AirBag: Boosting Smartphone Resistance to Malware Infection

Chiachih Wu†, Yajin Zhou†, Kunal Patel†, Zhenkai Liang*, Xuxian Jiang†

†Department of Computer Science
North Carolina State University
{cwu10, yajin_zhou, kmpatel4, xjiang4}@ncsu.edu

*School of Computing
National University of Singapore
liangzk@comp.nus.edu.sg

***Abstract*—Recent years have experienced explosive growth of smartphone sales. Inevitably, the rise in the popularity of smartphones also makes them an attractive target for attacks. In light of these threats, current mobile platform providers have developed various server-side vetting processes to block malicious applications ("apps"). While helpful, they are still far from ideal in achieving their goals. To make matters worse, the presence of alternative (less-regulated) mobile marketplaces also opens up new attack vectors, which necessitate client-side solutions (e.g., mobile anti-virus software) to run on mobile devices. However, existing client-side solutions still exhibit limitations in their capability or deployability.**

In this paper, we present *AirBag*, a lightweight OS-level virtualization approach to enhance the popular Android platform and boost our defense capability against mobile malware infection. Assuming a trusted smartphone OS kernel and the fact that untrusted apps will be eventually installed onto users' phones, AirBag is designed to isolate and prevent them from infecting our normal systems (e.g., corrupting the phone firmware) or stealthily leaking private information. More specifically, by dynamically creating an isolated runtime environment with its own dedicated namespace and virtualized system resources, AirBag not only allows for transparent execution of untrusted apps, but also effectively mediates their access to various system resources or phone functionalities (e.g., SMSs or phone calls). We have implemented a proof-of-concept prototype on three representative mobile devices, i.e., Google Nexus One, Nexus 7, and Samsung Galaxy S III. The evaluation results with a number of untrusted apps, including real-world mobile malware, demonstrate its practicality and effectiveness.

I. Introduction

Smartphone sales have recently experienced an explosive growth. Canalys [23] reports that the year of 2011 marks as the first time in history that smartphones have outsold personal computers. Their incredible popularity can be partially attributed to their improved functionality and convenience for end users. Especially, they are no longer basic devices for making phone calls and receiving text messages, but powerful platforms, with comparable computing and communication capabilities to commodity PCs, for GPS navigation, web surfing, and even online businesses. Among competing smartphone platforms, Google's Android apparently gains the dominance with more than half of all smartphones shipped to end users running Android [25].

One key appealing factor of smartphone platforms is the availability of a wide range of feature-rich mobile applications ("apps"). For instance, by September 2012, Google Play [9] and Apple App Store [6] are home to more than $650,000$ and $700,000$ apps, respectively. The centralized model of mobile marketplaces not only greatly helps developers to publish their mobile apps, but streamlines the process for mobile users to browse, download, and install apps, hence boosting smartphone popularity. With the increased number of smartphone users, malware authors are also attracted to the opportunity to widely spread mobile malware. As an example, the `DroidDream` malware infected more than $260,000$ devices within 48 hours, before Google took action to remove them from the official Android Market (now Google Play) [1]. Considering these threats, mobile platform providers have developed server-side vetting processes to detect or remove malicious apps from centralized marketplaces in the first place. With varying levels of success, many malicious apps are identified and removed from marketplaces. However, they are far from ideal as malware authors could still find new ways to penetrate current marketplaces and upload malicious apps.

From another perspective, a number of client-side solutions have been developed. As a mobile platform provider, Google provides the Android security architecture which sandboxes apps based on their permissions and runs them as separate user identities. However, they are still insufficient as malicious apps may masquerade as legitimate apps but request (and abuse) additional permissions [34] to access protected smartphone functionality or private information. In the face of these threats, traditional software security vendors have developed corresponding mobile anti-malware software. With the inherent dependence on known malware signatures, they are largely ineffective against new ones. To mitigate them, Aurasium [55] is proposed to enforce certain access control policies on untrusted apps. However, it requires repackaging apps to enable the enforcement and the enforcement is still ineffective against attacks launched from native code. L4Android [43] and Cells [19] take a virtualization-based approach to allow for multiple virtual smartphones to run side-by-side on one single physical device. However, they are mainly designed to

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23164

embrace the new "bring-your-own-device" (BYOD) paradigm and the offered isolation is too coarse-grained at the virtual smartphone boundary. For mobile users, it is desirable to have a lightweight solution that can strictly confine untrusted apps (including ones with native code or root exploits) at the app boundary.

In this paper, we present the design, implementation, and evaluation of *AirBag*, a new client-side solution that leverages lightweight OS-level virtualization to significantly boost our defense capability against mobile malware infection. Specifically, as a client-side solution, AirBag assumes a trusted smartphone OS kernel and considers users may unintentionally download and install malicious apps (that somehow manage to penetrate the vetting processes of mobile marketplace curators). To strictly isolate and prevent them from compromising normal phone functionalities such as SMSs or phone calls, AirBag dynamically instantiates an isolated virtual environment to ensure their transparent "normal" execution, and further mediate their access to various system resources or phone functionalities. Therefore, any damages that may be possibly inflicted by untrusted apps will be strictly isolated within the virtualized environment.

To provide seamless user experience, AirBag is designed to run behind-the-scenes and transparently support mobile apps when they are downloaded, installed, or executed. Specifically, when an user installs (or sideloads) an app, the app will be automatically isolated within an AirBag environment. Inside the AirBag, the app is prohibited to interact with legitimate apps and system daemons running outside. To accommodate its normal functionality, AirBag provides a (decoupled) *App Isolation Runtime (AIR)* whose purpose is to separate it from the native Android runtime, but still allow the isolated app to run as it is installed normally. Further, users can choose to run AIR in three different modes: (1) "incognito" is the default mode that will completely remove personally-identifying information about the phone (e.g., IMEI) or users (e.g., gmail accounts) to avoid unnecessary information leakage; (2) "profiling" mode will log detailed execution traces (in terms of invoked Android APIs or functionalities) for subsequent offline analysis; (3) "normal" mode will essentially execute the app without further instrumentation. For other normal phone features (e.g., networking and telephony), the AIR proxies related API calls to the external native Android runtime through an authenticated communication channel.[1] This brings us new opportunities to apply fine-grained access control on the isolated app (e.g., prompting users for outgoing SMSs or phone calls) without repackaging the app itself or affecting the native Android runtime. Besides, the default mode ("incognito") of AirBag allows users to "test" an app in the isolated runtime before running it in the native runtime. Throughout the "test" phase, users can check if the app has any abnormal or malicious behavior with the fine-grained access control logs provided by AirBag. This prevents end users from installing malicious apps in the first place. On the other hand, users can also use the "profiling" mode to gather detailed information of the identified malicious apps (in "incognito" mode) for analysis.

To develop a robust AirBag mechanism and strictly confine untrusted apps, a common wisdom is to encapsulate their execution in a separate virtual machine (VM) that is isolated from the rest of the system. However, challenges exist to create a lightweight virtual machine for commodity mobile devices. In particular, current mobile devices are typically resource constrained with limited CPU, memory, and battery capability. And most off-the-shelf mobile devices do not have the processors with hardware virtualization support, which makes traditional virtualization approaches less desirable [52]. As our solution, AirBag takes a lightweight OS-level virtualization approach but still obtains comparable isolation capability. Specifically, by sharing one single OS kernel instance, our approach scales better than traditional hypervisors and incurs minimal performance overhead. Also, by providing a separate namespace and virtualizing necessary system resources, AirBag still achieves comparable isolation.

We have implemented a proof-of-concept prototype on three mobile devices, Google Nexus One, Nexus 7, and Samsung Galaxy S III, running Linux kernel 2.6.35.7, 3.1.10, and 3.0.8, respectively. To ensure seamless but confined execution of untrusted apps, our prototype builds the app isolation runtime or AIR by leveraging the Android Open Source Project (AOSP 4.1.1) to export the same interface while in the meantime allowing users to choose different running modes. Specifically, the "incognito" mode prevents personally-identifying information from being leaked while the "profiling" mode logs the untrusted app behavior, which we find helpful to analyze malicious apps (Section IV) in a live phone setting. Security analysis as well as the evaluation with more than a dozen of real-world mobile malware demonstrate that our system is effective and practical. The performance measurement with a number of benchmark programs further shows that our system introduces very low performance overhead.

The rest of the paper is organized as follows: In Section II, we present the overall system design, followed by its implementation in Section III. We then evaluate our prototype and report measurement results in Section IV. After that, we further examine possible limitations and explore future improvements in Section V. Finally, we describe related work in Section VI and conclude in Section VII.

II. System Design

A. Design Goals and Threat Model

Our system is designed to meet three requirements. First, AirBag should reliably confine untrusted apps such that any damage they may incur would be isolated without affecting the native phone environment. The challenges for realizing this goal come from the fundamental openness design behind Android, which implies that any app is allowed to communicate with other apps or system daemons running in the phone (through built-in IPC mechanisms). In other words, once a malicious app is installed, it has a wide attack surface to launch the attack. The presence of privilege escalation or capability leak vulnerabilities [37] further complicates the confinement requirement.

Second, AirBag should achieve safe and seamless user experience throughout the lifespan of untrusted apps, from their installation to removal. Specifically, from the user's perspective, AirBag should avoid incurring additional burden on users. Correspondingly, the challenge to meet this goal

[1]A network connection which relies on the authentication protocols to provide secure communication.

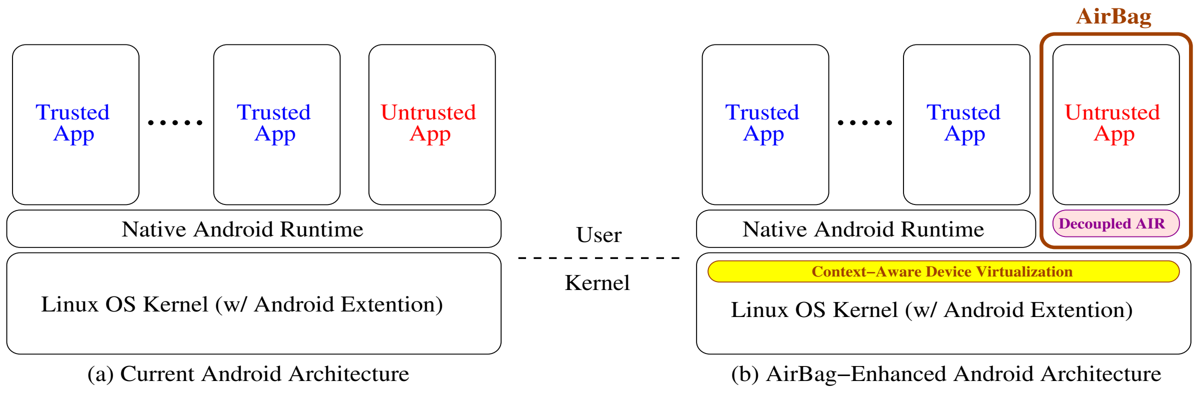

AirBag
Trusted App
Trusted App
Untrusted App
Native Android Runtime
Linux OS Kernel (w/ Android Extention)
(a) Current Android Architecture
User
Kernel
Trusted App
Trusted App
Untrusted App
Native Android Runtime
Decoupled AIR
Context–Aware Device Virtualization
Linux OS Kernel (w/ Android Extention)
(b) AirBag–Enhanced Android Architecture

Fig. 1. An Overview of AirBag to Confine Untrusted Apps

is to transparently instantiate AirBag's app isolation runtime when an untrusted app is being installed and seamlessly adjust different runtime environments when the untrusted app is being launched or terminated.

Third, because AirBag is deployed in resource-constrained mobile devices, it should remain lightweight and introduce minimal performance overhead. In addition, AirBag should be generically portable to a range of mobile devices without relying on special hardware or features (that may be limited to certain phone models).

Threat Model and System Assumption We assume the following adversary model while designing AirBag: Users will download and install third-party untrusted apps. These apps may attempt to exploit vulnerabilities, especially those in privileged system daemons such as `Zygote`. By doing so, they could cause damages by either gaining unauthorized access to various system resources or abusing certain phone functionalities in a way not permitted by the user or not known to the user.

Meanwhile, we assume a trusted smartphone OS kernel, including our lightweight OS extension to support isolated namespace and virtualized system resources. As a client-side solution, AirBag relies on this assumption to establish necessary trusted computing base (TCB). Also, such assumption is shared by other OS-level virtualization research efforts [43], [19]. With that, we consider the threat of corrupting OS kernels falls outside the scope of this work.

B. Enabling Techniques

In Figure 1, we show the overview of AirBag to confine untrusted apps and its comparison with traditional Android-based systems. The confinement is mainly achieved from three key techniques: *decoupled app isolation runtime (AIR)*, *namespace/filesystem isolation*, and *context-aware device virtualization*.

1) Decoupled App Isolation Runtime (AIR): Due to the openness design of Android, all apps share the same Android runtime and consequently any app is allowed to communicate with other apps on the phone. As mentioned earlier, from the security perspective, this exposes a wide attack surface. In AirBag, to minimize the attack surface and avoid affecting the original Android runtime, we choose to decouple the untrusted app execution from it. A separate app isolation runtime that allows apps to run on it and has (almost) no interaction with the original Android runtime is instantiated for untrusted app execution.

There are several benefits behind such a design: First, by providing a consistent Android abstract layer that will be invoked by third-party Android apps, AIR effectively ensures proper execution of untrusted apps without impacting the original Android runtime. Second, by design, AIR does not need to be trusted as it might be potentially compromised by untrusted apps. Third, a separate app isolation runtime also allows for customization to support different running modes (Section II-C). This is necessary as AIR mainly consists of essential Android framework classes and other service daemons that are tasked to manage various phone resources (e.g., device ID) or features (e.g., sensors). As a result, they likely access private or sensitive information that could be of concern when being exposed to untrusted apps.

2) Namespace/Filesystem Isolation: With a separate Android runtime to host untrusted apps, AirBag also provides a different namespace and filesystem to further restrict and isolate the capabilities of processes running inside. Because of namespace and filesystem isolation, an untrusted app inside AirBag is not able to "see" and interact with other processes (e.g., legitimate apps and system daemons) running outside. In fact, all processes running inside have their own view of running PIDs, which is completely different from external processes. In addition, to proactively contain possible damages, AirBag has its own filesystem different from the normal system. For storage efficiency, we extensively leverage unionfs [48] to compose AirBag's filesystem and isolate modifications from untrusted apps.

To elaborate, when an Android system is loaded, a number of service processes or daemons (e.g., `vold`, `binder` and `servicemanager`) are created. Inside AirBag, we similarly launch the same subset of processes but group them in their own `cgroup` [24]. By doing so, they are prevented from observing and interacting with processes in another group (i.e., processes in the original native Android system). The `cgroup` concept greatly facilitates AirBag management. Specifically, the set of processes inside AirBag is typically suspended until one untrusted app is being installed or launched. The newly installed untrusted app will automatically become a member of this `cgroup`. As a result, we can easily suspend the whole `cgroup` when no untrusted app is active to minimize the footprint or reduce the performance and power consumption.

Note that `cgroup` is provided by the OS kernel and is assumed to be trusted.

3) Context-Aware Device Virtualization: The presence of a separate AIR and namespace in AirBag unavoidably creates contentions for underlying system resources, even though AirBag delineates a boundary and by default disallows any interaction from inside to outside and vice versa. To resolve the contention, there is a need to multiplex various system resources. In our design, we develop a lightweight OS-level extension to mediate and multiplex the accesses from native and AirBag runtimes.

As an example, suppose two apps need to update the screen at the same time. Traditionally, a single service daemon `SurfaceFlinger` is in charge of synthesizing data from different sources (including these two apps) and generating the final output to be rendered on the device screen. However, with AirBag, these two apps run in two different runtimes and they will not share the same `SurfaceFlinger` service. Instead, AirBag has its own `SurfaceFlinger` service which will independently update the screen.

Our solution is to virtualize hardware devices in a context-aware manner. Specifically, our lightweight OS extension adds necessary multiplexing and demultiplexing mechanisms in place when the underlying hardware devices are being accessed. Also, our extension keeps track of the current "active" Android runtime (or namespace) and always allows the active runtime to access the hardware resources. Notice that an Android runtime is active if an app on it holds the focus, i.e., the user is currently interacting with the app. To maintain the same user experience, we disallow an user to simultaneously interact with two apps in different runtimes. As a result, in any particular moment, there exists at most one active runtime. Meanwhile, to gracefully handle contentious access from inactive runtime, we take different strategies base on the nature of relevant hardware resources. For example, for touch-screen and buttons, any press/release event will always be delivered to the active runtime only. For screen update, as the framebuffer device driver performs actual DMA operations from a memory segment to the LCD controller hardware, we accordingly prepare two separate memory segments such that each environment can independently render different output without interfering each other. The framebuffer driver can then choose the active memory segment to perform DMA and thus have an actual access to the LCD controller hardware.

C. Additional Capabilities

Beside the above key techniques, we also developed additional capabilities to facilitate the confinement and improve user experience.

1) Incognito/Profiling Modes: The decoupled AIR to host untrusted apps provide unique opportunities for its customization. Specifically, to prevent private information disclosure, we introduce the incognito mode that essentially instruments the AIR to exclude any sensitive data such as IMEI number, phone number, and contacts. For example, the device's IMEI number can be normally retrieved by apps through the services provided by the Android framework. When entering the incognito mode, such services are configured to return faked IMEI number to the calling app. Therefore, the isolated app transparently proceeds with fake data without additional risks. Also, AirBag prepares a separate root filesystem that allows for convenient "restore to default" to undo damages from untrusted apps. In addition, we also provide profiling mode that essentially records the execution trace of untrusted apps. The trace is mainly collected in terms of Android-specific `logcat`, which turns out to be very helpful for malware analysis (Section IV).

2) User Confirmation for Sensitive Operations: The decoupled AIR also provides interesting opportunities to further limit the capabilities of isolated apps. For example, a malicious app may attempt to stealthily send SMS text messages to certain premium-rate numbers or record your phone conversation. When such an app runs inside AirBag, the access to related phone features (e.g., `radio`, `audio`, and `camera`) will immediately trigger user attention for approval. In other words, the stealthy behavior from these apps will now be brought to user attention and the user also has the option to disallow it. It is interesting to notice that the latest Android release, i.e., Jellybean 4.2, introduces a built-in security feature called premium SMS confirmation [2] to avoid malware to rack up phone bills. While achieving similar goals, AirBag is different in restricting the access to certain phone features outside the AIR environment, thus providing stronger robustness than any inside solutions (as the *internal* built-in feature can be potentially compromised by untrusted apps for circumvention).

3) Seamless Integration: To achieve seamless user experience, AirBag introduces minimal user interaction when an app is being installed or launched. Specifically, when an untrusted app is being installed (or sideloaded), AirBag will prompt user with a (default) option to install it inside AirBag. If chosen, AirBag essentially notifies its own `PackageInstaller` to start the installation.[2] Note that for an app downloaded from Internet, the Android `DownloadManager` will store it in a specific directory located in microSD. In our prototype, we choose to export this directory read-only to AirBag so that its `PackageInstaller` can access it for installation. For improved user experience, AirBag will be installed as the default `PackageInstaller`. Inside AirBag, we have a daemon that listens to the command from it to kick off internal app installation. In other words, the isolated apps are physically installed in the AirBag instead of the original Android runtime. Moreover, for any app being installed inside AirBag, AirBag will automatically create an app stub that bears the same icon from the original app. (To indicate the fact that it is actually inside AirBag, we will attach a lock sign to the icon.) When the *app stub* is invoked, AirBag will be notified to seamlessly launch the actual app such that the user would feel just like invoking a normal app (without noticing the fact it is actually running inside AirBag). By doing so, the AIR becomes active and the original Android runtime goes to inactive. Once the user chooses to terminate the app, the original Android runtime is resumed back to active.

III. Implementation

We have implemented a proof-of-concept AirBag prototype on three different mobile devices, i.e., Google Nexus

[2]If not chosen, the normal installation procedure will be triggered without AirBag protection.

One, Nexus 7, and Samsung Galaxy S III, running Linux kernel 2.6.35.7, 3.1.10, and 3.0.8 respectively. Our prototype is portable without relying on any specialized hardware support. In the following, we present in detail about our prototype. For simplicity, unless explicitly mentioned, we will use Google Nexus One as the reference platform.

A. Namespace/Filesystem Isolation

Our system confines untrusted apps in a separate namespace and filesystem. In our prototype, we leverage and extend the namespace isolation feature of `cgroups` [24] in mainstream Linux kernels. At the high level, our prototype instantiates a new namespace and then starts from the very first process (i.e., `airbag_init`) inside AirBag. The `airbag_init` process will then bootstrap the entire AIR. Specifically, the new namespace of AirBag is created by cloning a new process with a few specific flags: `CLONE_NEWNS`, `CLONE_NEWPID`, `CLONE_NEWIPC`, `CLONE_NEWUTS`, and `CLONE_NEWNET`. Further, right before switching the control to the `airbag_init` program, we initialize a separate root filesystem for the newly `clone`'d process (and its decedent processes) by invoking `pivot_root` in the new root directory that contains essential AIR files. We then prepare `procfs` and `sysfs` filesystems inside AirBag so that subsequent processes inside AirBag can properly interact with the underlying Linux kernel. After that, we yield the control by actually executing the `airbag_init` program that then kicks off the entire AIR, including various service daemons (e.g., `SurfaceFlinger` and `system_server`). These service daemons as well as essential Android framework classes collectively allow untrusted apps to execute transparently when they are dispatched to the AIR.

With a new AirBag-specific namespace, all processes running inside cannot observe and interact with processes running outside. However, some features (mainly for improved user experience) may require inter-namespace communication. Specifically, when installing an untrusted app, our `PackageInstaller` needs to notify AirBag for seamless installation. To achieve that, we virtualize a network device [17] inside AirBag and connect it to a pre-allocated bridge interface on the native Android system. By building such an internal channel for "inter-namespace" communication, we can naturally enable networking and telephony support inside AirBag.

By instantiating two different namespaces on the same kernel, our prototype needs to keep track of the current active namespace, which is needed to enable context-aware device virtualization (Section III-B). Specifically, we need to export the related namespace information to corresponding OS components (e.g., framebuffer/GPU drivers) such that they can properly route or handle hardware device accesses from different namespaces. For instance, when a user-level process requests to update the framebuffer, we need to update the respective memory blocks associated with its namespace in OS kernel. Fortunately, when a process is `clone`'d with the `CLONE_NEWNS` flag, an instance of `struct nsproxy` would be allocated in Linux kernel to store the information such as `utsname` and filesystem layout of the new namespace. Given that all processes belong to the same namespace share the same `nsproxy` data structure, our current prototype simply uses it as the namespace identifier. When a process accesses system resources (e.g., via `ioctl`), we consult the `nsproxy` pointer of its `task_struct` via the `current` pointer and use it to guide proper access to virtualized system resources. For bookkeeping purpose, we maintain an internal mapping table which records the related `nsproxy` pointer for each namespace. In our prototype, we find it sufficient to support two namespaces, one for the native Android runtime and another for AirBag. The corresponding entry is dynamically created when the respective first process (i.e., `init` or `airbag_init`) is launched.

TABLE I. SUPPORTED ANDROID HARDWARE DEVICES IN AIRBAG

Hardware Device	Description
Audio	Audio Playback and Capture
Framebuffer	Display Output
GPU	Graphics Processor
Input	Touchscreen and Buttons
IPC	Binder IPC Framework
Networking	WiFi Network Interface
pmem	Physical Memory Allocator
Power	Power Management (Suspend/Resume)
RTC	Real Time Clock
Sensors	Temperature, Accelerometer, GPS
Telephony	Cellular Radio (GSM, CDMA)

B. Context-Aware Device Virtualization

Our prototype permits contentious accesses from the two running namespaces. To accommodate that, AirBag effectively multiplexes their accesses to various system resources in a way transparent to user-level apps (so that normal user experience will not be compromised). In Table I, we show the list of virtualized hardware devices supported in Airbag. Due to page limit, we will explain the six representative hardware devices in more details.

1) Framebuffer/GPU: In AirBag, one of the most important devices for virtualization is the device screen, which involves the respective framebuffer and GPU. Specifically, in Android, all the visual content to be shown by running apps are synthesized by the screen updater (`SurfaceFlinger`) to the framebuffer memory, which is allocated from the OS kernel but mapped to userspace. Any update will trigger the framebuffer driver to issue DMA operations and display the synthesized image to the device screen. Since we have only one device screen and there exist two screen updaters from two different namespaces, we need to regulate which one will gain actual access to the screen.

For isolation purposes, our prototype allocates a second framebuffer memory exclusively for the AIR runtime so that each updater can update its own framebuffer without affecting each other. But the underlying hardware driver will only deliver the framebuffer from the active namespace to the screen. In our prototype, since the framebuffer memory is mapped into the GPU's private page table and the page table can be dynamically updated at runtime, we choose to only activate the framebuffer memory in GPU from the active runtime.

Our solution works well in all three experimented mobile devices. However, the prototype on Nexus One deserves additional discussions. To efficiently manage and allocate physical memory for GPU, the Android support on Nexus One has a physical memory allocator called `pmem`. The user-level screen updater will request physical memory from the `/dev/pmem` device. In order for the GPU and the upper-layer screen updater to render on the screen, a 32MB contiguous physical

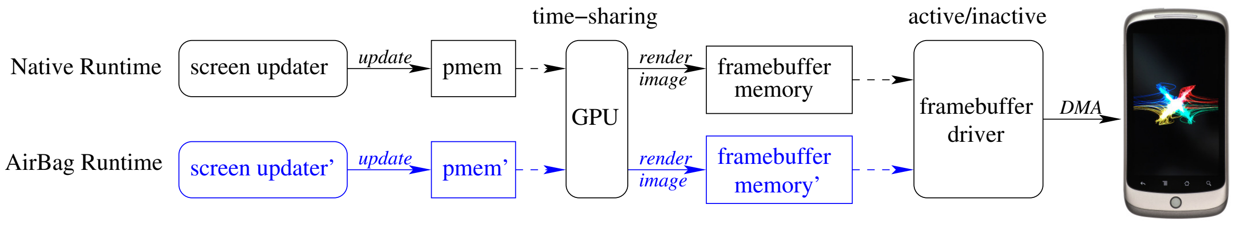


Fig. 2. Framebuffer Virtualization in AirBag (Nexus One)

memory block has been reserved for `/dev/pmem`. With two instantiated runtimes, an intuitive solution will be to double the memory reservation and dynamically allocate the first half for the original Android runtime and the second half for AIR. In fact, we indeed implemented this approach but painfully realized that there also exist lots of other meta information associated with `/dev/pmem`, which also need to be decoupled for namespace awareness. For portability, we aim to avoid changing the internal logic. We then devise another solution by creating a separate `/dev/pmem` device for each namespace (while still doubling the memory reservation). From the upper-layer runtime perspective, it is still accessing the same `/dev/pmem` device. But in our OS extension, we dynamically map the device file to `/dev/pmem_native` and `/dev/pmem_airbag` respectively to maintain transparency and consistency inside the original `pmem` driver as well as upper-layer screen updaters. In Figure 2, we summarize the interaction between the screen updaters, decoupled `pmem` device, GPU, and framebuffer drivers on our Nexus One prototype.

2) Input Devices: After creating a distinct framebuffer for each namespace, our next step is to appropriately deliver events from various input devices (e.g., touchscreen, buttons, and trackball) to the right namespace. Interestingly, Linux kernel has designed a generic layer, i.e., *evdev* (event device), which connects various input device drivers to upper-layered software components. The presence of such layer makes our prototype relatively straightforward. Specifically, the Android runtime (or its service daemons) will listen to input events (e.g., touchscreen and trackball) by registering itself as a client represented as `evdev_client` in OS kernel. When the underlying driver is notified with a pending input event from hardware (e.g. a tap on the touchscreen), the event is delivered to all the registered clients. Therefore, upon the input event registration, we will record its namespace into the `evdev_client` data structure. When an input event occurs, similar to the framebuffer driver, we deliver it only to the registered clients from the active namespace. In other words, all other clients from inactive namespace will not be notified about the event.

3) IPC: After handling basic input and (screen) output devices, we find they are still insufficient to properly set up the AIR environment. It turns out that the problem is due to the custom IPC mechanism in Android. Specifically, unlike the traditional Linux IPC that is already isolated by different namespaces (or `cgroups`), a custom IPC driver named `binder` is developed in Android. With the `binder` driver, a special daemon `servicemanager` will register itself as the binder context manager during the loading process of Android. After that, various service providers will register themselves (via *addService*) so that other service users can look up and ask for their services (via *getService*). Note that all these operations are performed by passing IPC messages through `/dev/binder`.

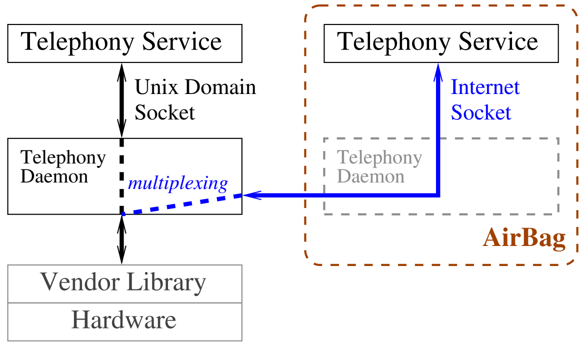


Fig. 3. Telephony Virtualization in AirBag

To virtualize `/dev/binder`, we create a separate context manager for AIR so that all subsequent services registration or lookup will be performed independently within AirBag. In our prototype, we have similarly created an array of context managers indexed by respective namespace. With that, both native runtime and AIR have their own `servicemanager` daemons registering as the context managers that handle follow-up *addService/getService* operations independently, such that all inter-app communications (e.g., *intents*) are fully supported within AirBag. Also, notice that `binder` is the first system resource the Android runtime acquires, we can also conveniently consider the moment when the device file `/dev/binder` is being opened as the indication that a new namespace needs to be created.

4) Telephony: The telephony support in Android largely relies on a service daemon, `rild`, which loads vendor-proprietary library (e.g., `libhtc_ril.so`) for controlling the underlying hardware. In particular, a Java class `com.android.internal.telephony.RIL` of Android runtime communicates with `rild` via an Unix domain socket (created by `rild`) to proxy various telephony services. To support necessary telephony functions inside AIR, as we do not have access to vendor-specific source code, we choose to multiplex the hardware access at the user level `rild`. Specifically, in our prototype, we create a TCP socket along with the normal Unix domain socket in `rild` that runs in the native runtime. The new TCP socket is used to accept incoming connections from the `com.android.internal.telephony.RIL` inside AirBag (Figure 3). In other words, the `rild` inside AirBag is disabled (by adjusting the internal startup script `init.rc`). By design, our current prototype allows for outgoing phone calls from AirBag, but any incoming phone calls will be automatically answered in the native runtime.[3]

5) Audio: For the audio device, we find the support on Nexus One straightforward as it exports a device file

[3]If the native runtime is currently not active when an incoming phone call is received, we will automatically activate it to achieve the same level of user experience.

/dev/q6dsp that allows for concurrent accesses. However, the support on Nexus 7 and Galaxy S III is rather complicated. Specifically, both devices adopt the standard ALSA-based audio driver [18] in OS kernel, which allows only one active audio stream. In other words, if one namespace is currently accessing the device, the other will not be able to access it. Specifically, the process trying to access the audio device would be put into a wait queue when the device is in use.

In our prototype, we take a similar approach with the /dev/pmem device. Specifically, we add a separate virtual audio stream for each namespace so that it will maintain exclusive use within respective namespace. The virtual audio stream from the active namespace will be bound to the hardware audio stream at runtime. For example, in ALSA, an ioctl operation, i.e., SNDRV_PCM_IOCTL_WRITEI_FRAMES is used to send audio data to the device. Such an ioctl from the inactive runtime would silently return without actually sending data to the hardware. But for other ioctls to retrieve or update hardware states such as SNDRV_PCM_IOCTL_SYNC_PTR, we maintain its own latest cache of the states, which will then be applied to hardware when its namespace becomes active. When an inactive namespace becomes active, it is allowed to preempt the use of the audio device.

6) Power Management: The presence of two runtimes also complicates the power management. For example, when an untrusted game app runs inside AirBag for a while, the native runtime may time out and attempt to perform early suspend on the entire phone, which includes turning off the screen. To avoid causing inconvenience, our current prototype chooses to disable any power-related operations from AirBag. In other words, we only allow the native runtime to turn off or dim the screen. In order to prevent the native runtime to sleep while AirBag is active, it will require a wakelock [13] in the native runtime before activating the AIR. The AIR still maintains its own timeout for screen turn-off. But instead of actually turning off the screen, it will release the wakelock. Also, when the app inside AirBag terminates, it will then release the wakelock and yield the control back to the native runtime.

C. Decoupled App Isolation Runtime

With a separate app isolation runtime, we have the opportunity to customize it to better confine untrusted apps without affecting the original native runtime. As mentioned earlier, we build the AIR by customizing Android Open Source Project (AOSP 4.1.1) to export the same interface while in the meantime allowing users to choose different running modes. In particular, the AIR's root directory is relocated with the pivot_root system call (so that any write operation issued in AirBag would not corrupt the original files in the firmware). Specifically, we build a unionfs [48] that copy-on-writes all updates in a file-based ext4 disk image and uses a base filesystem as a squashfs image for read-only operations. Such an organization enables us to readily provide the "restore to default" feature, which essentially removes the dirty file-based ext4 disk image. Also, our system eliminates all potential personally-identifying information from AIR for the "incognito" mode. For instance, the Android API TelephonyManager.getDeviceId() has been instrumented to return a faked IMEI number.

The layered design of AOSP also provides the opportunity to profile app behavior. For example, while analyzing a malware, we usually leverage logcat, to record various Android API calls we are interested in. We note that the collected log entries are pushed down from the namespace in which the untrusted app runs, which does lead to the concern of trustworthiness of collected log. However, from another perspective, the actual dumped message is maintained by the kernel-level log driver, which is assumed to be trusted (Section II). Moreover, the profiling mode will turn on the systemtap support [16] to record syscalls from AirBag (with confined apps) to external SD card for in-depth analysis.

In addition, our system also instruments the AIR to prevent untrusted apps from performing stealthy actions (e.g., sending SMSs to premium-rate numbers). In particular, by modifying the Android API in com.android.internal.telephony.RIL class, the untrusted app running inside AirBag mode is prevented from performing any stealthy telephony action. Further, thanks to the cgroup abstraction, we could white-list the devices for AirBag access. Specifically, before starting the AirBag namespace, we can write each allowed device file name with the corresponding permission to the cgroups virtual filesystem (e.g. /cgroup/airbag/devices.list). After that, all the access to the device files not listed in the white-list would be automatically blocked.

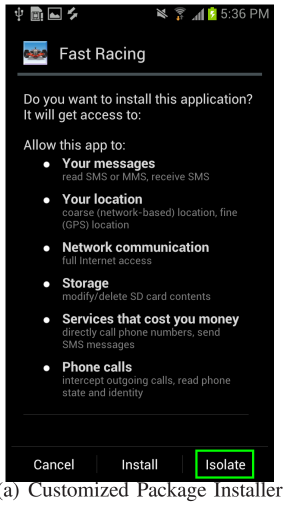
To maintain transparency, our scheme is seamlessly integrated with the native system without breaking user experience. Specifically, when the system boots up, the AirBag environment is automatically initiated and then suspended. Its suspension will be removed in two scenarios when the user either (1) dispatches an app to it for isolation or (2) launches a previously isolated app. In the first case, our customized PackageInstaller automatically guides the installation procedure by simply adding an "isolate" button (Figure 4(a)). For each isolated app, our system will register an "app stub" in the native Android runtime. In Figure 4(b), we show the example app stub for an isolated game app (com.creativemobi.DragRacing). For comparison, we also install the same game app inside the native runtime. The difference in their icons is the addition of a lock sign on the icon associated with the isolated app. When the user clicks the app stub, AirBag is activated to execute the isolated app, which transparently marks native runtime inactive and thus yields underlying hardware accesses to AirBag. When the app terminates, AirBag would make itself inactive and seamlessly bring the native runtime up-front.

D. Lessons Learned

In the process of developing our early prototype on Nexus 7, we encounter an interesting problem that a benchmark program running inside the AirBag always scores one fourth of normal system, which indicates that AirBag only utilizes one of the four available CPU cores. After further investigation, it turns out that Nexus 7 has a CPU hotplug mechanism that can dynamically put CPU cores online or offline based on the workload of the whole system. However, due to a bug [8] in Linux kernel 3.1.10, the CPU online events are not properly delivered to AirBag, which then fails to scale up the computation power when AirBag is fully loaded but the native runtime is idle. We then backport the patches from mainline

TABLE II. EFFECTIVENESS OF AIRBAG IN SUCCESSFULLY BLOCKING 20 REPRESENTATIVE ANDROID MALWARE

Malware Family	Malicious Behavior					
	Retrieve IMEI	Retrieve Phone Number	Send SMS	Intercept SMS	Record Audio	Damage Firmware (w/ root exploits)
BeanBot	√	√	√	√		
DKFBootKit	√	√				√
DroidKungFu	√	√				√
DroidLive	√		√	√		
Fjcon			√	√		
Geinimi	√	√	√	√		
GingerMaster		√				√
GoldDream	√	√	√	√		
HippoSMS			√	√		
NickiBot	√		√	√	√	
RogueLemon			√	√		
RogueSPPush			√	√		
RootSmart	√					√
SMSSpoof			√			
SndApps		√				
Spitmo			√	√		
TGLoader			√			√
YZHCSMS		√	√	√		
Zitmo				√		
Zsone			√	√		


5:36 PM

Fast Racing

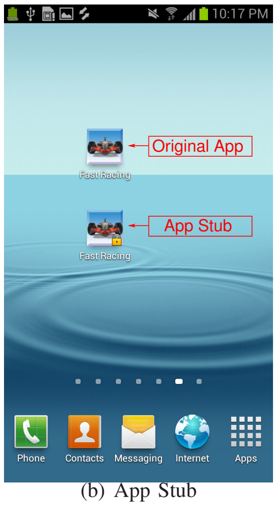
Do you want to install this application?
It will get access to:

Allow this app to:

- **Your messages**
  read SMS or MMS, receive SMS
- **Your location**
  coarse (network-based) location, fine (GPS) location
- **Network communication**
  full Internet access
- **Storage**
  modify/delete SD card contents
- **Services that cost you money**
  directly call phone numbers, send SMS messages
- **Phone calls**
  intercept outgoing calls, read phone state and identity

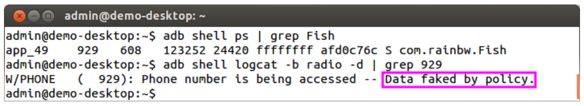
Cancel Install Isolate

(a) Customized Package Installer



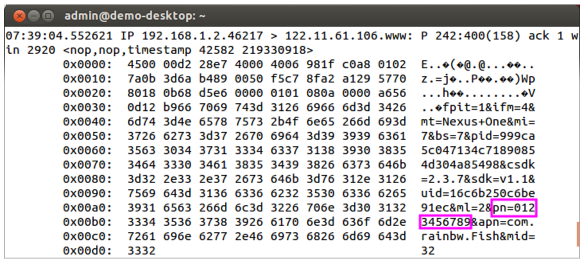
(b) App Stub

Fig. 4. Seamless Integration of AirBag



```
admin@demo-desktop: ~
admin@demo-desktop:~$ adb shell ps | grep Fish
app_49    929   608   123252 24420 ffffffff afd0c76c S com.rainbw.Fish
admin@demo-desktop:~$ adb shell logcat -b radio -d | grep 929
W/PHONE   (  929): Phone number is being accessed -- Data faked by policy.
admin@demo-desktop:~$
```

(a) Faked phone number is being accessed



```
admin@demo-desktop: ~
07:39:04.552621 IP 192.168.1.2.46217 > 122.11.61.106.www: P 242:400(158) ack 1 w
in 2920 <nop,nop,timestamp 42582 219330918>
        0x0000:  4500 00d2 28e7 4000 4006 981f c0a8 0102  E..◆(◆@.@...◆◆..
        0x0010:  7a0b 3d6a b489 0050 f5c7 8fa2 a129 5770  z.=j◆..P◆◆.◆◆)Wp
        0x0020:  8018 0b68 d5e6 0000 0101 080a 0000 a656  ...h◆◆........◆V
        0x0030:  0d12 b966 7069 743d 3126 6966 6d3d 3426  ..◆fpit=1&ifm=4&
        0x0040:  6d74 3d4e 6578 7573 2b4f 6e65 266d 693d  mt=Nexus+One&mi=
        0x0050:  3726 6273 3d37 2670 6964 3d39 3939 6361  7&bs=7&pid=999ca
        0x0060:  3563 3034 3731 3334 6337 3138 3930 3835  5c047134c7189085
        0x0070:  3464 3330 3461 3835 3439 3826 6373 646b  4d304a85498&csdk
        0x0080:  3d32 2e33 2e37 2673 646b 3d76 312e 3126  =2.3.7&sdk=v1.1&
        0x0090:  7569 643d 3136 6336 6232 3530 6336 6265  uid=16c6b250c6be
        0x00a0:  3931 6563 266d 6c3d 3226 706e 3d30 3132  91ec&ml=2&pn=012
        0x00b0:  3334 3536 3738 3926 6170 6e3d 636f 6d2e  3456789&apn=com.
        0x00c0:  7261 696e 6277 2e46 6973 6826 6d69 643d  rainbw.Fish&mid=
        0x00d0:  3332                                     32
```

(b) Faked phone number is being uploaded

Fig. 5. GoldDream Analysis

Linux kernel [10] to have AirBag informed about the status of available CPU cores whenever a CPU core is online or offline.

Another issue we encountered in our prototype is related to the low-memory killer, which will be waked up to sacrifice certain processes when the system is under high memory pressure. As our prototype supports two concurrent namespaces, the unknowing low-memory killer may pick up a process from the active namespace as victim for termination, which greatly affects user experience. Therefore, our prototype adjusts the algorithm and makes it in favor of choosing processes from inactive runtime as victims to maintain responsive user experience.

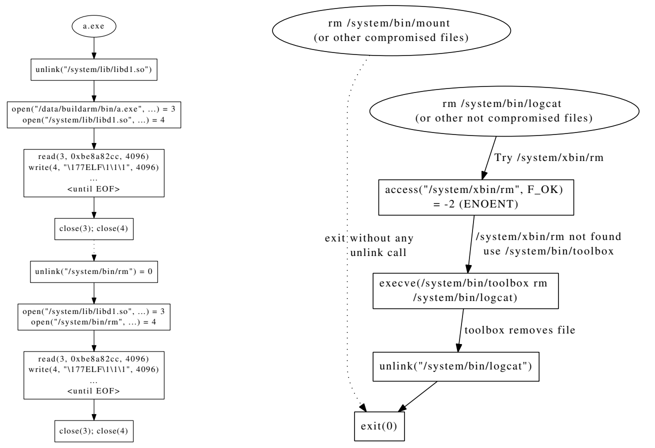
IV. EVALUATION

In this section, we present the evaluation results by first showing the effectiveness of AirBag with various mobile malware. We then measure the impact on performance as well as power consumption and memory usage.

A. *Effectiveness*

To evaluate the effectiveness, we selected 20 Android malware that present a good coverage of state-of-the-art mobile malware in the wild. In Table II, we show the list and their malicious behavior which is manually triggered. Specifically, AirBag is able to successfully isolate these malicious apps and prevent them from performing the malicious operations in either Android framework level or OS kernel level. For example, the way AirBag detects and prevents `NickiBot` from recording audio is done by hooking the corresponding ioctls (e.g., `SNDRV_PCM_IOCTL_READI_FRAMES`) of the ALSA-based audio driver [18] in OS kernel while the relocation of the AIR's root directory and the usage of `unionfs` (Section III-C) enable us to prevent firmware damages. We emphasize that AirBag in all three supported mobile devices is able to achieve the same results.[4] In the following, we present details of three representative experiments, to demonstrate the values from incognito mode, profiling mode, and flexible user confirmation for sensitive operations, respectively.

[4]The exceptional case is the Nexus 7 that is a tablet and does not have necessary telephony support. However, it does not affect AirBag's effectiveness in isolating these apps.


a.exe
unlink("/system/lib/libd1.so")
open("/data/buildarm/bin/a.exe", ...) = 3
open("/system/lib/libd1.so", ...) = 4
read(3, 0xbe8a82cc, 4096)
write(4, "\177ELF\1\1\1", 4096)
...
<until EOF>
close(3); close(4)
unlink("/system/bin/rm") = 0
open("/system/lib/libd1.so", ...) = 3
open("/system/bin/rm", ...) = 4
read(3, 0xbe8a82cc, 4096)
write(4, "\177ELF\1\1\1", 4096)
...
<until EOF>
close(3); close(4)
rm /system/bin/mount
(or other compromised files)
rm /system/bin/logcat
(or other not compromised files)
Try /system/xbin/rm
access("/system/xbin/rm", F_OK)
= -2 (ENOENT)
exit without any
unlink call
/system/xbin/rm not found
use /system/bin/toolbox
execve(/system/bin/toolbox rm
/system/bin/logcat)
toolbox removes file
unlink("/system/bin/logcat")
exit(0)

(a) Payload Execution (b) `/system/bin/rm` Execution

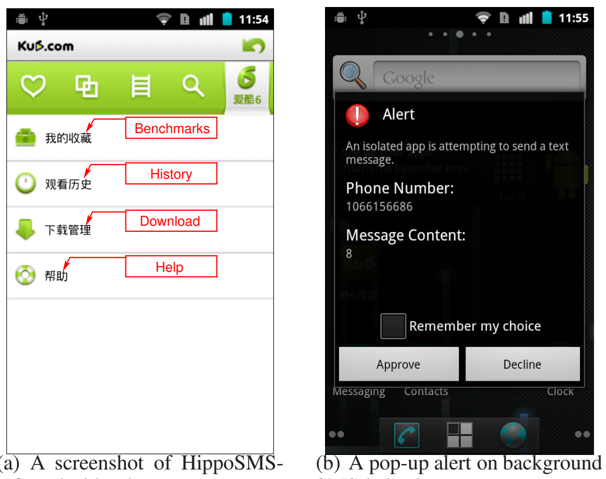
Fig. 6. DKFBootKit Analysis

1) `GoldDream` *Experiment:* This malware [4] infected Android systems by hiding in popular game apps. It spies on SMS messages received by users, monitors incoming/outgoing phone calls, and then stealthily uploads them as well as device information to a remote server without user's awareness. Specifically, by registering a receiver for various system events (e.g., when a SMS message is received), `GoldDream` launches a background service without user's knowledge to monitor and upload private information.

With AirBag, this malware is automatically dispatched to run inside the isolated AIR, instead of the native runtime. Also, the spying activities are effectively blocked as various system-wide events are by default isolated from the native runtime to AIR. In Figure 5, we show how the incognito mode is helpful to prevent real phone information from being leaked by a `GoldDream`-infected game app `com.rainbw.Fish`. In this experiment, we capture incoming/outgoing network traffic of AirBag with `tcpdump` when the malware runs. From the dumped log, we observed the collected IMEI number and phone number were being uploaded in an HTTP message to a remote server. Figure 5(a) shows the recorded malware behavior of retrieving the phone number (faked to be `0123456789` in our prototype). Figure 5(b) highlights the collected (fake) phone number being reported back to a remote server.

2) `DKFBootKit` *Experiment:* The previous experiment effectively blocks malware's spying behavior and prevents private information from being leaked. In this experiment, we further demonstrate how AirBag can prevent the firmware from being manipulated by malware. In this case, we experimented with `DKFBootKit` [14], an Android malware that infects the boot sequence of Android (not the bootloader) and replaces a few system utilities such as `ifconfig`, `rm`, and `mount` under the system partition.

With AirBag, `DKFBootKit` will not be able to cause any damage to our system. First, the native filesystem is completely isolated from the AIR on which the `DKFBootKit` runs. Second, the changes inflicted by `DKFBootKit`, while visible inside AirBag, are automatically copy-on-written to a separate file. With that, we can not only conveniently analyze the contamination from the malware (Section III-C), but also apply "restore to default" feature to undo the changes. Moreover, with profiling mode, we collected syscalls from AirBag including



Alert

An isolated app is attempting to send a text message.

Phone Number:
1066156686

Message Content:
8

Remember my choice

Approve Decline

(a) A screenshot of HippoSMS-infected video browser (b) A pop-up alert on background SMS behavior

Fig. 7. HippoSMS Analysis

TABLE III. BENCHMARKS USED IN OUR EVALUATION

Benchmark Name	Version	Workload Type
AnTuTu Benchmark [5]	2.8.3	Combination
BrowserMark [7]	2.0	CPU/IO
NenaMark2 [11]	2.3	GPU
Neocore [12]	1.9.35	GPU
SunSpider [15]	0.9.1	CPU/IO

confined processes to monitor the detailed infection sequence. From the infection sequence, we notice that `DKFBootKit` will release at runtime a payload file named `a.exe`, which when executed will copy it to `/system/lib/libd1.so` and further replace a few other files, such as `rm` and `mount` (Figure 6(a)). It turns out the replacement of `rm` is to protect various malware files. In Figure 6(b), we report the internal logic of the replaced `rm`, which basically checks arguments and avoids removing infected files. (For other files, the compromised `rm` proceeds normally by invoking `/system/xbin/rm` or `/system/bin/toolbox`.)

3) `HippoSMS` *Experiment:* In this experiment, we present the capability of exposing stealthy malware behavior and how users can dynamically block them. Specifically, we run an Android malware `HippoSMS` [3] inside AirBag. As the name indicates, this particular malware sends text messages to a premium-rate number that incurs additional phone charges. Notice that the only interface to access the telephony hardware is the `rild` daemon running in the native runtime. And any telephony-related operation inside AirBag will be tunneled out to native runtime. The user will then have the option to either allow or disallow it. By doing so, we can effectively expose any background behavior that is often go unnoticed in a normal system (without AirBag). In Figure 7(a), we show a screenshot of a `HippoSMS`-infected video browser that is involved in background SMS behavior. The background SMS-sending behavior is intercepted and reported to user in a pop-up window – Figure 7(b). The user then has the option to permit or deny it.

B. Performance Impact

To evaluate AirBag's impact on performance, we have performed benchmark-based measurements on three supported

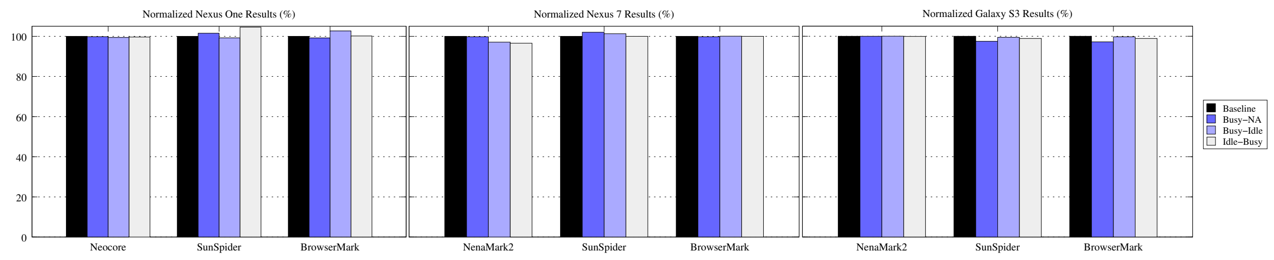


Fig. 8. Performance Measurement of AirBag on Google Nexus One, Nexus 7, and Samsung Galaxy S III

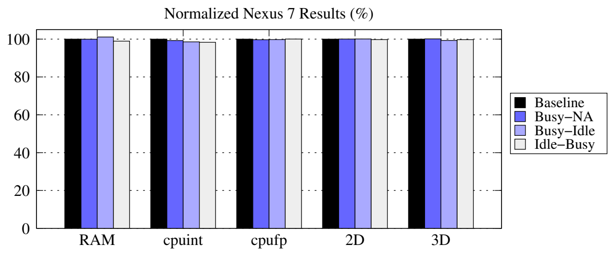


Fig. 9. AnTuTu Measurement Results

devices – with and without AirBag. Table III shows the list of benchmarks used in our measurement.

These benchmark programs are designed to measure various aspects of system performance. For each benchmark program run, we have measured the performance in four different settings: (1) "Baseline" means the results obtained from a stock mobile device without AirBag support; (2) "Busy-NA" means the results from a mobile device with our OS kernel extension for AirBag but without activating the AirBag; (3) "Busy-Idle" means results from an AirBag-enhanced system by running the benchmark program in the native runtime while keeping AirBag idle; and (4) "Idle-Busy" means results from an AirBag-enhanced system by running the benchmark program inside the AirBag while keeping the native runtime idle. All the performance results are normalized with the "Baseline" system to expose possible overhead introduced by AirBag. Figure 8 summarizes the measurement results. Overall, our benchmark experiments show that AirBag incurs minimal impact on system performance (with around 2.5%) in both GPU-intensive workloads (Neocore and NenaMark2) and CPU/IO-intensive workloads (SunSpider and BrowserMark). We also run AnTuTu [5], a comprehensive benchmark that reported similar small performance overhead (with around 2% – Figure 9). We point out that our experiments so far are conducted by entering the default incognito mode. When we turn the profiling mode on, the evaluation with Neocore benchmark indicates that our system introduces additional 10% overhead. We are not concerned as the profiling mode is only turned on when performing a forensics-style investigation of an untrusted app.

C. Power Consumption and Memory Usage

Beside the performance overhead, we also evaluate the impact of AirBag on battery use. With two concurrent namespaces, our system likely incurs additional battery drains. In our measurement, we perform two sets of experiments. In the first set, we start from a fully-charged Nexus 7 device, wait for 24 hours without running any workload, and then check its battery level. The stock system reports 91%, and AirBag-enhanced system shows 89%, indicating 2% more battery use. In the second set, we also start from a fully-charged Nexus 7 device, wait for 24 hours while keeping playing an audio file, and then check its battery level. The stock system reports 66%, and AirBag-enhanced system shows 63%, indicating 3% more battery use.

Finally, we also measure the memory footprint of AirBag. Specifically, we examine the percentage of in-use memory (by reading `/proc/meminfo`) of the Nexus 7 by repeating the previous two sets of experiments. Instead of waiting for 24 hours, we collect our measurement results in 4 hours. The results from the first set of experiments indicate that our system increases the percentage of in-use memory from 59.31% to 60.87%, an addition of 1.56%. In the second set of experiments (with repeated playing of an audio file), the percentage of in-use memory is increased from 60.25% to 63.70%. The additional memory consumption is due to the reserved memory blocks in OS kernel (e.g., for second framebuffer).

V. Discussion

In this section, we re-visit our system design and implementation for possible improvements. First, the current usage model of AirBag is to isolate untrusted apps when they are being installed. While it achieves our design goals, it can still be improved with a unique capability to dynamically migrate apps between native and AirBag-confined runtime environments. For example, users may want to try the new features of newly released apps in the AirBag without affecting the native environment but "move" it to the native runtime environment when the app is considered safe and stable. On the other hand, when an app is reported to have malicious behavior (e.g., sending text messages in the background), users can still use the app by limiting its capabilities within the AirBag. Obviously, one solution will be simply uninstalling the app in one runtime and then re-install it in another runtime. However, it will lose all internal states accumulated from previous installation. A better solution might lively migrate it from one to another. This is possible as both runtime environments share the same trusted OS kernel, though in different namespaces. Possible challenges however may include handling dependent libraries that may be inconsistent in different runtimes as well as other currently interacting apps in the previous namespace.

Second, to confine untrusted app execution, our prototype disallows confined apps to communicate with other legitimate apps and service daemons running on the native runtime and vice versa. As a result, various system events are isolated at the AirBag boundary. In other words, when there is an incoming SMS or phone call on the native runtime, such an event will not be propagated to the AIR runtime, which will affect certain functionality of untrusted apps. Also, automatic updates on AirBag-confined apps may also break because of the current AirBag confinement. While an intuitive solution is to allow these events to cross the AirBag boundary, it may however break the isolation AirBag is designed to enforce. From another perspective, we are motivated to explore a hybrid approach, which might be ideal in selectively white-listing certain events to pass through (so that we can support legitimate feature needs such as automatic updates) without unnecessarily compromising AirBag isolation. On the other hand, if AirBag is configured to deny all permissions, our system could be considered to be replaced by a customized Android system. However, with our system, users can still run apps normally in the native runtime on the same mobile device which cannot be achieved by customized Android systems.

Third, our current prototype is still limited in supporting one single AirBag instance and multiple untrusted apps will need to run within the same instance. This leads to problems when all apps are installed as untrusted. In particular, AirBag does not provide inter-app isolation within itself. Naturally, we can improve the scalability of AirBag by dynamically provisioning multiple AirBag instances with one for each untrusted app. It does raise challenging requirements for more efficient and lightweight AIRs. Note that our AirBag filesystem already made use of copy-on-write to keep all the updates in a separate data file, which should be scalable to multiple AirBag instances. However, context-aware device virtualization requires additional memory to be reserved (e.g., for smooth framebuffer support – Section III-B). It remains an interesting challenge and we plan to explore possible solutions in our future work (e.g., by leveraging hardware virtualization support in latest ARM processors).

Fourth, as an OS-level kernel extension, our approach requires updating the smartphone OS image for the enhanced protection against mobile malware infection. While this may be an obstacle for its deployment, we argue that our system does not require deep modifications in smartphone OS kernel. In fact, our kernel patch has less than $2K$ lines of source code and most of them are related to generic Linux drivers, not tied to specific hardware devices in different smartphone models. Furthermore, we can improve the portability of our system by implementing a standalone loadable kernel module that can be conveniently downloaded and installed.

Fifth, for simplicity, our current prototype does not provide the same runtime environment as the original one. Because of that, a malicious app can possibly detect the existence of AirBag and avoid launching their malicious behaviors. In fact, as an OS-level virtualization solution, our system shares with other virtualization approaches [43], [19], [35], [40], [49] by possibly exposing virtualization-specific artifacts or footprints. Note that with the capability of arbitrarily customizing the isolated runtime environment (AIR), we are able to further improve the fidelity of AirBag runtime and make it harder to be fingerprinted. However, this situation could lead to another round of "arms race." From another perspective, if a mobile malware attempts to avoid launching its attacks in a virtualized environment, our system does achieve the intended purpose by resisting or deterring its infection.

Last but not least, with a decoupled app isolation runtime to transparently support untrusted apps, AirBag opens up new opportunities that are not previously possible. For example, our current profiling mode basically collects `logcat` output as well as various syscalls from AirBag. However, it does not need to be limited in basic log collection. For example, recent development on virtual machine introspection [35], [40], [29], [36], [56] can be applied in AirBag to achieve better introspection and monitoring capabilities. Moreover, it also provides better avenues to integrate with current mobile anti-virus software so that they can reliably monitor runtime behavior without being limited in only statically scanning untrusted apps.

VI. Related Work

In this section, we categorize related work into different research areas and compare our system with them.

Server-side protection The first category of related work include systems that are designed to improve the walled garden model in detecting and pruning questionable apps (including malicious ones) from centralized mobile marketplaces. For example, Google introduces the bouncer service in February, 2012. Besides smartphone vendors, researchers also endeavor to develop various systems to expose potential security risks from untrusted apps. PiOS [30] statically analyzes mobile apps to detect possible leaks of sensitive information; Enck *et al.* [32] studies free apps from the official Google Play with the goal of understanding broader security characteristics of existing apps. Our system is different by proposing a complementary client-side solution to protect mobile devices from being infected by mobile malware.

Client-side protection The second category aims to develop mitigation solutions on mobile devices. For example, mobile anti-malware software scan the apps on the devices based on known malware signatures, which limit their capability in detecting zero-day malware. MoCFI [27] provides a CFI enforcement framework to prohibit runtime and control-flow attacks for Apple iOS. TaintDroid [31] extends the Android framework to monitor the information flow of privacy-sensitive data. MockDroid [21], AppFence [38], Kantola *et al.* [42], Airmid [44], Apex [45], and CleanOS [51] also rely on extensions on Android framework to better control apps' access to potential sensitive resources. Aurasium [55] takes a different approach by repackaging untrusted apps and then enforcing certain access control policies at runtime. With varying levels of successes, they share a common assumption of a trustworthy Android framework, which unfortunately may not be the case for advanced attacks (that could directly compromise privileged system daemons such as `init` or `zygote`). In contrast, our system assumes that the Android framework inside AirBag could be compromised (by untrusted apps) but the damages are still contained in AirBag to prevent the native runtime environment being affected.

From another perspective, a number of systems have been proposed to extend the Android permission system. For example, Kirin [33] analyzes apps at install time to block apps with a dangerous combination of permissions. Saint [47] enforces policies in both install time and run time to govern the assignment as well as the usage of permissions. Stowaway [34] identifies the apps which request more permissions than necessary. In comparison, our system is different in not directly dealing with Android permissions. Instead, we aim to mitigate the risks by proposing a separate runtime that is isolated and enforced through a lightweight OS-level extension.

Virtualization The third category of related work includes recent efforts to develop or adopt various virtualization solutions which can strengthen the security properties of mobile platforms [53]. Starting from the approaches based on Type-I hypervisors (e.g., OKL4 Microvisor [46], L4Android [43], and Xen on ARM [39]), they may have smaller TCB but require significant efforts to support new devices and cannot readily leverage commodity OS kernels to support hardware devices. In a similar vein, researchers have also applied traditional Type-II hypervisor approaches on mobile devices (e.g., VMware's MVP [20] and KVM/ARM [26]). Compared to Type-I hypervisors, Type-II hypervisors might take advantage of commodity OS kernels to support various hardware devices. However, it still needs to run multiple instances of guest OS kernels, which inevitably increase memory footprint and power consumption. Also, the world switching operation causes additional performance degradation, which affects the scalability in resource-constrained mobile device environments.

Beside traditional Type-I and Type-II hypervisors, OS-level virtualization approaches are also being applied to mobile devices. For example, Cells [19] introduces a foreground /background virtual phones usage model and proposes a lightweight OS-level virtualization to multiplex phone hardware across multiple virtual phones. Our system differs from Cells in two important aspects: First, as mentioned earlier, Cells aims to embrace the emerging "bring-your-own-device" (BYOD) paradigm by supporting multiple virtual phone instances in one hardware device. Each virtual phone instance is treated equally and the isolation is achieved at the coarse-grained virtual phone boundary. AirBag instead is an app-centric solution that aims to maintain a single phone usage model and the same user experience while enforcing reliable isolation of untrusted apps. Second, to support multiple virtual phones, Cells needs to maintain an always-on root namespace for their management and hardware device virtualization. In comparison, AirBag is integrated with the native runtime for seamless user experience without such a root namespace. At the conceptual level, the presence of a root namespace is similar to the management domain in Type-I Xen hypervisor, which could greatly affect the portability on new phone models. Being a part of native system, our system can be readily ported to new devices with stock firmware.[5]

In addition, researchers also explore user-level solutions to provide separate mobile runtime environments. For example, TrustDroid [22] enhances the Android framework to provide domain-level isolation that confines the unauthorized data access and cross-domain communications. Recent Android release (Jellybean 4.2) extends the Android framework to add multi-user support. Such a user-level solution requires a trustworthy framework that is often the target for advance attacks. Moreover, these solutions require deep modifications on the Android framework. In comparison, AirBag adds a lightweight OS-level extension to confine cross-namespace communications without affecting the native Android framework, achieving backward and forward compatibility.

Virtualization-based security The last category of the related work includes a long stream of research projects to improve host security with virtualization: [28], [40], [41], [50], [54]. For example, Ether [28] transparently traces malware with the help of hardware virtualization extensions. Lockdown [54] divides the runtime environment into trusted and untrusted with a lightweight hypervisor. These systems benefit from a layered architecture design as well as the strong isolation guarantee provided by underlying virtualization. With a decoupled runtime environment to transparently confine user-level apps, AirBag can be naturally combined with the above approaches for better protection of Android-based mobile devices.

[5]Our prototyping experience confirms that AirBag can be readily ported to a new phone model. In fact, the very first prototype on Google Nexus One is ported to Nexus 7 and Samsung Galaxy S III each within one week!

VII. Conclusion

We have presented the design, implementation and evaluation of AirBag, a client-side solution to significantly boost Android-based smartphone capability to defend against mobile malware. By instantiating a separate app isolation runtime that is decoupled from native runtime and enforced through lightweight OS-level virtualization, our system not only allows for transparent execution of untrusted apps, but also effectively prevents them from leaking personal information or damaging the native system. We have implemented a proof-of-concept prototype that seamlessly supports three representative mobile devices, i.e., Google Nexus One, Nexus 7, and Samsung Galaxy S III. The evaluation results with 20 representative Android malware successfully demonstrate its practicality and effectiveness. Also, the performance measurement with a number of benchmark programs shows that our system incurs low performance overhead.

Acknowledgment

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "260,000 Android users infected with malware," http://www.infosecurity-magazine.com/view/16526/260000-android-users-infected-with-malware/.

[2] "Android 4.2 potential security features unveiled: SELinux, VPN Lockdown and Premium SMS Confirmation," http://www.androidauthority.com/android-4-2-potential-security-features-unveiled-selinux-vpn-lockdown-premium-sms-confirmation-123785/.

[3] "Android Malware Genome Project," http://www.malgenomeproject.org/.

[4] "Android.Golddream|Symantec," http://www.symantec.com/security_response/writeup.jsp?docid=2011-070608-4139-99.

[5] "AnTuTu Benchmark," http://www.antutulabs.com.

[6] "App Store," http://www.apple.com/iphone/from-the-app-store/.

[7] "BrowserMark," http://browsermark.rightware.com.

[8] "Bug 714271," https://bugzilla.redhat.com/show_bug.cgi?id=714271.

[9] "Google Play," http://play.google.com/.

[10] "linux/kernel/git/torvalds/linux.git," http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git.

[11] "NenaMark2," http://nena.se/nenamark/view?version=2/.

[12] "Neocore," https://play.google.com/store/apps/details?id=com.qualcomm.qx.neocore.

[13] "PM: Implement autosleep and "wake locks", take 3," http://lwn.net/Articles/493924/.

[14] "Security Alert: New Android Malware DKFBootKit Moves Towards The First Android BootKit," http://www.csc.ncsu.edu/faculty/jiang/DKFBootKit/.

[15] "SunSpider JavaScript Benchmark," http://www.webkit.org/perf/sunspider/sunspider.html.

[16] "SystemTap," http://sourceware.org/systemtap/.

[17] "Virtual ethernet device (tunnel)," http://lwn.net/Articles/232688/.

[18] "Advanced Linux Sound Architecture (ALSA) project homepage," http://www.alsa-project.org/main/index.php/Main_Page.

[19] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A Virtual Mobile Smartphone Architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[20] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware mobile virtualization platform: is that a hypervisor in your pocket?" *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, 2010.

[21] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *Proceedings of the 12th International Workshop on Mobile Computing System and Applications*, 2011.

[22] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. 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*, 2011.

[23] "Smart phones overtake client PCs in 2011," http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011.

[24] "CGROUPS," http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[25] "comScore Reports December 2012 U.S. Smartphone Subscriber Market Share," http://www.comscore.com/Insights/Press_Releases/2013/2/comScore_Reports_December_2012_U.S._Smartphone_Subscriber_Market_Share.

[26] C. Dall and J. Nieh, "KVM for ARM," in *Proceedings of the Ottawa Linux Symposium*, 2010.

[27] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[28] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[29] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[30] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[31] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[32] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[33] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[34] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[35] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[36] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the 10th Network and Distributed System Security Symposium*, 2003.

[37] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[38] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[39] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in *Proceedings of the 5th Consumer Communications and Networking Conference*, 2008.

[40] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-based "Out-Of-the-Box" Semantic View Reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[41] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based Hidden Process Detection and Identification using Lycosid," in *ACM International Conference on Virtual Execution Environments*, 2008.

[42] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing Attack Surfaces for Intra-Application Communication in Android," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[43] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: A Generic Operating System Framework for Secure Smartphones," in *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[44] Y. Nadji, J. Giffin, and P. Traynor, "Automated Remote Repair for Mobile Malware," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[45] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[46] "OKL4 Microvisor," http://www.ok-labs.com/products/okl4-microvisor.

[47] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[48] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok, "UnionFS: User- and Community-oriented Development of a Unification Filesystem," in *Proceedings of the 2006 Linux Symposium*, July 2006.

[49] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[50] R. Ta-Min, L. Litty, and D. Lie, "Splitting Interfaces: Making Trust between Applications and Operating Systems Configurable," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[51] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting Mobile Data Exposure with Idle Eviction," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[52] P. Varanasi and G. Heiser, "Hardware-Supported Virtualization on ARM," in *2nd Asia-Pacific Workshop on Systems (APSys'11)*, 2011.

[53] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?" in *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, 2012.

[54] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, 2012.

[55] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[56] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012.