AppShield: Enabling Multi-entity Access Control Cross Platforms for Mobile App Management

Zhengyang Qu[1](✉), Guanyu Guo[2], Zhengyue Shao[2], Vaibhav Rastogi[3],
Yan Chen[1], Hao Chen[4], and Wangjun Hong[1]

[1] Northwestern University, Evanston, IL 60208, USA
{zhengyangqu2017,wangjunhong2015}@u.northwestern.edu,
ychen@northwestern.edu
[2] Zhejiang University, Hangzhou, China
{guanyuguo,szylover}@zju.edu.cn
[3] University of Wisconsin, Madison, WI 53706, USA
vrastogi@wisc.edu
[4] University of California, Davis, CA 95616, USA
hchen@ucdavis.edu

Abstract. Bring-your-own-device (BYOD) is getting popular. Diverse personal devices are used to access enterprise resources, and deployment of the solutions with customized operating system (OS) dependency will thus be restricted. Moreover, device utilization for both business and personal purposes creates new threats involving leakage of sensitive data. As for functionalities, a BYOD solution should isolate an arbitrary number of entities, such as those relating to business and personal uses and provide fine-grained access control on multi-entity management. Existing BYOD solutions lack in these aspects; we propose a system, called AppShield, which supports multi-entity management and role-based access control with file-level granularity, apart from local data sharing/isolation. AppShield includes (1) application rewriting framework for Android apps, which builds Mobile Application Management (MAM) features into app automatically with complete mediation, (2) cross-platform proxy-based data access mechanism, which can enforce arbitrary access control policies. The fully functional controller with data proxy is implemented for both Android and iOS. AppShield allows for enterprise policy management without modifying device OS. The evaluation shows that AppShield is successful at policy enforcement and is reliable. It induces little impact on application's performance and size, for example, our app rewriting introduces less than 5% code size increment in over 95% apps in our evaluation.

1 Introduction

Bring your own device (BYOD) enterprise policies have been growing in popularity. Employees use their personal devices to access an enterprise's proprietary resources. According to the survey by RCR Wireless News in 2015 [1], 85% of respondents indicated BYOD was incorporated into their organization's current

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 3–23, 2017.
DOI: 10.1007/978-3-319-59608-2_1

telecom offering. The popularity of BYOD represents both an opportunity and a challenge. On the one hand, it boosts productivity and reduces the cost of dedicated devices. On the other hand, using the same device for both business and personal activities incurs new security threats, such as data exfiltration and revenue loss due to lost devices, employee job hopping, and malware. For example, considering the threat of malware alone, both Android and iOS have been reported to be affected by malware or low-reputation content [16,19,30]. Used in a BYOD setting, infected devices could threaten the confidentiality and integrity of business data. The concept of Mobile Application Management (MAM) is thus proposed to secure the BYOD utilization. Specifically, MAM solutions are the software and services that control access to enterprise resources at the mobile application level.

Android and iOS have discretionary access control to isolate data among apps. Regarding data sharing, Android provides the world read-/writable external storage, and iOS maintains a similar directory `/Documents/Inbox/`. The system default data sharing/isolation mechanisms are insufficient for the complicated scenario of BYOD, given the numerous inter-app information flows from various entities. We also investigate existing BYOD commercial solutions (in Sect. 3.1), studies on information flow control [25,31–33,42] and application virtualization/sandboxing [21,29,43]. The following issues are not addressed.

- **Portability.** Many existing studies have been proposed to secure privileged resources in the enterprise environment [29,37], but they are rarely adopted by vendors. Users have to get the customized firmware in deploying the security extension on their devices; this may not be possible because most devices have locked boot loaders and even in cases where this is technically possible, users may lack the right skills. The fragmentation issue of Android is another dominant factor that hinders the solutions with customized OS dependency from deploying in large scale. A recent report [8] showed 599 distinct Android brands with 11,868 distinct devices in 2013 and 18,796 distinct devices in 2014. Moreover, each of Android OS versions 2.3, 4.0, 4.1, 4.2, 4.4 has more than 10% of the worldwide market share. A solid MAM solution should not have any OS-specific requirement, e.g. version, firmware, to bolster the portability.
- **Multi-entity management.** Given a device, parallel data access control among application sets of various business entities is essential in the scenario of external business partner collaboration. For example, when a consulting company works closely with multiple clients simultaneously, it requires privileged data from those companies. The data sharing within each company's application set should be orthogonal. Existing BYOD solutions cannot address this issue because they only support bisecting the apps on device into the personal set and the business set.
- **Role-based access control (RBAC).** Role-based access control (RBAC) [36–38] associate permissions with roles and users are made members of roles. It eases access management and is especially beneficial to large organizations like financial and medical institutions.

While some operating systems (such as Android 5.0 and above) offer multi-account based management, the approach is not as flexible and lacks multi-entity management and RBAC support. We believe a BYOD solution should provide greater flexibility to enterprise policy administrators with respect to these aspects.

- **Fine-grained access control.** More stringent privacy laws have recently imposed new levels of confidentiality on health care and insurance companies, and financial institutions. Existing solutions do not have the policy enforcement flexible enough to secure high-credential data. In a solid solution, the data access among apps is controlled at a file level. For example, a user can share normal attachments received via email to `Dropbox`, but for a patent document with high-credential, any file sharing app's access can be blocked.

To resolve these problems in existing MAM solutions, we take the approach of application rewriting and provide it in a fully implemented prototype AppShield with the consideration of portability, which is able to enforce arbitrary access control policies with no dependency of OS. AppShield includes two parts: (1) application rewriting framework for Android platform, which builds MAM features into an app, (2) cross platform proxy-based data access mechanism, which is able to enforce arbitrary access control policies.

The application rewriting framework automatically converts a personal app to the business version with almost no developer support. Specifically, the application using AppShield does not need to be developed in a certain way w.r.t storing/accessing documents. We hook into the `libc` [6] to capture all file system system-call related calls and those relevant to Android content provider [7]. This design enables AppShield to achieve complete mediation. AppShield protects privileged data access through the stealth channels: (1) native code, (2) dynamic code loading [34], and (3) Java reflection. The interposed low-level system calls can reliably intercept the privileged data request from the application level in all these scenarios. While we provide our proxy-based data access mechanism for both platforms, the application rewriting is available for Android only due to the closed-source nature of iOS. Nonetheless, with a little developer support (such as using an "AppShield" SDK), it is possible to provide iOS support.

The proxy-based data access mechanism is implemented within a controller application. Then we transparently proxy the data requests through our own controller that manages the applications' file-system-level data, content provider data and enforces access control policies. Apart from portability, the novel design of decoupling policy enforcement from OS also brings the benefit of cross platform. With the idea of data request proxy, we implement the fully functional controller application on iOS platform.

The AppShield Android app[1] has been released on both Google Play in North America, and Myapp in China. Our contributions are:

- We design a proxy-based data access mechanism that does not need OS support to enforce arbitrary access control policies, including those like

[1] https://play.google.com/store/apps/details?id=com.webshield.appshield&hl=en.

MAC/SELinux [39] also. It is easily extended to other platforms, which is implemented on both Android and iOS.

- We investigate applying our proxy-based data access mechanism to Android MAM. The system prototype supports the configuration/enforcement of four types of security policies. *File isolation.* The privileged files of business apps are isolated from personal apps. *Multi-entity management & RBAC.* Apps can be divided into an arbitrary number of logical sets. It is further utilized in modeling RBAC, with orthogonal intra-set data access and multicast security policy update. Although we are not the first to apply RBAC to Android platform [36,37], we propose a novel design without OS modification to boost portability. *Fine-grained file access control.* To provide special protection on high-credential data, the access control policy could be defined at file-level granularity. *Content provider isolation.* Other than managing the privileged structured data in system content provider, the data requests from the business apps are redirected to a private mirror content provider. For example, the business contacts are hidden from the personal apps.
- Our evaluation shows that AppShield has low overhead in memory, runtime, and package size and that it can reliably rewrite a large number of apps.

The remainder of this paper is organized as follows. Section 2 presents a brief background. Next, we cover the problem statement and AppShield design in detail in Sect. 3, followed by the implementation aspects in Sect. 4. Section 5 deals with the evaluation of AppShield. We have the relevant discussion and related work in Sects. 6 and 7. Finally, we conclude our work in Sect. 8.

2 Background and Threat Model

Background. Android apps are implemented in Java, which is compiled down to Dalvik bytecode. It is also possible to use native code in apps. Android runtime environment enforces the sandbox mechanism to separate running apps. An app is assigned a unique user identifier (UID), by which the Linux kernel enforces discretionary access control (DAC) on low-level resources. Specifically, each app holds a private directory to keep the data in the internal storage, which cannot be accessed by any other app. The middleware further offers a permission system [9]. An app is granted permissions during installation. Apart from the pre-defined permissions guarding the system services, an app can define its customized permissions to restrict the access to their own components: *Activities*, *Services*, *Content Providers* [7], and *Broadcast Receivers*. Android includes content providers to control the access to a structured set of data.

3 types of MAM solutions have been proposed for BYOD.

- *Application Rewriting.* This approach inserts management hooks into existing Android apps. It has the advantages that it requires no developer collaboration and that it is independent of the OS version. However, it fails on apps that have been protected by anti-decompilation techniques.

- *Software Development Kit (SDK).* MAM vendors provide software development kits (SDK) for developers to incorporate into their apps. This approach has the disadvantage that developers must build and distribute two versions of the same app, and users' choice of business apps is limited to the markets.
- *OS Modification.* MAM features are directly built into the OS, so it neither requires developer collaboration nor can be defeated by anti-decompilation. However, since it relies on OS customization, the portability is limited.

In the case of application rewriting, third-party BYOD services are deployed with enterprise mobile marketplace. The client company selects useful general app, and BYOD vendor generates the enterprise version. Application rewriting requests reverse-engineering the personal app. With developer's cooperation in an enterprise setting, the developers can be asked not to apply anti-decompilation techniques, and either the developer's certificate or the unique certificate generated by BYOD vendor can be used to sign the business app under the agreement. Thus, app update can be easily managed in a timely manner.

Permissions are associated with roles, and users are made members of appropriate roles. Compared with the traditional group-based access control that only involves a set of users, using the role concept to bridge the user set and the permission set largely simplifies management of permissions and brings extra semantics in access control, which is valuable in the scenario of MAM.

Threat Model. On the device, both personal apps and business apps are installed. The personal apps may contain malware, which is able to access and leak the privileged data to untrusted servers. Moreover, for the data owned by an enterprise, other companies are motivated to track it.

OS level protection sacrifices the portability. Considering Android fragmentation, a solution without portability cannot fulfill the needs of BYOD, where employees utilize their diverse personal smartphones for business usage also. We agree that our defenses can be compromised if a device is rooted. Root is however too strong a threat model. Only hardware or hypervisor-based solutions can ensure defense against superuser attacks. OS-level defenses remain vulnerable. Furthermore, a lot of modern devices are not rootable by any known means, meaning our defenses can offer complete protection.

3 System Design

3.1 Problem Statement

Security Model. The security model of AppShield is depicted in Fig. 1. An employee may install both personal and business apps on her device. A personal app may be any app that the user wishes to install, including possibly malicious apps. The business app, however, is issued by the IT administrator, who grants business apps as follows. First, he selects any off-the-shelf app from a mobile marketplace that is useful for his organization and submits the request to the BYOD vendor. Then, BYOD provider vets it using existing malware detection

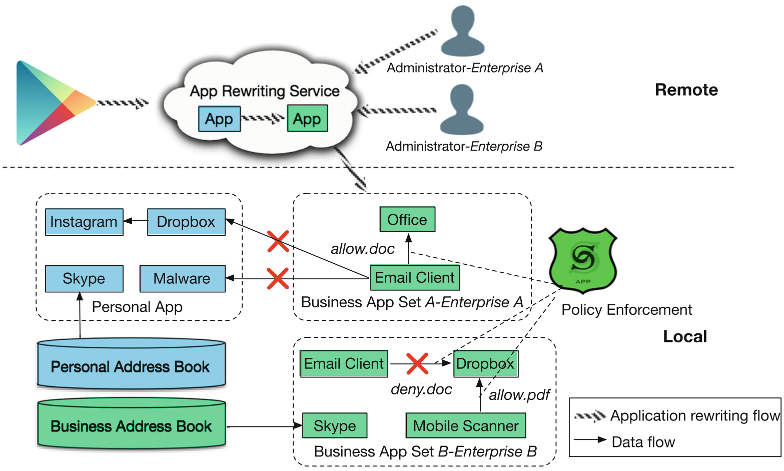


Fig. 1. Security model

systems, such as [20,26,35]. Finally, the app is converted into business version and deployed in the enterprise mobile application marketplace after getting the agreement from the application developer.

Personal apps share data by existing mechanisms, such as content provider and public external storage, on Android. For example, `Instagram` posts the photos managed by `Dropbox`. Business apps share corporate data using the mechanisms provided by AppShield. AppShield manages a secure space where all the business data are maintained and security policies can be dynamically configured and enforced at file-level granularity as the tuple:

$$Policy = (\texttt{App\_S}, \texttt{Obj}, \texttt{App\_R}, \texttt{D}), \tag{1}$$

where `App_S` and `App_R` are the apps to share and receive the data, `Obj` is the object to be shared, and `D` is the decision made. When the `Office` app, for example, opens a document "`allow.doc`" from the business `Email Client`, AppShield validates the identity of the `Office` app, verifies against the security policy, opens the attachment file, and provides the business version of `Office` with the file descriptor of the opened file, whereas the app `Dropbox` could not access the file "`deny.doc`" owned by `Email Client` due to the policy violation.

As for multi-entity management, business apps from different companies installed on a device can be classified into various logic sets by the IT administrator. Given the flexibility and simplicity of management, RBAC is introduced to model the capabilities assigned to the user through the user-role review phase. Specifically, in Fig. 1, the business app set A represents that a user is assigned the role holding the permissions to check the email and edit attached enterprise document belonging to enterprise A. The business app set B grants higher privilege to the user and allows the access to the address book and scanned document shared via the cloud service of enterprise B.

Table 1. Comparison with existing MAM solutions

Method	System	Isolation	Multi-entity management	RBAC	Granularity	Sharing	Portability
Rewriting	**AppShield**	**Sandbox**	**Yes**	**Yes**	**File-level dynamic**	**Local**	High
	AirWatch [2]	Sandbox	No	Yes	Static	Online	
	Mocana [15]	Sandbox	No	No	Static	Online	
SDK	Good [13]	Sandbox	No	No	Coarse dynamic	Online	High
	Citrix [12]	Sandbox & Encryption	No	Yes	Static	Local	
	AirWatch	Sandbox	No	Yes	Static	Online	
OS modification	Android L	DAC	No	No	Coarse dynamic	Local	Low

System Overview. Our system is organized into two parts: (1) an application rewriting framework for Android platform as the back-end that converts a personal app from mobile markets to a hardened business version by injecting MAM functionalities; (2) a front-end mobile app for both Android and iOS platforms that enforces the security policies with our proxy-based data access mechanism.

Table 1 lists existing MAM solutions on corporate data isolation/sharing and access control. The leading MAM vendors, except Citrix [12], fail to support local privileged data sharing, which requires the network connection and reduces the usability. Given the lack of fine-grained access control, these solutions are not able to provide special care of data with high-credential. All of the existing MAM solutions listed in Table 1 only bisect apps into the business set and the personal set. AppShield supports classifying the installed apps into an arbitrary number of groups, which enables multi-entity management. Some current BYOD systems provide RBAC support, but they deploy the access control module on the server side handled by their own administrators, which is not feasible in managing the data from multiple companies on the same device due to the lack of communication channel among IT administrators. Our solution jointly considers role modeling and multi-entity management.

To our best knowledge, Bring Android to work [11] deployed on Android 5.0 and above is closest to our framework but it still fails to satisfy all the requirements listed in Sect. 1. This system is implemented at the operating system level. It divides the external storage into two directories: `/storage/emulated/0/` for personal apps and `/storage/emulated/10/` for business apps. The two versions of an app run with different UIDs. The data in one directory is only publicly accessible and shareable by apps from the corresponding set.

On Android L, we found that enterprise data could be shared among them without proper regulation. Because Android L only enforces DAC at the root directories of the two application sets, the fundamental data sharing mechanism of authorized apps remains the same with general personal apps. When a privileged file is shared via file system, it goes through the public storage that is readable by other business apps, and the only difference is that data exchange is in the business root directory. It is not capable of setting up multiple business application sets, and thus neither the multi-entity management nor the fine-grained access control is supported.

Given our radically different design and methodology from existing studies, we summarize the following challenges:

- **Lack of OS support.** The existing Android storage mechanism can only support either data isolation by private internal storage or data sharing by the system-wide read-/writable external storage or by content providers. Previous work, such as TrustDroid [28,43], Maxoid [42], Aquifer [31], and DR BACA [37], need to modify Android middleware to achieve the domain-level data isolation or permission regulation, which strongly reduces the portability. Thus, it is non-trivial to enable allocating a selective set of apps privileged data access permission without OS modification and root privilege.
- **Diversity of data access behavior.** Developers could utilize a diverse set of methods to access privileged data. We need to abstract the data access behavior to completely enforce the data isolation/sharing policies.
- **Performance penalty.** Some previous studies employ virtualization-based approaches to provide isolation between private and corporate domains [22]. Such methods do not scale well on the resource-constrained mobile device. Moreover, deep virtualization reduces the battery lifetime given the duplication of complete OS.

3.2 Application Rewriting Framework

The developer can either call the OS API based on the framework interface written in Java or directly invoke the native libraries. All the OS-level API invocations go through libc, which then makes system calls into the kernel. The libc layer provides us with a reliable point that abstracts all the complex high-level data access requests. Overwriting the entries in the global offset table (GOT) during the dynamic linking procedure allows us to inject our hooks to monitor the app's data access behavior and enforce our security policies. Details of this application rewriting method were discussed in Aurasium [41]. We do not claim the application rewriting design as our contribution, but rather our investigation on its usage in data access control.

Android apps are distributed in APK, which is a JAR archive including compiled Java source files in Dalvik bytecode, compiled manifest file, resources such as layout, images, and native libraries. We first unpack the APK file and decompile the dex bytecode to an intermediate representation (IR) `smali` [17] to enable our modification on bytecode. Our rewriting modifies 3 parts of application:

- **Native code.** We implement our customized system call hooks in C/C++ to monitor the privacy-sensitive behavior, such as `open()` and `rename()` for file access and `ioctl()` for data exchange via the content provider. Java code cannot modify process memory space, so we include the native code to overwrite the GOT with the address of our detour hooks whenever any ELF file is loaded. Moreover, business apps have frequent communication with

AppShield, which includes information such as the identifier of business app to enforce security policies, and we thus implement the communication via the socket in the native layer for the latency performance.
- **Manifest file.** Android OS has the process `zygote` to initialize all the apps. When an app is running, its runtime environment is established. To enable GOT overwriting in ELF file, we modify the Manifest file to wrap the target app with our preprocess procedure. Specifically, we inject a service into the app that invokes the native code to modify the GOTs of all the loaded ELFs, and the preprocess procedure is configured in the parent class of the whole target app to guarantee it is running in the middle of `zygote` initialization and the start of the app. Moreover, AppShield front-end app manages the security policy repository set by the IT administrator and enforces the security policies that grant the app the access to privileged data. Thus, we need to declare the *Activities* in the manifest file, which are injected into the target app's bytecode to popup UI message about the violation of secure policies. Regarding the data sharing/isolation of content provider, we create a mirror content provider in the private internal storage of AppShield and guard it with a special permission. Therefore, if a business app needs access to this content provider, it must declare this permission in the Manifest file.
- **Bytecode.** We modify the bytecode to configure the preprocess procedure in the parent class of the app. For example, `class A` is the child class of `class B` whose parent class is `android.app.Application` [4]. Then we replace the parent class of `class B` with our injected service. The *Activities* showing UI message are written in Java, compiled and converted to Dalvik bytecode.

We then compile the IR into the rewritten version of bytecode and repack the app into an APK file. An app needs to be signed, but rewriting invalidates its original signature, and AppShield cannot sign the rewritten app using its original private key. The signature is mainly used for identifying the developer. Moreover, app updates require the new version of each app to be signed with the same private key as the old version. AppShield can achieve these functions by signing apps originally signed with same keys with same (but new) keys.

AppShield is deployed as a remote service and generates a random private key to sign each business app. When the app is installed, the client side AppShield keeps the mapping from the package name to its signature, which is used to differentiate business apps and personal apps. Due to the physical isolation of signature generation and the one-to-one mapping of original keys to new keys, it is difficult for an attacker to create a malicious app with the same signature as that of a legitimate business app to launch the privilege escalation attack. Our remote service can manage app update in the same way as mobile markets.

3.3 Proxy-Based Data Access Mechanism

Figure 2 illustrates our proxy-based data access mechanism. In Android, any operation on privileged data via file system and content provider goes through our customized low-level system calls. The injected bytecode collects the context

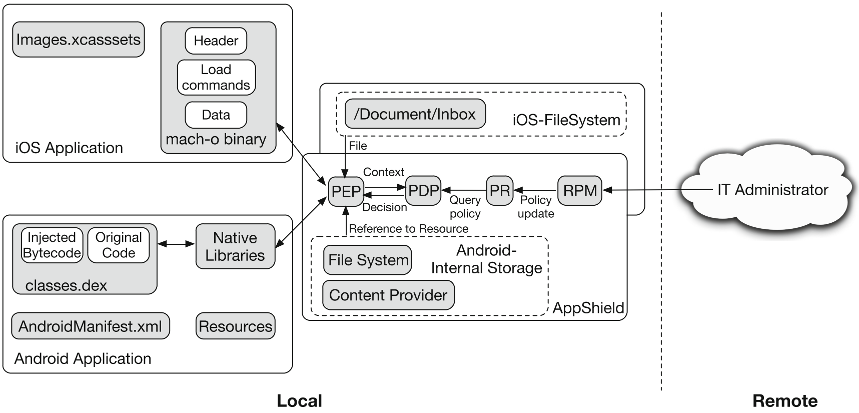


Fig. 2. Proxy-based data access mechanism

of the operation, such as the package name, signature, and data properties. The context is then sent to the **Policy Enforcement Point (PEP)**, which is implemented as a *Service* in AppShield and can be accessed by other apps through the socket in the native layer. On iOS platform, the request of file operation carrying the app's identity and target file object is sent to PEP, which is implemented as a handler. The **Policy Decision Point (PDP)** decides whether the operation is allowed based on the context from PEP and the query results from the **Policy Repository (PR)** that could be remotely updated by IT administrator via **Remote Policy Manager (RPM)**.

Android. AppShield virtually maintains a file system and content providers in its internal storage. If data sharing is allowed, AppShield generates a reference to the data, which is granted to the business app. The business app indirectly operates on privileged data based on the reference to avoid creating duplicated data for the sake of performance, security, and synchronization. Data isolation is achieved, because the file system and the content provider are privately stored in the internal storage, and PDP validates whether the app requesting data operation is a business one; if so, application identity is further verified against security policy set.

File-system. Wherever the original app stores the data, such as public external storage and privately accessible space, AppShield redirects the file operations from business apps to its own internal storage. We need to hook the following system calls:

- **open(), creat().** As an app invokes these two system calls, AppShield invokes the original system calls with a modified file path in the internal storage of AppShield and passes the flags and modes with a returned file descriptor.

- **rename(), mkdir(), remove().** The file paths in the parameters of these system calls are replaced with the business file paths in its internal storage.
- **stat(), lstat().** AppShield first gets a file descriptor to the business file in its internal storage and then invokes the `fstat()` to fetch the file status.

Content Provider. Content providers manage the access to a structured set of data, which is identified by `URI` [10]. Our proxy-based data access mechanism on content provider goes as follows:

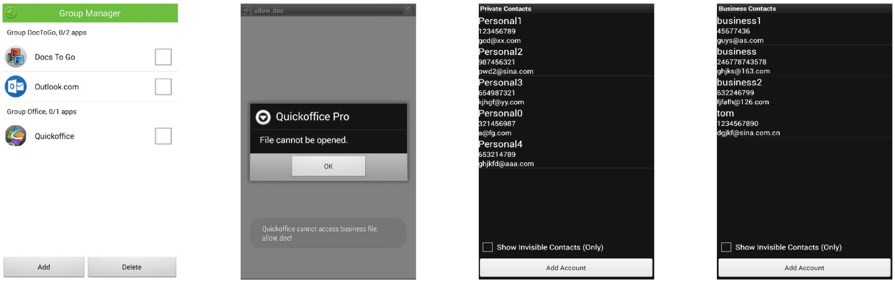
- **Mirror content provider.** The core of content provider is the *SQLite* database. AppShield duplicates the target content provider with the same schema and table definition in its private internal storage. AppShield guards the mirror content provider with a special permission.
- **System call ioctl().** This is the main system call through which all binder IPCs are sent. By interposing on this system call, AppShield replaces the URIs to the original content provider with the URIs to the mirror content provider to redirect the data operation. Using context in this system call, AppShield validates who initiates the operations on the content provider, and the PDP module decides whether to allow the access. The malicious app thus cannot operate on the mirror content provider by the overwriting URI and permission declaration.

iOS. Given the closed source iOS, it is difficult to have the rewriting framework inject the MAM features into general iOS apps without developer support. However, we easily extend our proxy-based data access mechanism on iOS platform and implement the AppShield iOS client in `Swift`, which manages the virtual file system in its private space. The business app, which owns the privileged file, could *create* and *update* privileged file by sending it to AppShield's directory `Documents/Inbox/`. At the same time, AppShield records the mapping between the app's identity and the file object, which is expressed as `App_S` and `Obj` in Eq. 1. The "Open-in management" feature, introduced from iOS 7 [14], allows AppShield to control which app the device uses to open a file. Thus, when an app `App_R` attempts to operate on the privileged file, AppShield validates the request against the policies in PR.

3.4 Security Policy

File Isolation. The file-related operations from personal apps to business apps are strictly prohibited. All the files owned by business apps are kept in the internal storage of AppShield client app, which is invisible to all the other apps. When an app initializes the file operation request, the package name bound with its signature are sent to AppShield, which verifies whether it is a business app against the record in a database. It is extremely challenging to evade this security check because it requires the attacker to get the mapping relation between package name to app signature, which is constructed on the remote server side and securely stored in the private space of AppShield client side.

Multi-entity Management and RBAC. Given the business apps from different companies, IT administrators can set up multiple app sets, where the union of the apps set's functionalities represents the permissions granted to this role (set). After a business app is pushed and installed on the device, it is assigned to a business app set following the configuration made by IT administrators, which can be dynamically adjusted on-the-fly. Once the business identity of the app requesting file access `App_R` is verified, AppShield would further check whether there is an app set including both the owner of the target file `App_S` and `App_R`. If the two apps are not grouped into the same set, the file operation will be denied, which thus guarantees the orthogonal data access among roles. The example is illustrated in Fig. 3a and b, where one app set includes email client `Outlook`, document editor `Docs to Go`, and another set consists of the app `Quickoffice`. When `Quickoffice` tries to open the file *allow.doc* as an attachment in `Outlook`, the request is denied because the policy maintains the parallel access among different roles.



Group Manager

Group DocToGo, 0/2 apps

Docs To Go □

Outlook.com □

Group Office, 0/1 apps

Quickoffice □

allow.doc

Quickoffice Pro

File cannot be opened.

Quickoffice cannot access business file: allow.doc!

Private Contacts

Personal1
123456789
gcd@xx.com

Personal2
987456321
pwd2@sina.com

Personal3
654987321
kjhgf@yy.com

Personal0
321456987
a@fg.com

Personal4
653214789
ghjkfd@aaa.com

□ Show Invisible Contacts (Only)

Business Contacts

business1
45677436
guys@as.com

business
246778743578
ghjks@163.com

business2
632246799
fjfafh@126.com

tom
1234567890
dgjkf@sina.com.cn

□ Show Invisible Contacts (Only)

(a) Application sets (b) Parallel access among application sets (c) OS contact provider access (d) Business contact provider access

Fig. 3. Multi-entity management, RBAC & Content provider isolation

Fine-Grained File Access Control. Android Lollipop allows all the requests across the business apps. In contrast, AppShield's file sharing is managed at file-level granularity for the apps in the same set. Given the sender app `App_S`, the receiver app `App_R`, and the file object `Obj`, AppShield checks the corresponding security policy in its repository, whose default value is `Allow`. This mechanism enables more flexible access control in protecting the high confidential file.

Content Provider Isolation. Business app conducts operations on the mirror content provider. If the app's identity is verified, the cursor of the mirror content provider will be returned, or AppShield will assign the app with the reference to the system content provider. This design guarantees the isolated operation on data in system default content provider and business privileged content provider. Note the example app in Fig. 3c and d, with the behavior of accessing the system's address book, the enterprise app fetches the business contacts in mirror content provider.

4 Implementation

We leverage the existing open source tools `apktool` [3] to unpack, decompile, and repack the app. We implement our customized system calls in C/C++. The open source tool AXML [5] allows us to modify the Manifest file at ease. The activities used to popup warning message are implemented in Java and those `.class` files are converted to bytecode using `dx` included in Android build tools. We also implement a script in `Python` to rewrite the bytecode in IR.

Android has 3 system content providers: contact provider, SMS provider, and calendar provider. The proxy-based data access mechanism is currently implemented on the contact provider. The calendar provider and SMS provider could be extended easily with small engineering efforts. For the content providers of third-party apps, our solution interposes on the system call `ioctl()` and blocks the operation when the app managing the content provider and the app accessing the data are from different sets.

5 Evaluation

We evaluated AppShield on a Samsung Galaxy Nexus with 4.3 Jelly Bean and an iPhone 5 s with iOS 8.1.1.

5.1 Security Policy Enforcement

We selected 50 apps from Google Play to evaluate the effectiveness of our proxy-based data access mechanism. These apps have common business functions, such as email, file-sharing, document editing/viewing, and contact management, which were classified into two sets by the type of sensitive data operation: (1) 35 file-related apps, and (2) 15 contact provider-related apps.

We first used AppShield to convert these 50 apps to business versions. Then we manually interacted with these apps. Only one app can not be rewritten due to its obfuscation, which crashed the reverse engineering toolchain during unpacking, decoding, and repacking. One app crashed after rewriting. Even if we just decompiled and repacked the app without any code modification or injection, this app still crashed, which is probably attributed to the usage of repackage-detection techniques, e.g. integrity verification.

We then tested each file-related app against three security policies. Specifically, whether the file owned by the business app was isolated from personal apps and business apps from another group; whether the request from other business apps in the same group can be allowed and blocked according to the configuration. The results are listed in Table 2. Two apps cannot enforce the security policies regarding multi-entity management and fine-grained access control. After investigating the reason through application reverse-engineering, we found that these two apps looked up files with the path starting with "/./sdcard", which was not considered when being converted to paths in the private space of AppShield and thus the business files cannot be located.

Table 2. 35 File-related applications

Package name	Isolation	Multi-entity management & RBAC	File-level granularity
com.pixatel.apps.filemgr	√	√	√
cn.wps.moffice_eng	√	√	√
com.aor.droidedit	√	√	√
com.dataviz.docstogo	√	√	√
net.appositedesigns.fileexplorer	√	√	√
com.ImaginationUnlimited.instaframe	√	√	√
com.joodioapps.DocToPdf	√	√	√
com.lyrebirdstudio.mirror	√	√	√
com.mail.emails	√	√	√
com.majedev.superbeam	√	√	√
com.microsoft.skydrive	√	√	√
com.outlook.Z7	√	√	√
com.outthinking.textonpic	√	√	√
org.devgiant.project.zipfileextracter	√	√	√
com.sketchpicture.pictutreeffect	√	√	√
com.taxaly.noteme	√	√	√
com.thomasgravina.pdfscanner	√	√	√
com.ToDoReminder.gen	√	√	√
com.youthhr.phonto	√	√	√
cz.awk.android.docconv	√	√	√
joa.zipper.editor	√	√	√
jp.ne.shira.csv.viewer	√	√	√
net.daum.android.solmail	√	√	√
com.acr.sdfilemanager	√	√	√
com.sapparray.docmgr	√	√	√
com.jellydog.freereader	√	√	√
com.olivephone.office	√	√	√
vn.esse.WordToText	√	×	×
couchDev.tools.DocxParser	√	×	×
com.qo.android.am3	√	√	√
com.probcomp.filexplorer	√	√	√
com.seeke.pdfreader	Crash		
com.topnet999.android.filemanager	√	√	√
com.nimblesoft.filemanager	√	√	√
com.infraware.office.link	Cannot rewrite		
Succeed	33/35	31/35	31/35

Table 3. 15 Contact provider-related applications

Package name	Isolation
com.appyown.contactsbackuprestore	√
com.globile.mycontactbackup	√
com.idea.backup.smscontacts	√
com.ijinshan.kbackup	√
com.mofinity.ui	√
com.payneservices.LifeReminders	×
com.tos.contact	√
net.IntouchApp	√
com.actimust.simplecontacts	√
com.netqin.contactbackup	√
no.uia.android.backupcontacts	√
com.xuecs.ContactHelper	√
digiteria.backup	√
nexg.contactbackup	√
com.brainworks.contacts.cuteblue	√
Succeed	14/15

The 15 contact provider-related apps were evaluated on content provider isolation. We checked whether each app loaded data from the system contact provider before rewriting and from the mirror contact provider as the business version. The results are abstracted in Table 3. One app failed in policy enforcement. Unlike the normal case where app loaded the address book data from contact provider, this app indirectly used `Intent` to start the system contact manager app. Our solution does not have the control over system apps.

Across the 120 times of policy enforcement (3 for each file-related app, 1 for each contact provider-related app), our mechanism achieves the success rate 109/120 (90.8%). The general reason for the failure is that our implementation does not consider developer's specific pattern of API invocation. e.g., the path of the privileged file.

5.2 Reliability

For the test on the reliability of AppShield, we picked top 250 apps by popularity on Google Play in September 2015 within the following categories: `Business`, `Finance`, `Medical`, and `Productivity`. We used AppShield to convert these 1000 apps to their business versions, and then automatically ran the apps using the UI/Application Exerciser Monkey [18]. The results are shown in Table 4.

12 apps failed during rewriting because their obfuscation crashed the reverse engineering tools `apktool` in unpacking, decoding, and repacking. While we

Table 4. Large-scale evaluation on 1000 applications

Total apps	Succeed	Cannot be rewritten	Crashed
1000	953(95.3%)	12(1.2%)	35(3.5%)

acknowledge that AppShield cannot reliably rewrite apps with anti-reverse engineering techniques, our large-scale test shows that the percentage of these apps is still low. Also, developers are actively improving the reverse engineering tools that AppShield relies on. For the 35 rewritten apps that crashed during execution, we ran their original versions and found 29 of them also crashed, which clearly were not caused by AppShield. To investigate the reasons why the remaining 6 rewritten apps crashed while their original versions did not, we just unpacked and repacked them without modifying their code or data, and found all of them still crashed after repacking. We hypothesize that they might use anti-repacking techniques, such as signature validation. We performed these tests on real-world apps without developer support. In an enterprise MAM situation, however, it is reasonable to assume that the MAM provider can work with the developers so as to enable successful rewriting of their apps. Developers have strong incentive to work with MAM providers as this allows their apps to be used across entire enterprises.

5.3 Impact of Application Rewriting

Latency. We evaluated AppShield's performance by both *micro-benchmark* and *macro-benchmark*. We implemented a test app that opens files and loads data from contact provider. Moreover, we developed an iOS app that can delegate the permission of accessing its private files to a selective set of apps. Given the closed nature of iOS, we could not modify the invocation of low-level system calls and hence cannot build an application rewriting framework. For evaluation, we implemented the proxy-based data access mechanism inside the app. Even though our rewriting framework is not cross platform, our proxy-based data access mechanism is. We expect that with reasonable developer support, our solution is still feasible on iOS platform.

- **Micro-benchmark.** We conducted a stress test with 1000 data access operations to investigate the latency introduced by AppShield. First, we recorded the accumulated time spent on getting the file descriptor on Android and getting the file contents from the iOS AppShield client with and without our security policies enforced. Because we cannot dig into low-level system calls of closed-source iOS, we measured the time of loading file contents on that platform. We also measured the total time of fetching the cursor, which is a reference to the content provider. Only the operations that we benchmarked contain the latency introduced by AppShield for policy enforcement, and the further operations on data remained the same with the unmodified app. The results are listed in Table 5. In the worst case, AppShield introduced an overall latency of 0.202 s on Android file system during 1000 operations, because acquiring each file descriptor involves one round of IPC with AppShield. For the performance on iOS, AppShield introduced a latency of 0.176 s. AppShield introduced a latency of 1.711 s when getting the cursor of a content provider. Since IPC is the dominant factor in the latency and has a fixed cost, the relative latency decreases, as the original operation takes longer.

- **Macro-benchmark.** We asked one user to manually load data via the file system and contact provider on the smartphone. We recorded the time from when the user started to access the data until when she closed the app after the data was fully rendered on screen. The user performed a series of data access operations for 5 times with and without AppShield. Table 5 shows the average of time. AppShield introduced a latency of 52 ms, 110 ms, and 126 ms in data operations on Android file system, iOS file system, and Android content provider, respectively. Such latency is barely perceptible. Although user experience on application response might not be accurate to the order of millisecond and there is a slight difference in each round of manual operation, we try our best to simulate user's daily usage manner.

Memory Consumption and Code Size. Figure 4 shows the cumulative distribution function (CDF) of the overhead in memory usage and code size caused by rewriting. To eliminate the side effect of Android garbage collection when calculating memory usage, we used the tool `dumpsys` in Android Debug Bridge (adb) to get the maximal memory usage during the execution of an app. To eliminate the side effect of compression during app packing, when calculating code size, we sum up the customized native libraries, Manifest file, and bytecode.

AppShield's rewriting introduced less than 5% code size increment in over 95% apps, and more than 85% apps incurred the memory usage overhead less than 60%. The average overhead was 8049.1KiB in memory usage and 121.2KiB in code size. Our system hooks into the low-level system calls, and the dynamic linking naturally supports the efficient memory utilization by avoiding code duplication. Moreover, we add our customized system calls, and the classes for UI notification just once rather than inlining them at every point where the original app accesses privileged data.

Table 5. Runtime latency introduced by AppShield

	File system				Content provider	
	Android		iOS		Android	
	Original	AppShield	Original	AppShield	Original	AppShield
Micro-benchmark×1000 (s)	0.180	0.382	0.171	0.347	7.303	9.014
Macro-benchmark (s)	1.472	1.524	1.643	1.753	1.068	1.194

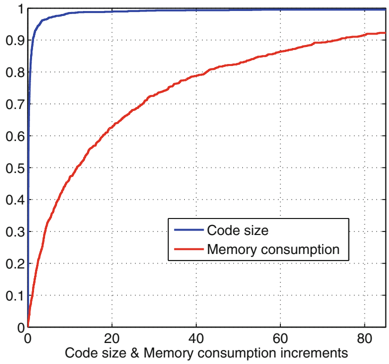


Fig. 4. Code size & memory usage overhead (CDF)

6 Discussion

AppShield does have some limitations because of its current implementation. Our rewriting mechanism involves unpacking the APK file and decompiling the dex bytecode to IR. App developers sometimes use anti-reverse engineering techniques to crash decompilation tools to protect their intellectual property. Moreover, when the IT administrators conduct the security verification on the apps to be selected as business ones, the obfuscated app may challenge the correctness of the verification. However, our large-scale evaluation shows that the percentage of these apps is low. Moreover, the app developer could be asked not to apply such tools, where tiny developer support is needed. Developers are often willing to work with enterprises as this offers them a large high-payoff user base.

Another limitation is that it depends on hooking on the dynamically-linked `libc`. Any system call invoked not via the system `libc`, such as by using a statically-linked libc, will bypass our hooking mechanism. The chance of this happening is very low, and can be detected statically. Regarding the iOS platform, it is extremely hard to automatically rewrite apps and hook those system calls, given its closed-source nature. However, the proxy-based data access mechanism is cross-platform, which is implemented as a client iOS app leveraging the "Open-in management" feature.

7 Related Work

Virtualization and Sandboxing. L4Android [29] combines the L4Linux and Google modifications of Linux kernel to enable executing Android OS on top of a microkernel. Running multiple Android OS instances in parallel on the same device enables the complete isolation but has high performance penalty. TrustDroid [43] addresses the performance issues. It introduces the logical domain isolation approach, where two single domains are considered and isolation is enforced as a data flow property between the logical domains without running each domain as a single virtual machine. Boxify [21] constructs virtual sandboxes to secure Android apps, but the decision on which app to be isolated relies on manual identification. We model the data access control problem in the scenario of MAM, and app identity is classified by its business/personal purpose. These approaches fail to consider the data-sharing problem to give a fine-granulated control that grants a selective set of apps the access to privileged data.

Rewriting. Davis et al. [24] rewrite the Dalvik bytecode to allow interposing on security sensitive APIs. Retroskeleton [23] supports the retrofit of app's behaviors by static and dynamic method interposition. These approaches are based on the high-level API interposition, and thus, they cannot completely enforce the security policies across all layers of Android framework. Aurasium [41] adopts the design most similar to us that provides reference monitor capabilities by repackaging Android apps to use a customized version of libc. AppShield extends the usage of this application rewriting technique with the proxy-based data access mechanism to achieve data access control, and multi-entity management. Similarly, ASM [27] provides a programmable interface for API hooking, which can also be leveraged to implement user-level access control.

RBAC. Vaidya et al. [40] propose RoleMiner to assist automatic role construction following a learning approach. Previous studies mostly focus on the general modeling of RBAC. Rohrer et al. [36,37] further investigate the specific RBAC problem when using Android device in sensitive environment, such as finance and health, but the mechanism involves the modification of system middleware and lacks a system prototype to be evaluated.

8 Conclusion

In this paper, we present the proxy-based data access mechanism, which can enforce arbitrary access control policies. Given the critical issues of MAM, our prototype system AppShield achieves multi-entity management and RBAC at file-level granularity, apart from privileged data isolation from personal apps and corporate data sharing across business apps. We implement it on both Android and iOS platforms to demonstrate its cross platform property. Our design has neither dependency on OS nor the root privilege, which thus has good portability. AppShield is successful at policy enforcement with low latency and is reliable.

Acknowledgments. We thank our reviewers for their valuable comments. This paper was made possible by the National Natural Science Foundation of China under Grant No. 61472209, by the U.S. National Science Foundation under Grant CNS-1408790. The statements made herein are solely the responsibility of the authors.

References

1. Predictions: The year of BYOD management (2016). http://www.rcrwireless.com/20160129/opinion/2016-predictions-the-year-of-byod-management-tag10
2. AirWatch: Enterprise Mobility Management. http://www.air-watch.com/
3. Android-apktool: A tool for reengineering Android apk files. http://code.google.com/p/android-apktool/
4. Android application class. http://developer.android.com/reference/android/app/Application.html
5. Android binary XML file parser. https://github.com/xgouchet/AXML
6. Android bionic. https://android.googlesource.com/platform/bionic/
7. Android content provider. http://developer.android.com/guide/topics/providers/content-providers.html
8. Android fragmentation report august 2014 - opensignal. http://opensignal.com/reports/2014/android-fragmentation/
9. Android manifest permission. http://developer.android.com/reference/android/Manifest.permission.html
10. Android Uri. http://developer.android.com/reference/android/net/Uri.html
11. Bring Android to Work. http://www.android.com/it/preview/
12. Citrix. https://www.citrix.com/
13. Good Technology. https://www1.good.com/
14. iOS Open-in management. http://searchmobilecomputing.techtarget.com/tip/Open-in-management-helps-secure-iOS-data
15. Mocana - Strong and Usable Security. https://www.mocana.com/
16. Significant iPhone and iPad malware threats will emerge (2015). http://www.ibtimes.co.uk/significant-iphone-ipad-malware-threats-will-emerge-2015-1490577
17. Smali: An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/
18. UI/Application exerciser Monkey. http://developer.android.com/tools/help/monkey.html
19. What You Need to Know About iOS Malware XcodeGhost. http://www.macrumors.com/2015/09/20/xcodeghost-chinese-malware-faq/
20. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of android malware in your pocket. In: Proceedings of NDSS (2014)
21. Backes, M., Bugiel, S., Hammer, C., Schranz, O., Von Styp-Rekowsky, P.: Boxify: Full-fledged app sandboxing for stock android. In: Proceedings USENIX Security (2015)
22. Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., Zoppis, B.: The VMware mobile virtualization platform: is that a hypervisor in your pocket? ACM SIGOPS Operating Syst. Rev. **44**(4), 124–135 (2010)
23. Davis, B., Chen, H.: Retroskeleton: Retrofitting android apps. In: ACM MobiSys (2013)
24. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: a rewriting framework for in-app reference monitors for android applications. In: IEEE MoST (2012)
25. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX OSDI (2010)

26. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: ACM MobiSys (2012)
27. Heuser, S., Nadkarni, A., Enck, W., Sadeghi, A.-R.: ASM: a programmable interface for extending android security. In: Proceedings USENIX Security (2014)
28. Kodeswaran, P., Nandakumar, V., Kapoor, S., Kamaraju, P., Joshi, A., Mukherjea, S.: Securing enterprise data on smartphones using run time information flow control. In: IEEE MDM (2012)
29. Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., Peter, M.: L4Android: a generic operating system framework for secure smartphones. In: ACM SPSM (2011)
30. Lever, C., Antonakakis, M., Reaves, B., Traynor, P., Lee, W.: The core of the matter: Analyzing malicious traffic in cellular carriers. In: NDSS (2013)
31. Nadkarni, A., Enck, W.: Preventing accidental data disclosure in modern operating systems. In: ACM CCS (2013)
32. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: ACM ASIACCS (2010)
33. Ongtang, M., Butler, K., McDaniel, P.: Porscha: policy oriented secure content handling in android. In: ACM ACSAC (2010)
34. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS (2014)
35. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: ACM ASIACCS (2013)
36. Rohrer, F., Feleke, N., Zhang, Y., Nimley, K., Chitkushev, L., Zlateva, T.: Android security analysis and protection in finance and healthcare. Comput. Sci. Educ. Comput. Sci. **8**(1), 80–89 (2012). Boston University MET
37. Rohrer, F., Zhang, Y., Chitkushev, L., Zlateva, T.: DR BACA: dynamic role based access control for android. In: ACM ACSAC (2013)
38. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Comput. **29**(2), 38–47 (1996)
39. Smalley, S., Craig, R.: Security enhanced (SE) android: Bringing flexible MAC to android. In: NDSS (2013)
40. Vaidya, J., Atluri, V., Warner, J.: RoleMiner: mining roles using subset enumeration. In: ACM CCS (2006)
41. Xu, R., Saïdi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: USENIX Security Symposium, pp. 539–552 (2012)
42. Xu, Y., Witchel, E.: Maxoid: transparently confining mobile applications with custom views of state. In: ACM EuroSys (2015)
43. Zhao, Z., Osono, F.C.C.: Trustdroid: preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In: IEEE MALWARE (2012)