Boxify
Bringing Full-Fledged App Sandboxing to Stock Android

MICHAEL BACKES, SVEN BUGIEL, OLIVER SCHRANZ, AND PHILIPP VON STYP-REKOWSKY

Boxify is the first concept for full-fledged app sandboxing on stock Android. Building on app virtualization and process-based privilege separation, Boxify eliminates the necessity to modify the code of monitored apps or the device’s firmware and thereby overcomes the existing legal concerns and deployment problems of prior approaches. As such, Boxify is a powerful tool for realizing privacy-protecting solutions on Android. In this article, we explain the Boxify concept and illustrate how it can benefit users in protecting their privacy on Android.

Background

Smart devices, like smartphones and tablets, in conjunction with the plethora of available apps, are very convenient companions in our daily tasks; they are our social hub to stay in touch with our friends and colleagues, help us organize our day, have replaced our digital cameras, and are our online banking portal or navigation system, among many more uses. From a privacy perspective, however, the data protection mechanisms in place on those platforms do not do justice to the rich functionality and data wealth that those platforms offer to apps. Although all popular platforms (like Apple’s iOS and Google’s Android) support permissions—that is user-granted privileges an app must hold to access user data and system resources [5]—permissions have been shown to be futile in creating more transparency of app behavior for users and in effectively protecting the users’ privacy [3, 8, 9, 10, 13].

First, permissions do not communicate how apps are actually (ab)using their granted privileges but only what apps could potentially do. For instance, let’s consider an on-demand transportation app that requests access to the user’s SMS—access also commonly requested by banking trojans to intercept TAN (transaction authentication number) messages—how can the user distinguish whether this access has benign or malicious intent?

Second, permissions are too coarsely defined, are not conditional, and hence cause apps to be overprivileged—a direct contradiction of the principle of least privilege that permissions were originally intended to realize. Picking up the example of the transportation app, it is reasonable why such an app would need access to some of the user’s address book data (e.g., a contact’s address to quickly choose a destination) but not all data (e.g., why share the contact’s email addresses and phone numbers?). And why should the app be able to always access those data and not only in the context of selecting a transportation destination?

Third, permissions apply to application sandboxes as a whole, including all third party libs included in the application, such as analytics and advertisement libraries. Currently, it is opaque to the user which security principal (app or lib) is leveraging the app’s privileges. This entangled trust relationship between user, app developer, and included libs can be abused by third party code. In fact, ad libs on Android have demonstrated dubious behavior [4, 6] that actively exploits their host app’s privileges to violate the user’s privacy by, for example, exfiltrating private information or even dynamically loading untrusted code.
Lastly, even when the user is well informed about the apps’ behavior (e.g., through app descriptions, reviews, or developer Web sites), she has very limited means to adjust the permissions to her own privacy preferences. Only very few selected permissions can be dynamically revoked by the user—on several platform versions of iOS and on Android 6—but many other privacy-critical permissions cannot be revoked or fine-grained data filtering enabled.

**Full-Fledged App Sandboxing on Stock Android**

What is required to improve on this situation and to shift the balance of power in favor of the users is an application sandbox controlled by the user and capable of enforcing user-defined privacy policies by reliably monitoring any interaction between apps and the Android system. The sandbox solution, additionally, must be easy to install on stock Android (e.g., as an app) and must refrain from modifying the underlying platform, that is, no jailbreak/rooting or reinstallation of the operating system, which entails unlocking the device and, usually, data loss and which forms a technical barrier for most end users.

With Boxify [1] we present a novel concept for Android app sandboxing that fills this gap. Boxify is based on app virtualization that provides full-fledged app sandboxing on stock Android devices. Boxify provides secure access control on apps without the need for firmware alterations, root privileges, or modifications of confined 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. By introducing a novel app virtualization environment that intercepts all interactions between the app and the system, Boxify is able to enforce established and new privacy-protecting policies on third party apps.

Additionally, the virtualization environment 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 leveraging on-board security features of stock Android, we ensure that the kernel securely and automatically isolates at process-level the sandbox app from the untrusted processes to prevent compromise of the policy enforcement code.

Boxify is realized as a regular app that can be deployed on stock Android versions higher than v4.1 and causes only a negligible performance penalty for confined apps (1.6–4.8% in benchmark apps). To the best of our knowledge, Boxify is the first solution to introduce application virtualization to stock Android.

In the remainder of this article, we outline the Boxify concept and how it can benefit users to efficiently and securely protect their privacy. For a full technical description of our solution, we refer the interested reader to our USENIX Security ’15 conference paper [1].

---

**Boxify Architecture**

Boxify sandboxes Android apps by dynamically loading and executing untrusted apps in one of its own processes. Thus, the untrusted application is not executed by the Android system itself but runs completely encapsulated within the runtime environment that Boxify provides (see Figure 1). This approach eliminates the need to modify the code of the untrusted application and works without altering the underlying OS, hence facilitating easy deployment by end users.

Boxify leverages the security provided by an on-board security feature of stock Android, called isolated processes, in order to isolate the untrusted code running within the context of Boxify by executing such code in a completely de-privileged process that has no permissions, very limited file system access, and highly constrained inter-application communication. However, Android apps are tightly integrated within the Android application framework (e.g., for application lifecycle management). 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 reintegrate the isolated app into the software stack. This reintegration is subject to privacy policies that govern very precisely to which functionality and data the untrusted app regains access (e.g., privacy-relevant services like location or filtering of data—like address book entries). 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(s).

**Target**

The Target hosts all untrusted code that will run inside the sandbox (see Figure 2). Each Target consists of a shim (SandboxService) that is able to dynamically load other Android services.

---

**Figure 1:** Architecture overview of Boxify
Boxify: Bringing Full-Fledged App Sandboxing to Stock Android

![Diagram](image)

**Figure 2:** Components of a Target process

**Shim code**

Target (Isolated Process)

Untrusted App Code

Binder IPC Interceptor

Syscall Interceptor

Binder IPC

Broker

Sandbox Service

Control Channel

All interaction between the app and the Android application framework occurs via IPC, and the Binder IPC Interceptor redirects all calls from the app to the application framework and other apps to the Broker. The IPC Interceptor does so quite efficiently by replacing all references to a central IPC service registry (ServiceManager) in the memory of the Target process with references to the IPC interface of the Broker process. Consequently, all calls directed to the ServiceManager are redirected to the Broker process instead, which acts as a proxy to the application framework and ensures that all subsequent interactions by the untrusted app with requested Android services are redirected to the Broker as well.

For system call interception, we rely on a technique called libc hooking. Android is a Linux-based software stack and ships with a custom C library, called Bionic, that acts as a wrapper around the underlying Linux kernel’s system call interface. Bionic is used by default by Android user-space processes, including application processes, to issue system calls to the kernel. By hooking Bionic’s libc, Boxify can efficiently 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.

It is important to notice that both interceptors, IPC and system calls, 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.

**Broker**

The Broker is the main Boxify application process and acts as a mandatory, bi-directional proxy for all interactions between a Target and the system. On the one hand, the Broker relays 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, the app is limited to the extremely confined privilege set of its isolated process environment (fail-safe defaults). As a consequence, the Broker is an ideal control-flow location in our Boxify design to implement a policy enforcement point (reference monitor). To protect the Broker (and hence reference monitor) from malicious app code, the Broker 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. On the other hand, the Broker dispatches IPC calls initiated by the system (e.g., basic lifecycle operations) to the correct Target.

The Broker is organized into three layers (see Figure 3): the API Layer abstracts from the concrete characteristics of the Android service IPC interfaces to provide compatibility across different Android versions. To this end, the API Layer bridges the semantic gap between the raw IPC transactions forwarded by the Target and the application framework semantics of the other layers in the Broker by transforming the raw Binder IPC parcels back into their high-level Java objects representation. This also facilitates the definition of more convenient and meaningful privacy policies.

The Core Logic Layer replicates a small subset of the functionality that Android’s core system services provide. Further, this layer decides whether an Android API call is emulated using a replicated service (e.g., a mock location service) or forwarded to the pristine Android service (through the Virtualization Layer). The Core Logic Layer is therefore responsible for managing the IPC communication between different sandboxed apps (abstractly like an “IPC switch”). Furthermore, this layer implements the policy enforcement points for Binder IPC services and syscalls. We emulate the integration of enforcement points into pristine Android services by integrating these points into our mandatory service proxies in the Core Logic Layer. This allows us to instantiate security and privacy solutions (including the default Android permissions) from the area of OS security extensions, but at the application layer. Similarly, syscall policy enforcement points enforce system call policies on network and file-system operations.

The Virtualization Layer is responsible for translating the bi-directional communication between the Android application framework and the Target. This technique can be abstractly best described as an IPC Network Address Translator. Thus, the sandbox is transparent to the Target, and all interaction with the application framework from the Target’s perspective appears as in any regular, non-virtualized app. At the same time, the sandbox is completely opaque to the application framework, and sandboxed apps are hidden from the framework and other regular installed apps, which can only detect the presence of the Boxify app.
Boxify: Bringing Full-Fledged App Sandboxing to Stock Android

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

Fine-Grained Permission Control
The TISSA [14] OS security 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. This brings TISSA’s enforcement strategy to Android as an application layer-only solution that does not require the user to exchange or alter her device’s firmware. To this end, we instrumented the mandatory proxies for core system services (e.g., location or telephony service) 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 UI added to Boxify. For instance, the transportation app from our motivating example could be restricted to the user’s contacts’ street addresses and barred from accessing their email addresses, phone numbers, etc. Similarly, access to the SMS data can be completely removed by returning only empty data sets to queries or be fine-grained, restricted to only SMS from whitelisted phone numbers (e.g., the transportation company’s service numbers).

Separation of Advertisements
For monetarization of apps, app developers frequently bundle their apps with advertisement libraries. However, those libraries have been shown to exhibit very dubious and even dangerous behavior for the user’s privacy [6]. To better protect the user from those unsafe practices, technical solutions [7, 11] for privilege separation have been brought forward that retrofit the Android middleware to isolate advertising libraries from their host apps and then subjugate them to a separate privacy policy.

In this spirit, we instantiate a similar solution on Boxify at application layer that extracts advertising libraries from apps, executes them in a separate Target, and reintegrates them with their host app through IPC-based inter-app communication via the Boxify Core Logic Layer. This is possible, since advertising libraries are by default only loosely coupled with their host application code. As a result, separate privacy policies can be applied to the ad lib sandbox on Boxify (e.g., preventing the ad lib from exfiltrating private information). The same technique for extracting advertising libraries from their host apps can even be applied to remove the advertising libraries in their entirety from apps (i.e., ad blocking).

Domain Isolation
Particularly for enterprise deployments, container solutions have been brought forward to separate business apps from other (untrusted) apps [2, 12]. We implemented a domain isolation solution based on Boxify by installing business apps into the sandbox environment. The Core Logic Layer of Boxify enables a controlled collaboration between enterprise apps, while at the same time isolating and hiding them from non-enterprise apps outside of Boxify.

To separate the enterprise data from the user’s private data, we take advantage of the Broker’s ability to run separate instances of system services (e.g., address book, calendar) within the sandbox. Our Core Logic Layer selectively and transparently redirects data accesses by enterprise apps to the sandboxed counterparts of those providers, thus ensuring that the data is not written to publicly available resources that can be accessed by non-enterprise apps (e.g., the default address book or calendar of Android).

Alternatively, the above described domain isolation concept can be used to implement a privacy mode for end users, where untrusted apps are installed into a Boxify environment with system services that return empty (or fake) data sets, such as location, address book, etc. Thus, users can test untrusted apps in a safe environment without risking harm to their private data.

Security Discussion
Lastly, we identify different security shortcomings of Boxify 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, for syscalls, a malicious process has the entire kernel API as an attack vector and might escalate its privileges through a root or kernel exploit.
To remedy this situation, additional layers of security could be provided by the underlying kernel to further restrict untrusted processes, e.g., program tracing or seccomp(). This is common practice on other operating systems, and we expect such facilities to become available on future Android versions with newer kernels.

**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. Because it is hard to predict a reasonable set of permissions beforehand, the Broker usually holds all available permissions. This makes the Broker an attractive target for attacks. A very elegant solution to this problem would be a Broker that drops all unnecessary permissions. 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 the untrusted app from gathering information about its execution environment that allows the app to deduce that it is being sandboxed. A malicious app can use this knowledge to change its runtime behavior when being sandboxed and thus hide its true intentions, or it can refuse to run in a sandboxed environment. While this might lead to refused functionality, it cannot be used to escalate the app’s privileges.

**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 by 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 and privacy policies without incurring significant runtime performance overhead. To make Boxify more accessible to security engineers, future work will investigate programmable security APIs that allow instantiation of various use cases in the form of code modules rather than patches to Boxify.

**Availability**

The IP and the corresponding patent of the Boxify technology are owned by the company Backes SRT, which plans to make a noncommercial version of Boxify for academic research available on their Web site (www.backes-srt.com).

**Acknowledgments**

This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy, and Accountability (CISPA).
Boxify: Bringing Full-Fledged App Sandboxing to Stock Android

References


