location, contacts, camera and external storage, respectively.

all of its optimization techniques [51]. Nevertheless, our proof-of-concept implementation shows the feasibility of incorporating PrivateEye-like privacy solutions as plugins in Dalf.

Finally, reading a file in Dalf with a storage plugin is significantly slower (≈ 80x) compared to Stock. We believe this is due to the inefficient implementation of the ptrace-based tracer in our prototype that intercepts every syscall made by the app as opposed to only those related to the file access. Android apps invoke a large number of syscalls over their lifetimes. For example, our microbenchmark app made over 15 K syscalls to perform a variety of tasks in addition to reading the files in our workload, such as invoking the binder driver to send and receive IPC calls. We discuss inefficiencies of the tracer and several ways of addressing them in Section 8.

Since Dalf invokes the plugin on demand, we also measure the latency due to launching a plugin. In our experiments, we found that the overhead of launching a plugin is very small (< 2 ms) across all the resources. However, it also depends on the implementation of the plugin. For plugins with large startup times, Dalf may be modified to start them together with their target apps.

7.3 Memory overhead

We used Android's `dumpsys` tool [5] to measure the memory usage of apps, plugins, and the system services. We use the PSS (Proportional Set Size) metric to report our results, which proportionally includes the memory shared by an app with other processes. For example, if a process A uses 100 MB of memory in total and 50 MB of it is shared with another process, A's PSS is 75 MB. The results of our memory experiments are illustrated in Figures 3a, 3b, and 3c.

Figure 3a shows that Dalf's memory overhead on the system process is up to 12 MB across all configurations. This suggests that the Dalf modifications to the system services have minimal impact on memory. This is desirable because the permissions framework code runs all the time, irrespective of the apps and plugins currently applied. Similarly, Figure 3b shows that the memory usage of the microbenchmark are also consistent across different configurations.

Finally, Figure 3c shows the memory used by plugins. It depends on the resource it arbitrates and the arbitration mechanism. In the cases of location, contacts and storage, the plugins only use a small amount of memory over the NoOp plugin as they perform relatively simple operations (e.g., adding noise to location data in case of GeoInd). However, in the case of the camera, GeoPrivateEye uses about 45 MB of memory on average because it performs intensive image processing. Note that the authors of PrivateEye also reported a similar memory overhead (40 MB) [51].

7.4 Battery usage

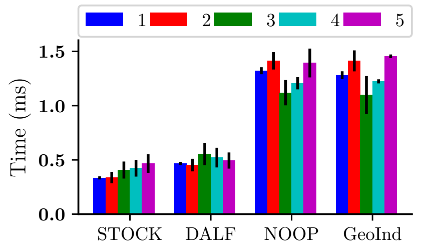
We used `batterystats` [7] to measure the usage of the phone's battery in Dalf. To keep the experiments consistent, we fixed the screen brightness, disabled bluetooth and the phone network, and controlled the phone wirelessly with WiFi-ADB. In our prototype, our modifications to the Android OS affect both apps and the system processes (Android framework). Therefore, we rely on the `discharge` metric of `batterystats`, which shows the amount of battery discharged since the device was last charge, to obtain end-to-end measurements. We reset `batterystats` before each experiment run so that `discharge` represents the amount of battery discharged during the experiment.

Figure 3d illustrates our results. Unlike the time and memory results, we could not observe a clear impact on power due to Dalf and found a lot of variations in the results. We attribute this to the `discharge` metric which is affected by many processes running in the phone (e.g., thermal-engine) that are difficult to account for. Moreover, the microbenchmark we used for evaluation, perform relatively small (but realistic) tasks requiring very little energy. Other components such as the display use far more energy resulting in high variations in our energy measurements. These results suggest Dalf's overall impact on the battery may be low.

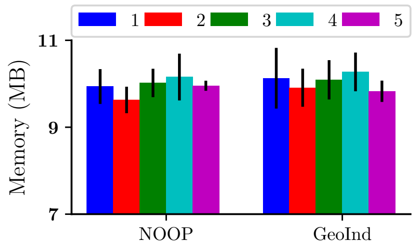
7.5 Scalability

Thus far, we evaluated Dalf with a single app using a single plugin. In this section, we evaluate the performance when multiple apps access a resource mediated by a single plugin. To answer this question, We simultaneously ran multiple instances of the LocFinder and FileReader microbenchmark apps. We focused on these two resources since according to our previous results, they are on the opposite ends of the performance-slowdown spectrum. As Android limits the number of foreground apps that can run simultaneously, we modified our microbenchmarks so that they run their workloads in a background service and ran multiple instances of the services. Our results are illustrated in Figure 4.

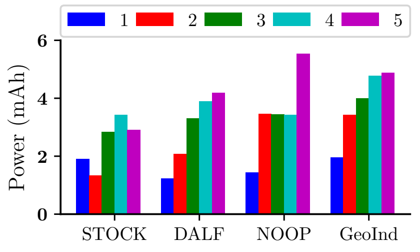
In the case of location, Dalf's impact on resource consumption remains low as the number of apps increases. However, observe that the absolute time taken to access the location has increased. In the case of Stock with a single app, it takes about 0.4 ms to access the location compared to 0.05 ms in Figure 2a. As we ran the previous experiments with an app running in the foreground, we



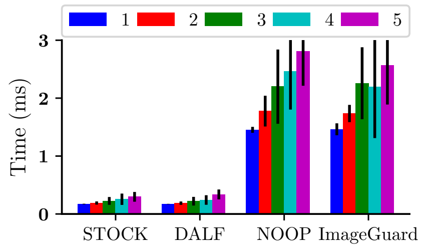
(a) Location retrieval time



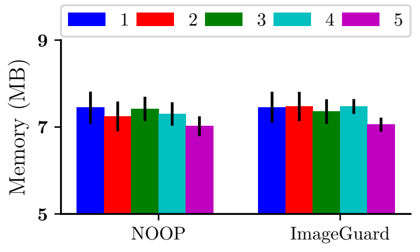
(b) Memory usage of plugins



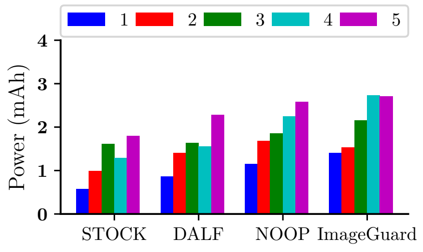
(c) Power consumption



(d) File read time



(e) Memory usage of plugins



(f) Power consumption

Figure 4: Performance of DALF as we apply plugins to multiple instances of LOCFINDER (top row) and FILEREADER (bottom row). The numbers in the legend indicate the number of apps (instances) running simultaneously.

attribute the increase in time to Android's prioritization scheme that prioritizes foreground apps over background services [8].

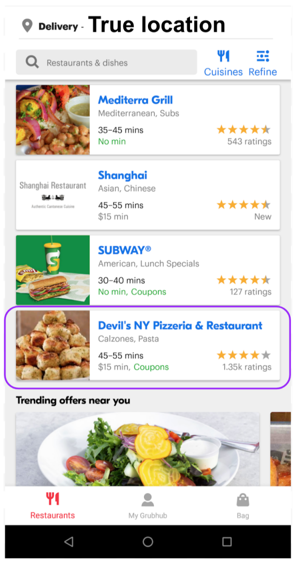
The results for the storage experiments exhibit a similar trend with one key exception: the absolute time taken to read files are lower than the results shown earlier in Figure 2d. We reran the storage workload experiments both in the foreground and background, and closely examined the timestamps observed by the tracer when the apps tried to read files. The results confirmed that the app running as a background service was indeed reading files faster than in the foreground. We speculate that the tracer, which runs as a separate process, interacts with Android's prioritization scheme in a way that favors the background service over the foreground.

7.6 Real world apps

We tested our plugins with real world apps downloaded from the Google Play Store and F-Droid [15]. Our goal was to understand how well the plugins worked and the issues either the plugin developers or users would face.

We demonstrate this with *Grubhub* [19], an app to order food from restaurants. First, we searched for nearby restaurants without any plugins. Next, we repeated the search with the GEOIND[3] plugin applied to perturb our location. In both cases, we sorted the results by the restaurants' distance to the current location. Figure 5a and Figure 5b show the screenshots of the app's results page from both experiments, respectively. First, observe that the list in the two experiments are different. This is expected as the location given to the app is perturbed when the plugin is applied. Second, the overlap of the top results in the two lists is a result of running the experiment in an area with a relatively small number of restaurants.

[3] We set ϵ, the differential privacy budget, to 0.1.



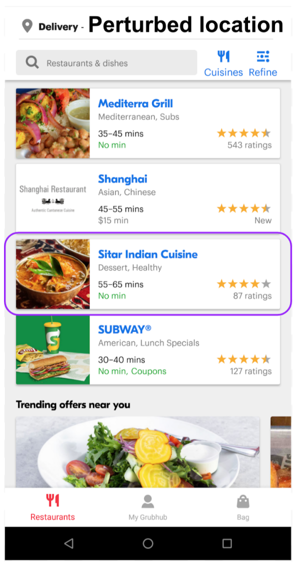
Mediterra Grill
Mediterranean, Subs
35–45 mins
No min
543 ratings

Shanghai
Asian, Chinese
45–55 mins
$15 min
New

SUBWAY®
American, Lunch Specials
30–40 mins
No min, Coupons
127 ratings

Devil's NY Pizzeria & Restaurant
Calzones, Pasta
45–55 mins
$15 min, Coupons
1.35k ratings

(a) STOCK



Mediterra Grill
Mediterranean, Subs
35–45 mins
No min
543 ratings

Shanghai
Asian, Chinese
45–55 mins
$15 min
New

Sitar Indian Cuisine
Dessert, Healthy
55–65 mins
No min
87 ratings

SUBWAY®
American, Lunch Specials
30–40 mins
No min, Coupons
127 ratings

(b) GEOIND

Figure 5: Results of finding nearby restaurants in *Grubhub* app using (a) user's true location, and (b) the location perturbed via GEOIND plugin. The marked restaurants are the ones that differ in both the results.

Interestingly, this suggests that there are cases where the utility of an app may only be minimally affected by the use of plugins that perturb resources.

We also discovered that a major benefit in Dalf, compared to approaches like Xposed [29], is that a plugin developer only has to focus on whether an app uses a particular resource such as location. She does not have to reason about the myriad Android APIs that provide location data and hook them all. Consequently, this significantly reduces developer effort (our GeoInd plugin is written in less than 200 lines of Java).

In the cases where we did observe issues between apps and plugins, they were due to bugs in our prototype which we eventually fixed. Given the promising results, we believe an important next step would be to work on the limitations that will prevent Dalf from being practically adopted.

8 LIMITATIONS

We now discuss the design and prototype limitations of our work.

8.1 Design

Reasoning about app semantics: In Dalf, it is difficult for a plugin to reason about *why* an app is requesting a particular resource. Consider the use of GeoInd with Google Maps: since it indiscriminately adds noise to the user's location each time the app requests it, the blue dot in the app that indicates the user's current location keeps moving. Consequently, features of the app such as finding directions from the current location become unusable.

One approach to address this limitation is to allow users to apply the plugins on a per-app-feature basis. For example, a user may want to allow Google Maps to access her location when she is using it for navigation, but not when she is merely browsing the maps. Alternatively, Dalf may provide plugins additional context surrounding the app's resource request, such as whether the app is running in the foreground, whether an app's resource request was initiated by a user input event, etc. In this latter approach, we need to take into account malicious plugins so that they are not provided excessive information about users or their apps. We consider this the most important limitation to address. Doing so will increase the granularity of the permissions model in Dalf and enable a wide class of plugins.

Collusion between apps: Apps on Android can collude in order to evade the existing permissions model. For example, an app permitted to access the location may send it to other apps that do not have the location permission. This is known as the *confused deputy* attack [42] and Dalf currently does not have any provisions against it. Techniques proposed in prior work such as Quire [42] and BinderFilter [54] may be applied in Dalf to address this limitation.

8.2 Prototype

Unimplemented features: Aside from the aforementioned limitations, the design of Dalf is general and there is a sheer number of possible actions that a plugin may wish to perform: interpose accesses to sensor data (e.g., accelerometer); interpose accesses to other content provider data (e.g., SMS); tweak the parameters of a camera capture request; control the network accesses of an app with a IP whitelist or blacklist; allow a resource request only for the next N minutes; etc. These actions are currently not supported by our prototype because the required machinery have not been implemented. We expect to be able to add these features using the binder and `ptrace` techniques described earlier.

Inefficient syscall tracer: The `ptrace`-based tracer used to control an app's access to the external storage is inefficient. First, `ptrace` traces *all* syscalls and not just `openat()`. Hence, the tracer traps twice for every single syscall called by the threads in an app. Second, for each `openat()` syscall, the tracer has to perform additional syscalls to identify the syscall called by the tracee and to read the filename argument. If the plugin returns a new file path, the tracer has to use more syscalls to copy the new path into the app thread's stack and modify the syscall argument. Third, a single instance of the tracer runs for all threads in an app. If multiple threads in the app call syscalls in parallel, the tracer will handle each of them in a serial manner, thereby unnecessarily impeding parallelism.

There are several ways of improving the tracer's performance. First, we can trace only the required syscalls using `BPF` (Berkeley Packet Filter) with `seccomp` [23, 47]. Second, we can adopt an experimental `seccomp` patch to defer syscall decisions to user-space [13]. This will reduce the number of syscalls required when plugins opt to load a different file than what the app requested. Finally, we can modify the Linux kernel to ensure `ptrace` works in a parallel manner.

In-band frame processing: Our Dalf prototype invokes the `onFrameAvailable()` callback on a plugin's camera interposer whenever a new frame is received from the camera stream. Since the plugin is required to finish perturbing the frame before the callback method returns, our prototype only supports in-band frame processing at the moment. It thus prevents camera plugins from performing certain kinds of optimizations, such as using multiple threads to process frames in a pipeline. Enabling out-of-band processing is a matter of modifying the prototype so that plugins are required to call a `sendFrameToApp()` method when they have finished perturbing a particular frame. In this approach, `onFrameAvailable()` will become a non-blocking call that immediately returns after delivering the newly available frame to the plugin.

9 CONCLUSION

In this paper, we presented Dalf, an extensible permissions framework for Android. It gives users flexible control over the resources accessed by apps using permissions plugins. The key in Dalf is that the plugins are apps themselves. This grants them a wide set of capabilities while also (i) limiting the amount of trust extended to the plugins, and (ii) preventing apps from circumventing the plugins applied on them. Our evaluation with a prototype implementation suggests that Dalf's design is practical. It generally exhibits low overheads and in those cases where they are high, they appear to be a result of an unoptimized prototype rather than limitations of the proposed framework. The source code of our prototype is available at https://github.com/dalfdroid/dalf.

10 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Chulhong Min for their invaluable feedback. This work was supported in part by the National Science Foundation grant 1253327 and by DARPA and SPAWAR under contract N66001-15-C-4067.

REFERENCES

[1] ActivityThread.getApplicationThread(). https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/app/ActivityThread.java#L2185.

[2] Adblock Plus. https://adblockplus.org.

[3] Android Developers: android.hardware.camera2. https://developer.android.com/reference/android/hardware/camera2/package-summary.

[4] Android Developers: Content providers. https://developer.android.com/guide/topics/providers/content-providers.

[5] Android Developers: dumpsys. https://developer.android.com/studio/command-line/dumpsys.

[6] Android Developers: Permissions overview. https://developer.android.com/guide/topics/permissions/overview.

[7] Android Developers: Profile battery usage with Batterystats and Battery Historian. https://developer.android.com/studio/profile/battery-historian.

[8] Android Developers: Who lives and who dies? Process priorities on Android. https://medium.com/androiddevelopers/who-lives-and-who-dies-process-priorities-on-android-cb151f39044f.

[9] Android Source: BufferQueue and gralloc. https://source.android.com/devices/graphics/arch-bq-gralloc.

[10] Android Source: Source Code Tags and Builds. https://source.android.com/setup/start/build-numbers#source-code-tags-and-builds.

[11] Android's BitTube source. https://android.googlesource.com/platform/frameworks/native/+/android-8.1.0_r1/libs/gui/BitTube.cpp.

[12] ApplicationThread.bindApplication(). https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/app/IApplicationThread.aidl#L53.

[13] Deferring seccomp decisions to user space. https://lwn.net/Articles/756233/.

[14] Electronic Frontier Foundation. https://www.eff.org.

[15] F-Droid: an installable catalogue of FOSS (Free and Open Source Software) apps. https://f-droid.org/en/.

[16] Ghostery Goes Open Source. https://www.ghostery.com/press/ghostery-goes-open-source/.

[17] Google May Remove Access To Undocumented/Hidden APIs in Android P. https://www.xda-developers.com/google-undocumented-hidden-apis-android-p/.

[18] Google Play Protect. https://www.android.com/play-protect.

[19] Grubhub. https://play.google.com/store/apps/details?id=com.grubhub.android.

[20] HTTPS Everywhere. https://www.eff.org/https-everywhere.

[21] Modifying System Call Arguments With ptrace. https://www.alfonsobeato.net/c/modifying-system-call-arguments-with-ptrace/.

[22] Privacy Badger. https://github.com/EFForg/privacybadger.

[23] SECCOMP: Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[24] SystemUI Plugins. https://android.googlesource.com/platform/frameworks/base/+/master/packages/SystemUI/docs/plugins.md.

[25] Telegram. https://telegram.org.

[26] Telegram FOSS. https://github.com/Telegram-FOSS-Team/Telegram-FOSS.

[27] uBlock Origin. https://github.com/gorhill/uBlock.

[28] UnsupportedAppUsage annotation. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/annotation/UnsupportedAppUsage.java#28.

[29] Xposed. http://repo.xposed.info.

[30] XPrivacy. https://play.google.com/store/apps/details?id=biz.bokhorst.xprivacy.installer&hl=en.

[31] XPrivacyLua. https://lua.xprivacy.eu.

[32] Yet Another Hook Framework for ART. https://github.com/rk700/YAHFA.

[33] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. Proceedings of CCS '13, November 2013.

[34] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Extensible Multi-Layered Access Control on Android. In *Proceedings of ACSAC '14*, December 2014.

[35] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proceedings of USENIX Security '15*, August 2015.

[36] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of HotMobile '11*, March 2011.

[37] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android. In *Proceedings of SPSM '15*, October 2015.

[38] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava. ipShield: A Framework for Enforcing Context-aware Privacy. In *Proceedings of NSDI '14*, April 2014.

[39] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, , and D. Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of NDSS '13*, February 2013.

[40] X. Chen, H. Huang, S. Zhu, Q. Li, and Q. Guan. SweetDroid: Toward a Context-Sensitive Privacy Policy Enforcement Framework for Android OS. In *Proceedings of WPES '17*, October 2017.

[41] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of ISC '10*, October 2010.

[42] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of USENIX Security '11*, August 2011.

[43] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra. INSPIRED: Intention-based Privacy-preserving Permission Model. (arXiv:1709.06654 [cs.CR]), September 2017.

[44] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of WiSec '12*, April 2012.

[45] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *Proceedings of IEEE S&P '13*, May 2013.

[46] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of SPSM '12*, October 2012.

[47] T. Kim and N. Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *Proceedings of ATC '13*, June 2013.

[48] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. TapPrints: Your Finger Taps Have Fingerprints. In *Proceedings of MobiSys '12*, June 2012.

[49] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. P. Hubaux. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *Proceedings of S&P '17*, May 2017.

[50] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of ACSAC '09*, December 2009.

[51] N. Raval, A. Srivastava, A. Razeen, K. Lebeck, A. Machanavajjhala, and L. P. Cox. What You Mark is What Apps See. In *Proceedings of MobiSys '16*, June 2016.

[52] N. K. Thanigaivelan, E. Nigussie, A. Hakkala, S. Virtanen, and J. Isoaho. CoDRA: Context-based dynamically reconfigurable access control system for android. *Journal of Network and Computer Applications*, 101:1 – 17, 2018.

[53] P. Wijsekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *Proceedings of S&P '17*, May 2017.

[54] D. Wu and S. Bratus. A Context-Aware Kernel IPC Firewall for Android. In *Proceedings of ShmooCon '17*, January 2017.

[55] Z. Xu and S. Zhu. SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones. In *Proceedings of CODASPY '15*, March 2015.

[56] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of TRUST '11*, June 2011.