AppBox: A Black-Box Application Sandboxing Technique for Mobile App Management Solutions

Maqsood Ahmad *Univ. di Trento, IT*
Francesco Bergadano *Univ. di Torino, IT*
Valerio Costamagna *Univ. di Torino, IT*
Bruno Crispo *Univ. di Trento, IT*
Giovanni Russello *Univ. of Auckland, NZ*

***Abstract*— Several Mobile Device Management (MDM) and Mobile Application Management (MAM) services have been launched on the market. However, these services suffer from two important limitations: reduced granularity and need for app developers to include third party SDKs. We present AppBox , a novel black-box app-sandboxing solution for app customisation for stock Android devices. AppBox enables enterprises to select any app, even highly-obfuscated, from any market and perform a set of target customisations by means of fine-grained security policies. We have implemented and tested AppBox on various smartphones and Android versions. The evaluation shows that AppBox can effectively enforce fine-grained policies on a wide set of existing apps, with an acceptable overhead.**

***Index Terms*—Android Security, App Sandbox, MAM**

I. Introduction

In the past decade, expensive company-owned laptops have been replaced by cheaper devices often owned by the employees, enabling the so called Bring Your Own Device (BYOD) paradigm, and applications are being rewritten w.r.t. this new context. In this scenario, it is important for enterprises to be able to configure secure policies for its employees' devices. Mobile Device Management (MDM) and Mobile App Management (MAM) services are the $de\ facto$ solutions to enforce such enterprise policies on mobile devices.

MDMs enforce policies only at the device level, while MAMs are app and/or employee-specific. MAMs provide Software Development Kits (SDKs), enabling developers to customise their apps. In this paper, we propose **AppBox**, a novel application sandboxing technique to enable an enterprise to select any app from the market and enforce any chosen security policy with minimum collaboration from the app developer. The developer will not have to disclose the app source code to the enterprise nor should he be involved with code customisations. Our contributions are:

1) we propose AppBox as a core technology for MAM solutions, enabling to regulate the behaviour of any app without SDKs
2) AppBox works on stock Android devices and does not require root privileges
3) we have implemented AppBox and tested it for performance and robustness on 1000 of the most popular real-world apps, using different Android versions.

II. Application Scenario

The reference application scenario we envisage involves the following parties: (i) a developer Dev and (ii) an enterprise Ent. Let us assume that Dev has created an app A that Ent wants to use. In a traditional MAM service, one has to have access to the source code of the app or the developer needs to apply the sandbox while developing her own app. Unfortunately, Ent **does not** have the source code of A, that could be heavily obfuscated, and Dev does not want to release the source due to IPR reasons. Dev is also not interested in customising A using a wrapper $sandbox$ because of the extra resources needed for managing the customised version of A and the cost of supporting updates.

In such a scenario, AppBox can be useful. We envision the developer's cooperation to offer an AppBox compatible version of A. However, Dev does not need to branch out any new version of A or to include third-party library/code within the app. Therefore, AppBox can be seen as an alternative to the SDKs offered by existing MAM services.

The only action needed is to run A through our *StubFactory* (see Section IV-A). This component takes A as input and returns two apps: the *StubApp* A'_{stub} and A'. The A'_{stub} will generate the sandbox where A' will be executed (details will be discussed in Section IV-C2). Here we stress that A' is an exact copy of A, except for a slightly different manifest file automatically modified by the *StubFactory* component. Moreover, *StubFactory* neither modifies A's bytecode nor does it insert any additional code in the app.

At this stage, Dev has to sign A'_{stub} and A' by a freshly generated self-signed certificate K. Since both the app A' and the stub A'_{stub} are now signed with the same K, the Android manifest attribute *android::sharedUserId* allows to execute both apps under the same UID. Finally, the signed new app can be distributed.

When Dev releases a new version of A, the only step required is to create a new version of A'. This is fully automated by means of the *StubFactory* . Each time Dev releases a new app's update she creates a compatible app. However, there is no need to update and distribute A'_{stub} again, because the stub code is independent from the app version. Finally, Dev normally signs the new version of A' with the certificate signing the previous version of the app.

III. Requirements and Related Work

Our aim is to provide a black-box app-level MAM solution that is also able to enforce enterprise policies. The following requirements are of interest:

- **R1:** Compatibility: do not require modifications to the Android OS or root privileges
- **R2:** BlackBox: provide customised apps without disclosing the source code and without SDK integration.
- **R3:** do not require more privileges than those originally requested by the app that will be sandboxed
- **R4:** Multilevel: cover both Java and at native levels

We summarise other approaches proposed in the literature in Table I and classify them according to our requirements.

We start reviewing the two state-of-the-art approaches most similar to our solution: DeepDroid [1] and AppShield [2]. The former, listed with approaches in the first column of Table I, proposes a dynamic enterprise security policy enforcement scheme on Android devices, allowing fine-grained capabilities but requiring the root privilege in order to operate (R1:✘). The latter, AppShield, has goals similar to our solution but it uses the inlined monitor technique. AppShield is basically a rewriting-based approach (R2:✘), and it is designed for scenarios where the app is modified by third-parties after it is distributed, thus implicitly assumes the app is not obfuscated.

Early works for enforcing security and privacy policies [3]–[9] modify part of the Android OS or require root privilege. They meet many of our requirements, but they need modifications to the Android codebase, making it impossible to deploy on stock devices (R1:✘). Extensive work on mobile security has shown that massive and partially automated app analysis is possible on stock Android [10]–[14], but massive policy enforcement and remediation are also needed.

To address these issues, researchers proposed sandbox solutions based on rewriting the bytecode of the apps using inline reference monitors (IRM) to enforce the security policy (R2:✘). The third column of Table I shows those approaches [11], [15], [16].

Although these systems work on stock Android, they present several limitations when it comes to rewriting obfuscated apps, as reported by previous research [19]. On the other hand, they do not require any additional permission and they have low impact on performance. However, there is still a big limitation. Since inlined monitors rewrite the app to enforce a specific policy, supporting many different policies generates many different custom versions of the same app, thus having a serious impact on app maintenance and updates.

Two other works have then been published: Boxify [17] and NJAS [18]. Both approaches do not need to modify the Android OS and do not require root privileges.

Boxify offers app virtualization by means of an isolated process, that executes without any privilege. If the app requires a privilege, Boxify evaluates the request and acts as a proxy for the app. This provides an isolation technique that works on stock Android devices. To manage the permissions required by the monitored apps, Boxify introduces a *Broker*. To work the Broker has to collect the permissions of all the apps monitored in Boxify thus creating a component that has the cumulative privileges of all sandboxed apps, introducing a single point of failure (R3:✘). To achieve a complete sandboxing of all app communications, Boxify must understand the semantics of the apps. This operation generates overhead and it would be very difficult for obfuscated apps (R2:✘).

NJAS provides a sandboxing solution based on the PTrace mechanism. NJAS does not require to access the app code and works on stock Android devices. In contrast with Boxify, its monitoring mechanism does not increase the permissions set from the apps. In NJAS the target app is installed on the user device and then executed within a stub app in which it is monitored by means of PTrace. For this reason, NJAS is tightly bound to the availability of PTrace on the user device, thus limiting its deployability on a wide number of real-world mobile devices (R1:✘). Furthermore, considering the high number of context switches added by the PTrace-based syscall filtering, it is not a lightweight solution. Finally, it enforces policies only at the native level (R4:✘).

Samsung Knox provides a commercial solution to protect Android from malware and isolate different working scenarios [20]. Its adoption is limited to Samsung devices (R1:✘). Knox requires ARM TrustZone hardware support, which limits its deployment to certain Android platforms.

Google's Android for Work [21] (AFW) defines a set of device features that separate personal apps and data from a work profile containing work apps and data. AFW management capabilities rely upon features that are part of newer Android operating systems. AFW supports a managed profile, where a device can be monitored by the IT department, that can distribute internal apps on Google Play for Work, a business-specific market offered by AFW. An app that is requested to run in a managed profile must make use of AFW APIs[1], for this reason the developer has to modify the app's codebase (R1:✘). AppBox can use some Google Play for Work features such as the distribution channel for the *StubApp* provided by AppBox , while the customised app can still be retrieved via the official market.

To conclude, to the best of our knowledge, no other proposed solution is able to satisfy all the requirements we have identified for a black-box lightweight sandboxing approach.

IV. AppBox Architecture

AppBox creates and runs the managed app within a dedicated container that enforces enterprise policies at runtime. It wraps each managed app in a sandbox that injects control hooks to intercept the app interaction with the external world. The installation of a single *StubApp* for each managed app is required. The *StubApp* is automatically generated by the app developer using the information contained in the manifest of the managed app, and requests the same permissions as

[1]https://developer.android.com/work/guide.html

TABLE I: Comparison of Sandboxing Approaches Based on Desired Requirements. (✔ = applies; ✘ = does not apply)

Requirements	OS/root Extension [1], [3], [4], [7] [5], [6], [8], [9]	Aurasium [15] Ref. Hijacking [11] Hybrid [16]	Boxify [17]	Njas [18]	AppShield [2]	AppBox
(R1) Compatibility	✘	✘	✔	✘	✘	✔
(R2) BlackBox	✘	✘	✘	✘	✘	✔
(R3) No Add. Permissions	✔	✔	✘	✔	✔	✔
(R4) Multilevel	✔	✔	✔	✘	✔	✔

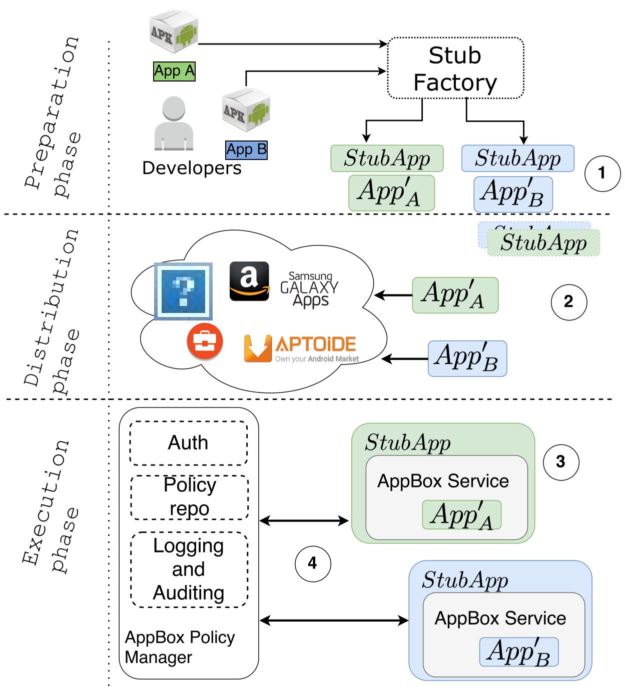

Fig. 1: AppBox design phases: Preparation, Distribution and Execution.

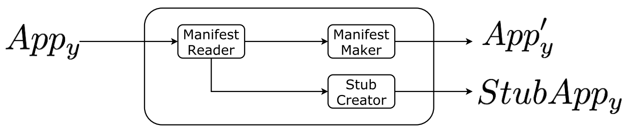

Fig. 2: StubFactory and its components

the managed app. The *StubApp* contains only the shim code responsible for loading the managed apps at runtime and for dynamically retrieving enterprise-defined security policies. It neither contains app code nor resources. For this reason, differently from repackaged apps, it can be submitted to the Google Play Store and installed as a regular app. The generation of a *StubApp* is done using the *StubFactory* provided by our framework and described in Section IV-A.

However, to generate a correct *StubApp* , the manifest of the managed app has to be modified to set the values for the `android:sharedUserId` and `android:process` attributes. The developer has to sign both the *StubApp* and the managed app with the same certificate. By using these two attributes, we are able to load the code of any app in the process space of the *StubApp* .

The AppBox workflow consists of three phases as shown in Fig. 1: *preparation*, *distribution* and *execution*. During the preparation, the app developer creates the managed app (App') and the *StubApp* (step 1). Then, the developer distributes the managed app via any supported Android market (step 2). Finally, the user installs both apps (App' and the *StubApp*) on the target device. During the execution phase, the *StubApp* creates a sandbox to execute the managed app. The sandbox is responsible for monitoring the managed app's behaviour and enforce the policies (step 3), offered via the AppBox Policy Manager instance (step 4).

In the following, we provide a description of the components involved in each phase.

A. Preparation phase

To be able to manage an app with AppBox , the developer has to create the *StubApp* and a managed app, as shown in Figure 1. This step is performed by the developer using the *StubFactory* , a set of python scripts along with a small DEX file containing the actual stub code. Another interesting aspect is that because the *StubFactory* only operates at the level of the manifest file, the managed app and *StubApp* can be created even if the original app code is obfuscated.

In the following, we assume that *App* is an app an enterprise wants to customise. Using the *StubFactory* , the developer generates the managed app, indicated as App', and the stub app, indicated as *StubApp*. Finally, both the *StubApp* and App' must be digitally signed by the developer.

As shown in Figure 2, the *StubFactory* first extracts and decodes the Android manifest from *App*, collecting package name, main activity and requested permissions. Furthermore, it checks if the manifest contains components of *App* defined to run across multiple processes. If this is the case, then the names of these components are added to the manifest of the *StubApp* so that any additional process created by the app will be monitored by a dedicated sandbox instance. Next, the **Manifest Maker** creates a new manifest[2] If *App*'s manifest contains the broadcast receiver for the boot completed intent, then this will be removed from the App''s manifest. This is to prevent the situation in which App' might be launched before *StubApp* .

[2]where App' will include both the attributes `android:sharedUserId` and `android:process` .

The last step is to create $StubApp$ through the **Stub Creator**. First, the manifest for the $StubApp$ is created, with the same permissions as *App*. By default, $StubApp$'s manifest will have the broadcast receiver component for the $BOOT\_COMPLETED$ system message. In this way, all the *StubApp* installed in a device will start as soon as the booting phase is completed. If the App's manifest also contained this broadcast receiver, then the $StubApp$ will act as a proxy and forward the boot completed intent to App'.

It is worth noting that *App* and App' have exactly the same bytecode. In fact, the *StubFactory* only operates on the App's manifest to output *StubApp* and App'. The developer is able to create as many *StubApp* as she may need to satisfy customers' requests. In fact, to iterate the preparation steps the developer is asked to create a new certificate, which will identify each customer. App updates are distributed via the app market as a normal APK file, the developer needs to sign them by the same certificates used initially.

B. Distribution phase

In this phase, the developer distributes apps as usual, via any supported market. AppBox does not require any additional user interaction to complete an app update and the developer does not need to accomplish any particular operation in order to distribute updates of managed apps. Whenever a new update is ready, the developer uses the *StubFactory* to automatically produce an updated managed app version. Thanks to our approach the developer does not need to redistribute the *StubApp* component.

C. Execution phase

After the preparation step, both $StubApp$ and App' are deployed on a device running stock Android OS. The $StubApp$ is totally transparent, there are no additional icons shown nor is there a management cost in charge of the end user. The execution of the managed app is done by the AppBox Service. The AppBox Service is a process created by the *StubApp* that loads and executes the code of the managed app App'. Because the *StubApp* and the AppBox Service share the same process space, it is possible to inject hooks at runtime into the managed app's virtual memory. These are the hooks that enforce the desired policy. By sharing the same UID through the use of the `android:sharedUserId` attribute, the AppBox Service is able to access all the private files of the managed app. In AppBox the managed app is dynamically instrumented by means of function interposition on both Java and native levels. This enables the enforcement of security policies related to Java APIs and native code. The AppBox Service modifies the memory of the managed app to inject hooks capable of intercepting calls to Java methods and to syscalls. The mechanism is transparent to the managed app and new versions of the target app can be easily managed without changing the *StubApp* or the AppBox Service.

The biggest technical challenge at this point is to guarantee that the managed app execution will be entirely confined within our sandbox. Android offers a considerable number of features for apps to communicate with each other and share functionality. These features are accessed through callbacks such as broadcast messages, intents, and IPC. Care must be taken to avoid that these mechanisms are exploited in order to let the managed app execute outside its sandbox.

The exported components of an app, including the main activity that is always exported by default, can receive explicit intents sent by any app. When this happens, usually Android starts the exported component into a new process. If not handled properly, this could be an issue because it could result in a managed app starting in a process outside its sandbox. However, before starting a new process, Android searches if there is already a process where (1) the process name matches the requested component's name; (2) the process UID is the same as the one assigned to the app in which the target component has been defined. Thanks to the combination of both attributes `android:sharedUserId` and `android:process` , the AppBox Service is sharing the same process name and UID, hence any intents sent to any exported component of a managed app will be captured and executed within the AppBox Service.

The following components are particularly relevant at execution time.

1) AppBox Policy Manager: AppBox Policy Manager is a console application intended to be used by security administrators. Through the AppBox Policy Manager an administration (e.g., the MAM service) can define new policies and deploy them on the enrolled devices. Once the managed app has been started, the AppBox instance manages the authentication process with the Policy Manager.

2) StubApp and AppBox Service: The *StubApp* is responsible for creating the sandbox process where the managed app will be executed as shown in Figure 3. When a managed app is launched, first the *StubApp* creates the AppBox Service in a separate process (step 1) and invokes the *prepare* method via the Binder to set up the interceptors (step 2). Then, the *StubApp* retrieves the set of policies and the hooking library (step 3) specific to the managed app from the *AppBox Policy Manager*. The AppBox Service loads the hooking library to instrument its virtual memory and the app's code will be loaded and executed within the AppBox Service (step 4). Before the managed app can be executed, the *StubApp* has to create the right Android context for that app. This operation is performed by calling the Android API method *createPackageContext* specifying the CONTEXT_INCLUDE_CODE flag. As an entrypoint, the *StubApp* declares in its manifest an *Application* class, that is the first app component loaded by Android before any other app code.

3) Java and Native Interceptors: The Java interceptors in AppBox are an extended versions of the ArtDroid hooking framework [22]. However, compared to ArtDroid, AppBox Java interceptors are able to hook static Java methods by implementing the approach proposed in [23]. We are able to intercept all calls to monitored Java methods including either

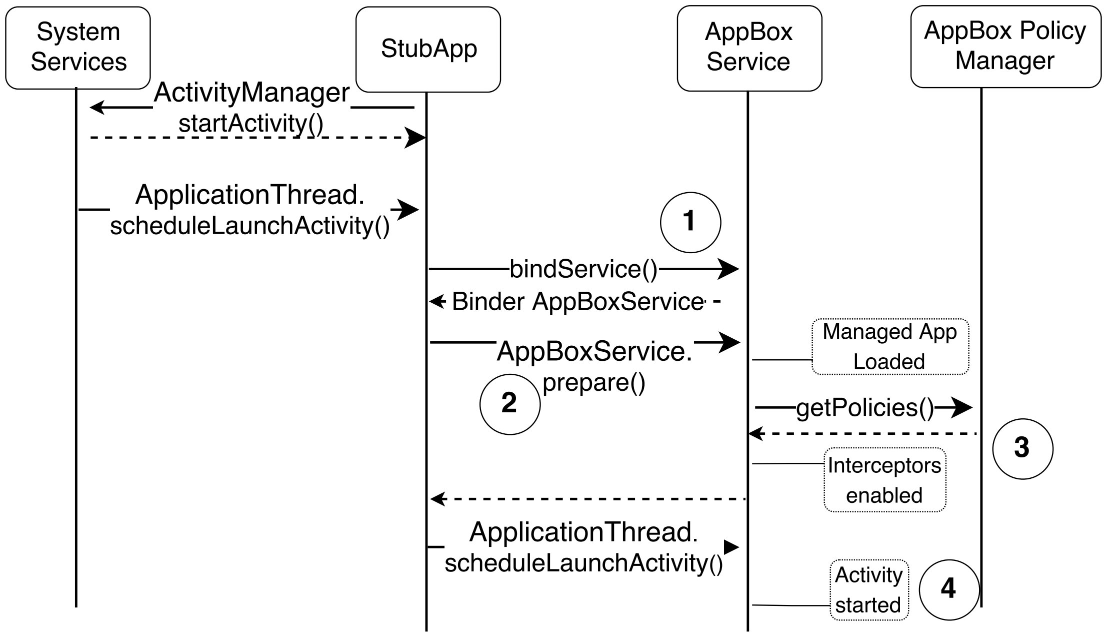

Fig. 3: AppBox enforcing managed apps

calls via Java reflection, native code or dynamically loaded code. The intercepted calls are redirected to the specific Policy Enforcement Point (PEP) where the actual user-defined policy is enforced. Java interceptors achieve transparent hooking by means of memory instrumentation. It fully supports both the DVM and ART Android runtime.

V. AppBox Policy Language

Figure 4 shows the syntax of AppBox policy. Policies are identified by a name and they define what *operation* a *Requester* app can execute on a *Resource*. In our prototype we defined two sets of *operations*: the first set contains getter methods that return data from the Resource to the Requester; the second set contains setter methods where data is being passed by the Requester to the Resource. Moreover, AppBox offers the possibility to define policies on events. The operations defined in the policies are then mapped to relevant Java methods or native functions.

```
PolicyName: Requester can do <operation> on <Resource>
            have to perform <action>
            [if <condition>]
```

Fig. 4: An AppBox Policy Language example

A resource identifies any sensitive data which could be retrieved via either Android middleware APIs (e.g., location, contact, camera) or native code (e.g., sensors, socket, microphone). The *have to perform* clause specifies which actions have to be performed if this policy is enforced. These actions are mapped to a set of functions to control the app's behavior (e.g., filtering, anonymisation) and to change the values of the parameters of the operation being executed. An action is a callback that is registered by AppBox to dynamically forward the execution to the corresponding function and can operate on both input parameters and returned values.

VI. Evaluation

In this section, we evaluate performance overhead, robustness, applicability and effectiveness.

A. Performance Overhead

To evaluate performance penalty on managed apps, we used benchmark apps and our custom micro-benchmarks. As benchmark apps, we used Quadrant and Vellamo. The former was selected because it has been used in other related works [5], [17], [18]. The latter is a highly accurate benchmark developed by Qualcomm and is specifically intended to stress the Binder communication channel. Given that the Binder is the most common means of communication for Android apps, it is important to measure the overhead AppBox introduces. Moreover, we executed the *webview* package of Vellamo that contains various benchmarks for Android's Webview API.

As shown in Table II, the impact of AppBox on the total score produced by the benchmarks is low. The test marked as *total* reports the cumulative score that the Quadrant benchmarking app produced, while the I/O test were oriented to stress disk read/write operations. The worst score (15.7%) is low w.r.t. to similar works, and can be attributed to the I/O test. AppBox overhead is lower than the one introduced by Boxify and NJAS in equivalent tests. As for the performance penalty introduced in the Binder communication (indicated as multi-core in Table II), the score indicated by the Vellamo benchmark is really low (up to 1%), due to the fact that AppBox does not need to perform extra marshalling operations.

TABLE II: BenchMark Apps Results For Nexus 5x (64 bit)

App	Test	AppBox	Native	Loss
Quadrant	Total	17736	17449	1.6 %
	I/O	9820	8277	15.7 %
Vellamo	multi-core	1948	1930	0.9 %
	webview	2803	2688	4.1 %

TABLE III: Native Micro-Benchmarks AppBox Performance, Compared against Boxify.

	AppBox: Nexus 5x (250k runs)			Boxify: Nexus 5 (15k runs)		
Func.	Native	AppBox	OvHead	Native	Boxify	OvHead
open	6.49 μs	6.7 μs	3.2%	9.5 μs	122.7 μs	1191%
mkdir	92.7 μs	95.2 μs	2.7%	88.4 μs	199.4 μs	125%
rmdir	80.7 μs	85.3 μs	5.7%	71.2 μs	180.7 μs	153%

To understand the performance implication of AppBox on method invocations and function calls, we developed a synthetic app in order to perform a micro-benchmark, testing the most significant native functions by means of AppBox native interceptors. In Table III, we report the overhead introduced by AppBox when hooking functions in libc (first column). In the same table, we also compare our overhead with Boxify performance overheads as reported in [17]. As the results show, our performance overhead is significantly lower.

Table V reports the results for the overhead introduced by AppBox when interposing Java Android APIs, and Table IV presents hooks that were in place during our experiments. Results show that the performance penalty introduced by API Hooking is acceptable. Micro-benchmark tests listed in Table V are sometimes responsible for triggering multiple hooks.

B. Security

The threat model considered in this paper is based on the assumption that AppBox will be used in an enterprise context as part of a comprehensive MAM/MDM solution. We also assume that developer apps are not malicious, but they may not comply with the enterprise security policies.

We would like to consider if sandboxing may be avoided. For example, the user might delete the stub app, and the managed app could be run alone so that policy enforcement could be bypassed. In the enterprise MAM context, however, this possibility can be excluded, or in any case it can be considered as external to our perimeter. Once the AppBox components are in place and running, the managed app execution context will always be the one instrumented by the *stubApp*.

In an enterprise scenario, attempts to uninstall monitored applications can be detected by other existing solutions. Sandboxing is enforced by the OS. Monitoring a running agent instance is a common problem for any deployed MAM solution: an external entity can monitor *stubApp* instances, detect any change and respawn them when they are killed.

It is also easy to prevent the managed app from running outside the *stubApp* container. In fact, by simply adding an Application class to the manifest file, we can prevent the managed app execution when the Application class defined by the *stubApp* has not been loaded. Thus, once the stub is installed, the stub and the managed app are bound together at execution time.

C. Effectiveness

During this evaluation our goal was twofold: (i) demonstrate that AppBox is easy-to-deploy and fully capable to wrap real-world apps and (ii) assess robustness. To this end, we executed 1000 free apps from the Google Play Store. Out of these 1000 analyzed apps, 66 were obfuscated, with an average app size of 20 MB. To recognize obfuscation, we employed APKiD (see https://github.com/rednaga/APKiD), that leverages on different heuristics in order to statically detect presence of packers, protectors and obfuscators.

To execute our tests, we had to modify the manifest of the collected apps adding the attributes requested by AppBox . This was required only for testing purposes - it is not required in the operational scenario, where the developer will be responsible for performing this task. We evaluated the runtime robustness of AppBox running the collected apps on a Nexus 5x with stock Android 8.0. We employed DroidBot [24] to exercise the managed app's functionality. DroidBot first statically analyses the target app then it allows to dynamically inject events to stimulate the app. We ran each managed app for 5 minutes as in the Google Play Bouncer [25], while we were collecting log information, seeking for app crashes. From the 1000 apps, only 56 apps reported a crash during testing. Manual investigation of the dysfunctional apps revealed that most errors were caused by pre-existing bugs.

D. Applicability

We manually executed 5 of the most popular free apps concerning business functionality. Our goal is threefold:

- test applicability on several Android versions;
- verify the correct interaction of the managed app with the OS: we completed the authentication process, if present, testing for correct delivery of system events and the interaction with apps such as Google Play Service;
- stress the SandboxService: we disabled it to detect issues when the managed app is executed outside AppBox .

Table VI lists the selected apps, with the following policies: (1) network-related, e.g., denying connections to known advertisement servers, and monitoring non-TLS connections; (2) file system monitoring to detect operations on the SD-card. Moreover, a fake Location Provider was in place during the tests. We stimulated those apps for 8 minutes, with tests such as visiting web pages and sharing location data via GPS and authentication through Google Play Services.

In the experiments none of the tested apps crashed and we verified policy-enforced behavior. AppBox effectively blocked any connection to the blacklisted addresses. Non-TLS connections were denied and reported. When location policy enforcement was in place we noticed that the actual location shared was referring to our previously set value.

TABLE IV: Android API: Micro-Benchmark Results

Nexus 5x (15k runs)			
Android API	Native	on AppBox	Overhead
Read Contact	6.55 ms	9.93 ms	3.38ms (51.6%)
Socket	23.23 ms	26.33 ms	3.1ms (13.3%)
openFileInput	0.19 ms	0.22 ms	0.03ms (15,7%)
openFileOutput	0.24 ms	0.28 ms	0.04ms (16.6%)
Create File	0.02 ms	0.03 ms	0.01ms (50%)
Open Camera	227.09 ms	237.02 ms	9,93 ms (4.4%)

TABLE V: APIs Monitored with Java Micro-Benchmarks

Alias	Class package	Method name
Read Contact	android.content.ContentResolver	query
Networking	java.net.Socket	<init>
File	android.app.ContextImpl android.app.ContextImpl java.io.File	openFileInput openFileOutput <init>
Camera	android.hardware. camera2.CameraManager	openCamera

TABLE VI: Apps for Testing Applicability/Effectiveness

App Name	Version	Category
Skype for business	6.13.06	Business
Slack	2.30.0	Business
Dropbox	38.2.4	Productivity
Intesa San Paolo Mobile Banking	2.1.0	Finance
Chrome	56.0.2924.87	Communication

VII. Conclusions

We presented a black-box app sandboxing solution for stock Android, suitable for enterprise domains, where apps running on employees' smartphones are often managed by specialised services such as MAMs and MDMs. We enforce fine-grained security policies covering both Java, native and third-party code. The managed app is confined within an instrumented process space, and we avoid modifications to its bytecode. Preliminary evaluations showed limited performance overhead, robustness and general applicability to real-world apps.

Acknowledgment

The work is partly funded by the European Union under Horizon Europe Programme - Grant Agreement 101070537 — CrossCon. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*, 2015.
[2] Z. Qu, G. Guo, Z. Shao, V. Rastogi, Y. Chen, H. Chen, and W. Hong. Appshield: Enabling multi-entity access control cross platforms for mobile app management. In *Int. Conf. on Security and Privacy in Communication Systems*, pages 3–23. Springer, 2016.
[3] M. Conti, V. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Int. C. on Inf. Sec.*, pages 331–345, 2010.
[4] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM T. on Computer Systems (TOCS)*, 32(2):5, 2014.
[5] G. Russello, A. Jimenez, H. Naderi, and W. van der Mark. Firedroid: Hardening security in almost-stock android. In *Proc. 29th Annual Computer Security Applications Conf.*, pages 319–328. ACM, 2013.
[6] S. Heuser, A. Nadkarni, W. Enck, and A. Sadeghi. Asm: A programmable interface for extending android security. In *USENIX Security*, volume 14, pages 1005–1109, 2014.
[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 51–62, 2011.
[8] A. Saracino, F. Martinelli, G. Alboreto, and G. Dini. Data-sluice: Fine-grained traffic control for android application. In *Proc. IEEE Symp. on Computers and Communication*, pages 702–709, 2016.
[9] Xposed repository. https://repo.xposed.info/. Accessed: 2018-12-17.
[10] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proc. 2015 Int. Symp. on Software Testing and Analysis*, pages 118–128. ACM, 2015.
[11] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang. Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted android devices. In *Proc. 38th Int. Conf. on Software Engineering*, pages 959–970. ACM, 2016.
[12] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Int. Conf. on Information Security*, pages 346–360. Springer, 2010.
[13] F. Bergadano, M. Boetti, F. Cogno, V. Costamagna, M. Leone, and M. Evangelisti. A modular framework for mobile security analysis. *Inf. Security Journal*, 29(5):220–243, 2020.
[14] V. Sembera, M. Paquet-Clouston, S. Garcia, and M. J. Erquiaga. Cybercrime specialization: An exposé of a malicious android obfuscation-as-a-service. In *Proc. IEEE European Symp. on Security and Privacy*, pages 217–228. IEEE, 2021.
[15] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security Symp.*, 2012.
[16] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid user-level sandboxing of third-party android apps. In *Proc. ACM Symp. on Inf., Computer and Communications Security*, pages 19–30, 2015.
[17] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symp.*, pages 691–706, 2015.
[18] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proc. 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38, 2015.
[19] H. Hao, V. Singh, and W. Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proc. 8th ACM SIGSAC symp. on Inf., computer and comm. security*, pages 25–36, 2013.
[20] Samsung knox. samsungknox.com. Accessed: 2023-03-03.
[21] Android for work. https://cloud.googleblog.com/2015/02/android-is-ready-for-work.html. Accessed: 2023-03-03.
[22] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. *Proc. Innovations in Mobile Privacy and Security (IMPS)*, pages 24–32, 2016.
[23] M. Wißfeld, P. von Styp-Rekowsky, and M. Backes. Callee-side method hook injection on the new android runtime art.
[24] Droidbot suite. github.com/honeynet/droidbot. Accessed: 2023-03-03.
[25] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012*, 2012.