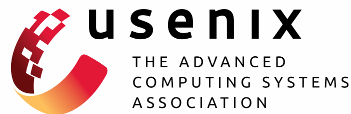
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE



Boxify: Full-fledged App Sandboxing for Stock Android

Michael Backes, *Saarland University and Max Planck Institute for Software Systems (MPI-SWS);* Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky, *Saarland University*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

Boxify: Full-fledged App Sandboxing for Stock Android

Michael Backes
CISPA, Saarland University & MPI-SWS
backes@cs.uni-saarland.de

Sven Bugiel
CISPA, Saarland University
bugiel@cs.uni-saarland.de

Christian Hammer
CISPA, Saarland University
hammer@cs.uni-saarland.de

Oliver Schranz
CISPA, Saarland University
schranz@cs.uni-saarland.de

Philipp von Styp-Rekowsky
CISPA, Saarland University
styp-rekowsky@cs.uni-saarland.de

Abstract

We present the first concept for full-fledged app sandboxing on stock Android. Our approach is based on application virtualization and process-based privilege separation to securely encapsulate untrusted apps in an isolated environment. In contrast to all related work on stock Android, we eliminate the necessity to modify the code of monitored apps, and thereby overcome existing legal concerns and deployment problems that rewriting-based approaches have been facing. We realize our concept as a regular Android app called Boxify that can be deployed without firmware modifications or root privileges. A systematic evaluation of Boxify demonstrates its capability to enforce established security policies without incurring a significant runtime performance overhead.

1 Introduction

Security research of the past five years has shown that the privacy of smartphone users—and in particular of Android OS users, due to Android's popularity and open-source mindset—is jeopardized by a number of different threats. Those include increasingly sophisticated malware and spyware [63, 39, 62], overly curious libraries [25, 32], but also developer negligence and absence of fail-safe defaults in the Android SDK [33, 29]. To remedy this situation, the development of new ways to protect the end-users' privacy has been an active topic of Android security research during the last years.

Status quo of deploying Android security extensions. From a deployment perspective, the proposed solutions followed two major directions: The majority of the solutions [26, 44, 45, 16, 21, 64, 52, 56] extended the UID-centered security architecture of Android. In contrast, a number of solutions [38, 59, 23, 49, 22, 15] promote inlined reference monitoring (IRM) [28] as an alternative approach that integrates security policy enforcement directly into Android's application layer, i.e., the apps' code.

However, this dichotomy is unsatisfactory for end-users: While OS security extensions provide stronger security guarantees and are preferable in the long run, they require extensive modifications to the operating system and Android application framework. Since the proposed solutions are rarely adopted [54, 53] by Google or the device vendors, users have to resort to customized aftermarket firmware [4, 6] if they wish to deploy new security extensions on their devices. However, installing a firmware forms a technological barrier for most users. In addition, fragmentation of the Android ecosystem [46] and vendor customizations impede the provisioning of custom-built ROMs for all possible device configurations in the wild.

In contrast, solutions that rely on inlined reference monitoring avoid this deployment problem by moving the reference monitor to the application layer and allowing users to install security extensions in the form of apps. However, the currently available solutions provide only insufficient app sandboxing functionality [36] as the reference monitor and the untrusted application share the same process space. Hence, they lack the strong isolation that would ensure tamper-protection and non-bypassability of the reference monitor. Moreover, inlining reference monitors requires modification and hence re-signing of applications, which violates Android's signature-based same-origin model and puts these solutions into a legal gray area.

The sweet spot. The envisioned app sandboxing solution provides immediate strong privacy protection against rogue applications. It would combine the security guarantees of OS security extensions with the deployability of IRM solutions, while simultaneously avoiding their respective drawbacks. Effectively,

such a solution would provide an OS-isolated reference monitor that can be deployed entirely as an app on stock Android without modifications to the firmware or code of the monitored applications.

Our contributions. In this paper we present a novel concept for Android app sandboxing based on *app virtualization*, which provides tamper-protected reference monitoring without firmware alterations, root privileges or modifications of apps. The key idea of our approach is to encapsulate untrusted apps in a restricted execution environment within the context of another, trusted sandbox application. To establish a restricted execution environment, we leverage Android's *"isolated process"* feature, which allows apps to totally de-privilege selected components—a feature that has so far received little attention beyond the web browser. By loading untrusted apps into a de-privileged, isolated process, we shift the problem of sandboxing the untrusted apps from *revoking* their privileges to *granting* their I/O operations whenever the policy explicitly allows them. The I/O operations in question are syscalls (to access the file system, network sockets, bluetooth, and other low-level resources) and the Binder IPC kernel module (to access the application framework). We introduce a novel app virtualization environment that proxies all syscall and Binder channels of isolated apps. By intercepting any interaction between the app and the system (i.e., kernel and app framework), our solution is able to enforce established and new privacy-protecting policies. Additionally, it is carefully crafted to be transparent to the encapsulated app in order to keep the app agnostic about the sandbox and retain compatibility to the regular Android execution environment. By executing the untrusted code as a de-privileged process with a UID that differs from the sandbox app's UID, the kernel securely and automatically isolates at process-level the reference monitor implemented by the sandbox app from the untrusted processes. Technically, we build on techniques that were found successful in related work (e.g., libc hooking [59]) while introducing new techniques such as Binder IPC redirection through ServiceManager hooking. We realize our concept as a regular app called Boxify that can be deployed on stock Android. To the best of our knowledge, Boxify is the first solution to introduce *application virtualization* to stock Android.
In summary, we make the following contributions:

1. We present a novel concept for application virtualization on Android that leverages the security provided by *isolated processes* to securely encapsulate untrusted apps in a completely de-privileged execution environment within the context of a regular Android app. To retain compatibility of isolated apps with the standard Android app runtime, we solved the key technical challenge of designing and implementing an efficient app virtualization layer.

2. We realize our concept as an app called Boxify, which is the first solution that ports app virtualization to the Android OS. Boxify is deployable as a regular app on stock Android (no firmware modification and no root privileges required) and avoids the need to modify sandboxed apps.

3. We systematically evaluate the efficacy and efficiency of Boxify from different angles including its security guarantees, different use-cases, performance penalty, and Android API version dependence across multiple Android OS versions.

The remainder of this paper is structured as follows. In §2, we provide necessary technical background information on Android. We define our objectives and discuss related work in §3. In §4, we present our Boxify design and implementation, which we evaluate in §5. We conclude the paper in §6.

2 Background on Android OS

Android OS is an open-source software stack (see Figure 1) for mobile devices consisting of a Linux kernel, the Android application framework, and system apps. The application framework together with the pre-installed system apps implement the Android application API. The software stack can be extended with third-party apps, e.g., from Google Play.

Android Security Model. On Android, each application runs in a separate, simple sandboxed environment that isolates data and code execution from other apps. In contrast to traditional desktop operating systems where applications run with the privileges of the invoking user, Android assigns a unique Linux user ID (UID) to every application at installation time. Based on this UID, the components of the Android software stack enforce access control rules that govern the app sandboxing. To understand the placement of the enforcement points, one has to consider how an app can interact with other apps (and processes) in the system:

Like any other Linux process, an app process uses syscalls to the Linux kernel to access low-level resources, such as files. The kernel enforces discretionary access control (DAC) on such syscalls based on the UID of the application process. For instance, each application has a private directory that is not accessible by other applications and DAC ensures

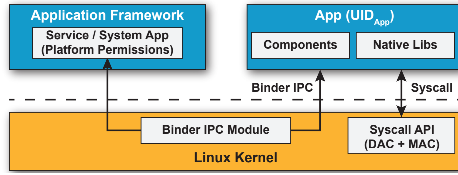


Figure 1: High-level view of interaction between apps, application framework, and Linux kernel on Android.

that applications cannot access other apps' private directories. Since Android version 4.3 this discretionary access control is complemented with SELinux mandatory access control (MAC) to harden the system against low-level privilege escalation attacks and to reinforce this UID-based compartmentalization.

The primary channel for inter-application communication is Binder *Inter-Process Communication* (IPC). It is the fundamental building block for a number of more abstract inter-app communication protocols, most importantly *Inter-Component Communication* (ICC) [27] among apps and the application framework. For sandboxing applications at the ICC level, each application UID is associated with a set of platform permissions, which are checked at runtime by reference monitors in the system services and system apps that constitute the app framework (e.g. `LocationService`). These reference monitors rely on the Binder kernel module to provide the UID of IPC senders to the IPC receivers.

In general, both enforcement points are implemented *callee-sided* in the framework and kernel, and hence agnostic to the exact call-site within the app process. This means that enforcement applies equally to all code executing in a process under the app's UID, i.e., to both Java and native code.

Additionally, Android verifies the integrity of application packages during installation based on their developer signature. The corresponding developer certificate is afterwards used to enforce a same-origin policy for application updates, i.e., newer app versions must be signed with the same signing key as the already installed application.

Isolated Process. The *Isolated Process*, introduced in Android version 4.1, is a security feature that has received little attention so far. It allows an app developer to request that certain service components within her app should run in a special process that is isolated from the rest of the system and has no permissions of its own [2]. The isolated process mechanism follows the concept of *privilege separation* [48], which allows parts of an application to run at different levels of privilege. It is intended to provide an additional layer of protection around code that processes content from untrusted sources and is likely to have security holes. Currently, this feature is primarily geared towards web browsers [35] and is most prominently used in the Chrome browser to contain the impact of bugs in the complex rendering code.

An isolated process has far fewer privileges than a regular app process. An isolated process runs under a separate Linux user ID that is randomly assigned on process startup and differs from any existing UID. Consequently, the isolated process has no access to the private app directory of the application. More precisely, the process' filesystem interaction is limited to reading/writing world readable/writable files. Moreover, the isolated process' access to the Android middleware is severely restricted. The isolated process runs with no permissions, regardless of the permissions declared in the manifest of the application. More importantly, the isolated process is forbidden to perform any of the core Android IPC functions: Sending Intents, starting Activities, binding to Services or accessing Content Providers. Only the core middleware services that are essential to running the service component are accessible to the isolated process. This effectively bars the process from any communication with other apps. The only way to interact with the isolated process from other application components is through the Service API (binding and starting). Further, the transient UID of an isolated process does not belong to any privileged system groups and the kernel prevents the process from using low-level device features such as network communication, bluetooth or external storage. As of Android v4.3, SELinux reinforces this isolation through a dedicated process type. With all these restrictions in place, code running in an isolated process has only minimal access to the system, making it the most restrictive runtime environment Android has to offer.

3 Requirements Analysis and Existing Solutions

We first briefly formulate our objectives (see §3.1) and afterwards discuss corresponding related work (see §3.2 and Table 1).

3.1 Objectives and Threat Model

In this paper, we aim to combine the security benefits of OS extensions with the deployability benefits of application layer solutions. We identify the following objectives:

O1 No firmware modification: The solution does not rely on or require customized Android firmware, such as extensions to Android's middleware, kernel or the default configuration files (e.g., policy files), and is able to run on stock Android versions. This also excludes availability of root privileges, since root can only be acquired through a firmware modification on newer Android versions due to increasingly stringent SELinux policies.

O2 No app modification: The solution does not rely on or require any modifications of monitored apps' code, such as rewriting existing code.

O3 Robust reference monitor: The solution provides a robust reference monitor. This encompasses: 1) the presence of a strong security boundary, such as a process boundary, between the reference monitor and untrusted code; and 2) the monitor cannot be bypassed, e.g., using a code representation that is not monitored, such as native code.

O4 Secure isolation of untrusted code: This objective encompasses fail-safe defaults and complete mediation by the reference monitors. The solution provides a reference monitor that mediates all interaction between the untrusted code and the Android system, or, in case no complete mediation can be established, enforces fail-safe defaults that isolate the app on non-mediated channels in order to prevent untrusted code from escalating its privileges.

Threat model. We assume that the Android OS is trusted, including the Linux kernel and the Android application framework. This includes the assumption that an application cannot compromise the integrity of the kernel or application framework at runtime. If the kernel or application framework were compromised, no security guarantees could be upheld. Protecting the kernel and framework integrity is an orthogonal research direction for which different approaches already exist, such as trusted computing, code hardening, or control flow integrity.

Furthermore, we assume that untrusted third-party applications have full control over their process and the associated memory address space. Hence the attacker can modify its app's code at runtime, e.g., using native code or Java's reflection interface.

3.2 Existing Solutions

We systematically analyze prior solutions on app sandboxing.

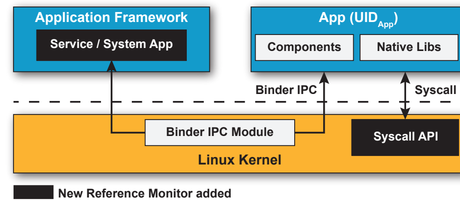


Figure 2: Instrumentation points for operating system security extensions.

Objectives	OS ext.	IRM	Sep. app	Boxify
O1: No system modification	✘	✔	✔	✔
O2: No application modification	✔	✘	✘	✔
O3: Robust reference monitor	✔	✘	✔	✔
O4: Secure isolation of untrusted code	✔	✘	✘	✔

✔= applies; ✘= does not apply.

Table 1: Comparison of deployment options for Android security extensions based on desired objectives.

3.2.1 Android Security Extensions

Many improvements to Android's security model have been proposed in the literature, addressing a variety of shortcomings in protecting the end-user's privacy. In terms of deployment options, we can distinguish between solutions that extend the Android OS and solutions that operate at the application layer only.

Operating system extensions. The vast majority of proposals from the literature (e.g. [26, 44, 45, 16, 21, 58]) statically enhance Android's application framework and Linux kernel with additional reference monitors and policy decision points (see Figure 2). The proposed security models include, for instance, context-aware policies [21], app developer policies [45], or Chinese wall policies [16]. More recent approaches [52, 43, 56] avoid static changes to the OS by dynamically instrumenting core system services (like Binder and Zygote) or the Android bootup scripts in order to interpose [47] untrusted apps' syscalls and IPC. Since in all approaches the reference monitors are part of the application framework and kernel, there inherently exists a strong security boundary between the reference monitor and untrusted code (O3: ✔). Moreover, this entails that these reference monitors are by design part of the callee-side of all interaction of the untrusted app's process with the system and cannot be by-

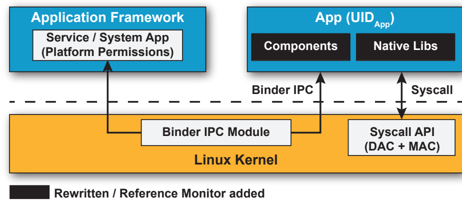


Figure 3: Instrumentation points for application code rewriting and inlining reference monitors.

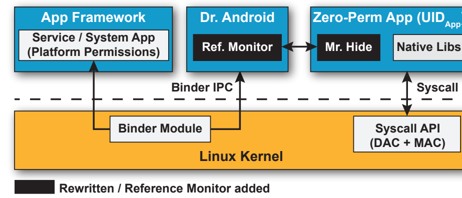


Figure 4: *Dr. Android and Mr. Hide* approach [38].

passed (O4: ✔). On the downside, these solutions require modification of the Android OS (image) or root privileges to be deployed (O1: ✘; O2: ✔).

Additionally, a number of solutions exist that particularly target higher-security deployments [17, 51, 40, 13], such as government and enterprise. Commercial products exist that implement these solutions in the form of tailored mobile platforms (e.g., Blackphone[1], GreenHills[2], or Cryptophone[3]). These products target specialized user groups with high security requirements—not the average consumer—and are thus deployed on a rather small scale.

Application layer solutions. At the application layer, the situation for third-party security extensions is bleak. Android's UID-based sandboxing mechanism strictly isolates different apps installed on the same device. Android applications run with normal user privileges and cannot elevate to root in order to observe the behavior of other apps, e.g., like classical trace or anti-virus programs on desktop operating systems [31]. Also, Android does not offer any APIs that would allow one app to monitor or restrict the actions of another app at runtime. Only static information about other apps on the device is available via the Android API, i.e., application metadata, such as the package name or signing certificate, and the compiled application code and resources.Consequently, most commercially available security solutions are limited to *detecting* potentially malicious apps, e.g. by comparing metadata with predefined blacklists or by checking the application code for known malware signatures, but they lack the ability to observe or influence the runtime behavior of other applications. As a result, their effectiveness is, at best, debatable [50, 62].

Few proposals in the academic literature [38, 59, 23, 49, 15] focus on application layer only solutions (see Figure 3). Existing systems mostly focus on access control by interposing security-sensitive APIs to redirect the control flow to an additionally inlined reference monitor within the app (e.g., Aurasium [59], I-ARM-Droid [23], RetroSkeleton [22], AppGuard [15]). DroidForce [49] additionally preprocesses target apps with static data flow analysis to identify strategic policy enforcement points and to redirect policy decision making to a separate app.

All these systems are based on rewriting the application code to inline reference monitors or redirect control flows, which works without modifications to the firmware and is thus suitable for large-scale deployment (O1: ✔; O2: ✘). However, app rewriting causes security problems and also a couple of practical deployment problems. First, inlining the reference monitor within the process of the untrusted app itself might be suitable for "benign-but-buggy" apps; however, apps that actively try to circumvent the monitor will succeed as there exists no strong security boundary between the app and the monitor. In essence, this boils down to an arms race between hooking security critical functions and finding new ways to compromise or bypass the monitor [36], where currently native code gives the attacker the advantage (O3: ✘; O4: ✘). Moreover, re-writing application code requires re-signing of the app, which breaks Android's signature-based same origin policy and additionally raises legal concerns about illicit tampering with foreign code. Lastly, re-written apps have to be reinstalled. This is not technically possible for pre-installed system apps; other apps have to be uninstalled in order to install a fresh, rewritten version, thereby incurring data loss.

Separate app. Dr. Android and Mr. Hide [38] (see Figure 4) is a variant of inlined reference monitoring (O1: ✔; O2: ✘) that improves upon the security of the reference monitor by moving it out of the untrusted app and into a separate app. This establishes a strong security boundary between the untrusted app and the reference monitor as they run in separate processes with different UIDs (O3: ✔). Additionally, it revokes all Android platform permissions from the untrusted app and applies code rewriting techniques to replace well-known security-sensitive Android API calls in the monitored app with calls to the separate

[1] https://blackphone.ch
[2] http://www.ghs.com/mobile/
[3] http://esdcryptophone.com

reference monitor app that acts as a proxy to the application framework. The benefit of this design is that in contrast to inlined monitoring, the untrusted, zero-permission app cannot gain additional permissions by tampering with the inlined/rewritten code. However, this enforcement only addresses the platform permissions. The untrusted app process still has a number of Linux privileges (such as access to the Binder interface or file system), and it has been shown that even a zero-permission app is still capable of escalating its privileges and violate the user's privacy [30, 33, 19, 18, 60, 42, 11, 12] (O4: ✘).

3.2.2 Sandboxing on traditional OSes

Restricting the access rights of untrusted applications has a longstanding tradition in desktop and server operating systems. Few solutions set up user-mode only sandboxes without relying on operating system functionality by making strong assumptions about the interface between the target code and the system (e.g., absence of programming language facilities to make syscalls or direct memory manipulation). Among the most notable user-space solutions are *native client* [61] to sandbox native code within browser extensions and the *Java virtual machine* [5] to sandbox untrusted Java applications.

Other solutions, which loosen the assumptions about the target interface to the system rely on operating system security features to establish process sandboxes. For instance, *Janus* [31], one of the earlier approaches, introduced an OS-supported sandbox for untrusted applications on Solaris 2.4, which was based on syscall monitoring and interception to restrict the untrusted process' access to the underlying operating system. The monitor was implemented as a separate process with necessary privileges to monitor and restrict other processes via the `/proc` kernel interface. Modern browsers like *Chromium* [9, 3, 8] employ different sandboxing OS facilities (e.g, seccomp mode) to mitigate the threat of web-based attacks against clients by restricting the access of untrusted code.

App virtualization. Sandboxing also plays a role in more recent *application virtualization* solutions [34, 10, 20, 41], where applications are transparently encapsulated into execution environments that replace (parts of) the environment with emulation layers that abstract the underlying OS and interpose all interaction between the app and the OS. App virtualization is currently primarily used to enable self-contained, OS-agnostic software, but also provides security benefits by restricting the interface and view the encapsulated app has of the system.

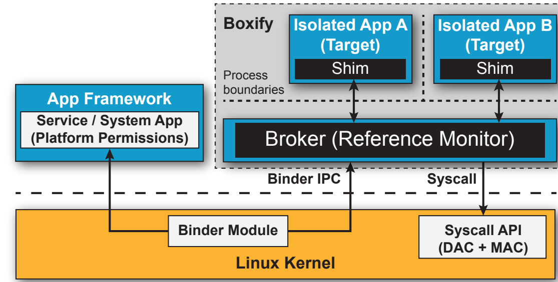


Figure 5: Architecture overview of Boxify.

Similarly to these traditional sandboxes and in particular to app virtualization, Boxify forms a user-mode sandbox that builds on top of existing operating system facilities of Android. Thereby, it establishes app sandboxes that encapsulate Android apps without the need to modify the OS and without the need to make any assumptions about the apps' code.

4 Boxify Architecture

We present the Boxify design and implementation.

4.1 Design Overview

The key idea of Boxify is to securely sandbox Android apps, while avoiding any modification of the OS and untrusted apps. Boxify accomplishes this by dynamically loading and executing the untrusted app in one of its own processes. The untrusted application is not executed by the Android system itself, but runs completely encapsulated within the runtime environment that Boxify provides and that can be installed as a regular app on stock Android (see Figure 5). This approach eliminates the need to modify the code of the untrusted application and works without altering the underlying OS (O1: ✔; O2: ✔). It thus constitutes the first solution that ports the concept of app virtualization to the stock Android OS.

The primary challenge for traditional application sandboxing solutions is to completely mediate and monitor all I/O between the sandboxed app and the system in order to restrict the untrusted code's privileges. The key insight for our Boxify approach is to leverage the security provided by *isolated processes* in order to isolate the untrusted code running within the context of Boxify by executing it in a completely de-privileged process that has no platform permissions, no access to the Android middleware, nor the ability to make persistent changes to the file system.

However, Android apps are tightly integrated within the application framework, e.g., for lifecycle

management and inter-component communication. With the restrictions of an isolated process in place, encapsulated apps are rendered dysfunctional. Thus, the key challenge for Boxify essentially shifts from constraining the capabilities of the untrusted app to now gradually permitting I/O operations in a controlled manner in order to securely re-integrate the isolated app into the software stack. To this end, Boxify creates two primary entities that run at different levels of privilege: A privileged controller process known as the Broker and one or more isolated processes called the Target (see Figure 5).

The Broker is the main Boxify application process and acts as a mandatory proxy for all I/O operations of the Target that require privileges beyond the ones of the isolated process. Thus, if the encapsulated app bypasses the Broker, it is limited to the extremely confined privilege set of its isolated process environment (*fail-safe defaults*; O4: ✔). As a consequence, the Broker is an ideal control-flow location in our Boxify design to implement a reference monitor for any privileged interaction between a Target and the system. Any syscalls and Android API calls from the Target that are forwarded to the Broker are evaluated against a security policy. Only policy-enabled calls are then executed by the Broker and their results returned to the Target process. To protect the Broker (and hence reference monitor) from malicious app code, it runs in a separate process under a different UID than the isolated processes. This establishes a strong security boundary between the reference monitor and the untrusted code (O3: ✔). To transparently forward the syscalls and Android API calls from the Target across the process boundary to the Broker, Boxify uses Android's Binder IPC mechanism. Finally, the Broker's responsibilities also include managing the application lifecycle of the Target and relaying ICC between a Target and other (Target) components.

The Target hosts all untrusted code that will run inside the sandbox. It consists of a shim that is able to dynamically load other Android applications and execute them. For the encapsulated app to interact with the system, it sets up interceptors that interpose system and middlware API calls. The interceptors do *not* form a security boundary but establish a compatibility layer when the code inside the sandbox needs to perform otherwise restricted I/O by forwarding the calls to the Broker. All resources that the Target process uses have to be acquired by the Broker and their handles duplicated into the Target process.

By encapsulating untrusted apps and interposing all their (privileged) I/O operations, Boxify is able to effectively enforce security- and privacy-protecting policies. Based on syscall interposition, Boxify has fine-grained control over network and filesystem operations. Intercepting Binder IPC enables the enforcement of security policies that were so far only achievable for OS extensions, but at application layer only.

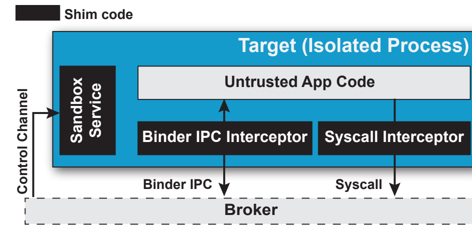


Figure 6: Components of a Target process.

Moreover, with this architecture, Boxify can provide a number of interesting novel features. Boxify is capable of monitoring multiple (untrusted) apps at the same time. By creating a number of Target processes, multiple apps can run in parallel yet securely isolated in a single instance of Boxify. Since the Broker fully controls all inter-component communication between the sandboxed apps, it is able to not only separate different apps from one another but also to allow controlled collaboration between them. Further, Boxify has the ability to execute apps that are not regularly installed on the phone: Since Boxify executes other apps by dynamically loading their code into one of its own processes and handles all the interaction between the sandboxed application and the OS, there is no need to register the untrusted app with the Android system. Hence, applications can be installed into, updated, or removed from Boxify without involving the `PackageInstaller` or having system privileges. A potential application of these features are application containers (e.g., enterprise app domain, see §5.4).

4.2 Target

The Target process contains four main entities (see Figure 6): The SandboxService (1) provides the Broker with a basic interface for starting and terminating apps in the sandbox. It is also responsible for setting up the interceptors for Binder IPC (2) and syscalls (3), which transparently forward calls issued by the untrusted application to the Broker.

1) SandboxService. Isolated processes on Android are realized as specifically tagged Service components (see §2). In Boxify each Target is implemented as such a tagged SandboxService component of the Boxify app. When a new Target should be spawned, a new, dedicated SandboxService is spawned. The Sand-

boxService provides an IPC interface that enables the Broker to communicate with the isolated process and to call two basic lifecycle operations for the Target: `prepare` and `terminate`. The Broker invokes the `prepare` function to initialize the sandbox environment for the execution of a hosted app. As part of this preparation, the Broker and Target exchange important configuration information for correct operation of the Target, such as app meta-information and Binder IPC handles that allow bi-directional IPC between Broker and Target. The `terminate` function shuts down the application running in the sandbox and terminates the Target process.

The biggest technical challenge at this point was *"How to execute another third-party application within the running isolated service process?"* Naïvely, one could consider, for instance, a warm-restart of the app process with the new application code using the `exec` syscall. However, we discovered that the most elegant and reliable solution is to have the Broker initially imitate the `ActivityManager` by instructing the Target process to load (i.e., *bind*) another application to its process and afterwards to relay any lifecycle events between the actual application framework and the newly loaded application in the Target process. The *bind* operation is supported by the standard Android application framework and used during normal app startup. The exact procedure is illustrated in Figure 7. The Broker first creates a new SandboxService process (❶), which executes with the privileges of an isolated process. This step actually involves multiple messages between the Broker process, the Target process and the system server, which we omitted here for the sake of readability. As a result, the Broker process receives a Binder handle to communicate with the newly spawned SandboxService. Next, the Broker uses this handle to instruct the SandboxService to prepare the loading of a sandboxed app (❷) by setting up the Binder IPC interceptor and syscall interceptor (using the meta-information given as parameters of the `prepare` call). The SandboxService returns the Binder handle to its `ApplicationThread` to the Broker. The application thread is the main thread of a process containing an Android runtime and is used by the `ActivityManager` to issue commands to Android application processes. At this point, the Broker emulates the behavior of the `ActivityManager` (❸) by instructing the `ApplicationThread` of the Target with the `bindApplication` call to load the target app into its Android runtime and start its execution. By default, it would be the `ActivityManagerService` as part of the application framework that uses this call to instruct newly forked and specialized Zygote processes to load and execute an application that

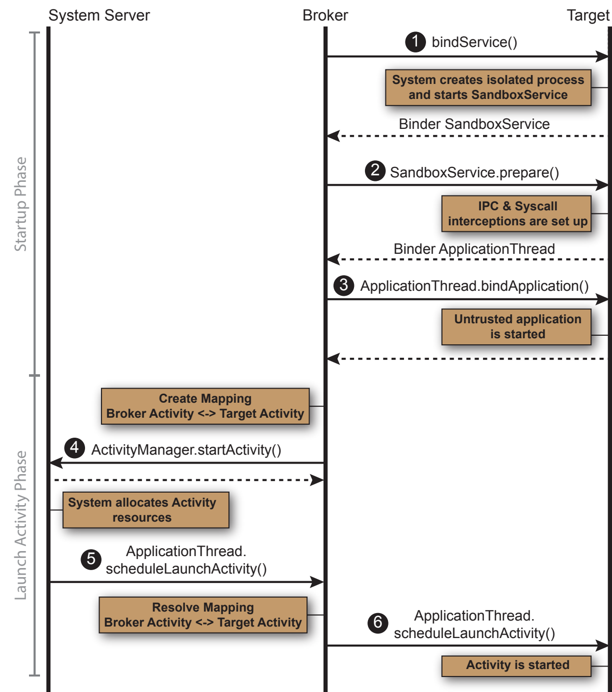


Figure 7: Process to load an app into a Target process and to launch one its Activities.

should be started. After this step, the sandboxed app is executing.

As an example how a sandboxed app can be used, we briefly explain how an Activity component of the sandboxed app can be launched, e.g., as result of clicking its entry in a launcher. As explained in §4.3, the Virtualization Layer creates a mapping from generic Boxify components to Target components. In this case, it maps the Activity component of Target to an Activity component of Boxify. The Broker requests the Activity launch from the `ActivityManager` in the `SystemServer` (❹), which allocates the required resources. After allocation, it schedules the launch of the Activity component by signaling the `ApplicationThread` of the targeted app (❺), which in this case is the Boxify app. Thus, the Virtualization Layer resolves the targeted Activity component and relays the signal to the corresponding Target process (❻).

2) Binder IPC Interceptor. Android applications use the Binder IPC mechanism to communicate with the (remote) components of other applications, including the application framework services and apps. In order to interact via Binder IPC with a remote component, apps must first acquire a Binder handle that connects them to the desired component.

To retrieve a Binder handle, applications query the ServiceManager, a central service registry, that allows clients to lookup system services by their common names. The ServiceManager is the core mechanism to bootstrap the communication of an application with the Android application framework. Binder handles to non-system services, such as services provided by other apps, can be acquired from the core framework services, most prominently the ActivityManager.

Boxify leverages this choke point in the Binder IPC interaction to efficiently intercept calls to the framework in order to redirect them to the Broker. To this end, Boxify replaces references to the ServiceManager handle in the memory of the Target process with references to the Binder handle of the Broker (as provided in the prepare function). These references are constrained to a few places and can be reliably modified using the Java Reflection API and native code. Consequently, all calls directed to the ServiceManager are redirected to the Broker process instead, which can then manipulate the returned Binder objects in such a way that any subsequent interactions with requested services are also redirected to the Broker. Furthermore, references to a few core system services, such as the ActivityManager and PackageManager, that are passed by default to new Android app runtimes, need to be replaced as well. By modifying only a small number of Binder handles, Boxify intercepts all Binder IPC communication. The technique is completely agnostic of the concrete interface of the redirected service and can be universally applied to all Binder interactions.

3) Syscall Interceptor. For system call interception, we rely on a technique called libc hooking (used, for instance, also in [59]). Applications use Android's implementation of the Standard C library *Bionic libc* to initiate system calls. With libc hooking, we efficently intercept calls to libc functions and redirect these calls to a service client running in the Target process. This client forwards the function calls via IPC to a custom service component running in the Broker. Due to space constraints, we refer to [7] for a detailed technical explanation of libc hooking.

In contrast to the IPC interception, which redirects all IPC communication to the Broker, the syscall interception is much more selective about which calls are forwarded: We do not redirect syscalls that would be anyway granted to an isolated process, because there is no security benefit from hooking these functions: a malicious app could simply remove the hook and would still succeed with the call. This exception applies to calls to read world-readable files and to

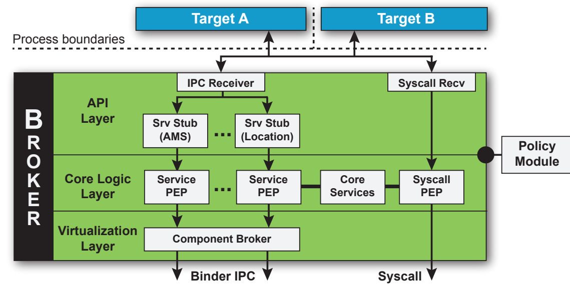


Figure 8: Architecture of the Broker.

most system calls that operate purely on file descriptors (e.g. read, write). Naturally, by omitting the indirection via our Broker, these exempted calls perform with native performance. However, Boxify still hooks calls that are security-critical and that are not permitted for isolated processes, such as system calls to perform file system operations (e.g. open, mkdir, unlink) and network I/O (socket, getaddrinfo). For a few calls, such as file operations, whose success depends on the given parameter, the syscall interception is parameter-sensitive in its decision whether or not to forward this operation to the Broker.

4.3 Broker

The Broker is the main application process of Boxify and is thus not subject to the restrictions imposed by the isolated process. It holds all platform permissions assigned to the Boxify app and can normally interact with the Android middleware. The Broker acts as a mandatory proxy for all interactions between the Target processes and the Android system and thus embodies the reference monitor of Boxify. These interactions are bi-directional: On the one hand, the untrusted app running in the Target process issues IPC and syscalls to the system; on the other hand, the Android middleware initiates IPC calls to Target (e.g., basic lifecycle operations) and the Broker has to dispatch these events to the correct Target.

The Broker is organized into three main layers (see Figure 8): The API Layer (4) abstracts from the concrete characteristics of the Android-internal IPC interfaces to provide compatibility across different Android versions. It bridges the semantic gap between the raw IPC transactions forwarded by the Target and the application framework semantics of the Core Logic Layer (5), which implements the fundamental mechanics of the virtual runtime environment that Boxify provides. All interaction with the system happens through the Virtualization Layer (6), which translates between the virtual environment inside of

Boxify and the Android system on the outside. In the following, we will look at every layer in more detail.

4) API Layer. The API Layer is responsible for receiving and unwrapping the redirected syscall parameters from the Syscall Interceptor in the Target and relaying them to the Core Logic Layer for monitoring and execution. More importantly, it transforms the raw Binder IPC parcels received from the IPC Interceptor into a representation agnostic of the Android version.

In order to (efficiently) sandbox applications at the Binder IPC boundary, Boxify must semantically interpret the intercepted Binder parcels. However, intercepted parcels are in a raw representation that consists only of native types that the kernel module supports and the sender marshalled all higher-level objects (e.g., Java classes) to this representation. This impedes an efficient sandboxing. To solve this problem, Boxify leverages the default Android toolchain for implementing Binder-based RPC protocols: To ensure that sender and receiver can actually communicate with each other, the receiver must know how to unmarshal the raw parcel data (exactly like Boxify). Android supports the developers in this process through the *Android Interface Definition Language* (AIDL), which allows definitions of Binder interfaces very similar to Java interfaces, including the names and signatures of remotely callable functions. The Android SDK toolchain generates the required boilerplate marshalling code from AIDL definitions both for the receiver (*Stub*) and the sender (*Proxy*). For system services, these Stubs are automatically generated during system build and Boxify uses the generated Stubs (which ship with Android OS and are conveniently accessible to third-party application) to unmarshal the raw Binder IPC parcel back to their application framework semantic (i.e., Java objects, etc). In essence, this allows us to generate the API layer of the Broker in an almost fully-automatic way for each Android version on which Boxify is deployed. Since Boxify is in full control of the Binder handles of the encapsulated app (i.e., calls to the `ServiceManager`, `ActivityManager`, etc.), it can efficiently determine which Binder handle of the app addresses which system service and hence which Stub must be used to correctly unmarshal the raw Binder parcel intercepted from each handle.

However, the exact structure of the unmarshalled data and the functions (name and signature) depend entirely on the AIDL file. Since the system service interfaces describe the internal Android API, these interfaces change frequently between Android versions. Hence Boxify would have to implement each possible version of a Stub for every available Android version. Since this Stub implementation, in contrast to the marshalling logic, can not be automated, this complicates efficient sandboxing of apps across multiple Android versions. Consequently, it is desirable to transform the unmarshalled IPC data into a version-agnostic representation and then implement each Stub once and for all for this version. To accomplish this in Boxify, we borrow ideas from Google's proprietary *SafeParcel* class: In contrast to the regular Binder parcel, the SafeParcel carries structural information about the data stored in it, which allows the receiver of an IPC request to selectively read parts of the payload without knowing its exact structure. We achieve the same result by transforming the version-dependent parcel into a version-agnostic key-value store (where keys are the parameter names of methods declared in the interface definitions) and adapting the Core Logic Layer and Stub implementations to work with these version-agnostic data stores. Thus, while the API layer is version-dependent and automatically generated for each Android version, the remaining layers of Broker are version-agnostic and implemented only once.

5) Core Logic Layer. The Core Logic Layer provides essential functionality required to run apps on Android by replicating a small subset of the functionality that Android's core system services provide. Most prominently, this layer provides a minimal implementation of the `PackageManager`, which manages the packages installed into the Boxify environment. Every call to a system service that is not emulated by the Core Logic Layer is passed on to the Virtualization Layer and thus to the underlying Android system. Other system services, such as the `LocationManager`, which are not necessarily required, can be instantiated at this layer as well, in case encapsulated apps are supposed to use the local, Boxify service implementation instead of the pristine Android service (e.g., servicing sandboxed apps with fake location data [64]). Hence, this layer decides whether an Android API call is emulated using a replicated service or forwarded to the system (through the Virtualization Layer). This layer is therefore responsible for managing the IPC communication between different sandboxed apps (abstractly like an *"ICC switch"*).

Furthermore, the Core Logic Layer implements the policy enforcement points (PEP) for Binder IPC services and syscalls. Because the API Layer already bridges the semantic gap between kernel-level IPC and Android application framework semantics, this removes the burden for dealing with low-level semantics in the IPC PEPs. We emulate the integration of enforcement points into pristine Android services by

integrating these points into our mandatory service proxies. This allows us to instantiate security models from the area of OS security extensions (see §3.2), but at the application layer. One default security model that Boxify provides is the permission enforcement and same origin model of Android. For instance, the replicated `ActivityManager` will enforce permissions on calls between components of two sandboxed apps. We present further security models from related work on OS security extensions that we integrated at this layer in §5.4 and for future work we consider a programmable interface for extending Core Logic Layer security in the spirit of ASM [37] and ASF [14]. For calls that are not protected by a permission, the Broker can also choose to enable direct communication between the target app and the requested Android system service. This can improve performance for non-critical services such as the `SurfaceFlinger` (for GUI updates) at the cost of losing the ability to mediate calls to these services.

The syscall PEP enforces system call policies in the spirit of [47] with respect to network and filesystem operations. Its responsibilities are twofold: First, it functions as a transparent compatibility layer by emulating the file-system structure of the Android data partition (e.g., `chroot` of sandboxed apps by emulating a home directory for each sandboxed app[4] within the home directory of the Boxify app). Second, it emulates the access control of the Linux kernel, i.e., compartmentalization of sandboxed apps by ensuring that they cannot access private files of other apps as well as enforcing permissions (e.g., preventing a sandboxed app without Internet permission from creating a network socket).

6) Virtualization Layer. The sandbox environment must support communication between sandboxed apps and the Android application framework, because certain system resources cannot be efficiently emulated (e.g., `SurfaceFlinger` for GUI) or not emulated at all (e.g., hardware resources like the camera). However, the sandbox must be transparent to the Target and all interaction with the application framework must appear as in a regular app. At the same time, the sandbox must be completely opaque to the application framework and sandboxed apps must be hidden from the framework; otherwise, this leads to inconsistencies that the framework considers as runtime (security) exceptions.

In Boxify, the Virtualization Layer is responsible for translating the bi-directional communication between the Android application framework and the Target. It achieves the required semi-transparent communication with a technique that can be abstractly described as *"ICC Network Address Translator"*: On outgoing calls from Target to framework, it ensures that all ICC appears as coming from the Boxify app instead of the sandboxed app. As described earlier, all Binder handles of a Target are substituted with handles of the Broker, which relays the calls to the system. During relay of calls, the Virtualization Layer manipulates the call arguments to hide components of sandboxed apps by substituting the component identifiers with identifiers of components of the Boxify app. On incoming calls from the framework, the Virtualization Layer substitutes the addressed Boxify component with the actually addressed component of the sandboxed app and dispatches the call. In order to correctly substitute addressed components, the Virtualization Layer maintains a mapping between Target and Boxify component names, or in case the Target component is not addressed by a name but a Binder handle that was given prior to the framework, the mapping is between the released Binder handle and its owning Target component.

A concrete example where this technique is applied is requesting the launch of a Target Activity component from the application framework (see Figure 7). The Virtualization Layer substitutes the Activity component with a generic Activity component of Boxify if a call to the `ActivityManager` occurs. When the service calls back for scheduling the Activity launch, the Virtualization Layer dispatches the scheduling call to the corresponding Target Activity component.

Lastly, we hook the application runtime of Boxify's Broker process (using a technique similar to [55]) in order to gain control over the processing of incoming Binder parcels. This enables the Broker to distinguish between parcels addressed to Boxify itself and those that need to be forwarded to the Target processes.

4.4 System Integration

Lastly, we discuss some aspects of integrating sandboxed apps into the default application framework.

Launcher. Since sandboxed apps have to be started through Boxify (and are not regularly installed on the system), they cannot be directly launched from the default launcher. A straightforward solution is to provide a custom launcher with Boxify in form of a dedicated Activity. Alternatively, Boxify could register as a launcher app and then run the default launcher (or any launcher app of the user's choice) in the sandbox, presenting the union of the regularly installed apps and apps installed in the sandbox environment; or Boxify launcher widgets could be placed

[4]Recall that sandboxed apps are not installed in the system but only in the Boxify environment, and hence do not have a native home directory.

Table 2: Microbenchmarks Middleware (200 runs)

API Call	Native	on Boxify	Overhead
`Open Camera`	103.24 ms	104.48 ms	1.24ms (1.2%)
`Query Contacts`	7.63 ms	8.55 ms	0.92 ms (12.0%)
`Insert Contacts`	66.49 ms	67.51 ms	1.02 ms (1.5%)
`Delete Contacts`	75.86 ms	76.81 ms	0.95 ms (0.9%)
`Create Socket`	120.83 ms	121.58 ms	0.75 ms (0.6%)

on the regular home screen to launch sandboxed apps from there.

App stores. Particularly smooth is the integration of Boxify with app store applications, such as the Google Play Store. Since no special permissions are required to install apps into the sandbox, we can simply run the store apps provided by Google, vendors, and third-parties in Boxify to install new apps there. For example, clicking install in the sandboxed Play Store App will directly install the new app into Boxify. Furthermore, Play Store (and vendor stores) even take care of automatically updating all apps installed in Boxify, a feature that IRM systems have to manually re-implement.

Statically registered resources. Some resources of apps are statically registered in the system during app installation. Since sandboxed apps are not regularly installed, the system is unaware of their resources. This concerns in particular Activity components that can receive Intents for, e.g., content sharing, or package resources like icons. However, some resources like Broadcast Receiver components can be dynamically registered at runtime and Boxify uses this as a workaround to dynamically register the Receivers declared statically in the Manifests of sandboxed apps.

5 Evaluation

We discuss the prototypical implementation of Boxify in terms of performance impact, security guarantees, and app robustness, and present concrete use-cases of Boxify. Our prototype comprises 11,901 lines of Java code, of which 4,242 LoC are automatically generated (API Layer), and 3,550 lines of additional C/C++ code. All tests described in the following were performed on an LG Nexus 5 running Android 4.4.4, which is currently the most widely used version in the Android ecosystem.

5.1 Performance Impact

To evaluate the performance impact of Boxify on monitored apps, we compare the results of common benchmark apps and of custom micro-benchmarks for encapsulated and native execution of apps.

Table 3: Microbenchmarks Syscalls (15k runs)

Libc Func.	Native	on Boxify	Overhead
`create`	47.2 μs	162.4 μs	115.2 μs
`open`	9.5 μs	122.7 μs	113.2 μs
`remove`	49.5 μs	159.6 μs	110.1 μs
`mkdir`	88.4 μs	199.4 μs	111.0 μs
`rmdir`	71.2 μs	180.7 μs	109.5 μs

Table 4: Benchmark Tools (10 runs)

Tool	Native	on Boxify	Loss
CF Bench v1.3	16082 Pts	15376 Pts	4.3%
Geekbench v3.3.1	1649 Pts	1621 Pts	1.6%
PassMark v1.0.4	3674 Pts	3497 Pts	4.8%
Quadrant v2.1.1	7820 Pts	7532 Pts	3.6%

Table 2 and Table 3 present the results of our micro-benchmarks for common Android API calls and for syscall performance. Intercepting calls to the application framework imposes an overhead around 1%, with the exception of the very fast `Query Contacts` (12%). For syscalls, we measured the performance of calls that request file descriptors for file I/O in private app directories (or external storage) and that are proxied by the Broker. We observe a constant performance overhead of $\approx 100\mu s$, which corresponds to the required time of the additional IPC round trip for the communication with the Broker on our test platform. However, the syscall benchmarks depict a worst-case estimation: The overall performance impact on apps is much lower, since high-frequent follow up operations on acquired file descriptors (e.g., `read`/`write`) need not to be intercepted and therefore run with native speed. We measured the overall performance penalty by excecuting several benchmarking apps on top of Boxify, which show an acceptable performance degradation of 1.6%–4.8% (see Table 4).

5.2 Runtime Robustness

To assess the robustness of encapsulated apps, we executed 1079 of the most popular, free apps from Google Play (retrieved in August 2014) on top of Boxify. For each sandboxed app we used the *monkeyrunner* tool[5] to exercise the app's functionality by injecting 500 random UI events. From the 1079 apps, 93 (8.6%) experienced a crash during testing. Manual investigation of the dysfunctional apps revealed

[5] `http://developer.android.com/tools/help/monkeyrunner_concepts.html`

Table 5: Android versions supported by Boxify.

Version	< 4.1	4.1	4.2	4.3	4.4	5.0	5.1
Supported	✘†	✔	✔	✔	✔	✔	✔

✔: supported; ✘: not supported
†: no *isolated proccess*

that most errors were caused by apps executing exotic syscalls or rarely used Android APIs which are not covered by Boxify yet and thus fail due to the lack of privileges of the Target process (fail-safe defaults). This leads to a slightly lower robustness than reported for related work (e.g., [59, 15]) where bypassed hooks do not cause the untrusted app to crash but instead silently circumvent the reference monitor. The remaining issues were due to unusual application logic that relies on certain OS features (e.g., the process information pseudo-filesystem `proc`), which the current prototype of Boxify does not yet support. However, all of these are technical and not conceptual shortcomings of the current implementation of Boxify.

5.3 Portability

Table 5 summarizes the Android versions currently supported by our prototypical Boxify implementation. Our prototype supports all Android versions 4.1 through 5.1 and can be deployed on nine out of ten devices in the Android ecosystem [1]. Android versions prior to 4.1 are not supported due to the lack of the *isolated process* feature.

5.4 Use-cases

Boxify allows the instantiation of different security models from the literature on Android security extensions. In the following, we present two selected use-cases on fine-grained permission control and domain isolation that have received attention before in the security community.

Fine-Grained Permission Control. The TISSA [64] OS extension empowers users to flexibly control in a fine-grained manner which personal information will be accessible to applications. We reimplemented the TISSA functionality as an extension to the Core Logic Layer of the Boxify Broker. To this end, we instrumented the mandatory proxies for core system services (e.g. `LocationManager`, `TelephonyService`) so that they can return a filtered or mock data set based on the user's privacy settings. Users can dynamically adjust their privacy preferences through a management Activity added to Boxify. In total, the TISSA functionality required additional 351 lines of Java code to Core Logic Layer.

Domain Isolation. Particularly for enterprise deployments, container solutions have been brought forward to separate business apps from other (untrusted) apps [56, 17, 53].

We implemented a domain isolation solution based on Boxify by installing business apps into the sandbox environment. The Broker provides its own version of the `PackageManager` to directly deliver intercomponent communication to sandboxed applications without involving the regular `PackageManager`, enabling controlled collaboration between enterprise apps while at the same time isolating and hiding them from non-enterprise apps and the OS.

To separate the enterprise data from the user's private data, we exploit that the Broker is able to run separate instances of system services (e.g., Contacts, Calendar) within the sandbox. Our custom `ActivityManager` proxy now selectively and transparently redirects `ContentProvider` accesses by enterprise apps to the sandboxed counterparts of those providers.

Alternatively, the above described domain isolation concept was used to implement a privacy mode for end users, where untrusted apps are installed into a Boxify environment with empty (or faked) system ContentProviders. Thus, users can test untrusted apps in a safe environment without risking harm to their mobile device or private data. The domain isolation extension required 986 additional lines of code in the Core Logic Layer of Boxify.

5.5 Security Discussion

Our solution builds on *isolated processes* as fundamental security primitive. An isolated process is the most restrictive execution environment that stock Android currently has to offer, and it provides Boxify with better security guarantees than closest related work [38]. In what follows, we identify different security shortcomings and discuss potential future security primitives of stock Android that would benefit Boxify and defensively programmed apps in general.

Privilege escalation. A malicious app could bypass the syscall and IPC interceptors, for instance, by statically linking `libc`. For IPC, this does not lead to a privilege escalation, since the application framework apps and services will refuse to cooperate with an isolated process. However, the kernel is unaware of the concept of an "isolated process" and will enforce access control on syscalls according to the process' UID. Although the transient UIDs of isolated

processes are very restricted in their filesystem access (i.e., only world readable/writable files), a malicious process has the entire kernel API as an attack vector and might escalate its privileges through a root or kernel exploit. In this sense, Boxify is not more secure than existing approaches that rely on the assumption that the stock Android kernel is hardened against root and kernel exploits.

To remedy this situation, additional layers of security could be provided by the underlying kernel to further restrict untrusted processes. This is common practice on other operating systems, e.g., on modern Linux distributions, where Chromium—the primary user of isolated process on Android—uses the *seccomp-bpf* facility to selectively disable syscalls of renderer processes and we expect this facility to become available on future Android versions with newer kernels. Similarly, common program tracing facilities could be made available in order to interpose syscalls more securely and efficiently [31, 47, 52].

Violating Least-Privilege Principle. The Broker must hold the union set of all permissions required by the apps hosted by Boxify in order to successfully proxy calls to the Android API. Since it is hard to predict a reasonable set of permissions beforehand, this means that the Broker usually holds all available permissions. This contradicts the principle of least privilege and makes the Broker an attractive target for the encapsulated app to increase its permission set. A very elegant solution to this problem would be a Broker that drops all unnecessary permissions. This resembles the privilege separation pattern [48, 57] of established Linux services like ssh, which drop privileges of sub-processes based on setting their UIDs, capabilities, or transitioning them to seccomp mode. Unfortunately, Android does not (yet) provide a way to *selectively* drop permissions *at runtime*.

Red Pill. Even though Boxify is designed to be invisible to the sandboxed app, it cannot exclude that the untrusted app gathers information about its execution environment that allow the app to deduce that it is sandboxed (e.g., checking its runtime UID or permissions). A malicious app can use this knowledge to change its runtime behavior when being sandboxed and thus hide its true intentions or refuse to run in a sandboxed environment. Prevention of this information leak is an arms race that a resolute attacker will typically win. However, while this might lead to refused functionality, it cannot be used to escalate the app's privileges.

6 Conclusion

We presented the first application virtualization solution for the stock Android OS. By building on isolated processes to restrict privileges of untrusted apps and introducing a novel app virtualization environment, we combine the strong security guarantees of OS security extensions with the deployability of application layer solutions. We implemented our solution as a regular Android app called Boxify and demonstrated its capability to enforce established security policies without incurring significant runtime performance overhead.

Availability and Future Work. We will make the Boxify source code freely available. Beyond the immediate privacy benefits for the end-user presented in this paper (see §5.4), Boxify offers all the security advantages of traditional sandboxing techniques and is thus of independent interest for future Android security research. As future work, we are currently investigating different application domains of Boxify, such as application-layer only taint-tracking for sandboxed apps [24], programmable security APIs in the spirit of ASM [37]/ASF [14] to facilitate the extensibility of Boxify, as well as Boxify-based malware analysis tools.

References

[1] Android developer dashboard. `https://developer.android.com/about/dashboards/`. Last visited: 06/20/15.

[2] Android developer's guide. `http://developer.android.com/guide/index.html`. Last visited: 02/19/15.

[3] Chromium: Linux sandboxing. `https://code.google.com/p/chromium/wiki/LinuxSandboxing`. Last visited: 02/10/15.

[4] Cyanogenmod. `http://www.cyanogenmod.org`.

[5] Java SE Documentation: Security Specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-specTOC.fm.html`. Last visited: 02/10/15.

[6] OmniROM. `http://omnirom.org`. Last visited: 02/19/15.

[7] Redirecting functions in shared elf libraries. `http://www.codeproject.com/Articles/70302/Redirecting-functions-in-shared-ELF-libraries`.

[8] The Chromium Projects: OSX Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`. Last visited: 02/10/15.

[9] The Chromium Projects: Sandbox (Windows). `http://www.chromium.org/developers/design-documents/sandbox`. Last visited: 02/10/15.

[10] Wine: Run Windows applications on Linux, BSD, Solaris and Mac OS X. `https://www.winehq.org`. Last visited: 02/13/15.

[11] Zero-Permission Android Applications. https://www.leviathansecurity.com/blog/zero-permission-android-applications/. Last visited: 02/11/15.

[12] Zero-Permission Android Applications (Part 2). http://www.leviathansecurity.com/blog/zero-permission-android-applications-part-2/. Last visited: 02/11/15.

[13] Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. Cells: A virtual mobile smartphone architecture. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (2011), ACM.

[14] Backes, M., Bugiel, S., Gerling, S., and von Styp-Rekowsky, P. Android Security Framework: Extensible multi-layered access control on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014), ACM.

[15] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard - enforcing user requirements on Android apps. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)* (2013), Springer.

[16] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[17] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and lightweight domain isolation on Android. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)* (2011), ACM.

[18] Cai, L., and Chen, H. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proc. 6th USENIX conference on Hot topics in security (HotSec'11)* (2011), USENIX Association.

[19] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. 23rd USENIX Security Symposium (SEC'14)* (2014), USENIX Association.

[20] Citrix. Xenapp. http://www.citrix.com/products/xenapp/how-it-works/application-virtualization.html. Last visited: 02/13/15.

[21] Conti, M., Nguyen, V. T. N., and Crispo, B. CRePE: Context-Related Policy Enforcement for Android. In *Proc. 13th International Conference on Information Security (ISC'10)* (2010).

[22] Davis, B., and Chen, H. Retroskeleton: Retrofitting android apps. In *Proc. 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'13)* (2013), ACM.

[23] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proc. Mobile Security Technologies 2012 (MoST'12)* (2012), IEEE Computer Society.

[24] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)* (2010), pp. 393–407.

[25] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In *Proc. 20th USENIX Security Symposium (SEC'11)* (2011), USENIX Association.

[26] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proc. 16th ACM Conference on Computer and Communication Security (CCS'09)* (2009), ACM.

[27] Enck, W., Ongtang, M., and McDaniel, P. Understanding android security. *IEEE Security and Privacy 7*, 1 (2009), 50–57.

[28] Erlingsson, Ú. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, January 2004.

[29] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[30] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *Proc. 20th USENIX Security Symposium (SEC'11)* (2011), USENIX Association.

[31] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proc. 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography (SSYM'96)* (1996), USENIX Association.

[32] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)* (2012), ACM.

[33] Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[34] Guo, P. J., and Engler, D. Cde: Using system call interposition to automatically create portable software packages. In *Proc. 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)* (2011), USENIX Association.

[35] Hackborn, D. Android Developer Group: Advantage of introducing Isolatedprocess tag within Services in JellyBean. https://groups.google.com/forum/?fromgroups=#!topic/android-developers/pk45eUFmKcM, 2012. Last visited: 02/19/15.

[36] Hao, H., Singh, V., and Du, W. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)* (2013), ACM.

[37] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. ASM: A Programmable Interface for Extending Android Security. In *Proc. 23rd USENIX Security Symposium (SEC'14)* (2014), USENIX Association.

[38] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proc. 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)* (2012), ACM.

[39] KASPERSKY LAB, AND INTERPOL. Mobile cyber-threats. http://securelist.com/analysis/publications/66978/mobile-cyber-threats-a-joint-study-by-kaspersky-lab-and-interpol/, 2014. Last visited: 02/19/15.

[40] LANGE, M., LIEBERGELD, S., LACKORZYNSKI, A., WARG, A., AND PETER, M. L4android: A generic operating system framework for secure smartphones. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)* (2011), ACM.

[41] MICROSOFT. Application Virtualization (App-V). http://www.microsoft.com/en-us/windows/enterprise/products-and-technologies/mdop/app-v.aspx. Last visited: 02/13/15.

[42] MOULU, A. Android OEM's applications (in)security and backdoors without permission. http://www.quarkslab.com/dl/Android-OEM-applications-insecurity-and-backdoors-without-permission.pdf. Last visited: 02/19/15.

[43] MULLINER, C., OBERHEIDE, J., ROBERTSON, W., AND KIRDA, E. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proc. 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013), ACM.

[44] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS'10)* (2010), ACM.

[45] ONGTANG, M., MCLAUGHLIN, S. E., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proc. 25th Annual Computer Security Applications Conference (ACSAC'09)* (2009), ACM.

[46] OPEN SIGNAL. Android Fragmentation Visualized (July 2013). http://opensignal.com/reports/fragmentation-2013/. Last visited: 02/06/2015.

[47] PROVOS, N. Improving host security with system call policies. In *Proc. 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)* (2003), USENIX Association.

[48] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proc. 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)* (2003), USENIX Association.

[49] RASTHOFER, S., ARZT, S., LOVAT, E., AND BODDEN, E. DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android. In *Proc. 9th International Conference on Availability, Reliability and Security (ARES'14)* (2014), IEEE Computer Society.

[50] RASTOGI, V., CHEN, Y., AND JIANG, X. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)* (2013), ACM.

[51] RUSSELLO, G., CONTI, M., CRISPO, B., AND FERNANDES, E. MOSES: supporting operation modes on smartphones. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12)* (2012), ACM.

[52] RUSSELLO, G., JIMENEZ, A. B., NADERI, H., AND VAN DER MARK, W. FireDroid: Hardening Security in Almost-stock Android. In *Proc. 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013), ACM.

[53] SAMSUNG ELECTRONICS. White paper: An overview of samsung KNOX. http://www.samsung.com/se/business-images/resource/2013/samsung-knox-an-overview/%7B3%7D/Samsung_KNOX_whitepaper-0-0-0.pdf, 2013. Last visited: 02/19/15.

[54] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)* (2013), The Internet Society.

[55] VON STYP-REKOWSKY, P., GERLING, S., BACKES, M., AND HAMMER, C. Idea: Callee-site rewriting of sealed system libraries. In *Proc. 5th International Symposium on Engineering Secure Software and Systems (ESSoS'13)* (2013), Springer.

[56] WANGY, X., SUN, K., AND JING, Y. W. J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15)* (2015), The Internet Society.

[57] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for unix. In *Proc. 19th USENIX Security Symposium (SEC'10)* (2010), USENIX Association.

[58] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., AND JIANG, X. Airbag: Boosting smartphone resistance to malware infection. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (2014), The Internet Society.

[59] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium – Practical Policy Enforcement for Android Applications. In *Proc. 21st USENIX Security Symposium (SEC'12)* (2012), USENIX Association.

[60] XU, Z., BAI, K., AND ZHU, S. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)* (2012), ACM.

[61] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Symposium on Security and Privacy (Oakland'09)* (2009), IEEE Computer Society.

[62] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *Proc. 33rd IEEE Symposium on Security and Privacy (Oakland'12)* (2012), IEEE Computer Society.

[63] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[64] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. Taming information-stealing smartphone applications (on Android). In *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST'11)* (2011), Springer.