Per-App Profiles with AppFork: The Security of Two Phones with the Convenience of One

Temitope Oluwafemi
University of Washington

Earlence Fernandes
University of Michigan

Oriana Riva
Microsoft Research

Franziska Roesner
University of Washington

Suman Nath
Microsoft Research

Tadayoshi Kohno
University of Washington

ABSTRACT

Employers increasingly allow employees to use their personal smartphones for work, but also impose strict security policies (e.g., wiping the device after a series of failed logins), which on one hand protects the corporation's data but on the other hand can affect a user's privacy and control of her own data. To address these issues, recent proposals securely partition work and personal data by means of virtualization techniques. Yet, virtualization comes with limitations. First, unless heavily optimized, it has a significant overhead on resource-constrained phones. Second, it constrains all apps to be in the same partition at a time, while users like having a mix of work and personal apps running on the device simultaneously.

To enable this functionality, we introduce a new point in the design space. We propose *AppFork*, an Android-based platform which allows users to switch a single app from one active profile (e.g., work) to another without switching the active profile of all other apps. AppFork still achieves the security of virtualization-based approaches, but with a smaller overhead. We built a tool for automatically identifying cross-profile channels in Android apps and applied it to 14,000 apps. Supported by this analysis, we craft the problem of cross-profile isolation for Android and implement our solution to it. AppFork can be used with existing unmodified apps. We evaluate it in depth with 24 Android apps. AppFork was able to successfully run all apps and provide two isolated user profiles within each app.

1. INTRODUCTION

"Bring your own device" or BYOD is the situation in which employers allow their employees to use their own personal devices, particularly smartphones and tablets, for work purposes [13, 34]. BYOD brings significant benefits to both the company and employees, including reduced equipment costs, improved employee engagement, and the convenience of carrying one dual-use device rather than a dedicated phone for each activity. Hence, the BYOD phenomenon appears to be here to stay.

Unfortunately, by using the same device for both work and personal activities, the user and the employer expose themselves to potential security and privacy risks [7, 17, 22, 24, 31]. A company's data is now stored and transmitted using devices and networks that the employer may not control. Applications (apps) on the phone may not all be controlled by the company and, in fact, could be untrustworthy or even malicious. Users may resist company-imposed policies, like data wipes after multiple unsuccessful attempts at unlocking the phone. Users may also worry about employers being able to mine personal data stored on their device, track their activities, and delete personal data unnecessarily [10]. In short, from a purely security and privacy perspective, there are important advantages—to both employers and employees—in having employees use *separate* phones for work and personal activities.

We seek to retain the benefits of BYOD (a single phone) while mitigating these types of security and privacy concerns. Toward addressing these concerns, recent work [3, 4, 18, 19] suggests virtualization to partition a device into a business and personal workspace, such that work and personal data are isolated and can be governed by distinct security and privacy policies. We argue that these approaches are insufficient to meet users' needs. First, classical virtualization approaches come with a significant overhead for mobile platforms because they require duplicating the phone operating system. Second, even lightweight virtualization, such as Cells [3], which replicates only the middleware, does not allow users to have both partitions running at the same time—as described in §2, we verified the importance of this requirement by surveying employees of a large IT company and the vast majority of respondents reported concurrently running both work and non-work apps at least a few times a day.

This paper presents the design, implementation and evaluation of *AppFork*, an Android-based platform designed to provide the security and privacy properties of *two* (or more) dedicated phones, but with a *single* phone. (We choose Android due to its popularity, but note that other phone platforms likely exhibit similar challenges.) From a security point of view, AppFork allows users to have on the same phone both a work and a personal profile, isolated and gov-

erned by profile-specific policies. From a functionality point of view, AppFork is designed to be compatible with existing unmodified apps and to enable the novel concept of a user profile that is per-app. Unlike user accounts, *per-app profiles* let users switch a single app from one active profile to another (e.g., from "work" to "personal") without switching all other apps' active profiles.

Although previous systems have considered our security goal of profile isolation (e.g., Cells [3], MOSES [26], TrustDroid [6]), to the best of our knowledge, AppFork is the first system to meet both the security and functionality goals described above. Enabling per-app profiles is especially challenging due to the difficulties involved in isolating profiles *within* an app and *between* apps. First, in addition to using the same phone for work and personal purposes, users often use the same app for both purposes—hence the need for isolation within the app. We accomplish this by creating partitions at the file system level and minimizing the number of files duplicated across profiles, thus keeping the storage overhead small. Second, although phone platforms provide app isolation (e.g., on Android, apps have private storage folders), there are many other channels that allow data to be shared across apps and therefore across profiles (e.g., apps running under different profiles can store data in a shared public directory).

To quantify these challenges, we first build ChannelCheck, a tool that performs both static and dynamic analysis to automatically identify cross-profile channels in an Android app. We present results from running ChannelCheck on more than 14,000 Android apps. Driven by this analysis, we implement a two-part solution to either automatically block those channels via AppFork, or to notify the user, the app, or the employer of an additional potential cross-profile violation via ChannelCheck. Our AppFork implementation ran successfully on all 24+ apps we tested in depth and was able to isolate channels that could lead to data leaks. Based on our implementation experiences, we outline additional recommendations to Android (and other mobile phone platforms) that, if adopted, could lead to even more robust AppFork-like systems.

Overall, this paper makes the following contributions:

- We provide a concrete definition of the BYOD problem for modern smartphones and particularly the Android platform. We introduce the concept of per-app profiles and motivate their need via a user study.
- We create ChannelCheck, a tool for automatically identifying which channels an Android app uses to share data (and possibly leak sensitive data across profiles), and we quantify how common these channels are via an analysis of 14,000 Android apps.
- We design AppFork, which automatically blocks the most prominent cross-profile leakage channels found in our measurements. AppFork supports per-app profiles without requiring modifications to existing apps. We implement and evaluate AppFork as a modified version of Android, with a small storage overhead.
- We make recommendations to Android that, if adopted, would allow for the further integration of AppFork-like capabilities in Android.

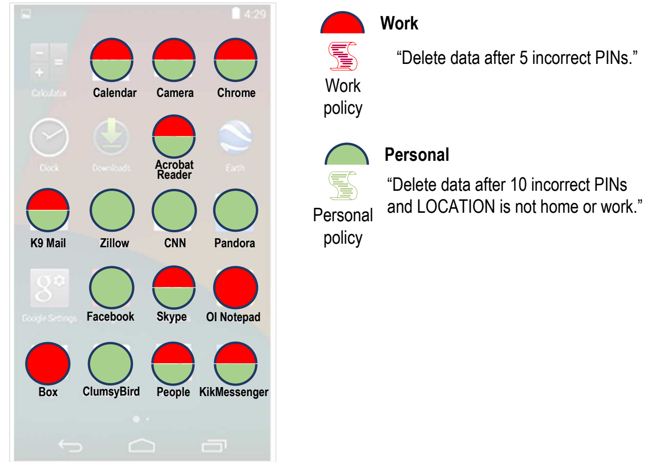

Figure 1: Examples of profiles with desired security policies.

2. MOTIVATION

AppFork is designed to enable owners of personal mobile devices, such as smartphones and tablets, to use the same physical device in different activities, such as work and personal (as in BYOD), with security and privacy properties comparable with having distinct devices for each activity.

Scenario. We start by describing a BYOD scenario. Consider a software developer working for an IT company. Some of the apps installed on his smartphone are shown in Figure 1. We highlight apps that contain sensitive information and distinguish those used for work (in red) and/or for personal (in green) activities. His employer allows him to use his personal phone at work to access corporate data via a restricted set of approved apps. In addition, for security reasons, his phone must have a PIN enabled and must implement a corporate policy wiping the phone after five consecutive incorrect PIN attempts. At work, he uses his phone mostly for emails (K9 Mail), calendar reminders, web browsing (Chrome), taking notes (OI Notepad), and accessing data in the cloud (Box). Occasionally, he uses the phone to take photos of whiteboard discussions or to chat with colleagues. However, while at work, he also uses apps without a direct work-related purpose. He uses K9 Mail not only to check work emails, but also for personal purposes. He uses Facebook and CNN for personal purposes, while other work apps are running at the same time. Though some of his apps can be uniquely associated with either work or personal activities, a significant number of apps, either for convenience or for efficiency, are multi-purpose (e.g., Calendar, Camera, K9 Mail, Acrobat Reader).

Desirable BYOD properties	One phone	Two phones	App Fork
P1. User: Low pocket weight.	✓	✗	✓
P2. Employer: Wipe all corporate data after authentication failure.	✓	✓	✓
P3. User: Employer cannot wipe personal data.	✗	✓	✓
P4. User: Two profiles active at the same time.	✗	✓	✓
P5. User: Same app can run under two profiles.	✗	✓	✓
P6. User: Same app can be *in use* in two profiles at once.	✗	✓	✗
P7. User: Protect privacy from employer.	✗	✓	✓
P8. Employer/User: Avoid network leakage between profiles.	✗	✗	✗
P9. Employer: Control all apps that access corporate data.	✗	✓	✓
P10. Employer: Monitor apps running on the corporate network.	✗	✓	✗

Table 1: Desirable BYOD properties supported by AppFork compared with state-of-the-art approaches.

Employer and user requirements. In the context of this scenario, we describe the users' and employers' requirements for a BYOD device, also summarized in Table 1. The advantages of being able to run work and personal apps on the same device (P1 in Table 1) and at the same time (P4-P6) are clear to our user, but he has two additional requirements.

First, he would like that his personal data, present in both single- and multi-purpose apps (P5), to be protected from access or deletion by his employer (P3, P7). Similarly, the employer wishes that the user's work-related data is not accidentally leaked via his use of personal apps (P8, P9). *Data separation* of this form is not a feature of modern phones. Unless apps explicitly build support for multiple isolated accounts, users cannot ensure their personal and work data are never co-mingled. This is difficult for multi-purpose apps, as well as apps that communicate via the phone platform's sharing channels (detailed in §4).

Second, as a father of two kids who often "borrow" his phone to play games, the user would like to protect his data in different ways, depending on whether the phone is at home, at work or elsewhere. For example, at home he would prefer that his personal data not be erased if his kids enter several wrong PINs (P3). Unfortunately, in current phone platforms, one policy rules all: a corporate security policy to wipe data applies to all apps and data with no distinction (P2). Instead, it should be possible to have *distinct policies* for work and personal data, to protect them according to different threat models.

From the employer's point of view, it is essential to prevent access to corporate data from malicious parties (P9) as well as ensure that personal apps cannot compromise corporate resources (e.g., a corporate wireless network). These security needs lead employers to enforce strict policies on their employees' devices (e.g., data wiping (P2)) and even monitoring apps' network activities (P10).

User survey: The need for per-app profiles. In the related work section (§8), we discuss various solutions to address the requirements above, among them virtualization approaches for separate work and personal accounts or "profiles", as we call them. In addition to requiring significant overhead (e.g., to duplicate the phone operating system or middleware), virtualization approaches (and traditional user accounts) do not allow users to simultaneously run multiple apps each associated with a different profile.

Instead, we argue that users demand ways to control their profiles at a finer granularity, on a *per-app basis*. As our scenario already illustrates, personal and work apps often need to be run simultaneously, but still be isolated from a security and privacy point of view. We verified this hypothesis with a small anonymous survey conducted in a large IT company.[1]. We surveyed 56 people (43 M and 13 F, age distribution 20–30 (8), 31–40 (26), 41–50 (18) and >50 (4)). We asked a total of 14 questions about their work and personal phone use. The survey did not explicitly mention BYOD until the very end, when we asked whether they had ever heard about BYOD (42 said "yes"). We summarize three key findings of this survey.

Dual-purpose phones regularly, not occasionally. All participants said they use their personal phone for work purposes, and 50 of 56 said that at work, they run apps needed both for work and personal purposes. At work, personal apps are used from a few times a week (28% of participants) to a few times a day (39%), to always (27%). Even more frequently, work apps are used at home, from a few times a day (38%) to always (48%). Finally, all but one participant reported having multi-purpose apps on their phone, most commonly email (90%), web browsing (87%), calendar (80%), contacts (76%), camera (67%), PDF viewer (64%) and maps (62%).

Work and personal apps running simultaneously. We then asked "How often do you happen to have work apps and non-work (personal) apps running on your phone at the same time?" Only 9% of participants answered "never" while most (44%) said "all the time". Other answers were "once or twice a day" (34%) and "5–10 times a day" (13%). These responses may represent a lower bound, as users are not always aware of background tasking. What is clear is that many users have a need to run such apps simultaneously.

Preference for per-app profiles. We asked 55 of our participants[2] "Consider the apps on your phone that you use for both work and personal purposes. Would you like if it were possible to separate the work and personal data for any of these apps, so that the work and personal data are never

[1]Our survey was reviewed by our institution's group responsible for reviewing human subject studies.

[2]55 instead of 56 because we excluded one participant who had no multi-purpose apps on his phone.

commingled?” 29 said “yes,” 15 were unsure, and 11 said “no.” The main reason given for responding “no” (5/11) was convenience. We asked participants who answered “yes” or “unsure” to select among four options for achieving such separation: “accounts” (our baseline, as users are already familiar with this option), “per-app switch,”[3] “both,” “none.” Only 4 out of 44 chose “accounts”; 22 selected “per-app switch” and 10 “both.”

Making strong claims about how representative these results are would require a large-scale user study, beyond the scope of this paper. Still, our study provides a starting point grounded in reality. It highlights the value of offering an alternative to traditional per-user accounts—an alternative that operates at the granularity of single apps. Moreover, it demonstrates the importance of multi-profile apps for users.

3. GOALS AND THREAT MODEL

Security goals. AppFork aims to achieve a primary security goal:

- *Multi-phone security and privacy comparability.* AppFork is designed to provide the security and privacy properties of two (or more) phones, but with a single phone.

A user who today might use separate work and personal phones should, with AppFork, be able to use a single phone with two *profiles*: a work profile and a personal profile.

To define our security goals for profiles more explicitly:

- *Profile isolation.* The apps and data associated with one profile should not interact, within the phone itself, with the apps and data associated with other profiles.
- *Per-profile policies.* Each profile may have its own security and privacy policies.

As an example of per-profile polices, AppFork may destroy all app data associated with the work profile after five incorrect PIN attempts (a policy some companies have today); all the app data associated with the personal profile may remain untouched.

Regarding our profile isolation goal, we note that it is impossible to guarantee isolation between apps installed on *different* phones if those apps have network access: those apps may use the network (and remote servers) to communicate (P8 in Table 1). Thus, we limit our own goal to providing isolation within the phone itself.

We additionally observe the challenge of side-channels: we cannot rule out the future discovery of novel new side-channels which may allow an app in one profile to infer information about apps in another profile. For example, researchers recently observed that background apps can use accelerometer data to infer private information about touch events in foreground apps [23]. Once discovered, it is possible to employ targeted mechanisms to mitigate known side-channels, as the authors of the above-cited paper discuss. However, since it is impossible to predict what future side-channels may arise, since the mitigation techniques may be ad hoc rather than principled and generalizable, and since side-channels are a potential concern for any single-phone solution to the BYOD problem, we view side-channel prevention as out of scope of this paper. Moreover, we observe that side-channels can also arise in two-phone scenarios, e.g., hypothesizing extensions to [14] that use the microphone on one phone to extract information from the other phone.

In addition to the above security goals, users should be able to know which profile an app is running under, and be able to control the switching of an app between profiles. While it is our goal to ensure that the user (and *not* the app) is allowed to switch apps from one profile to another, an explicit *non-goal* for our prototype implementation is the elegance and effectiveness of the user interface.

Functionality goals. We strive for a system that meets not only the above security goals, but also the following functionality goals:

- *Compatibility with existing apps.* The system should work with existing apps whenever possible, without requiring changes to their code bases.
- *Per-app profile switching.* It should be possible for the user to switch an individual app’s active profile without switching the active profile for all the apps on the phone.
- *Low storage overhead.* The storage overhead for using the system should be minimal.
- *Low switching overhead.* The overhead for switching an app’s profile should be minimal.

Achieving the per-app profile switching goal means that it is possible for a user to quickly switch (for example) his email app from his personal to his work profile, without switching Facebook out of his personal profile. An additional primary goal is to be compatible with the existing Android architecture. Nevertheless, a secondary goal is to extract insights into ways in which to modify Android (and phone platforms in general) and facilitate even stronger security properties.

Threat model. Our threat model is closely associated with our security goals above. An AppFork-equipped system has the following actors: the phone (including the OS and AppFork), the user, the profile owners, and the apps. We use *profile owners* to refer to the entities responsible for establishing the policies for different profiles. For example, the owner of the work profile might be the user’s employer, and the owner of the personal one the user himself. Apps can be installed under one or more profiles. All parties, i.e., all profile owners and the phone’s user, trust the AppFork system. This requirement is similar to the requirement today that these parties trust the phone hardware and underlying operating system.

[3] Described as “a switch attached to each app, so that you could selectively switch each app into work or personal mode. Using this solution, some apps could be running under your work profile while other apps could be running under your personal profile.”

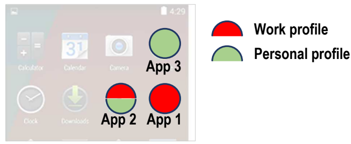

Figure 2: The owner of "work" approves App1 and App2, but does not trust App3 approved in "personal".

Our threat model is largely guided by our goal to achieve the security of two phones. Thus, we consider out of scope any threats that are also not addressed by using two phones. For example, we assume that profile owners trust device users not to be malicious. A malicious user, even with separate phones for each profile, can always maliciously leak information stored on one phone to another phone (e.g., by taking a picture of the first phone).

Similarly, if a profile owner (e.g., a business) approves the installation of an app under a profile (e.g., the work profile), then the profile owner trusts that app. To justify this trust, consider the following: if an app installed on the work profile is untrustworthy, it may be able to extract and leak information about the data of other apps installed on the work profile. But such an app could leak that information *even if* it was installed on a dedicated, work-issued phone (e.g., via the network). Hence, an AppFork profile owner must trust the apps he installs (or approves for installation) under that profile, just as he must trust the apps he would install on a dedicated phone.

A profile owner may *not*, however, trust apps installed under other profiles (P7 and P9 in Table 1). Consider the scenario depicted in Figure 2, in which App1 and App2 are installed in the work profile, and App2 and App3 are installed in the personal profile. The owner of the work profile may not trust App3. Indeed, if all three apps were installed on the same conventional (non-AppFork) phone, App3 might seek to extract and exfiltrate work-related data from App2. Hence, our profile isolation security goal above is critical: the work profile data must be isolated from App3.

In the dedicated-phone case, a profile owner who approves the installation of App1 and App2 on a phone must trust App1 and App2. However, we observe that in the dedicated-phone case, the profile owner would not evaluate the risks associated with App2 also being used in other profiles. Hence, we do *not* require the profile owner to trust that App2 will correctly handle data spanning multiple profiles; our profile isolation security goal is designed to address also this issue.

Goals in context. Other works have addressed goals similar to ours, but to our knowledge we are the first to consider all these goals simultaneously. Traditional virtualization (that duplicates the phone OS) or systems like MOSES [26] satisfy the security goals of AppFork, but not its functional goals. Lightweight virtualization, such as Cells [3], and security frameworks for domain isolation (e.g., TrustDroid [6]) also satisfy the security goals of AppFork with a smaller storage overhead, but they still do not address the functionality goals of per-app profiles and low-switching overhead.

Finally, we point out that it is not a goal of AppFork to make Android apps more secure. We assume profile owners trust the apps that they approve for their profile. We assume such an approval implies verifying that apps are not malicious (or accepting the risks) and ensuring the Android platform is used as per guidelines. We stress that this assumption is the same assumption used with today's solutions to the BYOD problem as well; if a large IT company approves the use of applications for employees' mobile phones, and if one of those applications proves to be untrustworthy, then that application can cause harm or leak data.

4. CROSS-PROFILE ISOLATION

AppFork seeks to enable per-app profiles in unmodified existing apps. Because we wish to allow a single app to support and switch between multiple profiles, we face two design challenges: (1) isolating profiles *within* an app, and (2) isolating profiles *between* apps. For example, K9 Mail when running under the work profile should not—even accidentally—leak work data to the K9 Mail personal profile, or to any other application running in personal mode.

We designed AppFork for Android due to its popularity. Here, we thus analyze in more detail why and how the above challenges arise in Android. However, other phone platforms likely exhibit similar challenges. In the following, we survey available cross-profile communication channels on Android. We then describe ChannelCheck, a tool we built for automatically detecting the use of these channels in Android apps, and the results obtained by applying it to 14,234 contemporary Android applications. We conclude with an overview of AppFork's approach to cross-profile isolation.

4.1 Android Background

Android isolates apps from each other by running them with separate Linux user identifiers (UIDs), and restricts their access to system resources and other apps by requiring that they request specific permissions at installation time. Further, an app's files are stored in a private folder accessible only to that app (unless the app's UID is shared with other apps). Thus, stock Android already provides some level of isolation. However, there are still many ways in which cross-profile communication can occur.

Explicit communication channels. Figure 3 summarizes key channels in which Android apps might share data. As discussed in §3, we do not consider network communication since the network is also available to apps running on two separate phones. Channels of type 1 usually apply to individual apps that span multiple profiles (*within-app channels*). In reality, they can also allow communication between different apps that share the same UID, but, as shown later, this is relatively rare. Channels of type 2–6 apply both to single apps running in multiple profiles (*within-app channels*) and multiple apps running in different profiles (*between-apps chan-*

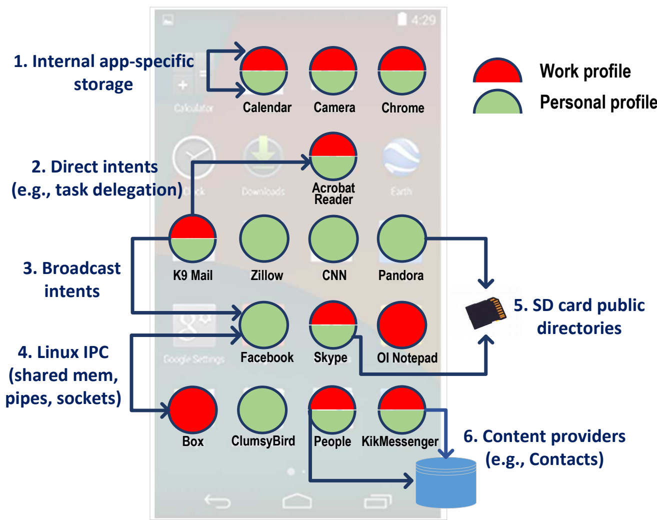

Figure 3: Cross-profile data sharing channels in Android (network and side-channels are not shown).

nels). We elaborate on these channels below.

Inter-Component Communication. Android apps consist of components, including Activities, Services, Content Providers and BroadcastReceivers. Components can communicate in many ways:

- *Direct intents*: One Activity or Service can launch another using a direct Android Intent. Intents can be used for task delegation, e.g., K9 Mail uses an Intent to launch Acrobat Reader to open an attachment. They can also be used to set up communication sessions between components, i.e., by binding to a Service, which can expose an AIDL (Android Interface Definition Language) interface.
- *Broadcast intents*: Apps may also send and receive Broadcast Intents. Broadcasts may originate from the system (e.g., notifying the device's screen is off) or from apps, and they are delivered to each registered receiver.
- *Content providers*: Content providers handle shared sets of data like SMSs or contacts. Apps can use built-in content providers or expose their own custom content providers. Two apps (or two profiles of the same app) can communicate by one writing to a content provider and the other reading from it.

External storage. Apps can share data by writing to world-readable locations on external storage (the SD card). Prior to Android 4.4, all files on external storage were accessible to any app with the READ_EXTERNAL_STORAGE permission. Starting with Android 4.4, external storage is structured like internal storage, using app-specific directories accessible only to that app. However, apps in Android 4.4 can still share data via the SD card through public shared directories, such as Music/.

Linux IPC. Apps can also communicate via standard Linux inter-process communication methods. Android offers Java APIs for Linux IPC (android.os.MemoryFile and android.net.LocalSocket) in addition to native Linux IPC. As shown later, these methods are rarely used in Android apps, which can instead achieve the same goals using more efficient ICC methods.

Side-channel capability. As discussed in §3, we focus on explicit data sharing channels and not side-channels. Nevertheless, we consider such possible attacks. Prior work has demonstrated that *System Services*, such as SensorService, WiFiManager or AudioManager, can be used as covert channels [30]. In addition, as we noted in §3 for accelerometer data, sensors can be used by an application to acquire information about another [23].

4.2 ChannelCheck: Detecting Channel Use

Above, we analyzed the possible cross-profile communication channels available to Android applications. To better understand how frequently these channels are used in real apps, and thus to ultimately inform our design and implementation of AppFork, we built a tool to detect their use in existing apps. This tool, *ChannelCheck*, performs static and dynamic analysis on an app's binary in order to identify whether and how the app uses all channels described above. ChannelCheck detects the use of both explicit communication channels and side-channels, but excludes the network channel for reasons given above.

ChannelCheck's static analysis involves processing every method call site and looking for the presence of Java APIs related to system services, Java bindings to Linux IPC, built-in content providers, custom content providers, access to SD card, and sensor services.[4] Moreover, ChannelCheck reports whether an app shares the same UID with any app of a given set of apps, implying the app's data is shared with those apps.

For dynamic analysis, ChannelCheck executes a given app and traces, using a kernel mode tracer, calls to APIs for exchange and broadcast of intents, filesystem accesses to data stored on external storage, and Linux IPC attempts (including sockets, pipes and Android custom shared memory[5]). The kernel mode tracer is context-sensitive, i.e. the tracer detects when the app is executing its own native code, and switches on tracing automatically. This reduces Linux IPC false positives arising from support libraries.

To automate the dynamic analysis (and be able to run it for many thousand apps), ChannelCheck relies on an existing

[4]For content providers, ChannelCheck detects use of methods for adding or retrieving data from built-in content providers (as specified in the android.provider package) and use of APIs that must be implemented by custom content providers. For sensors, it detects use of the getSystemService() method classified based on the SENSOR_SERVICE specified in the argument, for all 13 sensor types available in Android.

[5]More precisely, we trace system calls for socket (unixdomain), connect (unixdomain), bind (unixdomain), pipe, pipe2, msgget, semget, shmget, ioctl (ashmem), mknod (fifo) and mknodat (fifo).

Explicit communication channel	Num of apps	AppFork
Direct intents	21.1% (2994)	✓
Broadcast intents	14.8% (2100)	✓
Built-in content providers	14.3% (2005)	✓
Custom content providers	8.7% (1222)	×
SDcard access	49.3% (6937)	
SDcard - only app-specific paths	8.4% (1183)	✓
SDcard - public shared directory	31.3% (4399)	×
Linux IPC	0.5% (69)	×
Shared UIDs	0.9% (120)	×
None	44.1% (6200)	✓

Table 2: Use of *explicit* cross-profile communication channels in existing Android apps, based on static and dynamic analysis of 14,067 apps. The rightmost column indicates whether AppFork can automatically prevent cross-profile leakage via that channel, as described in §5.

Side-channel capability	Num of apps
System services	56.2% (7899)
Sensors - accelerometer	2.6% (363)
Sensors - gyro	0.1% (13)
Sensors - light	0.1% (17)
Sensors - magnetic field	0.2% (22)
Sensors - orientation	0.4% (60)
Sensors - proximity	0.4% (63)
None	43.8% (6167)

Table 3: Presence of side-channel capabilities in existing Android apps, based on static and dynamic analysis of 14,067 Android apps. Note that an app's use of a potential side channel does not necessarily mean that it is using that channel to actually leak information; in other words, the numbers in this table present an upper bound on the risk of side-channels. For sensors, we report results only for those that were used in at least 0.1% of apps.

automation framework called PUMA [15] that automatically runs an app and navigates to various pages of the app by emulating user interaction (e.g., by clicking a button, swiping a page, etc.). PUMA can be configured to explore all structurally distinct pages in an app, with a certain timeout (or a bound on the number of app interactions to perform).

4.3 Measurement Study

To assess the relative risk posed by the channels described above we used ChannelCheck with 14,234 Android apps. Apps were crawled from the Google Play Store during the third week of December 2013, and were listed as the 500 most popular free apps in each category provided by the store. We configured PUMA to explore all pages in an app with a timeout of 3 minutes. The tool successfully ran with 14,067 apps. Static analysis failed for 146 apps (1.0% failure rate) due to APK decode and unzip errors, and dynamic analysis failed for 19 apps (0.1% failure rate) due to Android or PUMA crashes.

Table 2 shows the fraction of apps communicating with other apps through one of the explicit cross-profile channels. A large fraction (49%) of the apps use external storage. 8% of the apps access external storage only at app-specific paths, but we expect with the changes introduced in Android 4.4 (see §4.1) more apps will use app-specific paths in the future. On the other hand, we found 31% of the apps using public shared directories on the SD card. Intents are also largely used with 21% of the apps using direct intents and 15% broadcast intents. Access to content providers is also relatively common: 14% of the apps use built-in content providers, with roughly half of them accessing at least one of the three most popular content providers: Settings, Contacts and Calendar. 9% of the apps use custom content providers. Linux IPC was found only in 0.5% of the apps (in fact developers are encouraged to use instead more efficient ICC methods provided by the framework), and shared UIDs were present in less than 1% of the apps. Finally, 44% of the apps do not use any of these communication channels.

Table 3 reports on the frequency with which applications use functionality that may be used as side-channels. These numbers represent an upper bound on the side-channel risk through these capabilities: the use of such a feature (e.g., system services or sensors) does not necessarily mean that the app is using this channel to leak cross-profile information. In fact, in the majority of cases, we expect that this is not the case; nevertheless, we report these numbers for completeness. We find that 44% of the apps do not use any functionality associated with a known side-channel.

We found that the numbers in Tables 2-3 are roughly the same if we consider only the apps in Business and Communication categories, which are the two most relevant categories for BYOD. The only major difference is the larger adoption of content providers and system services: 37.6% of these apps use built-in content providers, 14.5% use custom content providers, and 66.5% use system services.

These measurements help inform and validate our design of AppFork, discussed below. However, we also believe that these results are of independent interest to researchers studying the BYOD problem or other Android topics.

4.4 AppFork Overview & Deployment Model

The results above show that about half of the tested apps make use of several cross-profile channels. Without taking these channels into account, cross-profile data leakages can easily occur. To prevent such leakages, we propose a deployment model that combines the use of AppFork, which blocks many of these channels, and ChannelCheck, which can be used to verify the absence of any channels not explicitly handled by AppFork.

AppFork prevents cross-profile leaks via many of the channels described above, as we detail in the next section (§5) and summarized in Table 2. In particular, AppFork focuses on the most prominent channels detected in our measurements: it automatically blocks cross-profile communication using direct and broadcast intents, built-in content providers, and app-specific folders on external storage.

Recall from our threat model that corporations (and profile owners in general) must trust apps that they install for a given profile. AppFork helps prevent leakages through the most prominent communication channels, but additional, less frequently used communication channels remain. To verify that an app can be trusted for a given profile, the profile owner can run ChannelCheck to check for the presence of any cross-profile channel not handled explicitly by AppFork. In particular, an employer should reject apps that could leak data through public shared directories, custom content providers, or improperly configured permissions on Linux sockets or pipes. Employers can also choose to reject apps that use potential side-channels, or simply create per-profile policies to blacklist sensitive services and sensors, which are enforced by AppFork (see §5.5 for an example).

Thus, in addition to serving as a measurement tool, ChannelCheck provides employers with an easy and automatic way to decide whether a third-party application should be approved for use by its employees for work purposes. In the next section, we describe how AppFork achieves profile isolation for the channels considered.

5. DESIGN AND IMPLEMENTATION

We describe our AppFork design. As subtle yet important issues can arise when applying generic designs to real, complex systems, we also detail specific challenges with our implementation.

5.1 AppFork Overview

We designed AppFork with the goal of supporting existing apps with no modifications. We built AppFork on Android by modifying the Android Application Framework and provided an app for end users to manage their profiles.

Figure 4 shows the AppFork architecture. AppFork is implemented as an Android *system service*. This service, which starts when the phone boots, has three main components: the *Profile Manager* for managing user profiles and enforcing storage isolation, the *Cross Profile Filter* for preventing apps from leaking data across profiles, and the *Policy Enforcer* for implementing the specifications of all profile policies, by monitoring policy conditions through *Monitors* and executing policy actions through *Actuators*.

Peeking into details of our implementation, we restrict access to the AppFork service to processes with uid set to android.uid.system. This means that third-party apps cannot access this service and, for instance, arbitrarily change an app's profile. Instead, only first-party apps, if granted system privileges, can invoke the AppFork service API. We sign our AppFork app with the platform key of our Android custom build so that it inherits system privileges.

5.2 Per-App Profiles

In AppFork, a profile consists of a policy and a set of apps that are allowed to run under the profile. An app can be associated with multiple profiles. AppFork profiles are different

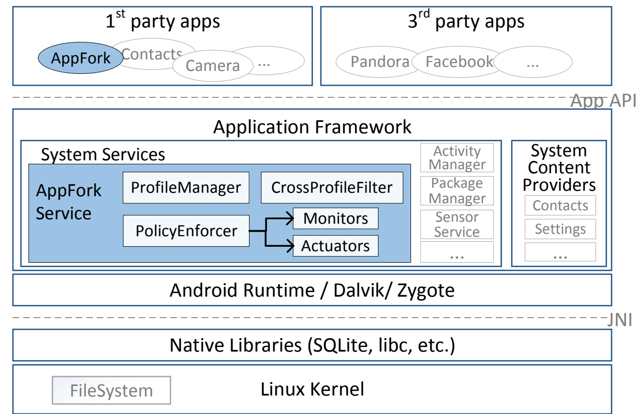

Figure 4: AppFork's architecture.

from user accounts because they are activated/deactivated on a per-app basis.

The profile owner determines the policy and the set of apps allowed in that profile. AppFork stores this information in the file system (with access restricted to system processes). A profile owner may be the owner of the device or an external actor, such as an employer. In the latter case, the employer determines the policy and provides a list of pre-approved apps that the device owner can selectively install. As discussed in §3, a profile owner approves apps for that profile, and AppFork trusts those apps within the scope of that profile (i.e., it is not AppFork's responsibility to block malicious apps approved by an employer); this is akin to trusting the work-installed apps on a work-issued phone. A profile's policy applies only to the apps in that profile. For instance, a work policy like the one in Figure 3 wipes only work-related data and apps after five consecutive failed logins.

Each time a user changes an app's active profile, AppFork checks whether the app is currently running and, if so, stops the app and all associated background processes. The app's profile is switched (more details on this below) and the app is started in the new profile.

5.3 Profile Partitions

AppFork maintains a separate storage partition for each profile, ensuring only data belonging to that profile is stored in that partition and is not accessible to other profiles.

Files saved to the internal storage are by default private to the app[6], and stored at the path /data/data/<packagename>. When the app is first installed, AppFork creates a partition for a "default" profile by moving the content of the app's original folder to /data/data/<packagename>-default and creating a symbolic link with a path of /data/data/<packagename> pointing to this folder. Suppose this app is added to both the "work" and "personal" profiles. The first time the app is switched into one of the profiles, AppFork dynam-

[6]Our analysis in §4 showed that very few apps have shared UIDs so we can make this assumption without loss of generality.

ically creates a storage partition for each profile, at location /data/data/<packagename>-work if the profile is "work" or /data/data/<packagename>-personal if the profile is "personal". These newly created folders have the same structure as the "default" profile's folder, except that AppFork creates symbolic links in them to point to the "default" lib subfolder located at /data/data/<packagename>-default/lib, which contains the app's precompiled libraries. With symbolic links, the lib folder is never replicated, minimizing AppFork's storage overhead.

When the user starts or switches an app into a given profile, AppFork creates a symbolic link in the original app folder /data/data/<packagename> pointing to the partition of the active profile. The file system permissions are set so that the folder of the active profile is accessible by processes with the app's uid, while folders containing inactive profiles have android.uid.system permissions. Thus, an approved app running in profile "personal" cannot maliciously access files within the "work" partition, even if it is aware of the symbolic link switch. This approach provides isolation for all file system operations, which includes apps' SQLite databases, since they are stored in the same app-specific folders.

Note that our approach based on symbolic links can possibly generate many replicated files across profiles. A more advanced solution is to use a copy-on-write approach in which symbolic links are maintained for previously unmodified or static files resident in their app's original folder. Files are copied into the appropriate partition only if a write is scheduled from any of the profiles. This solution, however, increases the implementation complexity and potentially the processing overhead, as it requires keeping track of all write operations. In the evaluation, we compare AppFork's storage overhead against this optimized implementation.

Android apps with the READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE permissions can read or write files in external storage (the SD card). Files saved here are world-readable, so accessible to any app with such permissions. Our solution to provide isolation for external storage builds on two observations. First, as per Android guidelines, external storage offers minimal protection for stored data, hence apps should not store sensitive data here, but instead in the app-private directories which can be effectively protected. Second, starting with Android 4.4, external storage is structured like internal storage, with package-specific directories such that apps can access their private partitions (e.g., /sdcard/Android/data/<packagename>) without holding the broad EXTERNAL_STORAGE permission.

We assume apps will follow the above guideline of using app-specific directories on external storage and not request the EXTERNAL_STORAGE permission. Then, AppFork can use a partitioning approach similar to that of internal storage, except that symbolic links cannot be created in external storage due to the vfat partition. At a profile switch, AppFork changes the name of the resident folder, /sdcard/Android/data/<packagename>, used in the previous profile to either /sdcard/Android/data/<packagename>-work or /sdcard/Android/data/<packagename>-personal, depending on the profile. However, AppFork does not isolate profile data stored in *shared public directories* on external storage, such as Music, Pictures, and Ringtones. Since these directories can be essential for some apps (e.g., to avoid a huge storage overhead or to simplify their syncing strategy), we do allow them. It is up to profile owners (e.g., corporations) to verify that approved applications comply with this policy and the Android guidelines for sensitive data. Recall AppFork's goal of approximating two phones with a single phone—since external storage is world-readable, profile owners may not approve an application violating these guidelines for sensitive data even on a dedicated phone for each profile. ChannelCheck logs how an app uses external storage and can support this assessment.

5.4 Cross-Profile Isolation

Android apps can share data across profiles in several ways (see §4). Table 2 summarizes which channels AppFork automatically blocks: we address potential leakages through ICC (discussed below), internal and external storage (discussed above), and built-in content providers (discussed below); we do not address the use of Linux IPC since it is rarely used in Android apps and the same functionality can be achieved with more lightweight ICC, and also because ChannelCheck can identify applications that use IPC; we do not address the use of networks to leak data across profiles as this is also possible with the use of two dedicated phones (see §3).

Direct intents. Android apps are allowed to start other apps or services through respective calls to either startActivity or startService(bindService). Additionally, Android allows apps to delegate tasks to other apps through calls to startActivityForResult. These features are facilitated by Android's Intent class. While useful, these feature pose security and privacy risks at odds with AppFork's goal of cross-profile isolation. For example, we found that a Book Catalogue app delegates scanning of barcodes to a Barcode Scanner app. If Book Catalogue runs under one profile, it may leak information to the Barcode Scanner, which might maintain a record of all scanned barcodes irrespective of Book Catalogue's current profile.

A solution to the potential cross-profile leakage threat in the case of task delegation is non-trivial. Consider the example shown in Figure 1, in which the user runs K9 Mail and AcrobatReader, both allowed in both profiles. K9 Mail may delegate opening an attachment to AcrobatReader, currently running under the other profile. To prevent data leakage across profiles, we envision at least four options which can be implemented at the system level: (1) The calling app cannot arbitrarily force another app to switch its current profile, so it has to wait for a timeout to expire or for the needed app to end; (2) the calling app has the right to force the called app to switch profile such that it can be used immedi-

ately; (3) the called app switches profile only after the user is prompted with a dialog and approves the switch; (4) the request of the calling app is rejected and a SecurityException is thrown or a friendlier failed status is returned to the calling app.

The first three options can lead to a denial-of-service attack: a malicious app running in the background may continuously invoke AcrobatReader and prevent other profiles from using the app. Even in the case of the third option, the user, unaware of what is happening, may keep approving the profile switch. Another drawback of the second approach is that it is not immediately clear what profile should be given precedence, and a drawback of the third approach is that dialogs are disruptive to users.

For these reasons, we argue that this class of conflict is better resolved by taking the apps' semantics into account. In fact, whether the profile switch should be automatically authorized depends on how trusted the apps are (e.g., first-party apps may be able to force a switch) and on the type of task (e.g., another app for viewing PDF files may be available for use instead). Our current solution is based on the fourth approach described above, in which the calling app receives a SecurityException thrown by AppFork from within the startActivityLocked member function of the ActivityStack class. This approach builds on the assumption that apps that delegate tasks to other apps should already be prepared to handle such exceptions, in the event that the needed app is unavailable.

For unresolved intents that result in Android's "chooser" activity, we modified the onCreate and rebuildList functions of the ResolverActivity and ResolveListAdapter classes respectively, to only display apps approved under the active profile of the intent creator.

Finally, we also ensure that app components cannot bind to services running under different profiles, with the exception of critical system services (e.g., Location Manager, Account Manager, Power Manager). This is implemented by intercepting calls to startServiceLocked and bindService of ActivityManagerService, where we deny requests to start or bind to services across profile boundaries.

Broadcast intents. Android apps may also send broadcast intents, which are delivered to all subscribed receivers (possibly subject to a predefined permission). Cross-profile data leakage can happen if a trusted app in one profile sends sensitive information to receivers in apps under a different profile. A data leak can even occur through subscriber registration because upon successful registration, the last available sticky broadcast is automatically sent to the new broadcast receiver. AppFork resolves this potential threat by filtering out registered or registering receivers with active profiles that are different from the one of the sending app. Specifically, we modified the broadcastIntentLocked and registerReceiver member functions of the ActivityManagerService class.

Content providers. Android apps can also share data through built-in and custom content providers. For built-in content providers like the Contacts provider, AppFork enforces a logical partition of the provider's database. Specifically, we modified the getDatabaseLocked API of the SQLiteOpenHelper class to fork and control access to the appropriate databases for each profile. At a profile switch, AppFork forces a switch of the database to the one belonging to the active profile, for any database function specified in the ContentProvider class.

For custom content providers, we cannot modify the corresponding ContentProvider classes (per our requirement of supporting unmodified apps) so we take a different approach. AppFork checks whether the calling app and the owner of the custom content provider are within the same profile (this happens by instrumenting the acquireProvider and acquireExistingProvider APIs of the ActivityThread class). If they belong to different profiles, a null reference is returned. Otherwise, the requested provider is returned. This solution provides isolation at the cost of making custom providers available only in one profile at the time.[7]

5.5 Policy Specification and Enforcement

Profiles are specified in XML and stored on the device. They are currently not encrypted, but could be in the future. We provide a simple template that can be extended as more policy constructors are introduced. Each profile specification consists of two parts: a list of packages approved for use under that profile and a policy. Each policy consists of one or more conditions to be monitored and one or more actions to be executed if those conditions are detected. Figure 5 shows an example of a "work" and of a "personal" profile. The "work" policy specifies that after five consecutive failed logins, data belonging to that profile must be wiped; the "personal" policy specifies that apps requesting the TYPE_ACCELEROMETER resource should be denied access.

Each time the phone boots or new profiles are created, the policy specifications are parsed. The map of all supported apps approved under the profile is stored in memory, and each policy is translated into subscriptions to policy monitors. As an example, we describe the monitors and actuators we implemented for the policies shown in Figure 5.

For the "work" policy, we implemented a *Password Monitor* that keeps track of incorrect password entries. We modified the reportFailedPasswordAttempt() method of the DevicePolicyManagerService class to send a sticky broadcast with information about the number of incorrect password entries each time a wrong password is entered. If the maximum number of wrong attempts is reached, the *Wipe Off Actuator* is invoked to erase all profile data. Once the profile has been deleted, a notification is sent to the other components and the AppFork app to reflect the changes.

[7] Because the support for custom content providers comes with this limitation, in Table 2 we list them as *not* supported by AppFork.

```
<profile name="work">
  <packages>
   <approved name="com.google.android.gm">
   <approved name="com.mobisystems.office">
  </packages>
  <policy>
   <sensor name="failed-login" maxOccurs="5"/>
   <actuator name="wipe-profile"/>
  </policy>
 </profile>
 <profile name="personal">
  <packages>
   <approved name="com.facebook.katana">
   <approved name="com.google.android.gm">
  </packages>
  <policy>
   <sensor name="access-resource" value="
        TYPE_ACCELEROMETER"/>
   <actuator name="block-access"/>
  </policy>
 </profile>
```

Figure 5: Examples of AppFork profile specifications.

For the "personal" policy, we implemented a *Blacklisted Resources Monitor* that keeps track of apps' requests for device resources, particularly sensors such as proximity, accelerometers, and light. We modified the ContextImpl, SensorManager and SystemSensorManager classes to monitor app's access to device sensors. If access is requested, the *Resource Block Actuator* grants or denies access depending on the policy. For simplicity, if access has to be denied it filters out the app's subscriptions for sensor readings. This approach prevents apps from crashing as opposed to if their requests were outrightly rejected. We envision other more advanced implementations where the sensor readings could be returned but in an obfuscated or generalized manner as proposed in [16].

Policy Enforcer is designed in a modular fashion such that new monitors and actuators can be easily plugged in. Additional monitors can cover important contextual information, such as home or work location, battery level, or WiFi network information. Additional actuators can provide a more comprehensive set of actions, such as blocking network traffic, switching network radio, and backing up data to the cloud. For instance, we envision policies such as "if at work and using the corporate WiFi network, use the cellular network for transmitting 'personal' data" or "block apps from communicating with blacklisted network domains".

6. EVALUATION

We discussed the security analysis of our system inline with its design in §5. In this section, we focus our evaluation on three goals: (1) AppFork is able to support unmodified Android apps; (2) AppFork's storage overhead is small compared to state-of-the-art solutions; and (3) the time required for switching an app from one profile to another is small enough to not impact app usability. We tested AppFork on a Samsung Nexus S phone running our custom build of Android 4.1.2. AppFork was configured with two profiles, "work" and "personal", with associated policies (see Figure 5).

6.1 Support of Unmodified Apps

Referring to the results of the app analysis described in §4 (Table 2), by counting the number of apps that make use of the channels explicitly handled by AppFork, we estimate AppFork being able to achieve automatic profile isolation for 65% (9089) of the 14,067 apps we tested using ChannelCheck. For the rest of the apps, ChannelCheck is provided to help profile owners (e.g., corporations) evaluate the possible leakage channels not explicitly handled by AppFork, particularly public shared directories and custom content providers.

We also tested AppFork in depth with a smaller set of 24 apps.[8] We selected 21 free apps based on popularity and functionality.[9] We then added 3 more apps (Box, K9 Mail, OI Notepad) because their functionality makes them popular in the BYOD context (e.g., email was the most popular multi-profile app identified by our survey respondents). Thus, we report our evaluation results on these 24 apps.

Methodology. We installed each app and added it to the two profiles. First, we verified that each app could be switched successfully from one profile to the other via AppFork. Second, we interacted with each app for 3 minutes in each profile, in a manner consistent with the app's functionality.[10] For example, interacting with K9 Mail involved syncing the inbox, composing an email, sending it, checking the sent folder, and returning to the inbox. For Facebook, it involved posting a status update, visiting a friend's page, and returning to the status update page. During these tests we also explicitly initiated actions that triggered task delegation occurrences. For example, in testing YouVersion Bible and Zillow, we performed activities that resulted in starting the browser to visit embedded URLs. For apps requiring an account, such as Facebook, K9 Mail, Netflix, and Skype, we created and populated dummy accounts.

We then verified that a partition was created for the app's data under each profile and that no data was shared across profiles via the channels described in §4. For file system and external storage, we inspected the files created under each profile. For content providers, we inspected the corresponding databases. For broadcast and direct intents, we inspected the execution logs and verified such operations were confined to a profile.

Results. Table 4 lists the tested apps. All 24 apps worked

[8] During development AppFork was successfully tested with a total of 35 apps including AngryBirds, Amazon and Marvel Comics.

[9] We took the top 13 free apps in the U.S. view of the Google Play Market and the top app in each of the top 15 app categories [2] (as of February 12, 2014). We excluded from our evaluation 3 apps in the Personalization, Tools, Arcade & Action categories, respectively, because they provide services that are not sensibly associated with a unique user profile. This gave us a total of 21 apps (not 25, because there were overlaps between the two selection criteria).

[10] We experimented also with longer interaction times and found 3 minutes of targeted and continuous activity to be sufficient for triggering an app's cross-profile channels.

App	Category	Support F	Support M	Possible Leak
Candy Crush Saga	Games	✓		
Clumsy Bird	Games	✓		
CNN	News & Magaz.	✓		
Duolingo	Education	✓		
Facebook	Social	✓		
FacebookMessenger	Social& Local	✓		
Google Earth	Travel	✓		
Guess the 90's	Brain & Puzzle	✓		
Indeed Jobs	Business	✓		
Instagram	Social		✓	SDcard
Ironpants	Arcade & Action	✓		
Kik Messenger	Communication		✓	SDcard
Myfitnesspal	Health & Fitness	✓		
NBCSportsLiveExtra	Sports		✓	SDcard
Netflix	Entertainment	✓		
Pandora	Music & Audio	✓		
Skype	Communication	✓		
SnapChat	Social	✓		
Unroll Me	Brain & Puzzle	✓		
YouVersion Bible	Books & Refer.	✓		
Zillow	Lifestyle	✓		
Box	Business	✓		
K9 Mail	Communication	✓		
OI Notepad	Productivity	✓		

Table 4: Apps tested on AppFork. 'F' means the app is automatically fully isolated and 'M' means the app is isolated but requires manual evaluation for a specific channel.

seamlessly on AppFork without any modification. For 21 apps out of 24, AppFork was able to "automatically" block any possible cross-profile data channel, so we considered them "fully isolated" (hence 'F' in the table). For 3 apps, also identified by ChannelCheck, although correctly running, AppFork detected they used folders on the SD card outside of their app-specific directories. Kik Messenger created a public folder but did not store any data in it while NBC stored temporary files in a folder outside its designated app directory. Instagram stored pictures and videos taken while using the app, in the shared "Pictures" and "Videos" directories. As discussed in §5, today AppFork has no means to automatically prevent such occurrences, and it is not intended to do so because, for non-sensitive data shared directories are a legitimate channel to share data. For this reason, AppFork defers the treatment of these cases to a manual testing of the app by the profile owner (hence 'M' in the table).

6.2 Storage Overhead

AppFork may increase the apps' storage overhead because its symbolic link-based implementation may cause duplicated files.

Methodology. We measure the storage space occupied by AppFork with two profiles and compare it to that of a baseline system, such as a virtualization-based system like Cells [3]. We ran each app on AppFork and interacted with

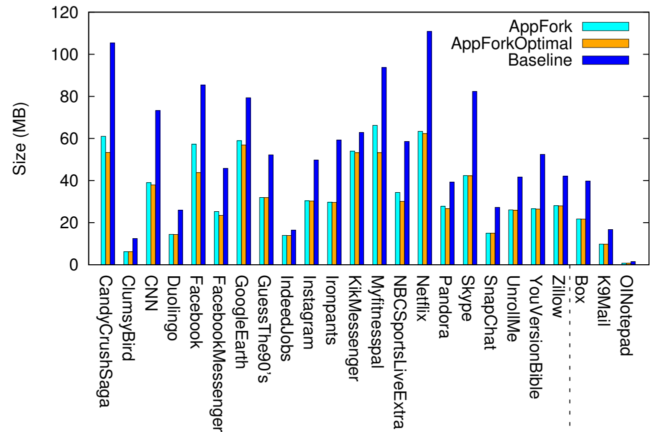

Figure 6: AppFork's storage overhead. Comparison between Baseline, AppFork and an ideal version of AppFork (called *AppForkOptimal*) that eliminates replicated files.

it for 3 minutes in each profile. We then ran the same app for the same duration and repeated the same actions on a clean build of Android. For both builds, we measured the amount of data stored in each profile under the app directories in internal and external storage. This includes the app's state and files as well as APKs and dex files stored in the /data/app and /data/dalvik-cache directories, respectively. To estimate the baseline overhead, we double the total size of APK and dex files on the clean build, since the app would need to be duplicated across VMs.

Results. Figure 6 shows the storage overhead. In addition to the results for AppFork and the device simulating the baseline, we report *AppForkOptimal*, which represents the ideal case of a (hypothetical) AppFork implementation that tracks and eliminates redundant files across profiles. We identify files duplicated across profiles by comparing the hashes of similarly named files.

AppFork reduces the storage overhead compared to the baseline by an average of about 36%. For most apps (e.g., Skype, Ironpants, CNN), the baseline requires almost double the storage because these apps maintain little state compared to the app's installation files. On the other hand, for apps like Facebook or IndeedJobs, the app's state is larger and the gap between baseline and AppFork is smaller. AppFork also performed well compared to *AppForkOptimal*. On average, the extra overhead introduced by our unoptimized implementation is 7%, not significant enough to justify the complexity of the *AppForkOptimal* implementation (see §5.3).

6.3 Profile Switching Overhead

Due to its design, AppFork's processing overhead for isolating profiles is negligible. In fact, switching symbolic links and filtering cross-profile operations (broadcast or direct intents) introduces negligible delays; also policy monitors listen to events already being broadcast. On the other hand,

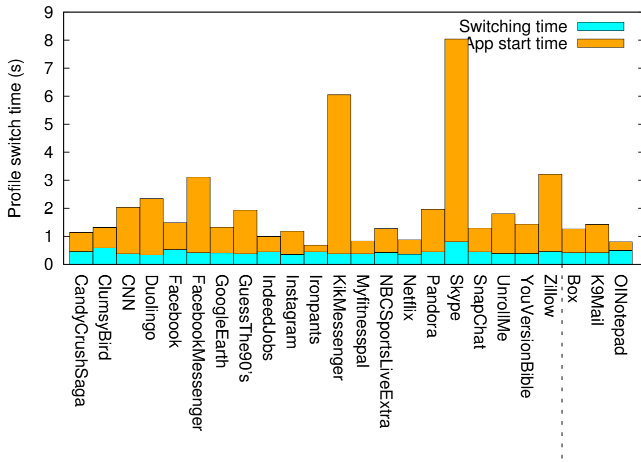

Figure 7: Time to switch an app between profiles.

while usability is not a goal of our implementation or evaluation, the time necessary for switching apps between profiles may be high for some use cases.

Methodology. We measure the time AppFork requires to stop an app, switch its profile, and reload it. For each app, we perform the switch 5 times and take an average.

Results. Figure 7 reports the switching time (including the time for terminating the app) and the app start time. The switching time is small, on average below 0.44 seconds. The time for starting an app (after being stopped) is unaffected by AppFork and, of course, varies by app; most apps took about 1.5 seconds, Skype over 7 seconds. Overall, the switching time is acceptable: it only slightly increases the time users already have to wait for apps to start.

We conclude with some observations about the AppFork user experience. In fact, one of us, who was not a developer of AppFork, used AppFork with his real work and personal accounts. He found AppFork to be functional and easy to use. His feedback solely had to do with the user interface which we stated as a non-goal. During his testing, the author accidentally entered his PIN incorrectly too many times in a row, thereby wiping his work profile. The author appreciated not having his personal profile data wiped—a direct but unintentional example of the benefits of the AppFork design, and the ability to separately enforce work and personal policies.

7. REFLECTIONS

Our study finds that it is feasible to offer the convenience of one phone, but the security properties of two phones, in a way that is compatible with many unmodified Android applications. There are, however, some limitations of AppFork, including corner cases that AppFork does not handle. These unhandled corner cases are often due to limitations in the underling Android platform and APIs. We begin this section with a broad discussion of several key issues related to our current AppFork prototype and the general design. We then reflect on how the Android platform might evolve to enable AppFork to overcome the limitations discussed herein.

7.1 Discussion

Multi-profile concurrent background apps. In AppFork, an app in one profile remains in that profile until the user explicitly switches it. Background processes associated with that app also run only in the current profile. Automatically switching the profiles of apps running in the background raises multiple questions, including: how often should the switch happen? How can AppFork avoid switching the app at times not expected by the user? To maximize profile awareness and user control, we chose not to support such switching in our current implementation.

Profile/app approval and verification. We expect remote profile owners (like corporations) to sign their profile-specific policies, which include the package names of apps authorized under that profile, with company credentials whose counterparts are installed on the device (e.g., with the help of the company's IT support staff). AppFork also supports the creation of local policies on the device.

Profile switching burden. Our threat model places the responsibility on users to switch between profiles. Future work should study the usability of manual profile switching and other approaches. For example, instead of relying on explicit user actions, profile changes could be suggested or made automatically based on context [26].

App versions. AppFork cannot run different versions of an app under different profiles, which may pose a problem if different profile owners have approved different versions of the same app.

Exposing app weaknesses. AppFork may expose existing flaws in Android apps that do not fail gracefully when denied permissions to start components in other packages. We argue that apps that depend on others to perform certain tasks should also anticipate the likelihood that those components may not be installed.

7.2 Recommended Android Enhancements

Profile-aware content providers. Due to Android's implementation of custom providers, AppFork only allows one profile's instance of a custom content provider to be active at a time. This poses problems when multiple apps running in different profiles wish to use the same content provider. To address this issue, we recommend that Android's custom content provider support be reimplemented to be profile-aware by providing ways to switch underlying database connections of derived content provider classes.

Profile-aware system services. To prevent potential side-channel attacks from queries to critical system services, system services should also be designed to be profile-aware by

detecting active profiles of clients and enforcing data isolation between profiles.

Support multiple phone instances. AppFork does not support profile isolation for telephony-related apps (e.g,. Phone, SMS) which would require the phone to have multiple phone numbers. To provide full cross-profile isolation, we thus suggest that mobile platforms intended for BYOD scenarios support multiple phone instances, either physically (with two SIM cards, which is a common feature on phones in some countries) or virtually (with multiple International Mobile Subscriber Identities (IMSIs) on a single SIM card, as in [4], or with VoIP support, as in Cells [3]).

Integrating content from multiple profiles. Users may wish to view content from multiple profiles simultaneously. For example, in a calendar app, users may want to see both their work and personal events at once. To display isolated content from two profiles within the same user interface, we recommend that Android adopt techniques from prior work on user interface isolation [25].

Linux IPC restrictions. One of the possible cross-profile leakage channel not prevented by AppFork is standard Linux IPC, such as local sockets or pipes. Since these communication methods are not commonly used by Android applications (they were found in 0.5% of the apps in our measurements study)—but are hard to prevent without hooking each method individually—and since applications can achieve the same goals using Android-specific communication channels (such as intents), we recommend that mobile platforms like Android restrict apps from using Linux IPC. In the meantime, ChannelCheck can be used to detect use of Linux IPC.

Stepping back. Our work on AppFork design provides a foundation for a single-phone solution to the BYOD problem, and our prototype demonstrates the feasibility of AppFork in practice. The current AppFork implementation already addresses important use cases which, based on our app analysis, are the most common in today's apps. The recommendations outlined above, if adopted by Android, would further facilitate an AppFork-like model that supports lightweight, per-app profile switching.

8. RELATED WORK

Some recent work on virtualization has been successfully applied to smartphones [3, 4, 18, 19]. "Classical virtualization" approaches require duplicating the phone OS and the Android stack, a challenge on resource-constrained mobile devices [6, 33]; optimizations can reduce this overhead. For example, Cells [3] is a lightweight OS-level virtualization architecture that enables multiple virtual phones to run on the same physical smartphone, enabling virtual work and personal phones. In contrast to AppFork, in such an approach, applications are duplicated across virtual phones with a clear overhead. Moreover, foreground and background applications must belong to the same domain (profile). User accounts, recently introduced on tablets (e.g., Microsoft Surface and Android 4.2 tablets) suffer from the same problems.

Other (non-virtualization-based) security frameworks exist for untrusted domain isolation. Examples include TLR [28] and TrustDroid [6]. We could apply this approach to the BYOD context, although it is unclear whether the trusted domain is work or personal—in fact, both profiles have sensitive information to hide from each other. Moreover, unlike AppFork, TrustDroid does not consider leaks through external storage, and allows an application to belong only to one domain.

Previous work on building trusted execution environments [12, 20, 21, 28, 29] typically rely on a Trusted Platform Module (TPM) and a trusted kernel. The Trusted Language Runtime (TLR) architecture [28, 29] is such a system that aims (unlike others) to also be easy to program. In the BYOD context, one could use TLR to run apps with sensitive data and avoid leakage to another profile. While extremely secure, solutions like TLR are not likely to scale for BYOD applications: they would require rewriting all apps, and they are not designed to run entire apps within the trustbox.

MOSES [26, 27], a policy-based framework for enforcing software isolation of applications and data, is closely related to AppFork. Unlike AppFork, MOSES does not support per-application profiles and requires heavyweight taint tracking [9] at run time.

Finally, Divide [1] provides a workspace application that supports work-related functions like calendar, email and office-like applications. Worklight [32] provides a secure SDK for developing applications. Such solutions are easy to deploy, but they cannot support existing unmodified applications. Moreover, their application isolation relies on the Android permission framework, which, as we discussed in §4, is not sufficient, and is subject to attacks such as privilege escalation [5, 8, 11, 30].

9. CONCLUSIONS

We studied the problems that BYOD raises on current mobile platforms, and proposed AppFork, a novel approach based on the notion of per-app profiles. AppFork provides the security and privacy properties of two phones, but with a single phone. Meeting both the security and functionality goals of AppFork is challenging because today's apps, despite being sandboxed, have many cross-application channels available for sharing data. These channels can turn into data leaks across profiles. We created ChannelCheck, a tool that uses static and dynamic analysis to automatically detect such leakage channels, and we reported on the results of running it on over 14,000 Android apps. Based on these results, we studied how to prevent cross-profile data leakage via the most prominent channels, and implemented our resulting design in AppFork. Our evaluation showed that AppFork is effective at supporting unmodified existing apps, with low overheads in terms of storage and profile switching time.

10. REFERENCES

[1] Divide - The Secure Workspace. http://www.divide.com/.

[2] Android. AppBrain Stats. http://www.appbrain.com/stats/android-market-app-categories.

[3] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187. ACM, 2011.

[4] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, Dec. 2010.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr. 2011.

[6] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *Proc, of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62. ACM, 2011.

[7] CIO. BYOD: Time to Adjust Your Privacy Expectations, May 2012. http://www.cio.com/article/707287/BYOD_Time_to_Adjust_Your_Privacy_Expectations.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proc. of the 13th International Conference on Information Security*, ISC'10, pages 346–360. Springer-Verlag, 2011.

[9] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–407. USENIX Association, October 2010.

[10] G. M. et al. The "Bring Your Own Device" to Work Movement, May 2012. http://www.littler.com/publication-press/publication/bring-your-own-device-work-movement.

[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. of the 20th USENIX Conference on Security*, SEC'11, pages 22–22. USENIX Association, 2011.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, 2003.

[13] Gartner. Gartner predicts by 2017, half of employers will require employees to supply their own device for work purposes, May 2013. http://www.gartner.com/newsroom/id/2466615.

[14] D. Genkin, A. Shamir, and E. Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. http://eprint.iacr.org/.

[15] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. Technical Report 14-941, University of Southern California, 2014.

[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proc. of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652. ACM, 2011.

[17] InformationWeek. BYOD Security Tops Doctors' Mobile Device Worries, October 2012. http://www.informationweek.com/mobile/byod-security-tops-doctors-mobile-device-worries/d/d-id/1107153?

[18] O. K. Labs. Ok:android. http://www.ok-labs.com/products/ok-android/.

[19] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 39–50, 2011.

[20] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158. IEEE Computer Society, 2010.

[21] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, 2008.

[22] K. Miller, J. Voas, and G. Hurlburt. BYOD: Security and Privacy Considerations. *IT Professional*, 14(5):53–55, Sept 2012.

[23] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proc. of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '12. ACM, 2012.

[24] RCR Wireless. BYOD grows, stirs privacy concerns, September 2012. http://www.rcrwireless.com/article/20120928/enterprise_mobile_and_wireless/byod-grows-privacy-concerns/.

[25] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium*, 2013.

[26] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 3–12, 2012.

[27] G. Russello, M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Demonstrating the effectiveness of moses for separation of execution modes. In *Proc. of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 998–1000. ACM, 2012.

[28] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 21–26. ACM, 2011.

[29] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, March 2014.

[30] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proc. of the 18th Annual Network and Distributed System Security Conference (NDSS'11)*, 2011.

[31] United States District Court, S.D. Texas, Houston Division. Rajaee v. Design Tech Homes, Ltd. Civil Action No. H-13-2517, Nov. 2014. http://scholar.google.com/scholar_case?case=17791284937819645574.

[32] WorkLight Inc. WorkLight Mobile Platform. http://www.worklight.com/.

[33] Y. Xu, F. Bruns, E. Gonzalez, S. Traboulsi, K. Mott, and A. Bilgic. Performance evaluation of para-virtualization on modern mobile phone platform. In *Proc. of International Conference on Computer, Electrical, and Systems Science, and Engineering (ICCESSE '10)*, 2010.

[34] D. Zielinski. Bring your own devices. *Society for Human Resource Management*, 57(2), February 2012.