Enhancing Mobile Security and Privacy through App Splitting

By

Andrew J. Davidson

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 1 Sep. 2016

The dissertation is approved by the following members of the Final Oral Committee:

Somesh Jha, Professor, Computer Sciences
Thomas Reps, Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Mihai Christodorescu, Researcher, Qualcomm Research
Xinyu Zhang, Assistant Professor, Electrical & Computer Engineering

Dedicated to Meghan Davidson

Acknowledgments

Research is what I am doing when I don't know what I am doing.

— Wernher con Braun (1957)

The work presented in this dissertation is most directly the result of the insights of Somesh Jha and Mihai Christodorescu. Of course, the indirect guidance and the support that I received throughout my graduate studies goes far beyond that. I owe a huge debt of gratitude to the faculty of the University of Wisconsin, most particularly to my advisor Somesh Jha, who has been a constant source of support. Thomas Ristenpart and Thomas Reps have helped me to grasp some of the fundamentals and finer points of research, and without these three individuals I would certainly have no idea what I was doing.

I have had the good fortune to have the lessons and examples of a number of mentors over the course of several internships, including Ben Livshits at Microsoft Research, Mihai Christodorescu at IBM T.J. Watson, and Vinod Yegneswaran at SRI, in addition to many others.

My family, friends, and peers have been instrumental in setting examples of success and providing a huge amount of encouragement and advice over the course of my studies. Most especially, Meghan Davidson has been an inspiration and an incredible comfort throughout my graduate career.

This work was supported, in part, by the National Science Foundation under grant CCF-0524051; by DARPA and AFRL under contract FA8650-10-C-7088. Any opinions, findings, recommendations or conclusions ex-

pressed herein material are my own and do not necessarily reflect the views of DARPA, AFRL, or NSF.

Contents

Figures, Tables, and Listings

Abstract

Mobile computers, especially in the form of smartphones and tablet computers, have rapidly become a major part of the computing landscape. The inherent portability of these devices enable new means of interaction with a user: mobile computers are carried and used constantly, for a mix of personal and professional uses. This continuous mode of interaction expose a wealth of information to the device. Sensors such as a Global Positioning Satellite receiver (GPS), Near Field Communication sensor (NFC), and accelerometer augment this data, allowing for accurate user tracking. This data is valuable to users, but it also presents a risk of privacy leakage. While mobile computer operating systems have taken steps to protect this data, their security model falls short of the needs of many users.

This dissertation presents several novel techniques for enhancing the security and privacy of mobile computer users. The primary technique described in this work is *app splitting*, whereby a single app is rewritten into a number of collaborating apps. These collaborating apps together fulfill the original purpose of the app, but enhance the privacy and security of the user by isolating functionality into fine-grained principals.

App splitting is a technique with several use cases. This dissertation discusses two such cases: splitting an app in order to isolate the permissions granted to an app, and splitting an app in order to isolate an app's web content from its bytecode.

1 Introduction

Mobile computers have emerged as ubiquitous computing devices accompanied by unique challenges for security and privacy. Through pervasive access, users present troves of personal data to these devices, both by manual interaction and through numerous sensors onboard the device. The misuse of such data can cause significant harm to a user's privacy.

One of the most important vectors by which a mobile computer might subvert the user's privacy is through applications (or *apps*) installed via an online marketplace. Marketplaces such as Google's Play store, Microsoft's Windows Phone App Store, and Apple's App Store are maintained by the respective platform distributors, but also host applications provided by 3rd party developers. Additionally, the Android OS allows applications to be installed from 3rd party marketplaces such as the Amazon Appstore [*sic*] and the independent marketplace Getjar. These marketplaces apply varying levels of scrutiny over the apps that they distribute, but each market allows apps to embed advertising for the purpose of monetization.

The presence of marketing content within apps is symptomatic of an important security consideration in apps: a single app may contain functionality representing multiple distinct entities, each with different goals. For example, the goal of a (benign) app developer is to provide utility to the user, while the goal of the advertiser is to profile the user. Furthermore, the app may integrate additional frameworks or provide access to distinct services (such as a single sign on service to allow password-less authentication) to the app.

Despite these distinct functionalities, an app is considered to be a monolithic security entity for all major mobile OSes: In the best case, a permission may be granted to a single program point, but the user lacks the context to understand what entity within the app they have granted permission. In the common case, a permission is simultaneously granted to a every entity within the app. Granting permissions coarsely violates an important security tenet: the principle of least privilege (PLP). The PLP states that *a principal should be given no more permissions than necessary to fulfill its purpose.* The user is given no means to determine how permissions map to these app-internal principals since the OS treats the app as a single entity. Previous work has found that intra-app entities may be assigned different levels of trust for the sake of user privacy [43].

A user may be willing to provide permissions in isolation to an app, but unwilling to grant capabiltiies allowed by the composition of those permissions. To illustrate this point, consider two common permissions used in Android: `ACCESS_FINE_LOCATION` must be granted to an app in order to access the location of the user via GPS and `INTERNET` allows an app to communicate over the network. Alone, `ACCESS_FINE_LOCATION` has little chance of leaking a user's location, since the GPS coordinates are isolated to the device. However, if an app is additionally granted the `INTERNET` permission, the user has no way to tell whether or not the additional permission is being used for a purpose other than to exfiltrate the user's location.

On Android and Windows Phone OS, a check is performed at runtime to ensure that an app holds the necessary permission to invoke a function when that function is invoked. However, the model in which permissions are assigned to apps is surprisingly coarse. Each app declares the set of all permissions that it may need in an XML manifest, which is bundled into the app package, irrespective of the many functionalities within the app. When the user installs an app, this list of permissions is presented in

a human-readable fashion to the user, but they are given no guidance as to how or when those permissions are used.

The key contribution of this dissertation is a technique called *app splitting*. This technique can improve the security and privacy of the user by partitioning security-relevant functionalities of a monolithic app into distinct, isolated apps. To demonstrate the effectiveness of app splitting, we discuss two instantiations of the technique:

- **Minionization** is a permission-based instantiation of app splitting, capable of breaking a single permission into its own app, or interposing on a flow between permissions. This instantiation shows the value of app splitting for the purpose of empowering the user with more fine-grained control over how permissions are granted and used by apps.

- **Web Isolation Rewriting (WIR** is a instantiation of app splitting for separating web content from the statically compiled code of the app. This instantiation shows the value of app splitting for the purpose of preventing security issues that occur when multiple entities within the security boundary of a single app are conflated.

App splitting itself has several advantages as a technique for securing apps. Because it relies on offline rewriting, it is backwards-compatible with existing mobile operating systems. This is especially important, given that mobile OSes exhibit a high degree of fragmentation: Android alone has 12 major consumer releases since its introduction in 2009. We also made a number of contributions in each instantiation of our splitting framework. We discuss each of these briefly below.

Contributions of Minionization: The core operation of minionization is to partition an app into a number of smaller, collaborating apps called *minions*. Minion apps contain a portion of the original app representing an action that the user can mediate.

The way in which app code is minionized is guided by a policy provided by the user when the app is downloaded but before the app is installed. This makes minionization a flexible technique for re-provisioning the capabilities of an app. For example, Brightest Flashlight Free could be partitioned into two minions: m_1, which includes the core and advertising functionality, and m_2, which includes only the calls that collect the location specifically to be sent to the network. The user may then choose to install m_1 only, effectively limiting the flow of their location to the network without denying the location to the core app.

Minionizing directly addresses the PLP by allowing users to identify principals within an app, and separate them into distinct entities that can be mediated and controlled by the OS. Minionizing also better supports the PLP: the user policy can list permission flows to be partitioned before the app is installed, thus deauthorizing fine-grained composite permissions. This policy-based approach to authorization has the benefit that it does not add to the user's prompt fatigue: no extra runtime action (such as approving a prompt) is required of the user. Furthermore, the user can write (or download from a trusted party) a single policy to apply to many apps, saving the cognitive overhead of examining the permissions of each app they use.

We have developed a tool to implement minionizing. In addition to breaking an app into minions, this tool ensures that the original functionality of the app is maintained when the minions are enabled. The instrumentation enables minions to communicate with each other via OS-level interprocess communication (IPC). This allows enforcement of a desired policy via access-control mechanisms (e.g., permissions on Android intents), with fall-back to unmodified execution faithful to that of the original app in the absence of any policy. Furthermore, inter-minion communication leads to graceful degradation of functionality when strict policies (e.g., absolutely no GPS access) are enforced.

Our work on Minionization consists of the following:

- We formalize app splitting as the problem of finding graph partitions and show how various classes of security policies map to minionizing strategies. Underlying app splitting is a notion of fine-grained, flow-based permission addressing the PLP.
- We introduce a tool for performing automatic, optimal app splitting of Android apps based on a specified security policy. This tool naturally generalizes the existing work on isolating advertising from the core functionality of an app [42, 47].
- We demonstrate experimentally that our tool is practical, supports a variety of app types (from book readers to translation apps to social networking tools), and incurs low overhead: operations that use permissions incur a low overhead and the total runtime of the app does not experience any measurable slowdown.

Contributions of WIR: Web content providers and app code developers have distinct security requirements that current web-embedding mechanisms are incapable of distinguishing or enforcing. As a result, app developers have no means of controlling the web's use of app data and code via the app-web bridge, except choosing to not expose interfaces to WebView at all and consequently give up app features. Similarly, web service providers cannot express their needs for isolating their sensitive web content from apps or only allowing limited access, and often have to sacrifice security and privacy for mobile integration.

WIR is a novel approach to web-embedding to secure both apps and web content. The contributions of this instantiation of app splitting are as follow:

- We demonstrate, through concrete attacks, that web-embedding mechanisms on mobile platforms provide insufficient protections

against malicious web service providers *and* against malicious local apps. We show the existence of severe app-to-web and web-to-app attacks predicated on the lack of configurable, fine-grained security enforcement.

- We formulate a system of *dynamic access policies* that allows both apps and web content to protect themselves from each other while maintaining the benefits of integrating apps and the web. We provide complete mediation between apps and their embedded web content. We create a technique called *origin tagging* to establish articulated security principals for app-web interactions.

- We implement a static/dynamic hybrid system, called the Web Isolation Rewriting Engine (WIRE) to deploy our protection mechanisms without modifying the operating system or requiring the cooperation of developers. Our evaluation shows that this system is effective in enhancing the security of web-embedding apps while incurring minimal overhead. Our rewriting tool targets a runtime component, called WIREFrame, that serves as a trustworthy provider of secure, isolated WebViews. Web-embedding apps use WIREFrame to render their embedded web content in decoupled, mediated WebView instances. WIREFrame allows both app developers (or app users) and web content providers to define their own dynamic access policies, which regulate the access to their respective resources. WIREFrame policy enforcement recognizes fine-grained security principals (i.e., origins) and controls all app-web interactions. WIRE automates the adoption of WIREFrame in existing apps by statically rewriting an app before installation. Each WebView in the app is replaced by a mediated WebView instance in WIREFrame. In addition to separating the app from its WebView, this also separates the individual WebViews in the same app.

A key advantage to the technique is that no rewrites to the OS are required. The benefit of this approach is that the user does not need to worry about their modifications being eliminated by updates to the OS. This is important for Android in particular, because OS updates are released frequently and occasionally deployed without warning. However, the changes are, by design, backwards compatible such that a (rewritten) app that worked on an older Android version will still work on a newer version. Furthermore, deploying a new OS requires that the bootloader of the device be unlocked. Many carriers do not allow a way to unlock the bootloader, thus forcing users to rely on third-party exploits to perform the unlock in contravention of the OS security model. These exploits have a number of disadvantages. They require the user to subject their device to a rootkit-level exploit, the exploits are device-specific, and in many cases they require the user to have a high level of technical expertise to perform the unlocking successfully. Should the exploit fail, it may also permanantly damage the device while simultaneously voiding the warranty.

While the two systems presented in this disseration are intended to demonstrate the usefulness and feasibilty of app splitting, many other instantiations are possible beyond what is explored in this dissertation. App splitting can be applied in many cases in which the user identifies entities within a single security boundary. However, our work focuses exclusively on Android for both reasons related to both principle and implementation:

- *Reliable Disassembly*. While incorrectly rewriting an app is less catastrophic for the system than incorrectly rewriting the OS, it nevertheless is unacceptable. Thus, the tool needs to be able to accurately analyze the app. Android is amenable to disassembly due to the wide array of existing analysis tools, and the simplicity of the Java bytecode that make up assembled Android apps.

- *Programmatic Splitting Boundaries*. In contrast to many other pro-

gram parititioning systems, both Minionization and WIR do not require program annotations to perform splitting effectively. This is because both systems rely on aspects of the target app that can be detected programatically: for Minionization, this is the presence of permission-using methods, while for WIR it is the use of WebViews. Reducing the burden on the user makes the systems more practical.

Despite the reasons listed above for targeting Android, we believe that additional implmenentation effort could be leveraged to port our app-splitting systems to other mobile OSes. Ultimately, the goal of this thesis is to show the benefits of disentangling priviledged operations from monolithic entities, even without the cooperation of OS providers or application developers. By allowing the user to choose an interposition mechanism (IPC mediation in the case of Minionizer, access policies in the case of WIRE), this work seeks to empower those users to take advantage of existing security mechanisms to build more powerful protections. For Minionizer, users can block permission flows directly, while for WIRE users can extend the Same Origin Policy.

While the work presented here does use process-style isolation to interpose upon the flow of privleged data (data derived from the web or permission-using functions in the instances presented here), a basic level of isolation is present in nearly any practical OS.

Outline: This dissertation is structured as follows: Chapter 2 reviews background in mobile OS security models, focusing in particular on Android, and discusses related work in program analysis. Chapter 3 details the techniques underlying minionization and explains our implementation of Minionizer. Chapter 4 provides a similar treatment of WIRE. Chapter 5 concludes and explores directions for future work.

2 Background

In this chapter, we review the background material relevant to this dissertation. Because Android is a quickly-evolving operating system, features that are newer or have become obsolete will be discussed with regards to a version number. For reference, a listing of the versions of Android is presented in Appendix A.

2.1 Android Security Model

We now review the security model and enforcement mechanisms of Android. Note that the focus of this document is on privacy leaks and exploits that operate within the security model of the OS. Thus, while a complete discussion of Android security should focus on cases in which the security model is violated (e.g., through a vulnerable system API, buffer overflow, or device misconfiguration), such attacks are out of scope for the discussion of this work. It should be further noted that the privacy leaks are of a less absolute nature than true exploits: while one user may be uncomfortable with their device recording their audio on an open microphone, another user may actually find this behavior desirable. Thus, the privacy violations discussed in this document cannot truly be called exploits, as they are not readily fixed by correcting implementation errors.

2.1.1 Android Permissions

ACCESS_CHECKIN_PROPERTIES	Allows read/write access to the "properties" table in the checkin database, to change values that get uploaded.
ACCESS_COARSE_LOCATION	Allows an app to access approximate location.
ACCESS_FINE_LOCATION	Allows an app to access precise location.
ACCESS_LOCATION_EXTRA_ COMMANDS	Allows an application to access extra location provider commands.
ACCESS_NETWORK_STATE	Allows applications to access information about networks.
ACCESS_NOTIFICATION_POLICY	Marker permission for applications that wish to access notification policy.
ACCESS_WIFI_STATE	Allows applications to access information about Wi-Fi networks.
ACCOUNT_MANAGER	Allows applications to call into AccountAuthenticators.
ADD_VOICEMAIL	Allows an application to add voicemails into the system.
BATTERY_STATS	Allows an application to collect battery statistics
BIND_ACCESSIBILITY_SERVICE	Required by an AccessibilityService, to ensure that only the system can bind to it.
BIND_APPWIDGET	Allows an application to tell the AppWidget service which application can access AppWidget's data.

BIND_CARRIER_ MESSAGING_SERVICE	This constant was deprecated in API level 23. Use BIND_CARRIER_SERVICES instead
BIND_CARRIER_SERVICES	The system process that is allowed to bind to services in carrier apps will have this permission.
BIND_CHOOSER_TARGET_ SERVICE	Must be required by a ChooserTargetService, to ensure that only the system can bind to it.
BIND_CONDITION_ PROVIDER_SERVICE	Must be required by a ConditionProviderService, to ensure that only the system can bind to it.
BIND_DEVICE_ADMIN	Must be required by device administration receiver, to ensure that only the system can interact with it.
BIND_DREAM_SERVICE	Must be required by an DreamService, to ensure that only the system can bind to it.
BIND_INCALL_SERVICE	Must be required by a InCallService, to ensure that only the system can bind to it.
BIND_INPUT_METHOD	Must be required by an InputMethodService, to ensure that only the system can bind to it.
BIND_MIDI_DEVICE_SERVICE	Must be required by an MidiDeviceService, to ensure that only the system can bind to it.

BIND_NFC_SERVICE	Must be required by a HostApduService or OffHostApduService to ensure that only the system can bind to it.
BIND_NOTIFICATION_LISTENER_SERVICE	Must be required by an NotificationListenerService, to ensure that only the system can bind to it.
BIND_PRINT_SERVICE	Must be required by a PrintService, to ensure that only the system can bind to it.
BIND_QUICK_SETTINGS_TILE	Allows an application to bind to third party quick settings tiles.
BIND_REMOTEVIEWS	Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
BIND_SCREENING_SERVICE	Must be required by a CallScreeningService, to ensure that only the system can bind to it.
BIND_TELECOM_CONNECTION_SERVICE	Must be required by a ConnectionService, to ensure that only the system can bind to it.
BIND_TEXT_SERVICE	Must be required by a TextService
BIND_TV_INPUT	Must be required by a TvInputService to ensure that only the system can bind to it.
BIND_VOICE_INTERACTION	Must be required by a VoiceInteractionService, to ensure that only the system can bind to it.

BIND_VPN_SERVICE	Must be required by a VpnService, to ensure that only the system can bind to it.
BIND_VR_LISTENER_SERVICE	Must be required by an VrListenerService, to ensure that only the system can bind to it.
BIND_WALLPAPER	Must be required by a WallpaperService, to ensure that only the system can bind to it.
BLUETOOTH	Allows applications to connect to paired bluetooth devices.
BLUETOOTH_ADMIN	Allows applications to discover and pair bluetooth devices.
BLUETOOTH_PRIVILEGED	Allows applications to pair bluetooth devices without user interaction, and to allow or disallow phonebook access or message access.
BODY_SENSORS	Allows an application to access data from sensors that the user uses to measure what is happening inside his/her body, such as heart rate.
BROADCAST_PACKAGE_ REMOVED	Allows an application to broadcast a notification that an application package has been removed.
BROADCAST_SMS	Allows an application to broadcast an SMS receipt notification.
BROADCAST_STICKY	Allows an application to broadcast sticky intents.

BROADCAST_WAP_PUSH	Allows an application to broadcast a WAP PUSH receipt notification.
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call.
CALL_PRIVILEGED	Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.
CAMERA	Required to be able to access the camera device.
CAPTURE_AUDIO_OUTPUT	Allows an application to capture audio output.
CAPTURE_SECURE_ VIDEO_OUTPUT	Allows an application to capture secure video output.
CAPTURE_VIDEO_OUTPUT	Allows an application to capture video output.
CHANGE_COMPONENT_ ENABLED_STATE	Allows an application to change whether an application component (other than its own) is enabled or not.
CHANGE_CONFIGURATION	Allows an application to modify the current configuration, such as locale.
CHANGE_NETWORK_STATE	Allows applications to change network connectivity state.
CHANGE_WIFI_ MULTICAST_STATE	Allows applications to enter Wi-Fi Multicast mode.

CHANGE_WIFI_STATE	Allows applications to change Wi-Fi connectivity state.
CLEAR_APP_CACHE	Allows an application to clear the caches of all installed applications on the device.
CONTROL_LOCATION_UPDATES	Allows enabling/disabling location update notifications from the radio.
DELETE_CACHE_FILES	Allows an application to delete cache files.
DELETE_PACKAGES	Allows an application to delete packages.
DIAGNOSTIC	Allows applications to RW to diagnostic resources.
DISABLE_KEYGUARD	Allows applications to disable the keyguard if it is not secure.
DUMP	Allows an application to retrieve state dump information from system services.
EXPAND_STATUS_BAR	Allows an application to expand or collapse the status bar.
FACTORY_TEST	Run as a manufacturer test application, running as the root user.
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
GET_ACCOUNTS_PRIVILEGED	Allows access to the list of accounts in the Accounts Service.
GET_PACKAGE_SIZE	Allows an application to find out the space used by any package.

GET_TASKS	This constant was deprecated in API level 21. No longer enforced.
GLOBAL_SEARCH	This permission can be used on content providers to allow the global search system to access their data.
INSTALL_LOCATION_PROVIDER	Allows an application to install a location provider into the Location Manager.
INSTALL_PACKAGES	Allows an application to install packages.
INSTALL_SHORTCUT	Allows an application to install a shortcut in Launcher.
INTERNET	Allows applications to open network sockets.
KILL_BACKGROUND_PROCESSES	Allows an application to call killBackgroundProcesses(String).
LOCATION_HARDWARE	Allows an application to use location features in hardware, such as the geofencing api.
MANAGE_DOCUMENTS	Allows an application to manage access to documents, usually as part of a document picker.
MASTER_CLEAR	Not for use by third-party applications.
MEDIA_CONTENT_CONTROL	Allows an application to know what content is playing and control its playback.
MODIFY_AUDIO_SETTINGS	Allows an application to modify global audio settings.

MODIFY_PHONE_STATE	Allows modification of the telephony state - power on, mmi, etc.
MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage.
MOUNT_UNMOUNT_ FILESYSTEMS	Allows mounting and unmounting file systems for removable storage.
NFC	Allows applications to perform I/O operations over NFC.
PACKAGE_USAGE_STATS	Allows an application to collect component usage statistics. Declaring the permission implies intention to use the API and the user of the device can grant permission through the Settings application.
PERSISTENT_ACTIVITY	This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.
PROCESS_OUTGOING_CALLS	Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether.
READ_CALENDAR	Allows an application to read the user's calendar data.
READ_CALL_LOG	Allows an application to read the user's call log.

READ_CONTACTS	Allows an application to read the user's contacts data.
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
READ_FRAME_BUFFER	Allows an application to take screen shots and more generally get access to the frame buffer data.
READ_INPUT_STATE	This constant was deprecated in API level 16. The API that used this permission has been removed.
READ_LOGS	Allows an application to read the low-level system log files.
READ_PHONE_STATE	Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device.
READ_SMS	Allows an application to read SMS messages.
READ_SYNC_SETTINGS	Allows applications to read the sync settings.
READ_SYNC_STATS	Allows applications to read the sync stats.
READ_VOICEMAIL	Allows an application to read voicemails in the system.
REBOOT	Required to be able to reboot the device.

RECEIVE_BOOT_COMPLETED	Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
RECEIVE_MMS	Allows an application to monitor incoming MMS messages.
RECEIVE_SMS	Allows an application to receive SMS messages.
RECEIVE_WAP_PUSH	Allows an application to receive WAP push messages.
RECORD_AUDIO	Allows an application to record audio.
REORDER_TASKS	Allows an application to change the Z-order of tasks.
REQUEST_IGNORE_ BATTERY_OPTIMIZATIONS	Permission an application must hold in order to use ACTION_REQUEST_IGNORE _BATTERY_OPTIMIZATIONS.
REQUEST_INSTALL_PACKAGES	Allows an application to request installing packages.
RESTART_PACKAGES	This constant was deprecated in API level 8. The restartPackage(String) API is no longer supported.
SEND_RESPOND_VIA_MESSAGE	Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.
SEND_SMS	Allows an application to send SMS messages.

SET_ALARM	Allows an application to broadcast an Intent to set an alarm for the user.
SET_ALWAYS_FINISH	Allows an application to control whether activities are immediately finished when put in the background.
SET_ANIMATION_SCALE	Modify the global animation scaling factor.
SET_DEBUG_APP	Configure an application for debugging.
SET_PREFERRED_APPLICATIONS	This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.
SET_PROCESS_LIMIT	Allows an application to set the maximum number of (not needed) application processes that can be running.
SET_TIME	Allows applications to set the system time.
SET_TIME_ZONE	Allows applications to set the system time zone.
SET_WALLPAPER	Allows applications to set the wallpaper.
SET_WALLPAPER_HINTS	Allows applications to set the wallpaper hints.
SIGNAL_PERSISTENT_PROCESSES	Allow an application to request that a signal be sent to all persistent processes.

STATUS_BAR	Allows an application to open, close, or disable the status bar and its icons.
SYSTEM_ALERT_WINDOW	Allows an app to create windows using the type TYPE_SYSTEM_ALERT, shown on top of all other apps.
TRANSMIT_IR	Allows using the device's IR transmitter, if available.
UNINSTALL_SHORTCUT	Allows an application to uninstall a shortcut in Launcher.
UPDATE_DEVICE_STATS	Allows an application to update device statistics.
USE_FINGERPRINT	Allows an app to use fingerprint hardware.
USE_SIP	Allows an application to use SIP service.
VIBRATE	Allows access to the vibrator.
WAKE_LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming.
WRITE_APN_SETTINGS	Allows applications to write the apn settings.
WRITE_CALENDAR	Allows an application to write the user's calendar data.
WRITE_CALL_LOG	Allows an application to write (but not read) the user's call log data.
WRITE_CONTACTS	Allows an application to write the user's contacts data.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.

WRITE_GSERVICES	Allows an application to modify the Google service map.
WRITE_SECURE_SETTINGS	Allows an application to read or write the secure system settings.
WRITE_SETTINGS	Allows an application to read or write the system settings.
WRITE_SYNC_SETTINGS	Allows applications to write the sync settings.
WRITE_VOICEMAIL	Allows an application to modify and remove existing voicemails in the system.

Table 2.1: Android Permissions List [24]

One of the most prominent features of Android is the *Permissions Model*. This is a capability system that guards system resources from apps, prohibiting access unless explicitly granted by the user. Table 2.1 contains a listing of the permissions available as of Android version 7.0 (codenamed *Nougat*). For an Android app to perform a security-sensitive operation, it must be granted the corresponding permission. As an example, the `READ_SMS` permission allows an app to read text messages sent to the device. Permissions represent binary flags of functionality, although in some cases distinct permissions can grant access at different levels to the same functionality. For example, the `ACCESS_FINE_LOCATION` permission, so named because it grants fine-grained location data to an app, is required to use the GPS receiver on the phone. The `ACCESS_COARSE_LOCATION` permission uses network information alone to get the device's location (e.g., via user triangulation based on the known location of nearby WiFi and cellular network resources).

Note that the permissions model changes between versions of Android. While permissions are backwards-compatible, certain permis-

sions are introduced as devices, and the OS itself, gain new capabilities. `BIND_SCREENING_SERVICE` is one such permission, which grants an app the ability to selectively drop incoming phone calls (i.e., to block a particular number or class of numbers). However. call-screening is a feature of Android 7.0, and thus no previous version of the OS mandated a call screening permission. Conversely, the capabilities granted by each permission can change between OS versions. As an example, with API version 19, the `WRITE_EXTERNAL_STORAGE` permission was no longer required when writing to app-specific storage locations (prior to this version the OS did not distinguish app-specific external storage, and thus all writes to external storage required this permission. Surprisingly, there is no canonical mapping from Android API functions to the permissions that function requires.

The existence of a permissions mechanism in Android represents the potential for more nuanced control over an app than what is available on a traditional desktop OS. However, the permissions model only serves to protect the user to the extent that he or she can make meaningful decisions about how permissions are granted to an app. We now describe how permissions are declared by developers, and presented to the user through the App's *Manifest*.

2.1.2 App Manifest

All permissions used by an Android app must be declared at compile time by the developer. This is achieved by bundling an XML *manifest* file, `AndroidManifest.xml`. This file allows each permission to be declared with the `uses-permission` XML element. Figure 2.2 shows a snippet of an example manifest file. Note that permissions exist in a namespace, with each of the system permissions being declared in the `android.permission` namespace. In addition to the system-defined permissions, the user may also choose to declare and use *custom* permissions in a namespace of

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

Figure 2.2: XML manifest for declaring the permissions shown in Figure 2.3. The permissions listed here are each system permissions and therefore include text descriptions that are automatically provided by the Android framework.

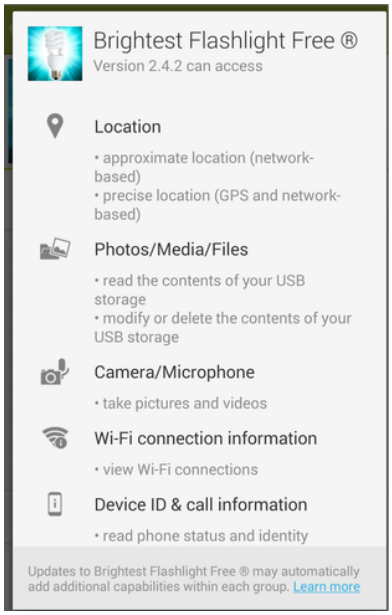

Figure 2.3: An example of how an app manifest is presented in the Google Play Store. This manifest is a snippet taken from the the well-known flashlight app *Brightest Flashlight Free*. Each permissions is described in plaintext and grouped according to a broad category. Note that the permissions shown here correspond to those listed in Figure 2.2

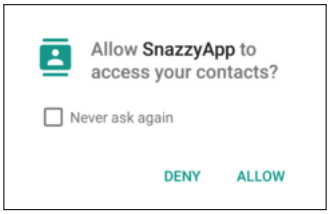

Figure 2.4: Dynamic permissions dialog, taken from the official Android documentation. Dialogs of this sort are required to be approved to access permissions that the system has tagged as dangerous.

their own creation, which allows an app to specify which other apps may communicate across intercomponent communication (ICC) channels.

Permissions are described differently to developers and to users. The developer's reference lists the permissions with the text descriptions presented in Table 2.1, whereas the Google Play store gives more high-level descriptions. Figure 2.3 shows a screenshot from the well-known app *Brightest Flashlight Free*. Note that this list continues to scroll and ultimately declares a total of 25 distinct permissions from eight distinct categories, but the permissions shown here correspond to those declared in Figure 2.2.

2.1.3 Granting and Revoking Permissions

As with many features of the permissions model, the way in which permissions are granted and revoked has evolved over time. Here, we discuss the major steps in this evolution over the history of Android.

Install-Time Permissions: Since the initial release of Android, the primary way in which permissions are granted to an app has been at install time. In the typical install-time workflow, the user initiates installation of an app, and is then presented with the permissions declared in the app manifest, as shown in Figure 2.3. By approving the dialog the user is granting each listed permission to the app. For the majority of Android's history, the user had no further control over the permission system: instal-

lation was tantamount to permanently approving all permissions, with the user's only alternative being to forego installation of the app entirely.

Group Name	**Group Description**
In-app purchases	An app can ask you to make purchases inside the app.
Device & app history	An app can do one or more of the following: • Read sensitive log data • Retrieve system internal state • Read your web bookmarks and history • Retrieve running apps
Cellular data settings	An app can use settings that control your mobile data connection and potentially the data you receive.
Identity	An app can use your account and/or profile information on your device. Identity access may include the ability to: • Find accounts on the device • Read your own contact card (example: name and contact information) • Modify your own contact card • Add or remove accounts

Contacts	An app can use your device's contacts, which may include the ability to read and modify your contacts.
Calendar	An app can use your device's calendar information, which may include the ability to: • Read calendar events plus confidential information • Add or modify calendar events and send email to guests without owners' knowledge
Location	An app can use your device's location. Location access may include: • Approximate location (network-based) • Precise location (GPS and network-based) • Access extra location provider commands • GPS access

SMS	An app can use your device's text messaging (SMS) and/or multimedia messaging service (MMS). This group may include the ability to use text, picture, or video messages. Important: Depending on your plan, you may be charged by your carrier for text or multimedia messages. SMS access may include the ability to: • Receive text messages (SMS) • Read your text messages (SMS or MMS) • Receive text messages (MMS, like a picture or video message) • Edit your text messages (SMS or MMS) • Send SMS messages; this may cost you money • Receive text messages (WAP)

Phone	An app can use your phone and/or its call history. Depending on your plan, you may be charged by your carrier for phone calls. Phone access may include the ability to: • Directly call phone numbers; this may cost you money • Write call log (example: call history) • Read call log • Reroute outgoing calls • Modify phone state • Make calls without your intervention
Photos/Media/Files	An app can use files or data stored on your device. Photos/Media/Files access may include the ability to: • Read the contents of your USB storage (example: SD card) • Modify or delete the contents of your USB storage • Format external storage • Mount or unmount external storage

Camera	An app can use your device's camera. Camera access may include the ability to: • Take pictures and videos • Record video
Microphone	An app can use your device's microphone. Microphone access may include the ability to record audio.
Wi-Fi connection information	An app can access your device's Wi-Fi connection information, like if Wi-Fi is turned on and the name(s) of connected devices. Wi-Fi connection information access may include the ability to view Wi-Fi connections. Note: Since apps typically access the Internet, you'll only see the Wi-Fi connection information permission group on the download screen when installing an app. Apps no longer display the "full internet access" permission on the download screen, but you can always see the full list of permissions by following the instructions under the "See all permissions for a specific app" section above.
Bluetooth connection information	An app can control Bluetooth on your device, which includes broadcasting to or getting information about nearby Bluetooth devices.

Wearable sensors / activity data	Allows the app to access data from wearable sensors, such as heart rate monitors. Can receive periodic updates on physical activity levels.
Device ID & call information	An app can access your device ID(s), phone number, whether you're on the phone, and the number connected by a call. Device ID & call information may include the ability to read phone status and identity.
Other	An app can use custom settings provided by your device manufacturer or application-specific permissions. Important: If an app adds a permission that is in the "Other" group, you'll always be asked to review the change before downloading an update. Other access may include the ability to: • Read your social stream (on some social networks) • Write to your social stream (on some social networks) • Access subscribed feeds You'll see all permissions from the "Other" group listed on the Play Store, including those that weren't shown on the app download screen.

Table 2.5: Android Permissions Groups, along with the textual descriptions given to the user for each group when approving an app's permission manifest [25].

Permissions Groups: As discussed in the previous subsection, permissions are categorized into groups. Table 2.5 provides a list of these groups as of Android 6.0, though the list is subject to change. When an app is updated, it requires approval from the user for any permissions that reside in new groups. However, new permissions that reside in existing groups do not need to be approved by the user. If the user has set their device to automatically update their apps, this update will occur silently.

Approval Dialogs: Beginning in Android 6.0, the use of certain permissions that are marked as *dangerous* require dynamic approval when they are used via a dialog. Figure 2.4 shows an example dialog taken from the official Android documentation on permissions [7]

AppOps: In Android 4.3, a developer feature called AppOps was released to production builds of the operating system. This feature allowed users to access a hidden permissions manager, which allowed for selectively revoking (and later re-granting) permissions that were granted at runtime. However, the feature was not intended for production use, and the feature was removed in the next major Android release cycle.

System-Level Revocation: Beginning in Android 6.0 (the latest released version of the OS), the user has the ability to revoke permissions on a per-app basis through a dialog in the system settings. This change in the OS largely mirrors the functionality of AppOps, but in an officially-supported capacity. However, none of these systems give the user any *context* for how the permission is being used. When the user disables the permission, they do so throughout the entire app, in any context in which that permission is used.

Inter-Component Communication

Android components are partitioned into *Components*. While components themselves do not represent a security boundary within an app, communication between components, called Inter-Component Communication (ICC), is the primary mechanism by which we connect apps that have been separated via app splitting. There are four types of components in Android:

1. *Activities* are the most common type of component, as they represent the entrypoint of an app that a user can launch, usually representing a UI screen which with the user is expected to interact.

2. *Services* are used to perform background processing. This component is particularly important given that Android will automatically kill an app if its UI is unresponsive after a timeout. Thus, any long-running action that the app needs to take is usually contained in a service.

3. *Broadcast Receivers* run in the background, but unlike services they are triggered by system events. Broadcast receivers subscribe to a list of system events of their own choosing, and are activated upon receipt of such an event. For example, a Broadcast Receiver might subscribe to the event of the system startup (thus allowing the app to be launched when the device is powered up) or to the receipt of a text message, or both.

4. *Content Providers* are used to store data and allow a passive means for multiple apps to get structured data from one another.

Much like permissions, the Activities, Services, and Content Providers of an app must be declared in the app's manifest (Broadcast Receivers can be created and registered at runtime).

The primary mechanism by which components (both within and between apps) communicate is through *Intents*. Intents are based on a complete reimplementation of traditional Linux Inter-Process Communication (IPC), specifically designed to be fast on Android. Intents can be sent in two ways:

1. *Explicit Intents* are addressed to a single component of a single app. The system enforces that the intent cannot be intercepted by other apps, and the package-signing system ensures that an app cannot impersonate the intended target of the intent.

2. *Implicit Intents* specify a class of data or functionality that will satisfy the intent. For example, an Implicit Intent may specify that it should be received by any application that can handle text messages. This allows any Broadcast Receiver that handles text messages to be notified of the message. While Implicit Intents offer a unique way to extend the functionality of an intent, they are not used in app splitting.

Android also supports automatic *Parcelization* of primitive data types and Objects, which allows apps to easily marshal data into (and out of) Intents in the form of key/value pairs. The Intent system helps to support a distributed workflow when using the Android OS: apps can specify particular components (particularly components of other apps) to satisfy particular functionality, and effectively "stub out" the implementation.

Chapter Summary

The Android OS, and even the security architecture of the alone, is a complicated software system that goes well beyond the scope of this dissertation. The purpose of this chapter is simply to introduce the concepts necessary for a complete understanding of the chapters to follow.

3

Web Isolation Rewriting (WIR)

3.1 Introduction

A common app-design paradigm is to embed web content directly in an app's UI. Apps that follow this paradigm, which we call *web-embedding apps*, combine the advantages of both the mobile web and native apps: web content is highly portable across platforms, and native app code can leverage the full power of the device. Unfortunately, these apps also introduce unique attack vectors in the interactions between web content and app code.

All major mobile platforms offer web-embedding support. The `WebView` class in Android and `UIWebView` class in iOS are UI widgets that display remote web elements or entire web pages natively within an app.[1] Web content and the embedding app can programmatically manipulate each other's data and behavior via the so-called *app-web bridge* APIs. For instance, an app can programmatically configure embedded WebViews and inject scripts. Conversely, JavaScript loaded in a WebView may call exported app code to access local resources, such as the file system, the camera, or GPS.

The popularity of web-embedding apps makes the app-web bridge an attractive target for attacks from both sides: a malicious app may seek to subvert or leak sensitive web content (i.e., *app-to-web attacks*); malicious web

[1] Though we focus on the security of Android WebViews, we believe that our observations and techniques are largely applicable to iOS.

content may attempt to misuse the app's permissions and local resources (i.e., *web-to-app attacks*). Both types of attacks are increasingly observed in the wild [39, 45].

Malicious apps can embed and manipulate web content from sensitive domains. Well-established web-security policies, such as the *same-origin policy* (SOP), are not enforced upon app-web interactions, largely due to the simplistic security design of WebView, which presumes apps always own embedded web content. As a result, web-embedding apps can easily disturb or spy on third-party web services, such as single sign-on (SSO) and in-app payment. Furthermore, apps can undermine inter-frame sandboxing by retrieving scripts from one page and injecting them into another. This means that a malicious app is not restricted by the SOP and can introspect on sensitive, third-party web content.

Conversely, malicious web content embedded in benign apps can abuse the app's resources. The permissions granted to an app are implicitly inherited by its embedded web content: the privileges meant for a trusted domain are universally available to sub-frames or elements loaded from untrusted domains in the WebView, allowing malicious web content from one domain to leverage permissions intended for a different domain. Moreover, the app-web bridge allows app developers to make portions of their app code invocable by JavaScript loaded in WebViews. This feature greatly facilitates web content's access to local data and resources such as the GPS location of the device. Unfortunately, this access is not restricted to a given origin. Therefore, developers are often forced to ignore attacks, such as those reported in [31, 45], in favor of adding app functionalities.

The key commonality amongst the attacks above is an intermingling of the privileges of separate security principals. Web content providers and app code developers have distinct security requirements that current web-embedding mechanisms are incapable of distinguishing or enforcing. As a result, app developers have no means of controlling the web's use

of app data and code via the app-web bridge, except choosing to not expose interfaces to WebView at all and consequently give up app features. Similarly, web service providers cannot express their needs for isolating their sensitive web content from apps or only allowing limited access, and often have to sacrifice security and privacy for mobile integration.

In this chapter, we introduce a novel and backward-compatible approach to web-embedding that is trustworthy for both apps and web content. The contributions of our work are as follow:

- We demonstrate, through concrete attacks, that web-embedding mechanisms on mobile platforms provide insufficient protections against malicious web service providers *and* against malicious local apps. We show the existence of severe app-to-web and web-to-app attacks predicated on the lack of configurable, fine-grained security enforcement.

- We formulate a system of *dynamic access policies* that allows both apps and web content to protect themselves from each other while maintaining the benefits of integrating apps and the web. We provide complete mediation between apps and their embedded web content. We create a technique called *origin tagging* to establish articulated security principals for app-web interactions.

- We introduce a static/dynamic hybrid technqiue to deploy our protection mechanisms without modifying the operating system or requiring the cooperation of developers. We call this technique web isolation rewriting (WIR). Our evaluation shows that this system is effective in enhancing the security of web-embedding apps while incurring minimal overhead.

We implemented WIR using a static, offline app rewriting tool called WIRE (for Web Isolation Rewriting Engine) and a secure, isolated WebView provider called WIREFrame. Web-embedding apps use WIREFrame

to render their embedded web content in decoupled, mediated WebView instances. WIREFrame allows both app developers (or app users) and web content providers to define their own dynamic access policies, which regulate the access to their respective resources. WIREFrame's policy enforcement recognizes fine-grained security principals (i.e., origins) and controls all app-web interactions. WIRE automates the adoption of WIREFrame in existing apps by statically rewriting an app before installation. Each WebView in the app is replaced by a mediated WebView instance in WIREFrame. In addition to separating the app from its WebView, this also separates the individual WebViews in the same app.

The rest of this chapter is organized as follows: In § 3.2, we introduce our threat model, discuss the security limitations of current WebView, and present example attacks. In § 4.2, we outline the designs of WIREFrame and WIRE, followed by their technical details in § 3.4 and § 3.5. We provide a security analysis of our system in § 3.6 and evaluate our prototype implementation of WIREFrame and WIRE in § 3.7. We compare our system with related work in § 4.7 and conclude in § 4.8.

3.2 Threat Model

Our system adopts a threat model that considers two separate classes of attacks exploiting the current WebView design:

- **App-to-Web Attacks:** an app may spy on or manipulate its embedded web content sourced from a third-party provider causing such harms as stealing passwords from forms or rewriting pages to aid in phishing attacks. In this case, the app, which controls the WebView, is the attacker; the embedded web content (and its provider) is the victim. To perform the attack, the malicious app may use the WebView inspection APIs or directly manipulate the WebView's data in memory. Moreover, the malicious app may employ obfuscation tech-

niques, including reflection and native code, to obscure its (ab)use of the WebView.

- **Web-to-App Attacks:** an embedded web page from a third-party may attack its host app, causing such harms as leaking personally identifiable information such as the device's unique identifiers or the user's contact list. Contrasting the previous class of attacks, in this case, the content embedded in a WebView (and its provider) is the attacker; the app that hosts the WebView is the victim. In such attacks, the malicious web content may exploit any web-facing interfaces exposed by the WebView and the host app, including the exported Java methods. However, the malicious web content is not expected to exploit arbitrary code execution vulnerabilities in the WebView. These vulnerabilities are extremely rare and out of the scope of this work, which addresses the insecure design, rather than implementation vulnerabilities, of WebView.

In either case, we assume that the OS is trusted, which is reasonable given that a compromised OS would obviate the need for launching the attacks studied in this chapter. We note that attacks in which an adversary controls both web content and app code simultaneously are out of the scope of this work.

3.2.1 Attack Scenarios

To illustrate the types of attacks that fall under our threat model, we introduce three representative examples. We use these examples to discuss the security limitations of WebView and the app-web bridge that make the exploits possible.

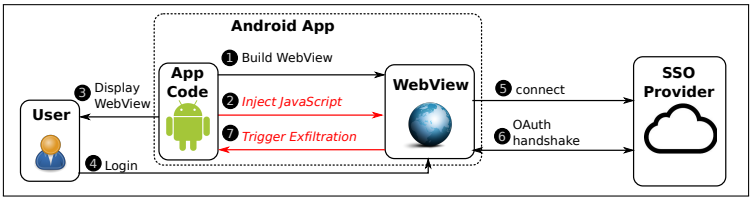

Figure 3.1: Workflow of an attack on an SSO client, as represented by the example app WebRSS. The app waits for the SSO dialog to appear in the WebView, then scrapes the username and password from the WebView via introspection, either through reflection or injected JavaScript.

SSO Credential Stealing

As one instance of an app-to-web attack, we implemented a malicious web-embedding RSS reader app, WebRSS. RSS readers are widely used on Android, with popular apps such as Feedly and Flipboard boasting hundreds of thousands of installs. WebRSS requires no additional permissions besides the `INTERNET` permission, which allows the app to access the network and is necessary for any legitimate RSS reader.

Like many account-based apps, WebRSS allows users to authenticate themselves using a third-party SSO service. SSO allows users to forgo the creation of a separate username and password combination for each account that they maintain. SSO services are popular precisely because they identify users without directly exposing secret credentials. Instead, users authenticate (by entering a username and password) to a dialog (inside a WebView) controlled by the SSO provider. Upon a successful login, the service passes an opaque authentication token back to the app, which attests to the user's identity without revealing credentials.

The security of an SSO dialog relies on the SOP to prevent web content of other origins from accessing the credential values. However, a malicious app like WebRSS can indirectly obtain these credentials by injecting JavaScript into the authentication WebView to scrape the username and

password from the text fields, even when the password field is blinded. Figure 3.1 illustrates the workflow of this attack at a high level. WebRSS goes through three steps in the attack (relevant snippets of code from WebRSS are shown in Figure 3.2, Figure 3.3, and Figure 3.4):

Construct WebView: The first step, shown in Figure 3.2, builds a WebView to load the authentication dialog. Note that the app code enables JavaScript on the WebView and interacts with a real SSO library, in this case LinkedIn. From the perspective of the library, no malicious behavior occurs as the app code is allowed to call `getRequestToken()` to get the opaque SSO token.

Attach JavaScript Bridge: Figure 3.3 shows the app code that will exfiltrate the user credentials. For the purpose of demonstration, this code outputs the username and password to a log file, but could send the values to an adversary over the internet using the permissions already granted to the app for legitimate RSS functionality.

Inject JavaScript Code: To complete the attack, the malicious app registers for a callback when the authentication dialog is loaded, as shown in Figure 3.4. When the callback is fired, the app injects the JavaScript code on Lines 8-15, which is stored as a string as part of the app. The script scrapes the credentials from the dialog and passes it to the code of Figure 3.3 through the app-web bridge. The JavaScript can extract the contents of the password field (Line 12) even though it is blinded to the user (i.e. it displays a series of dots on-screen rather than the literal characters that the user types in). To ensure that the characters of the username and password are exfiltrated after the user has completed the form, the code triggers when the user clicks the "Allow Access" button.

The use of a WebView in WebRSS also enables a web-to-app attack. For example, an iframe containing third-party content (e.g., an ad banner outside of the SSO provider's domain) may exist on the user login page or the redirection page following a successful login. Although the same-

```
1 public void setWebView(){
2   WebView v = (WebView)findViewById(R.id.w);
3   v.getSettings().setJavaScriptEnabled(true);
4   v.setWebViewClient(new WebClient());
5   v.addJavascriptInterface(new JS(), "js");
6   LinkedInRequestToken t = getRequestToken();
7   v.loadUrl(t.getAuthorizationUrl());
8 }
```

Figure 3.2: Code snippet from WebRSS to steal user credentials malicious app code, enabling the JavaScript code to be injected and run.

```
1 public class JS{
2     void harvest(String name, String pass){
3       Log.e("NAME", name);
4       Log.e("PASS", pass);
5     }
6  }
```

Figure 3.3: Code snippet from WebRSS to steal user credentials from an SSO dialog. This snippet shows the app code called to exfiltrate user data scraped from the authentication dialog.

origin policy prevents the third-party website from viewing web data from the SSO provider's domain, the third-party iframe can invoke, without restrictions, the Java interfaces exported by the local app and the SSO library. This includes sensitive interfaces solely intended for the web login (e.g., for retrieving user location or login history data). As a result, without breaking any existing web or app security policy, the malicious iFrame can steal sensitive data by abusing the unmediated app-web bridge.

Local Storage Inference

A powerful web-to-app attack involves web content loaded in WebView stealing content from the host app. Most recently, Son *et al.* observed several such attacks, including one where web content can infer the ex-

```
 1 public class WebClient extends WebViewClient{
 2
 3  public void onLoadResource(WebView v, String url){
 4    super.onLoadResource(v, url);
 5
 6    String tgtURL = "linkedin.com/uas/oauth/";
 7    if (url.contains(tgtURL)){
 8      v.loadUrl("javascript:function hack(){"
 9      + "var f = document.getElementById("
10      + "'session_key-oauthAuthorizeForm');"
11      + "var g = document.getElementById("
12      + "'session_password-oauthAuthorizeForm');"
13      + "js.harvest(f.value, g.value);};"
14      + "document.getElementById("
15      + "'Allow Access').onclick=hack()");
16    }
17  }
18 }
```

Figure 3.4: Code snippets from WebRSS to steal user credentials from an SSO dialog. This snippet shows how JavaScript is constructed from within the app and injected into the authentication site.

istence of local files and in some cases can completely read the contents of such files [45]. Such attacks have a severe privacy impact. Son *et al.* found instances in which the host app contains information on the user's medications, dating gender preference, social circle, and identity. The host app may contain credentials used to authenticate the user, allowing malicious web content to breach the user's security. The attack relies on specific configurations of the WebView. However, Son *et al.* found that such configurations are required and used in legitimate circumstances. Unfortunately, the current design of WebView and the app-web bridge cannot allow apps to selectively expose local resources to web content based on web content's origins or trust levels. Therefore, when an app needs to permit any trusted web content to access local files or other resources, the same level of access is given to all web content despite their

origins.

User Impersonation

Another abuse of the app-web bridge is for a malicious app to trick an embedded WebView and impersonate a user through JavaScript actions. Websites are largely defenseless against such actions: even if they require users to manually input credentials and prevent malicious credential stealing (e.g., through a use of a properly salted and encrypted password with every login), a malicious app can simply wait for the credentials to be input and then send surreptitious requests to the authenticated page in the guise of the user.

Such attacks are not just realistic but likely. For instance, attackers often repackage popular websites' official companion apps, which are usually thin wrappers around WebViews. The rogue companion apps can stealthily impersonate users, which is difficult for web servers or average users to detect. Furthermore, apps that allow for general-purpose browsing can include specific triggers on particular websites to launch user impersonation attacks.

3.2.2 Exploit Analysis

The common cause of the above attacks lies in two assumptions implicit to the design of WebViews: (1) apps always own web content embedded in them; (2) web content in a WebView is always from a single origin. Android provides only a weak form of isolation between the app and web content: the app loads the web content, and can cede coarse-grained control. Since both run in the same process, the app is expected to protect the user from malicious web content. Unfortunately, the weak isolation between apps and web content is insufficient to prevent attacks between apps and embedded web content. In the next section, we show how our

system improves upon this isolation while still allowing sharing when appropriate.

3.3 System Overview

In this section, we describe our system and show how it addresses the threats listed in §3.2. The key capability of the system is that it provides a secure service that runs web-content in a decoupled app. The most obvious benefit of this approach is that it places app and process boundaries between the web-content and embedding app, leveraging existing isolation mechanisms without modifying the underlying OS or framework. However, the true power of our approach is that it provides an opportunity for both the app and web-content to express dynamic access policies over their interactions. The secure service mediates all interactions between app code and web-content over an inter-process communication (IPC) interface subject to these policies.

3.3.1 System Design

As introduced in §3.1, our system consists of two components: (1) a runtime component, WIREFrame, that runs the secure WebView service. This component is distributed as a standalone Android app. (2) a static, offline rewriting tool, WIRE, that retargets apps to use WIREFrame. This component injects the protection mechanisms of WIREFrame without requiring apps to be redesigned. Thus, it ensures that the policies of each security principal are enforced. We describe the operation of this system by walking through the design diagram shown in Figure 3.5.

WIREFrame App: At runtime, WIREFrame registers a background service that waits for connections from client apps (i.e., third-party apps using WIREFrame). When a connection is created, the service binds a new *IPC Agent* to the client app and establishes a stateful connection via Android's

Binder mechanism. If the client app is allowed to display WebViews, the IPC Agent constructs a floating window that contains an actual WebView instance, called the *Concrete WebView*. The IPC Agent maintains an internal mapping between each WebView instance rendered by WIREFrame and its counterpart in the client app. Throughout the lifecycle of the WebView, the IPC Agent handles the client app's requests for WebView functionalities. For a given request, it first queries the *Policy Checker*, which serves as a security oracle. The Checker has a default configuration, but can also load policies from the client app side (i.e., defined by developers or app users) as well as policies from the web side (i.e., defined by the web-content provider). If allowed by the policies, the IPC Agent invokes the corresponding WebView API. The IPC Agent also forwards invocation results or callbacks back to the client app.

WIREFrame places mediated WebViews in individual `Service` components running in isolated processes [4], and therefore strictly separates them from each other and the embedding app. Process separation prevents reflection, memory mapping and other means of stealthy cross-origin memory introspection. This separation applies to not only WebViews' executions, but also their access to local storage, including the cookie database and the accessible paths in the file system, which prevents WebViews housed in WIREFrame, often from different apps, from influencing each other.

In-app WebView Proxy: The *WebView Proxy*, loaded inside the client app, initiates and maintains the connection to the IPC Agent. It also handles client-side data marshalling and unmarshalling. To maintain a correspondence to the look and feel of an embedded WebView, the Proxy builds an empty view component (called the proxy view) in the client app and registers callbacks to visual changes to the proxy view. Whenever these callbacks fire, the Proxy forwards them to the WIREFrame app to propagate the corresponding view change in the concrete WebView. The

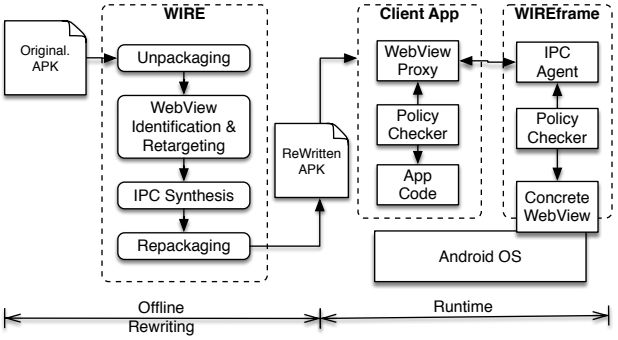

Figure 3.5: System diagram of WIRE and WIREFrame. WIRE is applied to a third-party app before install time, ensuring that it uses the protection mechanisms of WIREFrame at runtime.

proxy maintains the same syntactic interface as an Android WebView. For example, the typical way that a page is loaded in a WebView is by invoking the `loadUrl` method. Thus, the WebView proxy exposes a `loadUrl` method, which it translates into IPC, ultimately resulting in a concrete call to the Concrete WebView within the WIREFrame service. We discuss technical details of how this interaction works in §3.4.

WIREFrame Tool: Although app developers can interface with the WebView Proxy manually, our threat model assumes that developers can be malicious. As such, WIRE is needed to help app users and IT administrators automatically retarget WebViews in (untrusted) apps into proxy connections to WIREFrame. WIRE unpackages a given Android APK, and identifies all uses of WebViews. If any such WebViews exist, WIRE injects the WebView Proxy library and replaces all instances of WebViews with instances of the WebView Proxy. This process is aided by the fact that the Proxy has the same interface as the generic WebView. Finally, the app is repackaged, and can be installed on a device, where it will use WIREFrame. We discuss the implementation of WIRE in §3.5.

3.3.2 Dynamic Access Policies

As mentioned above, WIREFrame enforces access policies to protect web-content and app code from one another. By virtue of running each WebView in an isolated process, a web-embedding app can defeat many of the attacks listed in §3.2: web-content can no longer read files from the host app, thereby mitigating local storage inference. The app is disallowed from injecting JavaScript into the WebView, preventing SSO credential stealing and user impersonation.

In the remainder of this section, we discuss additional details of the policy mechanisms and introduce how these policies can be refined dynamically for fine-grained control by each side within a web-embedding app.

Web Protections: The effect of WIREFrame is to extend the SOP to treat the app code as a distinct origin. A web-embedding app can launch a WebView, but cannot inspect its content. Furthermore, the app is completely disallowed from injecting JavaScript in the WebView. This policy is safe, but it can limit the capabilities of web-embedding apps. For instance, a common behavior of web-embedding apps is to source web-content from a remote origin belonging to the app developer, which should be considered as a single origin.

To support this use case, WIREFrame allows web-content owners to declare exceptions via a dynamic policy-update mechanism. When the WIREFrame connects to a remote website, it makes a request for a special set of WIREFrame specific headers. If the headers are absent, the default policy is employed. If the headers exist, they contain a list of policy objects $\langle A_1, A_2, \ldots, A_n \rangle$. Each policy object A_i specifies a pair (S_i, P_i) where S_i is a security principal and P_i is a policy to enforce over S_i. In our implementation of WIREFrame, the security principal S_i is an app, identified by its unique app signature and developer's certificate. WIREFrame verifies the principal identity using the existing signature-checking mechanism pro-

vided by the OS. A website can also use the ANY principal as S_i, which will apply P_i to all embedding apps. The policy P_i is a set of WebView APIs that S_i is allowed to access. For example, if $P_i = \{$ setJavascriptEnabled $\}$, then S_i is allowed to inject JavaScript. There is also a special LOCKDOWN policy object, which puts the WebView into a high-security mode: JavaScript injection is disabled for the remainder of the session.

WIREFrame and its dynamic policy-update mechanism allows web providers to protect their sensitive content or services that are embedded in untrusted apps. For instance, by defining a simple policy that restricts embedding apps' control over the WebViews, web-content providers can easily prevent the currently unstoppable app-to-web attacks discussed in §3.2.1. Note that more complicated policies or more granular principals could be enforced by WIREFrame (e.g., a policy automaton to prohibit certain sequences of API calls), but our current implementation is sufficient for common use cases. Note that policies are reloaded per-page. Thus, if the user navigates to a new page, policies for previous pages are no longer enforced.

App Protections: A key enhancement that WIREFrame uses to protect apps from malicious web-content (e.g., remote JavaScript calling an exported local Java method) is to regulate requests to the client app on a per web-origin basis. Note that identifying the web origin of a remote request for local resources is not trivial because current WebView design does not provide such information explicitly via its APIs. We obtain the origin information without modifying WebView using a technique called *origin tagging*. By using existing WebView callback interfaces, WIREFrame rewrites JavaScript invocations of WebView interfaces in the web page being rendered. It extends the parameter list of such a invocation to include a string that indicates the origin of the JavaScript (more details in § 3.4). The integrity and confidentiality is guaranteed by the enforcement of the same-origin policy inside WebView. Besides enabling origin-based policy

enforcement, origin tagging also ensures that distinct WebViews within WIREFrame cannot introspect on each other. For example, WIREFrame intercepts WebViews' access to the local file system (via URI loading override) and transparently redirects such access to per-origin private paths, unless a client app defines a less restrictive policy.

Developers can take advantage of origin tagging to define custom policies, placed in the app's manifest. An app-defined policy object follows the same format as that of a web-defined policy object: (S_i, P_i). But in this case, the security principal S_i is a web origin and the policy P_i is a list of local interfaces that the app exposes to S_i. For example, a legitimate location-service app can define a policy whose S_i is the app's own domain and P_i contains a local Java interface `getGpsLocation`, which returns the GPS location. This policy informs WIREFrame that only web elements from origin S_i are allowed to invoke `getGpsLocation` via the app-web bridge, whereas web elements from other origins, even if loaded inside the same WebView, are disallowed.

Such policies enable app developers to expose sensitive interfaces solely to intended web origins, which is a missing capability in today's WebView that allows the web-to-app attacks discussed in §3.2.1. With this capability, app developers no longer have to bear high-security risk while adding local support to their own or trusted web services.

3.4 WIREFrame Technical Details

In the previous section, we described the high-level protection mechanisms of our system. We now discuss the implementation of the runtime component, WIREFrame, and show how it achieves the security goals introduced above. WIREFrame is implemented as a standalone third-party app that acts as a secure and trusted provider of WebView for regular apps. WIREFrame completely mediates all interactions between an app and its

embedded web content while enforcing fine-grained security policies.

Internally, WIREFrame wraps one or more default WebView instances and use them to service apps requests for WebView features. Apps make such requests and receive results via well-defined IPC interfaces exposed by WIREFrame. Each IPC interface corresponds to a public WebView API and provides the equivalent functionality, except that it performs comprehensive security checks and enables policy enforcement. When in operation, WIREFrame overlaps its WebView UI on top of the invoking app's UI in the exact area where the original WebView is expected, providing a consistent and seamless user experience (i.e., the user is not aware that a web-embedded UI is in fact composed and supported by two separate apps). To keep the UIs of both apps synchronized, WIREFrame and the client app collaborate to captures user-interaction events (i.e., touches) and ensure that the proper UI receives the event based on its position.

App developers may directly interface with the WIREFrame service by making IPC calls in place of WebView APIs. However, doing so adds an additional level of complexity to using WIREFrame: the developer needs to manage the IPC channel on top of embedding WebViews. To enable easy adoption, we developed a proxy library that developers can easily import into their apps. This proxy includes a `WebViewProxy` class that has the same interface as the default Android WebView. When the WebViewProxy is started it establishes an IPC connection to the WIREFrame app. The developer can simply call the methods of the WebViewProxy, which will translate each call into an IPC operation on the connected WIREFrame. Furthermore, we developed WIRE to automatically patch the proxy library into legacy apps and refactor the usage of WebView into IPC invocations to WIREFrame without any developer assistance (WIRE is discussed in §3.5). Therefore, our system can be easily and quickly adopted in practice. An advantage of this deployment is that a developer or an end user can transition an app from using WebViews to using WIREFrame mediation

easily. It also allows for deploying regular WebViews and WIREFrame side-by-side. We discuss the security implications of this deployment further in §3.6.

In the remainder of this section, we discuss the implementation of WIREFrame by discussing how it handles the key challenges in its design.
Serialization: Android requires that objects passed via IPC have methods to handle their internal data marshalling and unmarshalling by implementing the `Parcelable` or `Serializable` interface. A few complex class types referenced in the WebView APIs do not implement these interfaces, and therefore, cannot be passed via IPC. Although data marshaling for IPC is a well studied problem, the unique constraints that we faced in designing WIREFrame make the existing solutions non-applicable. For instance, adding serialization support to complex class types is not feasible without changing WebView or Android middleware. Furthermore, even if serialization methods could be added, a type may have volatile state that prevents if from being fully serialized or passed across app boundaries. In other words, such objects are inherently bound to their app contexts.

We handle unserializable types using a technique we call *object shadowing*. The intuition behind object shadowing is that, if an object cannot be moved to, or duplicated in, the remote process, we keep it in the original program context while creating a shadow copy of the object in the remote process. The shadow object acts as a transparent proxy for the original object: it only contains the public interfaces of the original object. The shadow copy's implementation of these interfaces simply invokes the corresponding interface exposed by the original object via IPC. As a result, the shadow object allows code in the remote process to invoke public methods or access public fields as if the original object were passed to the remote process. At the same time, when its methods are invoked, the original object functions properly without suffering from broken dependencies that would otherwise occur if the object had been copied or duplicated

in the remote process. Figure 3.6 shows an example of applying object shadowing to the second parameter of `WebView.evaluateJavascript`, a `ValueCallback` object. In the example, the original object, `callback` is kept at the client app side while a shadow object, `shadowCallback`, is automatically created in the WebView instance in WIREFrame. The shadow object forwards calls to the public interface, `onReceiveValue`, back to the original object via the IPC channel provided by WIREFrame.

Object shadowing can be recursive when a shadow interface takes or returns complex objects. The recursion is bounded due to the fact that object interfaces always converge to primitive types that can be directly transferred over IPC. The generation of these objects and classes is straightforward and automated. Thanks to object shadowing, non-serializable objects involved in WIREFrame IPC interfaces are invoked in their original app context, rather than copied across app boundaries, which allows IPC-unfriendly objects to be used in a cross-app fashion.

Visual Fidelity: WebViews running in WIREFrame need to appear and function as native UIs of their embedding apps. This includes not only displaying at the same scales and locations as native WebViews but also responding to events, for instance, indicating device rotation from landscape mode to portrait mode, in which case the content rendered in the WebView should automatically rotate and resize. Simply using the floating UI feature of Android does not enable synchronization among the UIs belonging to two apps. For instance, when the device is rotated, a series of events is sent down the view hierarchy of the embedding app, updating the layout of each element. This context is not available to the WIREFrame and is necessary to calculate the final position and size that the WebView would have occupied.

To achieve visual fidelity, the Proxy WebView maintains an invisible view (i.e., a transparent placeholder) that takes the size and shape of the original WebView and forwards all view events to the WIREFrame via

```
/* An example of a WebView API involving
complex parameter */
void evaluateJavascript(
     String script,
     ValueCallback<String> resultCallback)
```

```
/* Pseudo-def of a ValueCallback object in the
original app context */
ValueCallback<String> callback =
new ValueCallback<String>(){
     @Override
     public void onReceiveValue(String value) {
          ... // Original callback handler
     }};
```

```
/* Pseudo-def of a shadow ValueCallback object
in the WIREFrame app */
ValueCallback<String> shadowCallback =
new ValueCallback<String>(){
     @Override
     public void onReceiveValue(String value) {
          // Auto-generated IPC stub
          WebHarborIPCAgent.remoteInvoke(
                    OrigValueCallBackObjID,
                    value);
     }};
```

Figure 3.6: An illustration of object shadowing

IPC. Android supports several types of floating UI, by which an app in the background can draw UI elements on top of the currently foregrounded app. We leverage the floating UI feature to place the trusted WebView managed by WIREFrame over the rewritten app while the latter is running in the foreground. The WIREFrame WebView occupies the exact screen area where the original WebView would have been rendered had the app not been rewritten or WIREFrame not deployed. To avoid marshaling all of the necessary context, the invisible element inserted by the proxy (i.e., a transparent placeholder) is placed in the client app's view hierarchy where the WebView would be. This element responds to layout events and automatically forwards its new size and position to the WIREFrame, which mirrors these updates appropriately.

Origin-based Policy Enforcement: To achieve fine granularity, our policy enforcement needs to track the origins of web content and the origins of

web-initiated calls to the app-web bridge. Without this capability, WIREFrame cannot enforce useful policies such as allowing only a particular origin to invoke the GPS-reading method exposed by a client app. However, realizing this capability in WIREFrame is challenging because none of the WebView APIs are aware of the notion of web origins (i.e., their parameters and return values do not carry information about origins).

To retain the origin information for each web-to-app data access or code invocation, WIREFrame employs a dynamic HTML rewriting technique, which we call *origin tagging*. This technique is built on the standard WebView callbacks that the embedding app (WIREFrame in this case) can register to handle web-navigation events. Upon each page (re)load or DOM element refresh event, WIREFrame receives a callback from WebView's rendering-event inspector. During this callback, WIREFrame rewrites every Javascript-to-WebView invocation in the to-be-loaded page by appending an origin label to the parameter list (i.e., as a new final parameter). WIREFrame then resumes processing the page. Using the origin tagging technique, WIREFrame attaches origin labels to the invocations of the app-web bridge in a webpage before the page is loaded. Any obscured invocation that is not labeled will be rejected by the Policy Checking during invocation. Note that an origin label is an encoded string that can only be decoded into a plain origin string with the secret key for the current webpage. The encrypted labels prevent malicious web content from faking or tampering with their origin labels. Later on when a rewritten invocation is triggered, the Policy Checker retrieves the origin label by inspecting the last parameter of the call. It decodes the label, verifies its integrity, and then checks the invocation against the origin-based policy.

Complete Mediation: An important guarantee that our system provides is that *all* app-web interactions are subject to policy enforcement. However, there is an inherent difficulty in maintaining this guarantee without modifying the Android framework: An adaptive adversary may attempt

to hide the use of a default WebView from rewriting by WIRE, or may re-implement web-embedding features in third-party code. To address these scenarios, we imbue WIREFrame with the ability to intercept all packets coming to and from a client app. Thus, a sensitive website can require that it must be accessed from WIREFrame, in which case, WIREFrame disallows packets originating from the client app destined for that website.

We realize this feature using the `VpnService` class, which allows an app to act as a VPN client without requiring root privilege. While the intended usage of the class is for building a tunnel interface, we repurpose it for packet inspection on selected apps. By implementing a *per-app VPN*, WIREFrame can force the client apps to send all traffic through it while not affecting other app's network connections.

Note that using a mediated tunnel in this way leverages a key advantage of our approach: the static analysis of WIRE mandates that the secure service is set up at the entry points and torn down at the exits of a client app. The complete mediation enforced by the secure service ensures that any WebViews missed by the static analysis are detected at runtime.

3.5 WIRE Technical Details

The security mechanisms discussed in §3.4 only take effect if WIREFrame is used by a web-embedding app in place of its regular WebViews. While benign developers might choose to deploy our mechanisms, malicious developers have no incentive to do so. Our offline rewriting tool, WIRE, addresses this concern by replacing all uses of WebView with uses of the secure WIREFrame proxy. This section provides details on the design and implementation of WIRE.

Packaged App Analysis: One of the key advantages of our approach is that it does not require assistance from developers. Without developer

support, the tool can rely only on the packaged app (.apk file) and compiled bytecode. To handle this challenge, WIRE leverages previous work on reverse-engineering and re-compiling Dalvik bytecode. In particular, we use the open source Apktool to unpackage and repackage code and resources from an apk [8]. We use the Soot Java Optimization Framework [48] and Dexpler [11] to extract Dalvik to an intermediate representation and recompile the rewritten code.

Furthermore, WIRE is designed as a modular pipeline, with the rewriting phase decoupled from unpackaging and repackaging the app. Thus, improvements to the underlying tools can be easily integrated into our workflow.

Identifying WebView Usage: Because WIREFrame prevents the use of the default WebView, it is crucial for the proper operation of the client app that all legitimate uses of WebViews are identified and replaced. Unfortunately, this identification can be challenging. In addition to WebViews that are programmatically constructed and configured at runtime, an app can define the WebView UI and its layout using an XML manifest that the system loads at runtime. Thus, WIRE introspects and modifies the applications code, resources, and XML metadata files.

Satisfying Lifecycle Constraints: Android apps run in an event-driven *lifecycle* managed by the system. Events are fired by the Operating System in response to events or system notifications. An implicit ordering exists between the lifecycle events: one event cannot happen until the component's lifecycle has gone through preceding events. Without considering component lifecycle and the implicit constraints, app rewriting can cause erroneous or interrupted app execution. Thus, WIRE includes a model of the Android lifecycle, ensuring that the WIREFrame is properly running and bound before each invocation.

3.6 Security Analysis

We now discuss the security and robustness of our system against evasion. Our discussion concerns attacks launched by either a malicious client app or a malicious webpage—two types of adversaries allowed in our threat model. We explain how our design addresses each adversary, and discuss limitations of our approach.

Malicious client apps: Adversarial apps may attempt to evade our bytecode rewriting process to maintain the usage of an unprotected WebView, and in turn preserve an attack on the WebView's content. A sufficiently advanced adversary may be able to evade WIRE through obfuscation (e.g., using Java reflections), native code, or dynamically loaded code. However, the per-app `VPNService` implemented in WIREFrame would intercept the traffic coming from the elicit WebView and block it. This behavior highlights the fail-safe nature of our system: if a hidden web connection avoids WIRE, it will cause the app to break rather than allowing unmediated web access.

Malicious apps may hijack the IPC channel through which the client-side proxy and WIREFrame communicate, leading to unchecked or forged WebView API calls. The adversary may employ IPC spoofing (i.e., communicating to WIREFrame directly without going through the local proxy) or compromise the local proxy. The client app is considered as a single, untrusted entity from the perspective of the web content, and all calls to the IPC interface are mediated on the WIREFrame side. In other words, the WIREFrame treats all apps as if they were under the control of an adversary spoofing the local proxy.

Malicious web content: When rendered inside WIREFrame, a malicious web page may attempt to break the isolation and security checks enforced by the trusted WebView. Since the web origin plays a central role in regulating untrusted web content, the origin-tagging mechanism of WIREFrame can be an obvious target for attackers. For example, malicious JavaScript

can either obfuscate its invocation of Java interfaces to avoid tagging, or spoof its origin by stealing a tag assigned to scripts from other domains. Although it is possible to hide Java invocations, such invocations are rejected by WIREFrame because they are not tagged. Stealing tags is impossible because reading tags of scripts from other domains is prevented by the SOP. Moreover, origin tags cannot be forged or reused because they are randomly generated on a per-session basis. In very rare cases, attackers may successfully exploit vulnerabilities in the web-rendering engine, and possibly compromise the TCB of WIREFrame. While not designed to mitigate such low-level attacks, our system does significantly reduce the potential damage that such attacks can cause to either client apps or WIREFrame thanks to the process-based separation of each WebView instance.

3.7 Evaluation

Our evaluation seeks to answer the following questions:

1. *Correctness*: Do apps have the same appearances and functionalities after adopting WIREFrame?
2. *Effectiveness*: Does WIREFrame enforcement effectively prevent attacks on the app-web bridge?
3. *Efficiency*: What is the performance impact of replacing in-app WebViews with WIREFrame?

Experimental Highlights: Our experiments validate our approach and show encouraging results. All 20 popular apps of different categories continued to run correctly after being rewritten using WIRE to use WIREFrame, with 90% showing no visual differences at all. We found that WIREFrame effectively prevents both web-to-app and app-to-web attacks:

App Name	Category	Functional	Visual
Dictionary.com	Reference	✓	✗
Flappy Bird	Entertainment	✓	✓
Facebook	Social	✓	✓
LinkedIn	Social	✓	✓
The Hindu	News	✓	✓
NY Times	News	✓	✓
The Economic Times	News	✓	✓
Groupon	Social	✓	✓
IMDB	Reference	✓	✓
Amazon Shopping	Shopping	✓	✓
Ebay	Shopping	✓	✓
Textgram	Social	✓	✓
Jewels Saga	Entertanment	✓	✓
Ask.fm	Social	✓	✗
Photodirector	Media	✓	✓
Angry Birds	Entertainment	✓	✓
Instant Inventory	Shopping	✓	✓
Fun Run	Entertainment	✓	✓
LivingSocial	Social	✓	✓
QuickPic	Media	✓	✓

Figure 3.7: Table of benign apps rewritten using WIRE. A ✓ indicates that the given app uses an overlay over a WebView, while a ✗ indicates that the given app does not.

WIREFrame successfully stopped the attacks against four popular third-party WebView libraries that were otherwise vulnerable, and prevented real web exploits targeting apps found in the wild.

In the remainder of this Section, we describe our methodology for arriving at these conclusions, and provide a more in-depth analysis of our results.

3.7.1 Methodology

We used four sets of apps to evaluate our system: a set of *correctness* apps found in the wild, gathered to ensure that our system can handle the wide

variety of apps available, a set of *attack apps* that we designed to mount attacks against 4 popular third-party WebView libraries, a set of *benign apps* that use WebViews and are popular in the Google Play market, and a set of *benchmark apps* for precisely measuring the performance of our approach. All experiments were run on devices running Android 5.0.

Correctness Apps: To ensure the external validity of WIRE, we applied it to a collection of 7166 apps downloaded from Google Play and 3rd party markets. Given the size of this sample, running each app manually is infeasible. Our goal with this sample is to ensure that the transformations applied by WIRE are correct and produce valid bytecode even on apps found in the wild.

Attack Apps: Our set of attack apps exploits WebViews used in four popular third-party libraries: LinkedIn, Facebook, Twitter, and Foursquare. The basic flow of the attack is very similar to that of our example attack discussed in Section 4.2. Each attack app creates a WebView and uses the API of the third-party library to get a sign-on URL from the associated provider. The attack app then injects JavaScript into the login page to read the username and password fields on that page.

To apply the extra security protection of a login page, WIREFrame needs to know when it is on a secure login site. In a production system, the secure web page would provide a dynamic policy to indicate to WIREFrame that the page should selectively allow JavaScript to be injected or any web content to be introspected upon. However, in our experiment, instead of altering the HTML headers of the login page and installing a dynamic policy on behalf of the SSO providers, we simply rely on the default and the most restrictive policy of WIREFrame: by default, without cooperation from the site, WIREFrame does not allow apps to inject scripts to or inspect on an embedded WebView.

Benign Apps: Our suite of benign apps is composed of 20 popular apps off of the Google Play store. The apps come from a variety of categories

Tested API		API Invocation Time (in milliseconds)				API Invocation Overhead (relative)	
		w/ WebHarbor		w/o WebHarbor			
Name	*Type*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*
`clearCache`	basic	2.38	2.23	1.22	0.82	**0.95**	**1.72**
`getTitle`	basic	0.58	0.183	0.30	0.11	**0.93**	**0.72**
`capturePicture`	complex	6.08	6.97	1.16	1.64	**4.25**	**3.24**

Figure 3.8: Added Runtime Overhead of WIREFrame protection mechanisms (Thus 0.95 represents a nearly 2x slowdown). Overhead includes the IPC invocation and policy checks. Note that the complex object shadowing of `capturePicture` includes the time needed to copy an entire screenshot of a WebView between apps.

including reference (for reference material, such as a dictionary), entertainment (for games), Social (for social content such as Facebook), and Media (for traditional media apps such as image viewers). Figure 3.7 shows the full table of apps in our suite, along with their categories.

Benchmark Apps: To characterize per-operation overheads associated with WIREFrame, we manually insert timing checks into a set of synthetic apps. We are broadly interested in three measures of overhead: the space cost of having an additional app on the device, the per-launch overhead of establishing the communication channel between client apps and the WIREFrame services, and the per-use overhead of the IPC-based interaction between a client app and its embedded WebView.

3.7.2 Analysis

Correctness

We performed two experiments to ensure the correctness of our approach. In the first, we ensured that the app rewriting performed by WIRE produced valid bytecode. In total, we found that 46 of our 7166 apps (approximately 0.6% of apps) failed to complete the rewriting successfully. We note that all of these apps also fail to complete a null transformation in

	Time				Relative Overhead	
	w/ WebHarbor		w/o WebHarbor			
	Nexus 5	*Samsung S5*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*
Load URL w/ origin tagging (ms)	13.24	15.16	12.63	14.30	**0.05**	**0.06**
Load URL w/o origin tagging (ms)	12.38	14.43	12.63	14.30	**-0.02**	**0.01**
Average app boot and load (s)	5.37	6.12	5.09	4.68	**0.05**	**0.08**

Figure 3.9: Runtime Overhead of the Load URL API. Note that loading URLs without origin tagging has a low enough overhead that it is within the margin or error.

	w/ WebHarbor		w/o WebHarbor	
	N5	*S5*	*N5*	*S5*
Client – Kernel time (s)	0.6	0.3	0.8	7.6
Client – User time (s)	1.8	1.1	8.7	3.7
WHbr – Kernel time (s)	0.7	2.4	-	-
WHbr – User time (s)	3.7	9.6	-	-
Client – VSS (KB)	945	965	1021	1061
Client – RSS (KB)	66.6	37.4	72.7	100
WHbr – VSS (KB)	947	952	-	-
WHbr – RSS (KB)	46.7	47.1	-	-

WHbr = WebHarbor App

Figure 3.10: Resource Utilization of CPU and Memory. WIREFrame incurs modest overhead, mostly composed of time and memory in user space.

Soot (our underlying analysis engine). Thus, we believe these limitations are not intrinsic to our technique.

In our second correctness experiment, we tested that the apps in our benign sample of apps continued to perform correctly when run manually. Figure 3.7 shows the results of this experiment on our 20 web-embedding apps.

Functional Correctness: The *Functional* column indicates that the functionality of the app was preserved: no crashes were detected in a manual session of operating the app, and all web and app tasks completed using the WIREFrame just as using a plain WebView.

Visual Fidelity: The *Visual* column of Figure 3.7 indicates if the app using WIREFrame versus the in-app WebView appeared to be identical. We discovered none but two apps that did not meet this criteria, which were expected corner cases. As a security feature, WIREFrame does not allow client apps to overlay UI over any part of WebView, and therefore, prevents clickjacking and other UI confusion attacks. The 2 apps failed the visual fidelity test because of this deliberate security restriction of WIREFrame. In the `Ask.fm` app, a loading widget from the app is placed over the WebView while it loads, and is thus not visible in the rewritten app. In the `Dictionary.com` app, a widget from the app displays an advertising message for a premium version of the app over web content. In both cases, the workflows of the apps remain undistorted. Furthermore, these offending overlays could have been embedded directly into the web content or displayed elsewhere in the apps.

Effectiveness

For each of the four attack apps that we tested, we found that WIREFrame was effective in preventing the malicious behavior that we inserted.

Effective Enforcement: The attack apps import and exploit the authentication libraries from Facebook, Foursquare, LinkedIn, and Twitter, all

of which use WebViews. To exploit the library, the attack apps inject JavaScript into the login window for each service according to the techniques described in Section 4.2. For all four libraries, we successfully extracted the username and password when the app used a default in-app WebView. We then rewrote each app using WIRE, and replayed the attacks. In each case, WIREFrame successfully prevented exfiltration of credentials.

In addition, we simulated the web-to-app attacks and examined WIREFrame's origin-based policy enforcement. We created a test app which, employing dynamic policies, exports a range of sensitive Java interfaces exclusively to web content from a trusted origin. We also composed a mash-up page with multiple iFrames and scripts from different origins that all try to access the exported app-web interfaces. During the test, the app first loads the mash-up page using a regular in-app WebView and then does the same using WIREFrame. Our results show that the sensitive interfaces were universally accessible to all web content loaded in the regular WebView but were only accessible to the trusted domain from within WIREFrame.

Those tests show that WIREFrame's enforcement is effective at isolating the threats that apps and embedded web content may impose on each other.

Efficiency

The extra security protections afforded by our approach have overheads in terms of resource utilization (CPU and memory) and runtime overhead. While correctness and effectiveness are the primary concerns of our system, we also evaluate if the mechanism is efficient enough to use.

Resource Utilization: Figure 3.10 lists the resources used by an app with and without WIREFrame. VSS lists the virtual set size (VSS), which is a measure of the maximum utilization of virtual memory. RSS lists the

resident set size, which measures the maximum footprint in resident memory. An app using WIREFrame has a smaller memory footprint across both metrics, because web content is now being loaded in the WIREFrame process. There is also a constant overhead of less than 1 MB for running the additional process, but given that modern Android devices such as the S5 are equipped with 2GB of RAM, we consider this overhead to be negligible.

Runtime Overhead: Rewritten apps incur overhead from the extra bookkeeping performed for WIREFrame protection mechanisms. We measured the runtime increase across representative web APIs of both types.

Figure 3.8 shows the runtime of invocations of two *basic* APIs, in which the arguments to the call do not require object shadowing and *complex* APIs which do. These functions measure the additional overhead of app to-web protections, which is accounted for by the actual IPC invocation and related marshalling. For basic APIs, we experience an approximately 2x overhead. For complex APIs, we experience a 3-4x increase.

Figure 3.9 shows the overhead of loads with and without origin tagging. This overhead is accounted for by building and inspecting the web origin. As expected, we experience negligible overhead without origin tagging (within the margin of error of our timing tool, `DDMS`).

Although these overheads are high in relative terms, they are mitigated by the fact that the absolute overheads are small. Given that these WebView APIs are called infrequently in an app, the runtime overhead accounts for a negligible factor of the total runtime of the app. We have found these latencies to be acceptable in use, but we note that there is room to optimize our techniques, especially with regards to object shadowing. Furthermore, interacting with web content is especially amenable to absorbing the overheads introduced by IPC, runtime of such operations will often by dominated by network latency.

3.8 Related Work

Studying WebView-related Attacks: Previous studies have reported several types of WebView attacks that exploit the app-web bridge. Luo *et al.* [31] demonstrated that, using WebView APIs, apps may inject malicious scripts into embedded web content, and at the same time, unauthorized web code may invoke app-exported Java methods. Roesner *et al.* have noted that apps can read passwords from the embedded WebViews [40]. Many works have noted the scope and severity of malicious web content on benign apps (*web-to-app* attacks, in our terminology: Chin *et al.* [13] studied two types of WebView attacks whereby malicious JavaScript scripts perform unauthorized Java invocations and file system access in vulnerable apps. Neugschwandtner *et al.* [35] showed that WebViews can serve as a powerful attack vector when the server is compromised. Thomas *et al.* [46] formulated a model for determining the lifetime of a vulnerabilities in Android using Javascript attacks on WebView as a case study. This model notes the slow deployment of patches in Android, a point that supports our technique of app rewriting rather than patching the system-provided WebView. Wang *et al.* [49] demonstrated the origin-confusion attacks and provided a mitigation that requires OS modifications. More recently, Son *et al.* [45] found that untrusted advertisements rendered in WebViews may infer user profiles by testing the mere existence of certain files, an operation that the current WebView design cannot forbid. Motivated by the findings of those previous studies, our work solves an open and pressing issue—generalizing and preventing WebView-related attacks.

Isolating External Web in Apps: There is a rich body of work [19, 37, 44, 54] on mobile ads isolation. The proposed solutions isolate ads from hosting app by placing ads in a separate process or app. NativeWrap [34] performs a similar kind of isolation to cover web applications in WebViews. Our work also uses process boundaries to separate apps and web content, but is compatible with all kinds of WebView usages and considers both

web-to-app and app-to-web attacks. Unlike previous work, our system allows for policy-driven and origin-based security, and includes a static rewriting tool, WIRE, to help app users conveniently apply WIREFrame to existing apps that use WebView.

Securing Sensitive Web Content in Untrusted Apps: Web-based logins are a common embedded web element that previous research set to secure [12, 28, 41] by means of trusted devices, verified UI, and scrutinized implementation of authentication protocols. In contrast, WIREFrame prevents the web content manipulations unique to WebView. Such manipulations are caused by the faulty security assumptions of WebView and the coarse security control over the app-web bridge. LayerCake [40] is a modified version of Android that prevents UI confusion and clickjacking attacks. It supplies secure user interfaces elements, including SecureWebView that can be embedded in an app but run in a separate process. SecureWebView statically disallows the use of JavaScript and the app-web bridge. Therefore, it can prevent the SSO attacks that partly motivated our work. However, SecureWebView only aims to protect sensitive web content whereas WIREFrame protects both apps and web content as per the policies from both sides. In addition, WIREFrame is backward compatible with the existing Android architecture While the goals of our systems are different, it would be interesting to combine the systems: LayerCake could enable the app-web bridge but enforce the policies that we describe in this chapter, and WIRE could retarget legacy apps to use the OS-provided SecureWebView. An alternative approach used by Mutchler *et al.* [33] and Hassanshahi *et al.* [22] is to scan web-embedding applications offline for possible web-app bridge vulnerabilities. While these chapters do not specifically mention SSO credential stealing, they share a similar threat model to our own in that they consider malicious apps as well as malicious web traffic. Unlike our work, these techniques do not propose defenses other than reporting the possibile vulnerabilities.

Hybrid Frameworks: Frameworks such as PhoneGap / Cordova [2] allow developers to write apps in web languages, including HTML and JavaScript. The abstractions provided by such frameworks could implement some of the protections against malicious web content that we describe. For example, Cordova can hook URL loading and inject filtering. However, it is the responsibility of the developer to use the framework correctly, and thus enforcement is not mandatory. Some recent works [18, 26] attacked hybrid apps via local-code injection or remote-resource abuse. They proposed mitigations that are specific to hybrid apps and require changes to the frameworks. In comparison, WIREFrame does not directly protect hybrid apps and instead focuses on native apps that embed web content. However, since the hybrid frameworks all use WebView as their building blocks, they may in principle adopt WIREFrame's policy-driven, origin-based security model to govern web elements in hybrid apps.

3.9 Chapter Summary

As discussed in this work and previous work, Web-embedding apps increasingly attract attacks from different angles. Several current threat vectors they remain unprotected, due to the lack of practical security mechanisms that can meet security requirements of all parties, including app developers, app users and web content providers.

We propose the use of a secure, third-party app called WIREFrame to provide trustworthy web-embedding while enforcing configurable and origin-based security policies on the interactions between Android apps and embedded web content. Both apps and web content can define policies to secure their own resources at fine-granularities. We have shown that our solution is effective in preventing abuses of the app-web bridge by either malicious web content or malicious apps. At the same time, our system maintains the appearance and functionality of client apps.

Our solution is easy to deploy. It requires no modification to the Android operating system or framework. Furthermore, through the use of our offline app-rewriting tool, WIRE, we can retarget legacy apps to benefit from the enhanced security of WIREFrame without requiring developers' cooperation.

4 Minionizer

Note that for typographical reasons, several figures appear at the end of this chapter

4.1 Introduction

Smartphones have emerged as ubiquitous computing devices accompanied by unique challenges for security and privacy. Through pervasive access, users present troves of personal data to these devices, both by manual interaction and through numerous sensors onboard the device. The misuse of such data can cause significant harm to a user's privacy. Mobile-operating-system (OS) providers have increasingly integrated priviledge mechanisms to lock down the use of privacy-sensitive operations. iOS presents permission requests dynamically while the app runs and when it needs them, in the hope that the UI context hints to the purpose of the permission request. Windows Phone 8 requires the user to approve an app's full set of permissions at install-time. Android recently adopted a hybrid permissions model where permissions are granted at install time, but the user can revoke permissions dynamically through prompts or by disallowing a single permission to an app. We believe that many of our findings can be applied to many mobile OSes, including the Windows Phone and iOS. However, we focus our exclusively on Android. While these mechanisms mitigate some of the most egregious abuses of permissions, they fail to satisfy a number of important privacy needs.

To describe these needs, we introduce an example that we will use

throughout this chapter, *Brightest Flashlight Free*. Brightest Flashlight Free is a popular Android flashlight app, downloaded from the Google Play store over 1.2 million times and carrying a 5-star rating. We consider mobile-OS-privacy mechanisms for this app in the context of two best-practice security tenets: the *principle of least priviledge* (PLP) and the *principle of informed authorization* (PIA).

1) PLP: The PLP states that a principal should be given no more permissions than necessary to fulfill its purpose. However, multiple principals of diverse provenance and trust levels can exist within a single app. Brightest Flashlight Free includes code for the core flashlight functionality and for several advertising libraries. The user is given no means to determine how permissions map to these app-internal principals because the OS treats the app as a single entity. As such, should the install-time permissions to use the camera be granted to the core, the advertising library would also be able to use it. Previous work has emphasized the importance of recognizing different levels of trust for these entities [43].

2) PIA: The PIA states that authorization should be supported by enough context to make an informed decision. Implementing this principle is difficult, because the composite effect of multple permissions may have a greater impact than each permission in isolation. The user might feel comfortable letting the advertising library of Brightest Flashlight Free use the network (to fetch ads), and use their location (to light up at local sundown), the two permissions in tandem can be used to leak the user's location to the Internet. Even when dynamic permission prompts are presented to the user, they are not accompanied by enough context to tell whether permissions are being used together or in isolation. Further complicating matters, the inclusion of such information in dynamic prompts can exacerbate prompt fatigue, in which users become overwhelmed by authorization dialogs and simply begin accepting them [30].

Some previous work has attempted to address these challenges by

identifying undesirable behavior through static analysis [9, 15]. These approaches do not provide any way for users to determine if a flow is actually occurring at runtime. Other previous work has attempted to rewrite the Android permissions model entirely [17]. However, such approaches require updates to the OS itself. At best, such updates are slow to reach users. Historically, handset manufacturers have taken as much as several years to apply upstream patches. At worst, legacy device support may be discontinued entirely and changes to the permission model may require the explicit cooperation of app developers [21].

In this chapter, we introduce a technique called *minionizing* that addresses the problems listed above without incurring the limitations of these previous approaches. This technique works by partitioning an app into a number of smaller, collaborating apps called *minions*. Minion apps contain a portion of the original app representing an action that the user can mediate. The key insight behind minionization is that *splitting the application into smaller pieces converts sensitive code and data flows from intra-app (indistinct to the user and to the OS) to inter-app (distinguishable by the user and the OS).*

The way in which app code is minionized is guided by a policy provided by the user when the app is downloaded but before the app is installed. The policy-driven nature of minionization makes it a flexible technique for re-provisioning the capabilities of an app. For example, Brightest Flashlight Free could be partitioned into two minions: m_1, which includes the core and advertising functionality, and m_2, which includes only the calls that collect the location specifically to be sent to the network. The user may then choose to install m_1 only, effectively limiting the flow of their location to the network without denying the location to the core app.

Minionizing directly addresses the PLP by allowing users to identify principals within an app, and separate them into distinct entities that can be mediated and controlled by the OS. Minionizing also better supports the

PIA: the user policy can list permission flows to be partitioned before the app is installed, thus deauthorizing fine-grained composite permissions. This policy-based approach to authorization has the benefit that it does not add to the user's prompt fatigue: no extra runtime action (such as approving a prompt) is required of the user. Furthermore, the user can write (or download from a trusted party) a single policy to apply to many apps, saving the cognitive overhead of examining the permissions of each app they use.

We have developed a tool called Minionizer to implement minionizing. In addition to breaking an app into minions, this tool ensures that the original functionality of the app is maintained when the minions are enabled. The instrumentation enables minions to communicate with each other via OS-level interprocess communication (IPC). By exposing operations as IPC, a desired policy can be enforced using IPC access-control mechanisms (e.g., permissions on Android intents), with fall-back to unmodified execution faithful to that of the original app in the absence of any policy. Furthermore, inter-minion communication leads to graceful degradation of functionality when strict policies (e.g., absolutely no GPS access) are enforced.

This chapter makes the following contributions:

- We formalize app splitting as the problem of finding graph partitions and show how various classes of security policies map to minionizing strategies. Underlying app splitting is a notion of fine-grained, flow-based permission addressing the PLP and PIA.

- We introduce a tool, Minionizer, for performing automatic, optimal app splitting of Android apps based on a specified security policy. Minionizer naturally generalizes the existing work on isolating advertising from the core functionality of an app [42, 47].

- We demonstrate experimentally that Minionizer is practical, sup-

ports a variety of app types (from book readers to translation apps to social networking tools), and incurs low overhead: operations that use permissions incur a low overhead of less than 3% and the total runtime of the app does not experience any measurable slowdown.

The remainder of the chapter is structured as follows:

Section 4.2 overviews and motivates our approach by focusing on the concepts of our running example. In Section 4.3 we detail our technique for choosing strategies to implement minionization policies. Sections 4.4 and 4.5 discuss the technical details of how Minionizer preserves app functionality across minions, allowing minion apps to collaborate. In Section 4.6, we evaluate how applications split with Minionizer perform against their monolithic counterparts. We review related work in Section 4.7 and conclude in Section 4.8.

4.2 Overview

In this section, we motivate the need for fine-grained permission controls in accordance with the PLP and PIA. We then present the design of Minionizer, and introduce how each component of the system uses minionizing to sharpen the permission mediation of an app.

4.2.1 Motivation

To illustrate the permission problems identified in Section 4.1, we present an example app, `Bright Flashlight`, that demonstrates the challenges users face in the current Android ecosystem. This app is a synthentic example designed to present the concepts of Brightest Flashlight Free in a simplified manner. The source code presented here is distinct from that of Brightest Flashlight Free, but requires the same (commonly used) permissions. Unless otherwise specified, behavior is shared by both apps.

`Bright Flashlight` is a flashlight app that works by placing the camera's flash into torch mode. It has the distinct functionality that it tracks the local time of sunset, using the system date and time (which requires no permission), a built-in timetable, and the user's current location.

There are three functions of `Bright Flashlight` that use permissions, as shown in Figure 4.1. These functions illustrate different ways in which the same permissions can be used. The `sendLoc` method uses the permission `ACCESS_FINE_LOCATION` to collect location information at program point P_0 which is sent to the network using `INTERNET` permission at P_1. The `sunsetLight` method also gets location information using `ACCESS_FINE_LOCATION`, at P_2. The data flows to P_3 which uses the location to light the flash using `CAMERA`. The `getAd` method also uses `INTERNET` to download an advertisement from the network at P_4, which it returns. The app can execute all three of the above methods by declaring the use of permissions `INTERNET`, `ACCESS_FINE_LOCATION`, and `CAMERA` in its manifest.

Consider a policy of a user who wants to ensure that their location is never leaked to the network. By installing the app, the user grants permission to the app to send data to the network, and read from the contact list, as it does in `getAd`. Android does not expose sufficient information to determine if such a flow actually exists in the program or not. If the user wanted to ensure that no such flow occurred at runtime, they could completely shut down the network, but doing so stops all content from reaching the device.

The user's desire to ensure that their location does not use the network, without completely disallowing use of the network, cannot be expressed in Android. The OS does not allow the user to view the interactions between permissions, only the static set of permissions that the app might use. Even if the user could determine that such a flow was possible, they have no recourse but to shut down the entire permission.

Fine-grained permission control

At the most basic level, the user's goal is to mediate the ways in which an app uses its permissions in a fine-grained way. This mediation must use existing OS mechanisms and user interfaces for security, in order to simplify the overall user experience. Because Android supports mediation of operations and enforcement of security only at the app level, the best approach is to separate the app into multiple minions, each containing one permission, that collaboratively operate to achieve the original functionality of the app. Once the app is separated into minions, each one installable by itself, the user can effectively express any security policy on flows inside the original app in terms of the minions that she chooses to install.

Separating each use of any permission into a distinct entity that can be independently addressed by the user is thus the fundamental operation supported by MINIONIZER. In the case of `Bright Flashlight`, MINIONIZER can isolate each of the program points P_0, P_1, P_2, P_3, and P_4 into distinct minions, and replace their invocations with inter-process communication code to retrieve the original behavior of these program points.

The goal of the user can be expressed as a list of instruction pairs $\langle s, t \rangle$, where s is an instruction that is a source of sensitive information and t is a sink instruction, each such pair written as $s \overset{!}{\rightsquigarrow} t$. For example, a user may have a general policy of the form shown in Figure 4.2.

```
ACCESS_FINE_LOCATION ⇝! INTERNET
READ_PHONE_STATE ⇝! INTERNET
READ_PHONE_STATE ⇝! INTERNET
GET_ACCOUNTS ⇝! INTERNET
ACCESS_FINE_LOCATION ⇝! SEND_SMS
READ_PHONE_STATE ⇝! SEND_SMS
GET_ACCOUNTS ⇝! SEND_SMS
ACCESS_FINE_LOCATION ⇝! WRITE_EXTERNAL_STORAGE
READ_PHONE_STATE ⇝! WRITE_EXTERNAL_STORAGE
GET_ACCOUNTS ⇝! WRITE_EXTERNAL_STORAGE
```

Figure 4.2: Example Minionizer policy, designed to prevent device identifiers and fine grained location information from being leaked from the device

Note that the policy listed here does not map to a single program point. Instead, it requires that *every* flow that meets these criteria must be mediated. In effect, this policy prevents the exfiltration of the user's location (`ACCESS_FINE_LOCATION`), account data (`GET_ACCOUNTS`), or phone identifiers (`READ_PHONE_STATE`) from being leaked via the internet (`INTERNET`), text message (`SEND_SMS`), or SD card (`WRITE_EXTERNAL_STORAGE`).

In practice, a user is likely to employ a much larger policy than presented above. However, the user need only build (or obtain from a trusted 3rd party) a policy once, after which it can be reused for any app. For example, this policy can be applied to `Bright Flashlight`, in which case only the first rule will apply. For `Bright Flashlight`, this means that the flow from the source P_0 to the sink P_1 will be subject to minionization. However, since the policy does not concern any permission involving the camera, the flow from P_2 to P_3 remains untouched, as does the isolated permission use at P_4.

4.2.2 System Design

For MINIONIZER to work, we need to solve two fundamental problems: 1) given a policy of the form described in Section 4.2.1, what region of code should be extracted to minion apps? The task of partitioning the app in this way requires careful consideration to ensure that the policy is satisfied, while at the same time mitigating performance impact. 2) How should the minion apps communicate to collaboratively maintain the functionality of the original app? In the remainder of this section, we provide an overview to our solutions to these problems by walking through the workflow of MINIONIZER.

The Unpackager

In order to operate on compiled Android apps, MINIONIZER takes a `.apk` file as input, and uses the dexpler [11] frontend to construct a Jimple representation of the app's code, denoted *Soot IR* in Figure 4.3. The component responsible for this operation is called the `Unpackager`. This intermediate code is then forwarded to two custom components: the `Split Director` and the `Splitter`.

The Split Director

App splitting enables diverse security or functionality modifications to an app, which can be realized though different methods of selecting minion apps. We refer to such methods as *splitting strategies*. A `Split Director` implements a single such splitting strategy. While we focus on enhancing the Android permission model in this paper, MINIONIZER allows the user to drop in a new `Split Director` at runtime. Splitting strategies are implemented as plaintext XML files that identify how instructions should be partitioned amongst the minion apps. As such, it is possible for a user to create a splitting strategy by hand, or to manually modify the output

of a director. To demonstrate the modularity of this component, we have implemented directors for two distinct strategies in MINIONIZER:

- The *Isolation strategy* ensures that every permission-using instruction is split into its own minion. This strategy provides support for legacy devices to effectively toggle which permissions an app can use at runtime, giving them the ability to select the exact set of permissions points in the program to remove. However, the strategy does not account for runtime performance when it places split points, and it will split permissions that the user may not care about.

- The *Flow strategy* ensures that data flows are paritioned into minions. Most method calls that require a permissions can be categorized as either a *source* that can gather sensitive data on the device, or a *sink* that can send data to a potentially-untrusted entity. As discussed in Section 4.1, a user may be comfortable with the independent operation of a source and sink method, but unwilling to allow data to travel from a source to a sink. The Flow strategy ensures that for every source/sink pair (s, t) of method calls, data cannot flow from s to t without passing through a minion. This strategy addresses the limitations of the isolation strategy: the director attempts to finds program points at which a split has a small performance impact. Furthermore, the user can express which permissions that they care about splitting, and the Director will ignore other permissions.

Note that in contrast to previous work, the goal of MINIONIZER is not to identify data flows within apps, but rather to mediate flows. In keeping with this focus, we require the user to provide permission points (i.e. individual method calls that they would like to split). In the case of the Isolation strategy, this is a list of permission points that the app should split. In the case of the Flow strategy, this is a list of pairs of permission points to separate. This requirement allows the split directors to be more

flexible, as the user may only care about a subset of the permissions that an app uses. By default, the split directors automatically wrap calls to Flowdroid [9], an existing data flow tool, to identify permission points and identify flows.

We explore the algorithm used by our split directors in Section 4.3. Although both of the directors that we have defined are based on the notion of permissions, a split director could just as easily be built that will split based on application package, or accept regions of the program that perform some action that the user cares about mediating but does not require an explicit permission. For example, one could easily imagine a Split Director that partitions all native code into minions, though this is not a direction that we have explored.

The Splitter

The `Splitter` takes the intermediate representation of the app and uses the splitting strategy to create a collection of new, collaborating apps, called minions. This component of Minionizer represents the bulk of our implementation effort, because it must ensure that splitting the app does not alter its behavior when all minions are present. Thus, the splitter ensures that all minions have enough program state to perform their task while accounting for Android-specific concepts such as app lifecycle and callbacks.

The Support Generator

In addition to performing the actual partitioning of a monolithic app into minions, Minionizer also generates a number of additional artifacts. The Support Generator is responsible for building these artifacts. The Support Generator outputs an install script that assists the user in installing minion apps *en masse*, and also outputs information on the provenance of each

minion, such as the package from which the code was partitioned and the permissions required.

A key advantage of minionizing, as discussed in Section 4.1, is that opaque functionality within the original app is exposed to OS-level security mechanisms. The support generator can interface with such mechanisms to enable policies that cannot currently be expressed on a monolithic app. The support generator can serve as a framework upon which configuration for these tools is generated in tandem with the minion partitions. As an example of this behavior, we have built an example support generator module to configure Android's *Intent Firewall*, which allows the user to blacklist communication between apps [3, 52]. Creating Intent Firewall rules in this way also lightens the burden on the user. Rather than having to manually pick and choose which minion apps to install, the user can install all minions and let the automatically-generated Intent Firewall policy mediate communication between minions. This approach also has the advantage of allowing runtime configuration of the permissions that an app can use: if a user decides to allow a flow post facto, they need only tweak the Intent Firewall rules.

The Repackager

The final step of the tool is to package each minion into its own executable Android app. The backend of Minionizer, called the `Repackager` is built upon the Soot dex compiler, but also uses apktool [8] to recover resources from the original app.The compiler also rewrites a new manifest for each minion, constraining it to the reduced set of permissions that it needs.

4.3 Splitting Strategies

In this section, we discuss our algorithm for building the splitting strategies used by the Split Director, which are described in Section 4.2.

Recall that a *program dependence graph (PDG)* has two kinds of edges: data-dependence and control-dependence edges [32]. First, we formalize the problem in terms of a *labeled program-dependence graph (LPDG)* of an application. LPDGs are essentially PDGs whose nodes are labeled with permissions. Let $G = (V, E, L)$ be a LPDG of an A, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $L : V \to \mathcal{P}$ is a function that labels each node with an element (called *permission*) from a set $\mathcal{P}$). We assume that there is a special element $\perp \in \mathcal{P}$ which represents the *null* permission. Intuitively $L(v) = \perp$ means that the statement corresponding to node $v \in V$ does not need any special permissions. Formally, the problem, which we call the *permission separation problem (PSP)* can be defined as follows:

Problem 4.1. *Given a LPDG* $G = (V, E, L)$ *and a relation* $X \subseteq \mathcal{P} \times \mathcal{P}$. *The problem is to find a partition* $\Pi = \{V_1, V_2, \cdots, V_k\}$ *of* V, *which satisfies the following condition: for all pairs of nodes* (v_1, v_2), *if* $(L_A(v_1), L_A(v_2)) \in X$, *then* v_1 *and* v_2 *are in different sets of the partition* Π.

Our definitions and algorithms also work for other graphs related to an application. For example, all the algorithms work equally well for control-flow graphs (CFGs). However, partitioning based on CFGs provide weaker guarantees than partitioning based on PDGs because PDGs are semantically richer than CFGs (i.e., PDGs capture data and control dependences of programs/applications). Given a partition $\Pi = \{V_1, V_2, \cdots, V_k\}$, we can create k applications $\{A_1, \cdots, A_k\}$ such that A_i consists of all instructions corresponding to nodes in V_i. We call applications A_i $(1 \leqslant i \leqslant k)$ *minions*. A naive algorithm for solving PSP creates a partition as follows: each $v \in V$ such that $L(v) \neq \perp$ is put in its own set and there is a set that consists of all nodes w such that $L(w) = \perp$. We call this naive algorithm *permission isolation splitting*. Of course, our naive algorithm can create a lot of minions. Our goal is to construct as few minions as possible and

also minimize data transfer between the minions. Next we present our algorithm to accomplish these goals.

Our Algorithm

Our algorithm works in two stages: (1) We compute a vertex multicut using dominators and post-dominators. (2) We use the vertex multicut found in step (1) to find a solution to the PSP. The two steps of the algorithm are described below.

(Step 1) An algorithm for finding vertex multicuts. The *vertex multicut problem (VMP)* is defined below.

Problem 4.2. *We are given a graph* $G = (V, E)$*, where* V *is the set of nodes,* $E \subseteq V \times V$ *is the set of edges and a collection of* k *pairs of vertices* $H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$*. The problem is to remove the minimum number of vertices* $V' \subseteq V$ *such that in the resulting graph there is no path from* s_i *to* t_i *for all* $1 \leqslant i \leqslant k$*. In other words, every path from* s_i *to* t_i *(for* $1 \leqslant i \leqslant k$*) goes through at least one vertex in* V'*. This problem is called the directed graph vertex multicut problem (VMP).*

Although the problem of computing optimal vertex and edge multicuts is NP-complete, there exist approximation algorithms to solve these problems [5, 20]. However, these existing algorithms ignore the structure of the program (i.e., the PDGs and CFGs resulting from an application have a very special structure). We present an algorithm that is based on the structure of the program. Specifically, we present here an algorithm for computing vertex multicuts that is based on the concept of pre- and post-dominators. Recall that dominators and post-dominators are used to find *control dependences* in programs [32] and there are efficient algorithms to compute dominators and post-dominators [27].

Assume that we are given a graph $G = (V, E)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges and a collection of k pairs of vertices

$H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$. We present an algorithm which demonstrates that an algorithm for finding hitting sets can be used to find a vertex multicut. With each pair (s_i, t_i) we associate a set M_i with the following property: for all $v \in M_i$, every path from s_i to t_i passes through v. The collection of k pairs of vertices $H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$ corresponds to a collection of sets $\mathcal{M} = \{M_1, \cdots, M_k\}$. Let U be a universe and $\mathcal{C} = \{S_1, \cdots, S_k\}$ be a collection of sets such that $S_i \subseteq U$ for all $1 \leqslant i \leqslant k$. $Z \subseteq U$ is called a *hitting set* for $\mathcal{C}$ iff $Z \cap S_i \neq \emptyset$ for all $1 \leqslant i \leqslant k$.

The problem now is to associate with a pair of nodes (s, t) a set M such that all vertices in M appear on all paths from s to t. For this, we use the concept of dominators and post-dominators. We assume that the graph $G = (V, E)$ has two distinguished vertices $r \in V$ (called the *start node*) and $e \in V$ (called the *exit node*) such that every vertex in V is reachable from r and e is reachable from every vertex in V.

Dominators and post-dominators: A vertex v dominates w (denoted using v dom w) iff every path from r to w passes through v. A vertex z post-dominates w (denoted as z pdom w) iff every path from w to e passes through z. The set of dominators and post-dominators of a vertex w are denoted by DOM(w) and PDOM(w), respectively. Consider a path π from s to t. Since s is reachable from the start node $r \in V$, π can be extended to a path from r to t. Similarly, since the exit node e is reachable from the node $t \in V$, π can be extended to a path from s to e. These observations lead to the following proposition:

Proposition 4.3. *Let (s, t) be a pair of vertices and let $M = DOM(t) \cap PDOM(s)$. Every path from s to t passes through every vertex in M.*

Based on the proposition given above we can formulate an algorithm for finding a vertex multicut, which is based on the dominator and post-dominator structure. (see Figure 4.4).

(Step 2) From Vertex Multicut to Partitions

An algorithm for solving VMP can be used to solve PSP. The description is as follows:

- Assume that we are given an *application* A whose (LPDG) is $G = (V, E, L)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $L : V \rightarrow \mathcal{P}$ is a labeling function. We are also given a relation $X \subseteq \mathcal{P} \times \mathcal{P}$.

- Relation X corresponds to a collection $H(X)$ of pairs of vertices as follows: $(v_1, v_2) \in H(X)$ iff $(L(v_1), L(v_2)) \in X$.

- Now consider the graph $G_1 = (V, E)$ and set $H(X)$. Let $V' \subseteq V$ be a vertex cut for G_1 and $H(X)$. Let G' be the graph obtained from G_1 where outgoing edges from all vertices in V' have been removed. G' induces a partition as shown in Figure 4.5. It is not hard to see that the partition $\mathcal{P} = \{V_1, V_2, \cdots, V_k, V_{k+1}\}$ solves the corresponding PSP problem, i.e., for all pairs of nodes (v_1, v_2) such that $(L_A(v_1), L_A(v_2)) \in X$, then v_1 and v_2 are in different sets of the partition $\mathcal{P}$.

Discussion

Our algorithm based on dominators and post-dominators allows a designer to have control over how the split is performed. First, we introduce some notation from [27]. Vertex v is the *immediate dominator* of w (denoted by v i-dom w), if v dominates w and every other dominator of w dominates v. Similarly, vertex v is the *immediate post-dominator* of w (denoted by v i-pdom w), if v post-dominates w and every other post-dominator of w post-dominates v. The relations i-dom and i-pdom each form a directed rooted tree. Intuitively, a node "higher" up in the tree corresponding to i-dom represents a statement closer to the entry point of an application (similar intuition can be applied to the tree corresponding to the relation

i-pdom). Therefore, if there are two nodes v and w in a set in the collection Z (see Figure 4.4) and v is an ancestor of w in the tree corresponding to i-dom, then v can be preferred over w while constructing the hitting set for Z. Similarly, a designer can specify other conditions. For example, some vertices from the collection Z can be eliminated based on certain conditions before computing the hitting set. Examples of some of these conditions are given below (there are several other domain-specific possibilities).

- Eliminate vertices that correspond to statements in some specific functions (e.g., belonging to a third-party library).
- Eliminate vertices from Z that belong to loops (having the split point in the loop might result in expensive IPC calls because of marshaling and un-marshaling of arguments).

Figure 4.6 shows the CFG for the `Bright Flashlight` code of Figure 4.1, with line numbers preserved from the original figure. Although simplistic, this example shows the importance of picking good split points: consider the naive solution of including blocks 1 and 2 for a partition: since the variable buffer is live across the boundary from block 2 to 3, making the call to the minion corresponding to the partition will require copying the entire buffer. While this behavior might be acceptable for a single transfer to a minion, but altering the minions in this way causes a transfer on every iteration of the loop that begins on line 32. Thus, the heuristics presented above places blocks 1, 2, and 3 into the minion.

Our algorithm to solve an instance of PSP does not take the weight of an edge into account (i.e., we solve an *unweighted* version of PSP). The above-mentioned example highlighted the ramifications of this limitation of our algorithm. In the future, we will investigate the weighted version of PSP, which can be defined as follows: given a LPDG $G = (V, E, L)$, X, and a function $w : E \rightarrow \Re^+$ find a partition $\Pi = \{V_1, V_2, \cdots, V_k\}$ of V, which satisfies the following conditions: for all pairs of nodes (v_1, v_2), if

$(L_A(v_1), L_A(v_2)) \in X$, then v_1 and v_2 are in different sets of the partition Π, and the weight $w(\Pi)$ of the partition is *minimized*. The weight of the partition $\Pi = \{V_1, V_2, \cdots, V_k\}$ is defined as sum of weights of the cross edges $E(\Pi)$ defined as follows: (u, v) is in $E(\Pi)$ iff $(u, v) \in E$ and u and v are in different sets in Π (i.e., these edges cross two different partitions). Intuitively, a weight of an edge corresponds to the amount of data that needs to be marshaled. The weighted version of PSP tries to find a partition that minimizes the amount of data transferred between minions.

4.3.1 Split Director Implementation Details

Edge Repair

An important goal of the split director is preserving the original functionality of the app across cut points. This is a non-trivial task, especially in the case of a PDG-directed split, in which case nodes in the same partition may not even be contiguous. or most instructions, its uses will be generated by a previous instruction in the same parititon and its definitions will be consumed by a following instruction in the same partition. However, if a use is generated outside of the partition, it must be marshalled in and it a definition is required outside of the parition it must be marshalled out.

The Split Director passes over every instruction in the partition to ensure that the its use/def relations will be satisfied by collecting values to be marshalled in and out. In the common case, marshalling in can occur in a single batch at the entry to the partition and marshalling out can occur in a single exit from the partition. However, if the partition is not contiguous the minion may be specialized and split into multiple methods, allowing marshalling to occur in stages where control returns to the core, and is then re-established in the minion with another round of marshalling.

Permission Mapping

Both of the split directors that we present above rely on a permission labeling function L that maps nodes of the app's CFG to a permission. For simplicity, we assumed $\mathsf{L} : V \to \mathcal{P}$, so a node is only mapped to a single permission. In practice, a node may require a finite set of permissions. Extending the strategies support this behavior is trivial, and in practice our labeling function does not assume any structure on the set of permissions $\mathcal{P}$.

Previous work has noted that the Android Open Source Project does not maintain a canonical "permissions map" of the permissions required to invoke each Android function [16]. Fortunately, a number of previous systems have been built to infer a permissions map. In the spirit of the modular nature of the Split Director, Minionizer uses an external permissions map, and provides parsers for the formats of PScout [10], Stowaway [16], and Flowdroid [9]. By default, our splitter uses the permission map provided by Flowdroid.

4.4 Minion App Generation

In the previous section, we explained how Minionizer defines points at which to split a monolithic input app into disconnected regions. This information is used by the splitter to induce new minion apps based on the split points. Although the split points that the Split Director provides attain a security goal, the task of the Splitter to automatically refactor an Android app based into multiple collaboration apps, is a useful goal in and of itself. For this reason, we plan to release the Splitter as a stand-alone tool. In this section, we explore the implementation details of how Minionizer rewrites apps, beginning with a brief discussion of the background of Android IPC, and then explain how Minionizer uses it to perform inter-app computation.

This level of app rewriting is non-trivial to implement, and relies on a number of unique circumstances that are fortunately present for Android apps. In particular, there needs to be an efficient IPC mechanism that allows for objects to be quickly moved from one app to another, and a way to ensure that the semantics of an application are not altered when those objects are mutated in a remote minion. Fortunately, Android provides such a mechanism in Binder [1].

4.4.1 Implementation Details

The primary concern for a minion is to preserve the app's functionality because the splitting strategies that our tool uses yield regions in which the entry block to the region dominates the exit of the block, and the exit postdominates the entry, there is no need to worry about relocating control flow transfers[1]. However, Minionizer needs to ensure that dataflow that passes through a minion is preserved. While Minionizer could simply instrument an app to copy out all variables that the minion defines, and copy in all variables that the minion uses, doing so would unnecessarily copy data that is not live. Instead, we perform a simple reaching definition analysis to only copy uses that are live at the beginning minion region and only copy definitions that are live at the end of the minion.

Parcelling Objects: Android has a mechanism for cross-app communication, called *Binder*, which allows for performant transfer of file descriptors and "active" objects across process boundaries. This mechanism provides a means for transferring use-values into a minion and def-values out of a minion. Unfortunately Binder achieves its fast IPC by expecting that the communicated objects specify how they will be packed and unpacked (every such object must inherit from a class that implements the `Parcelable` interface). This additional constraint is a challenge for Minionizer, as it

[1]We believe that supporting regions that do not have this property is a feasible implementation detail that we leave to future work

is unlikely that every object that must be transferred to the minion will implement this interface. Fortunately, we can again use the object shadowing presented in Chapter 3. While object shadowing incurs additional overhead, it allows the proper transfer of state from the core application to a minion and back.

Minion Lifecycle: Android apps operate according to a lifecycle, dictated by environmental events. To ensure that the services exposed by minion apps are available to the app as it is launched, Minionizer calls the `bindService` function to binds each minion field at the entry point lifecycle functions of the app. In response to the `bindService` call, the system will invoke the onServiceConnected callback of the app (added by Minionizer, as appropriate), where the service is connected. While this works well for most apps, consider the case in which an entrypoint function itself requires the use of a minion: the onServiceConnected callback cannot be invoked until the app returns from the function, but the function uses minions initialized in onServiceConnected. We resolve this paradox in the following way: For each such callback C that uses a minion, we create a new function C′. The body of C is replaced with code that checks if the service is available, and if so calls C′. If it is not, C raises a global flag f_C indicating that a call to C is necessary, and assigns all arguments of C to newly-added instance fields $args_C$ of the app. When the service is available, onServiceConnected checks f_C, and calls C′ with $args_C$.

In effect, this modification of the app results in services being connected before any entrypoint function of the app takes effect. Note that because the C′ are all called at entry to onServiceConnected, any user code in onServiceConnected will not run until the entrypoint functions are run. This handles any dependencies in the original body of onServiceConnected to data touched in C.

4.4.2 Deployment Details

A potential concern of using Minionizer is that it requires the user to manage more apps than they would otherwise. For example, it would severely hurt usability of the system if each minion app cluttered the home screen. To avoid this circumstance, Minionizer modifies the manifest of each minion app (except for the core), so that it does not subscribe to the intent `android.intent.category.LAUNCHER`. As such, the system recognizes that the minion cannot be launched directly from the home screen, and will only display the core minion, which will maintain the title and icon of the original, monolithic app. Thus, the user's home screen launcher is unchanged by Minionizer.

Installation: Although launching and running split apps is identical to the monolithic version from the perspective of the user, the goal of Minionizer is to give extra control over installation of apps. To this end, we supply a number of scripts that ease the installation and uninstallation of minion apps, which we have made available as part of the Minionizer distribution. In particular, the script allows a user to install all minions associated with an app, uninstall all minions associated with the app, or selectively install individual minions.

App-Store Integration: While the scripts that we provide are sufficient for Minionizer to work, a useful deployment scenario would be to integrate the tool directly into the app download and installation. It would be relatively straightforward to implement a browser extension that splits an app according to a policy written by the user, thus guaranteeing that all apps that the user installs are safe according to that policy.

4.5 Minion Support Artifacts

The key advantage of minionization is that opaque, internal functionality of a single app can be exposed to app-centric security mechanisms. How-

Display Name	# Instructions	# Permissions	# Minions
Bible	575472	16	1
CNN	440211	13	1
Duolingo	562020	14	1
Facebook	272534	17	4
Job Search	153580	8	2
Original Borders	54	0	0
MyFitnessPal	859176	13	2
Pandora	296037	13	1
Pocket Manga	150417	4	0
Ringtone Maker	135487	9	1
Zillow	788544	16	

Table 4.7: Characteristics of the apps used in evaluating the correctness of Minionizer, and the number of minions yielded when the app is split according to our example policy.

ever, the resultant set of minion apps can be more complicated to manage than a single app. To address this complexity, and to fully take advantage of minionization, Minionizer generates several management artifacts. In this section, we describe the artifacts generated by our current implementation of Minionizer. We focus on the artifacts that are created to assist in the mediation of interactions between multiple permissions. However, we note that our approach is suitable to generating minion artifacts to support other types of minionization goals or mediation mechanisms.

4.5.1 Install Script

The most immediate drawback of app splitting is that a user needs to manage multiple apps instead of one. Previous work on static app rewriting does not address this consideration, as users can reasonably expect to sideload a single app after rewriting. However, as the number of minion apps increase, the task of sideloading each minion separately becomes daunting. To address this concern, the policy generator outputs a script

that can be invoked to install minion apps *en masse*.

The support generator can be configured to build two types of install scripts. By default, each of the minions generated by Minionizer are included in the install script. Alternatively, the user may specify a set of constraints that preclude a minion from the install script. Our current implementation allows constraints to specify a permission or package from which minions should not be installed. For example, in the case of `Bright Flashlight`, the user may specify that no minions with the `READ_CONTACTS` permission should be installed.

4.5.2 Intent Firewall Rules

An important goal of deploying apps rewritten with Minionizer is to ensure that policies applied at rewriting time are obeyed at runtime. To meet this goal, we leverage the *Intent Firewall*, an integrated feature of the Android framework. The Intent Firewall accepts an XML-based set of rules that are enforced at runtime and prevent Intents from being passed from one component to another. Figure 4.8 shows an example of an intent firewall XML file for the example app, NetDialer, of Section 4.2. In this case, the firewall enforces, from within the OS, the policy that the core app may not send any intent to minion1, which corresponds to GPS use.

4.6 Evaluation

In this Section, we present the results of our evaluation on Minionizer. We performed several experiments to characterize three key aspects of our technique:

1. *Correctness*: Can apps rewritten with Minionizer continue to provide their desired functionality?

Category	# Apps	# Minions (Avg)
AndroidSpecific	12	1.25
ArraysAndLists	7	1.57
Callbacks	15	1.47
EmulatorDetection	3	2.33
FieldAndObjectSensitivity	7	2.14
GeneralJava	23	1.65
InterComponentCommunication	18	1.0
Lifecycle	17	1.35
Reflection	4	2.0
Threading	5	1.2

Table 4.9: Minion partitioning for the DroidBench categories in which FlowDroid detected leaks. For each of the flows detected by the underlying FlowDroid analysis, Minionizer correctly separates the permission into its own minion. Note that for two categories, Aliasing and ImplicitFlows, FlowDroid (erroneously) did not detect any leaks. However, we consider this a limitation of the underlying system not of Minionizer itself. Any improvements to the flow analysis will in turn lead to new minions.

2. *Effectiveness*: Are apps rewritten with Minionizer prohibited from performing disallowed functionality?

3. *Performance*: Does the rewriting process of Minionizer incur manageable overhead on apps?

We begin by briefly presenting the highlights of these experiments before discussing our methodology and results in greater depth. Each of our experiments used its own collection of Android apps, suitably selected for the question at hand as described in the following sections.

Experimental Highlights: We find that Minionizer ensures utility by preserving the desired functionality of apps while blocking the disallowed functionality, thus providing the security guarantees. Minionized apps exhibit an average runtime overhead of 3% over their original variants and use a trivial amount of additional disk space.

4.6.1 Correctness

We evaluated the correctness of our system in two ways. First, we tested that our rewriting is valid to a wide variety to apps found in the wild. Second, we tested that apps rewritten using Minionizer continue to function as expected. Our set of apps for these tests is a collection of 7000 apps collected from the Google Play store and 3rd-party markets.

The most basic test of correctness that we performed is to pass each of our 7000 apps to Minionizer and use the dexopt tool to verify that the bytecode of each minion is correct. We find that of the total set, 46 fail to complete the rewriting process (approximately 0.7%). We note that these apps also fail to pass through the unmodified Soot engine upon which our tool is built. We believe that these limitations are an artifact of the infrastructure we used and not a failure of our approach.

While dexopt shows that the code output by Minionizer is valid, it does not guarantee that runtime behavior is being preserved by minionization. However, our sample set is too large to dynamically exercise each app in a meaningful way. Previous work has noted that testing Android apps is challenging [6, 50]. Apps are frequently interactive, with significant functionality triggered by user interaction with a GUI. In the absence of a comprehensive testing tool to explore an app's behavior, one must employ either human-generated or semi-random event sequences. Both of these options have disadvantages. Scalability quickly becomes a problem for human users, while semi-random input sequences can be shallow in the functionality they elicit from the app [38]. As the purpose of this test is to determine whether the user experiences the same behavior from an app in both its original and minionized versions, we opted for a manual evaluation approach where a human interacts and observes with the apps. This necessarily limited the number of apps in this experiment.

For our manual test, we built a subsample of apps by randomly selecting eleven top apps (one for each Google Play appstore category) from

the collection of 7000. These apps are listed in Table 4.7. By focusing on this subsample, we could evaluate the effects of minionization in depth. To ensure that minionization did not cause any errors or changes in the functionality of the app, we executed the two app variants (original and minionized) on the same sequence of user interactions, and then manually inspected the resulting user interface (UI) states. We noted any differences in functionality caused by the Minionizer transformation in a side-by-side comparison.

In practice, we executed and manually interacted with the original variants of these apps while recording all interactions in trace files with the help of the Robotium tool [53]. These interaction traces were later replayed using the instrumentation framework built into the Android OS. To ensure that the replay mechanism does not introduce any side effects, we replayed the interaction sequence on both the original and the rewritten app and used these executions to make Minionizer-utility determinations. For each app in the experiment we collected two interaction traces, each sufficiently long to perform a logical task in the corresponding app. On average a logical task took 5 seconds to complete. Our experiments did not show any change in behavior in the minionized apps compared to their original variants. A number of statistics about each app are shown in Table 4.7.

4.6.2 Effectiveness

We evaluated the effectiveness of minionization by testing if Minionizer properly mediates permission flows. An important consideration for this experiment is to have a reliable ground truth on the set of possible permission flows as determined by a system other than our own. For this reason, we built our sample from the 119 applications of DroidBench 2, a test suite originally developed as part of FlowDroid [9]. DroidBench 2 was built specifically for the purpose of evaluating static analyses for information-

flow tracking. As such, apps in DroidBench are crafted by authors from a variety of institutions to provide challenging data flows. We used these applications to test the security of Minionizer, with the expectation that every flow in every DroidBench app should be mediated by a split.

The DroidBench apps are deterministic, thus not depending on phone state. In our experiments, we used the information flows statically reported by FlowDroid as input to Minionizer, with the goal of minionizing the DroidBench apps such that all of the FlowDroid-discovered flows are mediated by a cross-minion IPC. The results of our effectivness experiments showed all of the statically detected information flows (as discovered by FlowDroid) were split such that permission-requesting operations were separated by an IPC call into a minion. For each of the 119 programs that we tested, Minionizer was able to successfully expose each flow to OS-level mediation. As shown in Table 4.9, the number of minions varied between apps, with some apps having no unwanted information flows (and thus no minions in the split version), while others having two or more.

4.6.3 Performance

The primary overhead introduced by Minionizer is due to the cost of each IPC call when data is transferred back and worth among minions. While the cumulative cost of Minionizer IPC over the lifetime of an app execution is low enough to be invisible to the user, mostly because apps typically do not cross cut boundaries frequently, we also wanted a precise estimate of the overhead for isolated instructions at cut points. To isolate the overhead, we crafted a number of apps that only create permission-to-permission flows and do nothing else. These apps do not represent the behavior and performance of a useful app, but provide a worst-case analysis and thus an upper bound on the performance impact of minionization.

The apps chosen for performance evaluation are fully deterministic, do not depend on user input or any environment settings, and behave as

follows.

- *Direct Flow:* In this app, we measure the performance penalty of minionizing the most common form of permission leak on Android, a flow of a device-specific identifier (IMEI) to the network. This microbenchmark measures the cost of a single IPC call to a minion. Our measurements were averaged over 12 runs and compared the original app versus the minionized app.

- *Loop Flow:* Here we modify the direct-flow experiment so that source data is repeatedly queried in a loop. Once the loop is finished, the results of the final query is leaked to the network. The purpose of this microbenchmark is to determine if the mechanism can properly identify good candidate regions for including in a single minion: Minionizer should include the entire loop in the minion and perform a single transfer, rather than performing a per-iteration transfer.

- *Large Flow:* This app tests the overhead of moving a large amount of data into the minion. While a typical minion app will only include a snippet of code and the small amount of data that the snippet uses, in this app we ensure that a large, user-defined class is tainted with source data. We measured the overhead of transferring progressively larger classes.

Our first set of findings deal with the *Direct Flow* microbenchmark. We found that transferring IMEI to and from a minion is inexpensive, costing an average of 3% more in a minionized app. Given that the actual copy operation of the IMEI value is nearly negligable, this overhead is dominated by the cost of the IPC mechanism itself.

For the *Loop Flow* microbenchmark, we observed overheads that were similar to those of the *Direct Flow* microbenchmark. Since we are evaluating

Minionizer using the flow strategy, this is an unsurprising result, as the only difference between the minionized *Loop Flow* and the minionized *Direct Flow* is the extra instructions placed in the minion due to looping.

The *Large Flow* microbenchmark showed that the runtime overhead scaled with the size of the data being transferred to the minions, as captured by Figure 4.6.3.

4.6.4 Discussion

The results of our experiments show that the app-splitting approach that we propose in Minionizer offers security from permission leaks while still maintaining the usability of apps with reasonable performance overheads. Our performance microbenchmarks show that our basic splitting mechanism does not introduce an excessive overhead, while our correctness tests show that in practice the cost is amortized over the runtime of the app. Furthermore, we have found that splitting does not encounter significant usability issues while still mitigating permission leaks. Finally, our tests show that Minionizer is usable over a large corpus of commonly available applications.

4.7 Related Work

The goal of giving programs the least privilege necessary to fulfill their desired functionality is well studied. We note the most closely related work to Minionizer regarding several considerations:

Program Partitioning: Several systems exist to partition applications. In general, the Android permissions model allows our system to bootstrap simple policies without the cooperation of the developer which is a benefit of our domain that much previous work did not have available. Chong *et al.* propose a system for splitting web applications [29]. Unlike that work, Minionizer does not require the placement of annotations, nor does

it require source code, or any effort on the part of the app's developer. However, granting such conditions could potentially improve the performance of Minionizer, though it would require a different threat model. Zheng *et al.* propose a system to partition applications across multiple, mutually distrusting hosts [55]. This scheme also requires annotations to the program source.

Advertising Isolation: There has been a line of work in isolating advertising from the rest of an application, such as [42, 47]. special-case operation of Minionizer [36, 42, 47]. The most closely related work to our own is AdSplit, which automatically rewrites an app to use an isolated advertising library [42]. Unlike Minionizer, AdSplit uses Quire [14], which requires modifications to Android itself. Minionizer runs on an unmodified Android device, and thus has no presence on the actual device. Although the approach of Minionizer is similar to AdSplit, the goals of the systems are different and both users may benefit from using both tools in parallel.

Android Rewriting: Aurasium [51] rewrites apps to specify policies by hooking system calls, and employing a runtime security monitor. Unlike Minionizer, Aurasium does not separate apps into multiple pieces, and does not give the user the chance to control permissions in any way.

Android Isolation: Previous work has explored advantages of application level isolation. In particular, Roesner *et al.* developed a modified version of the Android OS, called LayerCake, that allows entities of different trust levels to be embedded into a single app [40]. At a high level, the goal of this work is similar to that of app splitting in that it sharpens the boundaries of security principals. However, the approach taken by Roesner *et al.* differs from our own in that it requires action on the part of the developers to employ new programming practices to comply with a new version of Android. In constrast, our work focuses on enabling existing security mechanisms to work within the current Android security model. Furthermore, the goal of LayerCake is to enable trusted UI components,

whereas the goal of our work is to isolate fine-grained functionality of apps.

4.8 Chapter Summary

The Android operating system enables misbehaving apps to violate the user's privacy, yet lacks mechanisms for finer-grained control over how apps can use their privileges. Minionizer presents an opportunity to leverage the full potential of the privilege system, while putting control back into the hands of users. Minionizer provides a practical enforcement strategy that does not require modification to the Android OS, and can be used as a framework to enhance or enable other application enforcement mechanisms.

In this chapter, we have presented a concrete application of app splitting for this purpose. We believe that our approach can be used for a variety of security and functionality purposes beyond permission isolation and permission-flow mediation.

```
1  // Flow location to the network
2  public void sendLoc(){
3    LocationManager lm = (LocationManager)
         getSystemService(LOCATION_SERVICE);
4    Location l = lm.getLastKnownLocation("gps"); // P0
5    URL url = new URL(urlBase + l.toString());
6    Object content = url.getContent(); // P1
7  }

9  // Flow location to camera
10 public void sunsetLight(){
11   LocationManager lm = (LocationManager)
         getSystemService(LOCATION_SERVICE);
12   Location l = lm.getLastKnownLocation("gps"); // P2
13   Calendar sunset = timetable.sundown(location);
14   if (sunset < currentTime()){
15     Camera cam = Camera.open(); // P3
16     Parameters p = cam.getParameters();
17     p.setFlashMode("torch");
18     cam.setParameters(p);
19     cam.startPreview();
20   }
21 }

23 // Pull information from the network
24 public byte[] getAd(){
25   URL url = new URL(urlContactIcon + strurl);
26   Object content = url.getContent(); // P4
27   InputStream is = (InputStream) content;
28   byte[] buffer = new byte[8192];
29   ByteArrayOutputStream bkg = new ByteArrayOutputStream();
30   int bytesRead;
31   while ((bytesRead = is.read(buffer)) != -1) {
         bkg.write(buffer, 0, bytesRead); }
32   return bkg.toByteArray();
33 }
```

Figure 4.1: Snippet of code from `Bright Flashlight` demonstrating limitations of the Android permission model. The methods that are shown here use an overlapping set of permissions in different ways that are indistinguishable to the user

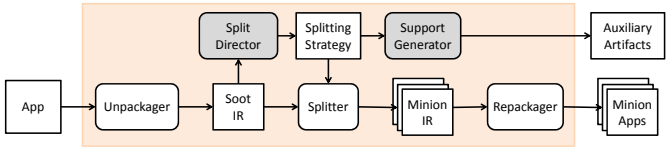

Figure 4.3: Workflow of MINIONIZER. Rounded components indicate code modules, rectangles indicate artifacts. Shaded components of the diagram indicate can be configured at runtime; the `Split Director` can be configured to use a different splitting strategy for partitioning the app into minions, and the `support generator` can be configured to produce policies and install scripts for minion apps. The workflow takes a packaged Android app, such as one downloaded from the Google Play store.

Input: A graph $G = (V, E, r, e)$, set H of k pairs of vertices $\{(s_1, t_1), \cdots, (s_k, t_k)\}$.
Compute M_i (for $1 \leqslant i \leqslant k$) as $\mathrm{DOM}(t_i) \cap \mathrm{PDOM}(s_i)$ Compute hitting set Z for the collection $\{M_1, \cdots, M_k\}$
Output: The hitting set Z.

Figure 4.4: Finding vertex multicuts using dominators, post- dominators, and hitting sets.

Inputs: A collection $H(X) = \{(s_1, t_1), \cdots, (s_k, t_k)\}$, a graph G' such that there is no path from s_i to t_i (for all $1 \leqslant i \leqslant k$).
Consider the sequence $s_1, s_2, \cdots, s_k$ of source vertices and let $G_0 = G'$. **For** $1 \leqslant i \leqslant k$, define V_i as all vertices reachable from s_i in G_{i-1}. To construct G_i, remove all vertices in V_i from G_{i-1}. Let V_{k+1} be the set $V \setminus \bigcup_{i=1}^{k} V_k$.
Output: $\{V_1, V_2, \cdots, V_{k+1}\}$

Figure 4.5: Algorithm for creating partitions from vertex multicuts. Removing a vertex v also means we remove all edges of the form (w, v) and (v, w).

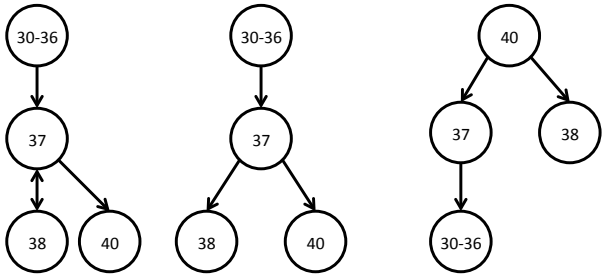

Figure 4.6: Control-Flow Graph (left) Immediate Dominator-tree (middle) and Postdominator-tree (right) for our sample app.

```
<rules>
  <activity block="true" log="false">
    <component-filter name="com.brightflashlight.core/" />
  </activity>
  <broadcast block="true" log="true">
    <intent-filter>
      <action name="com.brightflashlight.minion1" />
    </intent-filter>
  </broadcast>
</rules>
```

Figure 4.8: A sample Intent Firewall ruleset that blocks broadcast intents from the Bright Flashlight core app to a minion.

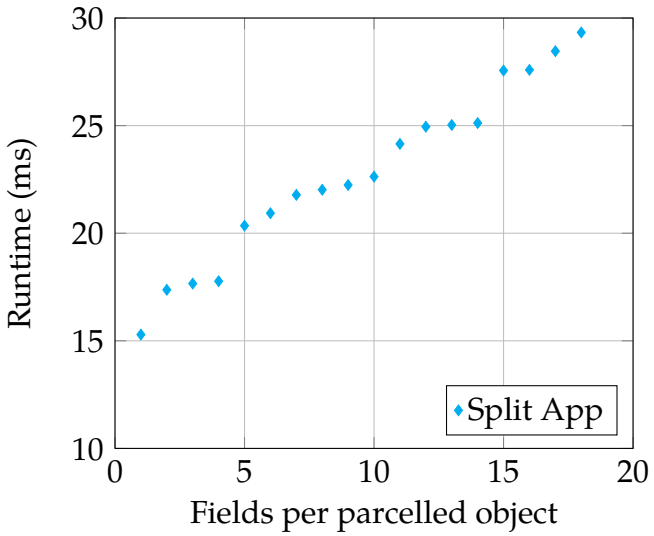

Figure 4.10: Runtime measurements of the *Large Flow* microbenchmark. The minion-IPC overhead increases with the amount of data transferred.

5 Conclusion

In this chapter, we conclude by discussing limitations of the techniques described in this thesis, and explore directions for future work. In §5.1, we discuss limitations of app splitting. In §5.2 we discuss future work.

5.1 Limitations of App Splitting

Applicability: There are some programs, in principle, for which app splitting will not apply. This prevents app splitting from being a general mechanism by which any patch to a program can be applied. There are two cases in which app splitting broadly does not apply:

1. For some program properties, particularly those that are distributed across the program, app splitting becomes impractical. To illustrate this point via a thought experiment, consider implementing a Java Security Manager via app splitting. In a workstation implementation of Java, the Java Security Manager can be used to examine call chains and implement program-wide policies. However, the Java Security Manager is not implemented in Android. While it would, in theory, be possible create an app-splitting instance to replicate the checks performed by the Java Security Manager, the aggressive rewriting necessary to perform the checks would mean that nearly all method calls would be replaced by IPC. The penalties of transferring so much

state into the minions, both in terms of complexity and overhead of the split app, would quickly become untenable.

2. Some programs include behavior that cannot be supported in a split app. As an example, an app could, in principle, create a device-wide singleton object and prohibit any references to be copied. An object that can neither copied nor referenced in the minion is, by definition, inapplicable to app splitting. While it is difficult to imagine that such an object would be present in a practical program and actually required to be referenced in a minion, it nevertheless represents a case beyond the powers of the app splitting technique. A more realistic scenario is that an app contains features beyond the capabilities of our current app-splitter implementation. For example, Minionizer cannot disassemble native code and therefore cannot split native functionality.

Overhead: Both WIRE and Minionizer introduce overhead in the rewriting process. We believe that the amount of overhead is acceptable, as detailed in Chapters 3 and 4. However, optimization to reduce the time taken in transferring context between an app and its split pieces (whether a minion or a WIREFrame) is still an important goal. A natural direction in this regard is to use shared memory or other mechanisms to avoid the full overhead of copying context. However, care must be taken to ensure that shared memory is still isolated enough that no implicit channel is opened.

5.2 Future Work in App Splitting

As discussed in chapter 1, app splitting as a technique has potential beyond the instantiations discussed in this dissertation. Some of the most obvious directions for future work lie in addressing the implementation limitations described in the previous section, and in extending the technique

to additional platforms. Improvements to the core techniques will also benefit the instantiations of the technique. In the remainder of this chapter, we discuss future work for the minionization and WIR instantiations described in this dissertation.

Minionization

Partial-object reconstruction: Currently, when an object is passed between minions, a Parcel Wrapper handles each field of the object, so that it can be rebuilt in the same state as it was on transfer. Many of these fields can be shown statically to hold dead values that are never used in the minion. A useful extension to Minionizer would be to specialize the Parcelization process to only propagate live fields across the minion object.

Inline mediation: Minion boundaries form natural "choke points" in the app, across which all data between a source/sink pair is mediated. Minionizer takes advantage of these choke points by empowering the user to turn functionality of the app on or off statically, before the app is run. This is certainly a useful feature, but we note that minion boundaries also form a convenient point to enforce dynamic policies. For example, a user may wish to check if a URL conforms to a certain domain, and then only allow transfer to the minion if it falls within an acceptable subnet.

Privacy-enhanced minions: The goal of Minionizer is to ensure that the app's behavior remains consistent through the splitting process as long as the relevant minions are installed on the device. However, Minionizer provides a opportunity to alter the behavior of the app by installing minions that expose the same interface to the core app, but behave differently. For example, consider the case of an app that connects to a hard-coded HTTP URL to send data. The user might split the URL connection operation into a minion, and distribute different minions that connect to proxy URLs.

5.2.1 WIR

App Updates: A consequence of using offline rewriting to induce enforcement mechanisms on apps is that apps can no longer be automatically updated on the device. Users are inconvenienced by having to re-apply the WIRE rewriting at each update. However, this inconvenience can be justified by the much enhanced security of web-embedding apps without requiring OS changes. Furthermore, we expect that WIRE will mostly be applied to legacy apps (which are updated less frequently) and untrusted apps that benefit from additional static checking before install time in any case. Apps that do not include WebViews or adopt WIREFrame during development do not need to be rewritten. In cases where app markets can adopt WIRE and perform app rewriting before app release, such as in an enterprise app store, app users can enjoy the security benefits of WIREFrame without facing app-update inconvenience.

WebView State Sharing: As shown by the attack in §3.2, allowing multiple WebViews to run in the same process enables implicit sharing of states, such as history and cookies. WIREFrame runs each mediated WebView in a separate process to disable cross-WebView attacks. It also restricts each WebView's file system access to a per-origin private path by default. However, sharing states among WebView instances created by a same app may be required for legitimate functionalities. While we did not encounter any such cases in our experiments, WIREFrame could be extended to allow multiple WebViews to share a process. We leave this implementation detail, and the design of when to allow sharing, to future work.

A

Appendix A: Android Versions

The Android OS refers to a number of production versions. Due to the rapid release of versions and the role of OEMs in restricting smartphones to particular versions, many of these versions are still in wide use. The below table is due to [23].

Note that since several non-production versions of the OS were released internally for development, the version history does not start at 1.0. However, since the focus of this dissertation is in security, these versions are omitted from the table presented here.

Code name	Version Number	Release Date	API level(s)
Cupcake	1.5	4/27/2009	3
Donut	1.6	2/9/2009	4
Eclair	2.0 - 2.1	10/26/2009	5-7
Froyo	2.2 - 2.2.3	5/20/2010	8
Gingerbread	2.3 - 2.3.7	12/6/2010	9-10
Honeycomb	3.0 - 3.2.6	2/22/2011	11-13
Ice Cream Sandwich	4.0 - 4.0.4	10/18/2011	14-15
Jelly Bean	4.1 - 4.3.1	6/9/2012	16-18
KitKat	4.4 - 4.4.4	10/31/2013	19-20
Lollipop	5.0 - 5.1.1	11/12/2014	21-22
Marshmallow	6.0 - 6.0.1	10/5/2015	23
Nougat	7.0	TBA	24

Table A.1: Android versions

Bibliography

[1] Android Developers: Binder. http://developer.android.com/reference/android/os/Binder.html. Last Accessed: 04/19/2016.

[2] Apache cordova. https://cordova.apache.org. URL https://cordova.apache.org/.

[3] IntentFirewall Source Code. https://android.googlesource.com/platform/frameworks/base/+/633dc9b/services/java/com/android/server/firewall/IntentFirewall.java. Last Accessed: 11/11/2015.

[4] Android isolated service.

[5] Amit Agarwal, Noga Alon, and Moses Charikar. Improved Approximation for Directed Cut Problems. In *STOC*, 2007.

[6] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351717. URL http://doi.acm.org/10.1145/2351676.2351717.

[7] Android Open Source Project. Requesting permissions at run time. https://developer.android.com/training/permissions/requesting.html.

[8] APKTool. Android apktool: A tool for Reengineering Android apk files. code.google.com/p/android-apktool/. Last Accessed: 11/11/2015.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, June 2014.

[10] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382222. URL http://doi.acm.org/10.1145/2382196.2382222.

[11] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the International Workshop on the State Of the Art in Java Program Analysis (SOAP'2012)*, 2012. doi: 10.1145/2259051.2259056. URL http://hal.archives-ouvertes.fr/hal-00697421/PDF/article.pdf.

[12] Elie Bursztein, Chinmay Soman, Dan Boneh, and John C Mitchell. Sessionjuggler: secure web login from an untrusted terminal using session hijacking. In *Proceedings of the 21st international conference on World Wide Web*, pages 321–330. ACM, 2012.

[13] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications*, Lecture Notes in Computer Science, pages 138–159. Springer International, 2014. ISBN 978-3-319-05148-2. doi: 10.1007/978-3-319-05149-9_9. URL http://dx.doi.org/10.1007/978-3-319-05149-9_9.

[14] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, August 2011.

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924971`.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046779. URL `http://doi.acm.org/10.1145/2046707.2046779`.

[17] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and Enhancing Android's Permission System. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 1–18, 2012. doi: 10.1007/978-3-642-33167-1_1. URL `http://dx.doi.org/10.1007/978-3-642-33167-1_1`.

[18] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. *Sat*, 2014.

[19] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[20] Anupam Gupta. Improved Results for Directed Multicut. In *SODA*, 2003.

[21] Dan Han, Chenlei Zhang, Xiaochao Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92, Oct 2012. doi: 10.1109/WCRE.2012.18.

[22] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *European Symposium on Research in Computer Security*, pages 577–598. Springer, 2015.

[23] Google Inc. Android Developers: History. `https://www.android.com/history,`.

[24] Google Inc. Android Developers: Permissions. `https://developer.android.com/reference/android/Manifest.permission.html,`.

[25] Google Inc. Android Developers: Permission Groups. `https://developer.android.com/guide/topics/security/permissions.html#perm-groups,`.

[26] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[27] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1997.

[28] Dongtao Liu and Landon P Cox. Veriui: attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 7. ACM, 2014.

[29] Benjamin Livshits and Stephen Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2013.

[30] Benjamin Livshits and Jaeyeon Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the Usenix Conference on Security*, 2013.

[31] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM, 2011.

[32] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[33] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, 2015.

[34] Adwait Nadkarni, Vasant Tendulkar, and William Enck. Nativewrap: Ad hoc smartphone application creation for end users. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 13–24, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2972-9. doi: 10.1145/2627393.2627412. URL `http://doi.acm.org/10.1145/2627393.2627412`.

[35] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A view to a kill: Webview exploitation. In *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, 2013. USENIX. URL `https://www.usenix.org/conference/leet13/workshop-program/presentation/Neugschwandtner`.

[36] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 71–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1648-4. doi: 10.1145/2414456.2414498. URL `http://doi.acm.org/10.1145/2414456.2414498`.

[37] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[38] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika Mäntylä. Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey. In *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*, pages 36–42, 2012. doi: 10.1109/IWAST.2012.6228988. URL `http://dx.doi.org/10.1109/IWAST.2012.6228988`.

[39] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS*, 2016.

[40] Franziska Roesner and Tadayoshi Kohno. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 97–112, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL `http://dl.acm.org/citation.cfm?id=2534766.2534776`.

[41] Mohamed Shehab and Fadi Mohsen. Towards enhancing the security of oauth implementations in smart phones. In *Mobile Services (MS), 2014 IEEE International Conference on*, pages 39–46. IEEE, 2014.

[42] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362821`.

[43] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362821`.

[44] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, pages 553–567, 2012.

[45] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[46] Daniel R. Thomas, Alastair R. Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, chapter The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface, pages 126–138. Springer International, Cham, 2015. ISBN 978-3-319-26096-9. doi: 10.1007/978-3-319-26096-9_13. URL http://dx.doi.org/10.1007/978-3-319-26096-9_13.

[47] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[48] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–. IBM Press, 1999. URL http://dl.acm.org/citation.cfm?id=781995.782008.

[49] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646. ACM, 2013.

[50] Anthony I. Wasserman. Software Engineering Issues for Mobile Application Development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.1145/1882362.1882443. URL http://doi.acm.org/10.1145/1882362.1882443.

[51] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362820.

[52] Carter Yagemann. Intent Firewall. http://www.cis.syr.edu/~wedu/android/IntentFirewall/. Last Accessed: 11/11/2015.

[53] Hrushikesh Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing, 2013. ISBN 178216801X, 9781782168010.

[54] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.

[55] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 236–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1940-7. URL `http://dl.acm.org/citation.cfm?id=829515.830549`.