

Smartphone Virtualization

Tzi-cker Chiueh Houcheng Lin Ares Chao Anthony Tan-Gen Wu

Industrial Technology Research Institute
tcc@itri.org.tw

Abstract—Virtualization plays a pivotal role in the success of cloud computing service models and is applied extensively in modern public and private data centers. However, its adoption on end user devices such as laptop/desktop computers and cell phones is relatively scarce, mainly because convincing use cases for client device virtualization have proven elusive so far. As smartphones emerge as the linchpin of everyday computing and communication for regular people and application download becomes a fact of life, the Bring Your Own Cloud (BYOD) problem, in which corporate employees connect their own smartphones to the corporate networks for office work, has put most enterprises in an unenviable position of making a difficult choice between corporate security and employee productivity. One effective solution to the BYOD problem is *smartphone virtualization*, which provides multiple virtual smartphones on a physical smartphone, and enables a user to use a highly secure but not so flexible virtual smartphone in the work environment and a less secure but more flexible virtual smartphone when outside the work environment. This paper describes the design and implementation of a comprehensive smartphone virtualization system called *Brahma*, which consists of a *virtualized smartphone* element and a *virtual mobility infrastructure* element, and presents the detailed evaluation results of the first *Brahma* prototype on a commercial smartphone.

Index Terms—Smartphone; virtualization; virtual mobility infrastructure; sensor redirection; state isolation

I. INTRODUCTION

Virtualization instantiates multiple virtual devices, each with a distinct personality, from a single physical device, which could be a server, a storage appliance or a network of switches. Hardware abstraction layer (HAL) virtualization enables multiple virtual machines to run on a single physical machine, and opens up myriad optimization opportunities for modern data centers that are designed to support cloud computing services, such as thin provisioning, consolidation, load balancing, and live upgrade and migration. Despite its enormous success in data centers, application of HAL virtualization to end user devices such as desktop/laptop computers and smartphones is almost non-existent, because most of the virtualization benefits in data centers cannot easily carry over to end user devices. However, the situation is changing, and the catalyst is the Bring Your Own Device (BYOD) problem.

As smartphones become the linchpin of everyday computing and communication for most people in modern societies, it is only natural that corporate employees choose to connect their personal smartphones to their corporate networks when they are in the office, and perform many workplace tasks right from their smartphones. Connecting personal smartphones to corporate networks in itself would not create any problems,

had these smartphones not been laden with software programs with questionable quality and pedigrees. Unfortunately, most people download applications to their smartphones all the time and do so unscrupulously, and the majority of smartphones today are not equipped with any anti-virus software that purges malicious applications. As a result, the BYOD problem puts corporations in a dilemma, which forces them to choose between company security and employee productivity, an impossible choice.

One way to solve the BYOD problem is to support multiple contexts on a physical smartphone, where each context corresponds to a specific use case. For example, one context could be set up for conducting business tasks, another context could be reserved for personal use like gaming or social network interactions, and the third is for testing of unknown applications. Because different contexts serve different purposes, they should be configured with different security policies and privileges that match their intended use cases. For example, only the business-use virtual smartphone is allowed to connect to the corporate network and accordingly its security policy should conform to the corporate security policy, whereas the security policy of the personal-use virtual smartphone could be more relaxed but it cannot be connected to the corporate network. In addition to different security policies, these smartphone contexts should also be isolated from one another, in the sense that there is no easy way for programs running in one context to read or write data or code belonging to any other contexts running on the same smartphone. The ITRI smartphone virtualization system, code-named *Brahma*, is designed to provide multiple isolated contexts on modern smartphones.

Brahma offers two mechanisms to support multiple usage contexts on a smartphone: *virtualized smartphone*, which runs an HAL hypervisor so as to enable multiple virtual smartphones to run concurrently on the physical smartphone, and *virtual mobility infrastructure* (VMI), which runs, in a cloud-based infrastructure, virtual machines whose inputs/outputs are redirected to physical smartphones with which users interact directly, and presents end users the illusion of using virtual smartphones through their physical smartphones. *Brahma* judiciously combines these two virtualization mechanisms in order to provide the best user experience for different usage scenarios. Although the two mechanisms are conceptually different, their implementations are actually quite similar to each other architecturally, and share a common set of technical challenges as follows:

- The set of virtual smartphones with which the user directly interacts, be they running on a physical smartphone or in the cloud, must be able to reliably and efficiently receive the user's inputs, e.g., touch and speech, and the input values associated with the wide array of sensors installed on the user's physical smartphone, e.g., camera, GPS, gyroscope, accelerometer, proximity sensor, and magnetometer.
- The audio/video outputs of virtual smartphones with which the user directly interacts, be they running on a physical smartphone or in the cloud, must be fluently delivered to the speaker and video display of the user's physical smartphone, respectively, without any adverse impacts on user experiences.
- The virtualization overhead in the form of performance penalty and additional resource usage at run time must be kept to the minimum, especially when the set of virtual smartphones running on a physical smartphone are based on the same kernel and libraries.

The rest of this paper is organized as follows. Section 2 describes previous research and development efforts related to virtualized smartphone and VMI. Section 3 provides an introduction to the basis of *Brahma*'s development: Secure Virtual Mobile Platform (SVMP) [13] and Android x86 [2]. Section 4 and 5 detail the design and implementation of the virtualized smartphone and the VMI, respectively. Section 6 presents the results of an evaluation study of the first *Brahma* prototype, and their analysis. Section 7 concludes this paper with the main research contributions of this work, and an outline of the future work.

II. RELATED WORK

A standard solution to the smartphone security problem is Mobile Device Management (MDM) [21], which monitors and controls the use of a smartphone remotely. However, the inconvenience associated with MDM-enabled smartphones triggers the BYOD problem in the first place. The Cells project in Columbia University [15] developed an open-source solution that allows multiple virtual smartphones to run on a single physical phone in an isolated manner. The architecture of Cells allows only one virtual phone to run in the foreground with other virtual smartphones running in the background. This limitation arises from their integration of a device namespace mechanism with device proxies as a lighter-weight virtualization approach. The resulting prototype supports fully accelerated 3D graphics, complete power management features, and caller ID, and imposes only modest runtime and memory overhead. The technology has since been licensed to a start-up, Cellrox.

In 2011, VMware introduced Mobile Virtual Platform (MVP) [16], a type 2 hypervisor for smartphones that enables multiple OSes to be run on a single smartphone. Later that year VMware acquired Trango, a French software company that had created a type 1 smartphone hypervisor, which offers stronger isolation, occupies only 20KB, and can be run from ROM. The MVP hypervisor was later rebranded as VMware Horizon

Mobile [18], which virtualizes the storage, networking and telephony, and allows IT administrators to remotely manage the phone such as wiping and locking the device when it is lost, delivering application updates, adding or removing applications, pushing down new templates, etc. MVP was eventually shut down in 2014.

Samsung Software Laboratories developed an Xen-based hypervisor for non-virtualizable ARM CPUs called Xen on ARM [17]. The main technical challenge of Xen on ARM is to separate the address spaces of a guest OS and the applications running on top of it because ARM CPU has only one unprivileged mode. To solve this problem, Xen on ARM split the user mode into two logical modes (user process mode and kernel mode), and was responsible for properly switching between the user process mode and the kernel mode. The virtualization overhead of Xen on ARM was relatively moderate. However, no complete virtualized smartphone based on this hypervisor was built and demonstrated.

Agawi, previously known as iSwifter, developed a low-latency app streaming technology [20] that enables users to play games from the cloud on their smartphones. Such app streaming technology is a critical building block for fluent UI experiences in Virtual Mobile Infrastructure (VMI) VMI. Google acquired Agawi in June 2015, and integrated this low-latency streaming technology with its Deep Link technology to allow its search engine to return search results that point to apps and users can then click on these apps to look at their contents through app streaming.

Android 5 (Lollipop) or newer versions support a multi-user feature [10] that is meant to ameliorate the BYOD problem. It includes a greatly improved Device Policy Manager with new APIs in order to support corporate-wide security policies, e.g., app restrictions, silent installation of certificates, and cross-profile sharing intent control [3]. Although downloading and installing apps, setting wallpapers and arranging home screens in one user account will not affect other user accounts, changes to system settings, such as adding a Wi-Fi network, are applied to all other user accounts on the device. Therefore, the inter-user isolation of this multi-user implementation leaves much to be desired.

Virtual desktop infrastructure (VDI) [12] is the practice of hosting a desktop operating system within a virtual machine (VM) that runs on a centralized server. Citrix, VMware and Microsoft dominate the VDI market, because they provide effective solutions to the following three problems: (1) How to support video-based and graphics-based applications over the network with low latency? Multimedia redirection [7] is one such technology. (2) How to seamlessly relay client-side hardware devices, e.g., USB devices, to the VMs running in the cloud? (3) How to reduce the complexity of administering a large number of largely identical VMs deployed in a VDI system, e.g. applying a patch? Although there are a large number of open-source VDI solutions such as the Red Hat Enterprise Virtualization Manager for Desktops [9], they generally fare worse in these three aspects, which are critical to corporate users.

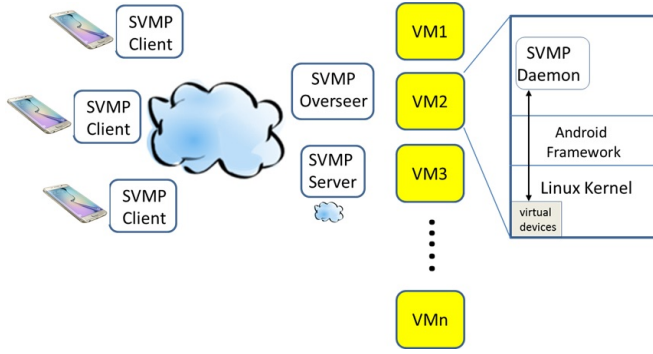


Fig. 1: The system architecture of SVMP and its main components: SVMP overseer, SVMP server, SVMP client, and SVMP daemon

III. BACKGROUND

The first *Brahma* prototype is designed to support Android-based virtual smartphones running on X86-based smartphones, which themselves run Android, and X86-based servers, which runs standard Linux. In addition, we started with the Secure Virtual Mobile Platform (SVMP) [13] and used it as the unified base for *Brahma*'s virtualized smartphone implementation and VMI implementation.

A. Android for X86

Android provides an application development framework [5] to simplify the development of Android applications. Developers write Android applications using the Java language so that the resulting applications can run on Android devices with different hardware configurations. An Android application typically calls framework APIs to access services provided by an Android system, for example, camera, media, and network. Each of these services is implemented as a standalone server process that is written in C so as to efficiently access the relevant aspect of the hardware abstraction layer (HAL). A framework API call made by an Android application invokes the target service's client-side JNI library, which in turn communicates with the corresponding server process via Android's binder IPC mechanism.

Instead of ARM-based smartphone hardware, *Brahma* is built on X86-based smartphone hardware. There are currently two versions of Android that run on X86 CPU, Android-x86 [2], which is supported by Asus, and Android-IA [1], which is supported by Intel. Both of them were derived from Android Open Source Project (AOSP), and include a series of patches and new low-level components to enable Android to better run on the X86-based architecture. The current *Brahma* prototype is based on Android-x86.

B. Secure Virtual Mobile Platform (SVMP)

SVMP [13] is designed to allow a smartphone user to access remote Android-based VMs running on virtualized servers. As shown in Figure 1, it consists of the following system components:

- The front-end *SVMP client* is an application running on a user's Android or iOS phone or tablet, and works

similarly to conventional remote desktop clients such as those for VNC or RDP. The SVMP client application forwards to a remote VM a user smartphone's touch screen events, sensor inputs from compass, accelerometer, gyroscope, etc., GPS-based location information, and two-way inter-application messaging for notifications and Intents, through protocol buffers messages. The relay of all these inputs enables a remote VM to feel like running on a physical smartphone. In addition, the client also creates a WebRTC [14] connection with a remote VM, and this enables the audio and video outputs from the VM to flow back to the user via this connection. WebRTC is chosen because it is capable of traversing NAT and firewalls by supporting STUN, TURN and ICE protocols, and because it provides support for several resilient video codecs, such as VP8, that can effectively handle fluctuating wireless network conditions.

- The *SVMP Overseer* is internet-facing, and is responsible for receiving login requests from users, performing authentication, and creating and managing virtual machines on demand. It supports a RESTful API that SVMP clients and SVMP Servers can request for service.
- The *SVMP Server* is also internet-facing, and takes care of receiving connection requests from users that the SVMP Overseer already authenticated, and routing various types of sensor input messages from SVMP clients to SVMP daemons running inside VMs. The SVMP server is designed to run behind a load balancer to scale up with the input loads.
- The *SVMP daemon* runs inside a VM to replay inputs from the client application to the corresponding virtual devices in the VM, and subscriptions or `Action_DIAL` intents from the VM to the client application.

SVMP sets up a set of virtual devices inside each VM that emulates the input and output devices of a virtual smartphone. The touch input events from the client application on a user's smartphone are forwarded to the SVMP server, passed through the SVMP daemon, injected into the Touch Input virtual device, and eventually delivered to applications. Other sensor events generated on a user's smartphone go through a similar process to be delivered to a virtual smartphone. The SVMP daemon on a VM injects these input events to a local socket on the VM, the `/dev/svmp_sensors` socket, and a SVMP HAL module, `libsensors`, listens on the `svmp_sensors` socket and processes these input events accordingly.

SVMP also supports exchanges of intents between a SVMP client and a smartphone VM. For example, for location information, SVMP passes any subscription intents requested by the LocationManager on a smartphone VM back to the client application on a user's smartphone, which in turn passes the location information to the smartphone VM. In addition, the SVMP client is capable of receiving the `ACTION_DIAL` intent when a phone number URI is pressed in the smartphone VM, showing any notifications received from the smartphone VM, and forwarding URLs with `ACTION_VIEW` intent to the

smartphone VM to open the URL inside the VM.

For video, a Virtual Frame Buffer (VFB) device is set up in the kernel of each smartphone VM. When a frame is written to the VFB, the Android `SurfaceFlinger` module generates a VSYNC event, which causes the contents of the VFB to be fed into the WebRTC module, which compresses it and streams to the SVMP client.

We used SVMP as the basis to implement *Brahma*'s virtual mobility infrastructure system, in which smartphone VMs run on X86 servers, and *Brahma*'s virtualized smartphone system, in which smartphone VMs run together with the SVMP client on the user's physical smartphone

IV. VIRTUALIZED SMARTPHONE

A. Use Case

A virtualized smartphone enables multiple virtual machines (VM), each of which corresponds to a virtual smartphone, to run on top of an HAL hypervisor, which in turn runs on the physical smartphone. A typical set-up for a virtualized smartphone is shown in Figure 2, and consists of four types of VMs. The *Work* VMs are work-related, strictly locked down, least flexible, and allowed to connect to the corporate network and the Internet. The *Personal* VMs are for day-to-day regular use, offer more flexibility in application execution, rely on conventional anti-virus protection, and cannot connect to the corporate network. The *Secure* VMs are more secure and less flexible than *Personal* VMs, cannot connect to the corporate network either, and are designed for carrying out secure transactions, such as entering credit card numbers. The *Playground* VMs are the least secure, and are meant to be disposable VMs that users can throw away after playing with unknown applications or potential malware. After a *Playground* VM is deleted, all the side effects in the VM are wiped completely clean from the underlying physical smartphone.

In terms of security policy enforcement, *Brahma* applies attestation and white-listing to completely lock down *Work* VMs, applies white-listing to *Secure* VMs to prevent malware such as key loggers, applies black-listing (traditional anti-virus protection) to *Personal* VMs to balance between security and convenience, and imposes no restrictions on *Playground* VMs, but includes a malware detection mechanism that constantly looks out for suspicious symptoms and alerts users when these VMs appear to be compromised.

B. The Host

Although most Android smartphones are based on ARM SOC, we chose to implement the first *Brahma* prototype on an X86-based smartphone, specifically Asus's Zenfone2, because the virtualization feature on most ARM-based SOC is turned off by default and cannot be turned on without the approval and support of smartphone chip vendors. More specifically, we were able to get the open-source ARM hypervisor [6] to successfully run on the development board using a 64-bit virtualizable ARM SOC such as Cortex-A53, but not on any of the commercially available smartphone reference designs from Qualcomm or Mediatek, because of this restriction.

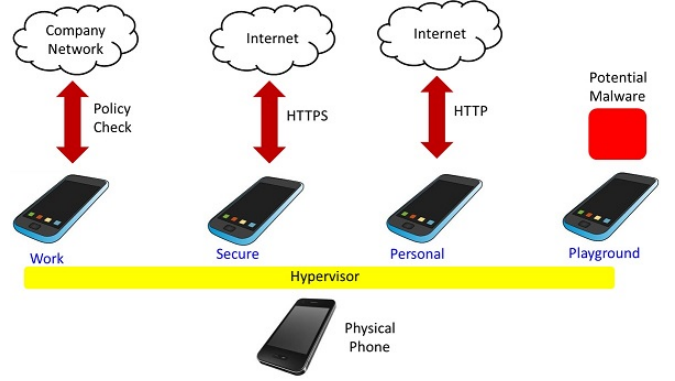


Fig. 2: A typical set-up for a virtualized smartphone that offers a smartphone user a work virtual smartphone for work use, a personal virtual smartphone for day-to-day use, a secure virtual smartphone for performing secure on-line transactions, and a disposable playground VM for experimentation

In contrast, we were able to successfully implement *Brahma* on a Zenfone2 purchased from the open market, which is based on Intel's Atom CPU, without any help from Asus or Intel. In addition to running Android-x86 inside the VMs, the *Brahma* prototype also runs Android-x86 on the physical smartphone because this presents to the user the illusion that Android VMs themselves are just like apps running on a standard Android phone.

To turn Android-x86 into a kernel that supports virtualization, we first applied a binary translator to convert parts of the Android framework and Android applications that include ARM binary instructions, e.g. JNI code, into their corresponding X86 binary code so that they can run on X86 CPU. Then we compiled KVM into the Android-x86 kernel, turned on QEMU, and successfully ran Android-x86 virtual machines on a Zenfone2 that itself also ran Android-x86. In the current *Brahma* prototype, each Android-x86 VM is packaged into an app running as a separate process on top of the host, which also runs the Android-x86 kernel, and each such process includes a QEMU component, which runs as multiple threads and supports I/O device emulation among other things.

C. Direct Frame Buffer Access

Because user interaction fluency is crucial to a smartphone's usability, the first design challenge of *Brahma* is how to provide the same level of UI fluency inside a virtual smartphone as on a physical smartphone. The key to achieving UI fluency is to eliminate unnecessary data movement when a virtual smartphone's frame buffer is copied to the underlying physical smartphone's frame buffer. Accordingly, we replaced SVMP's WebRTC-based mechanism for VM frame buffer delivery with a zero-copy frame buffer write mechanism.

QEMU presents to each VM a virtual VGA card, which a VM's guest OS could configure with respect to its display mode, resolution and color depth during initialization. At run time, the guest OS writes its display content into a video memory area corresponding to its virtual VGA card and

notifies QEMU accordingly. A VM's video memory area is accessible to both the VM's guest kernel and QEMU threads, because they both run in the same address space. In SVMP, QEMU sends the contents of this video memory to a process running on the host over a network connection, and incurs at least one data copying operation. To eliminate this data copying overhead, QEMU could write the video memory area directly to the host's frame buffer using a Linux system call, or a Java thread could be created to write this video memory area to the host's frame buffer using an Android graphics API call. In the current *Brahma* prototype, we chose the latter approach because it is considered more future-proof with respect to newer Android versions. Specifically, we wrote a JNI function to retrieve the address of this video memory area, and exposed it to a Java-based frame buffer write app, which is responsible for writing the VM's display contents to the host's frame buffer using an OpenGL library function called the `texture loading` function. Android's graphics stack enables applications to render into buffers called surfaces, which are composited by SurfaceFlinger, and rendered through the OpenGL pipeline, which could be implemented by a software rendered or accelerated by a GPU. The current *Brahma* prototype treats a VM's frame buffer as a texture, and copies it into the host's frame buffer directly without going through the WebRTC protocol as supported by SVMP.

In summary, the process hosting each VM in *Brahma* is comprised of three components: a set of vCPU threads for the VM, the QEMU threads, and another Java thread for writing the VM's virtual frame buffer to the physical smartphone's frame buffer.

If a guest VM's frame buffer is not modified, its QEMU does not need to copy its corresponding video memory area to the host's frame buffer. To cut down unnecessary video memory copying, the virtualization app of the current *Brahma* prototype applies QEMU's dirty page tracking mechanism to the guest VM's video memory area, detects modifications to this memory area when they are written, and propagates these and only these dirty pages to the host's frame buffer at a frequency consistent with the guest VM's video frame rate.

D. Fast VM Cloning

A virtualized smartphone enables its user to create a copy of a current VM to just run some unknown app, and cleanly remove any side effects of that app later on by deleting the new VM if the user does not like the app in any way. To support this use case, *Brahma* incorporates a VM cloning mechanism that can quickly create a new VM from a running VM. The goal of fast VM cloning is to reduce the cloning time to that of cloning a process. We have gone through the following four designs of VM cloning before settling down to the last design as the choice for the current *Brahma* prototype.

- **All-in-one Image:** A VM image is represented as a single file, which contains the VM's OS and data storage. Cloning a VM means stopping the VM, copying the VM's image file to a new file, and booting a new VM from the copied file.

- **Image and Overlay:** Leveraging the file overlay capability of QCOW2 [19], *Brahma* places a VM's base image in a backing file and then creates an overlay file on top of it. After a VM is booted from its base image, all the changes it makes during runtime are directed to its overlay file. With this set-up, cloning a running VM becomes stopping the VM, making a copy of the VM's overlay file and booting a new VM from the copied overlay file. Because a VM's overlay file is much smaller than its base image file, this VM cloning mechanism is more efficient than the previous one as a result of reduced data copying.
- **Image and Overlay with Warm Boot:** When cloning a running VM, this design stops the VM, saves the VM's memory and other states to the overlay file, makes a copy of the resulting overlay file, and boots a new VM from the copied file. Because the file used in booting contains the run-time memory state of the VM being cloned, this VM cloning mechanism substantially reduces the new VM's boot-up time by eliminating the system initialization and checking overhead associated with a cold boot.
- **Image and Overlay and State with Warm Boot:** This design saves the memory state of a VM being cloned to a separate state file rather than the VM's overlay file, makes a copy of the overlay file, and boots a new VM from the state file and the copied overlay file. Because a VM's memory state is typically much bigger than its overlay file, this design improves over the previous design by reducing the amount of data involved in the file copying operation. It is not necessary to make a copy of the state file, because the source VM has no use for the saved memory state file. In contrast, the third design must make a copy of the combined overlay and memory state file before booting the new VM, because the source VM still needs the overlay part of the combined file.

In all of the above implementations, when a new VM is created, some of its system configuration parameters, such as IP address or IMEI code, are modified properly to distinguish the newly spawned VM from the VM being cloned. The current *Brahma* prototype's VM cloning mechanism is based on **Image and Overlay and State with Warm Boot**, because it is more efficient than the other three by minimizing both the memory state saving overhead and the boot-up time.

E. VM Memory Deduplication

Because the resources on a smartphone are much more limited than on a server, *Brahma* makes every attempt to minimize the resource consumption of every new VM. VM cloning reduces the net disk space consumption of every new VM. VM memory deduplication further cuts down the net memory space consumption of every new VM.

Because the VMs running on a smartphone are likely to be based on the same OS image, there is a significant amount of redundancy among the memory states of the VMs co-residing on the same physical smartphone. *Brahma* uses Kernel Shared Memory (KSM) [4] to eliminate these redundancies. Being a kernel service, KSM incurs non-trivial

performance overhead and is controlled through the `sysfs` at `/sys/kernel/mm/ksm`. The run flag (0 or 1) starts or stops the KSM thread, the `pages_to_scan` parameter controls the number of pages scanned in one pass, and the `sleep_millisecs` parameter represents the time period between passes. In addition, KSM only merges anonymous pages that applications designate as likely candidates for merging using the `madvise` system call: `int madvise(addr, length, MADV_MERGEABLE)`.

When applying KSM, *Brahma* specifically aims to eliminate the redundancy among VMs that results from read-only pages holding kernel code, user-level application code, and shared libraries. Therefore, *Brahma* only scans each new VM's pages once to minimize the performance overhead introduced by KSM. Specifically, after a VM is created, *Brahma* turns on the KSM thread, instructs KSM which areas in the new VM's guest physical memory space to work on, and then turns off the KSM thread after the new VM's memory pages have been traversed and checked.

V. VIRTUAL MOBILITY INFRASTRUCTURE

A. Use Case

Instead of running on a physical smartphone, a virtual smartphone could be supported by a VM running in an X86-based enterprise cloud data center, and a smartphone user interacts with such a virtual smartphone through proper input/output redirection to his/her smartphone, as shown in the remote VM case of Figure 3. With a virtual mobility infrastructure (VMI), a smartphone user's *Work* virtual smartphone is instantiated by a VM that runs in his company's private cloud and automatically follows the company's security policy, and his *User* virtual smartphone is his physical smartphone. Compared with the virtualized smartphone approach, VMI depends critically on the availability of network connectivity and entails larger application response time. In-cloud virtual smartphones are generally preferable when applications running on them have the following properties:

- When applications require a great deal of network or computational resource, e.g. browsing large CAD drawings, and machine learning training using GPUs,
- When applications require superuser privilege and thus cannot run on non-rooted smartphones, and
- When data accessed by applications are not allowed to leave a company.

Brahma's VMI also supports *single-app* mode, which starts up a new VM to run a given application and displays only the application's results on the user's smartphone, as shown in the app on remote VM case of Figure 3. This application streaming capability makes possible display-only files, i.e., files that can only be displayed on but never physically come to a user's smartphone. In addition, this capability eliminates the need for installation of rarely used applications. For example, modern search engines return both web pages and applications that are relevant to a user query, and use application streaming (e.g. Google's Agawi) [20] to enable the user to view the

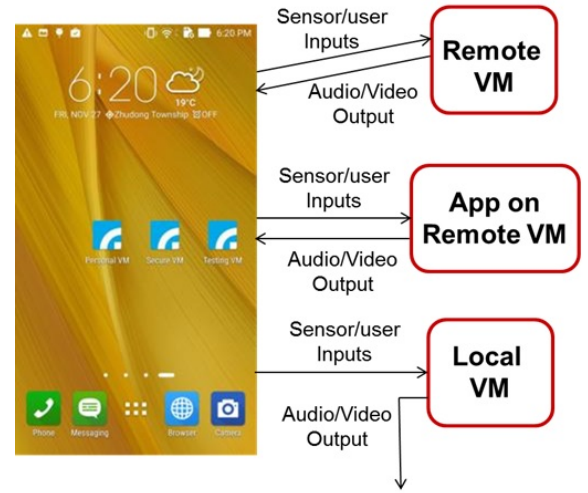


Fig. 3: A generalized Virtual Mobility Infrastructure supports running an entire VM or a specific app in a VM in a remote cloud data center or on a local physical smartphone and enables users to interact with these VMs through input/output redirection to their physical smartphones as if they are normal apps. In the local VM case, a VM's frame buffer is directly written into the physical smartphone's frame buffer without going through the SVMP protocol.

applications' results without installing them. As shown in Figure 3, *Brahma* packages a remote VM, an app on a remote VM, or a local VM into an app on the user's smartphone.

B. SVMP Adaptation to Android-x86

We use SVMP as the basis to develop *Brahma*'s VMI prototype. SVMP was built on top of Android 4.4, which lacks GPU support and limits SVMP's usability because the applications and UI on modern smartphones heavily rely on GPU support. To overcome the issue, we ported SVMP to Android-x86 4.4, which uses the `drm_gralloc` HAL implementation rather than the `gralloc` HAL implementation in Android 4.4, and provides full support for a wide range of GPUs used in X86 servers.

Porting SVMP to Android-x86 involves four major tasks. The first task is recompiling the user-mode SVMP daemon for Android-x86. The recompilation is straightforward except for some minor changes to the launcher setting. Specifically, the SVMP daemon implements its own launcher for the single app mode, but the default launcher used by Android-x86 is Trebuchet [11]. We modified the SVMP daemon a little bit to make it compatible with Trebuchet. The second task is migrating the modifications in the Android framework code made by SVMP. The framework codes of Android 4.4.4_r2 and Android-x86 4.4.4_r2.0.1 are mostly the same. We manually went through each of these changes to account for the minor differences in their code structures.

The third task is porting SVMP's HAL libraries. The SVMP project provides its own HAL libraries to implement audio pass-through and sensor input data injection. We replaced Android-x86's original HAL with SVMP's HAL libraries by

properly modifying the build parameters of Android x86 . Finally, to provide OpenGL support, we turned on the GPU pass-through mechanism to expose the GPU hardware to Andorid-x86 VMs running on our VMI infrastructure. Given a GPU, the current *Brahma* prototype allows only one VM to use it at a time. To support more fine-grained sharing and multiplexing of the GPU resource, one may need to leverage the new Virgl (virtual GPU) mechanism in Linux kernel 4.4 for OpenGL support.

C. Choices of Virtualization Infrastructure

Because the VMs in a VMI system typically run Android-x86, there are multiple possibilities for back-end the virtualization infrastructure of a VMI system, which are listed below:

- 1) A virtual smartphone runs inside an Android-x86 virtual machine over KVM/Linux on an X86 server,
- 2) A virtual smartphone runs inside a Linux container on an Android-x86-based X86 server,
- 3) A virtual smartphone runs inside a Linux container on an Android-based ARM SOC-based server, and
- 4) A virtual smartphone runs on an Android-based ARM SOC-based server.

The first configuration serves as the baseline. The second configuration removes the performance and resource overhead of HAL-based virtualization associated with the first configuration. The third configuration removes the need for Android-x86 by using a server that is based on the same hardware as a modern smartphone. The fourth configuration dedicates an entire ARM SOC-based server to a smartphone VM and thus incurs no virtualization overhead. The ARM SOC-based server in the third and fourth configuration is based on a single-board computer Odroid-XU4 [8], which is equipped with a Samsung Exynos5422 CPU, containing four 2.0GHz Cortex?-A15 cores and four 1.4GHz Cortex?-A7 cores, a Mali-T628 MP6 GPU that is compliant with OpenGL ES 3.0 and OpenCL 1.1, 2Gbyte LPDDR3 RAM, and a Gigabit Ethernet interface. At the retail price of \$40-\$74 USD per node, an Odroid cluster makes a cost-effective execution platform for the smartphone VMs of a VMI system.

VI. PERFORMANCE EVALUATION

A. Methodology

The first *Brahma* prototype is built from SVMP 2.0.0 and Android-x86 4.4.4_r2.0.1, and runs on Zenfone2, which contains an 2.3GHz 4-core Intel Atom Z3580 CPU, a 533MHz PowerVR G6430 GPU, and 4GB of LPDDR3 RAM and runs on Android 5.0 and QEMU 2.4.50, on an X86 machine, which contains an 3.1GHz i5-2400 CPU and 16GB of DDR4 RAM and runs Ubuntu 14.04.3 LTS with Linux 3.19.0-42-generic x86_64 and QEMU 2.3.0, and on a cluster of 8 Odroid single-board computers that run Android 4.4.2 on Linux Kernel LTS 3.10 and are connected by a Gigabit Ethernet switch. To measure the memory and power consumption overhead of a virtualized smartphone, we use the Android Debug Bridge (adb) shell command `dumpsys meminfo` and `dumpsys batterystats`, respectively. Each energy

Configuration	Memory (MB)	Power (mW)
Android without KVM	854.81	88.30
Android with KVM	858.37	92.98
A native browser app	858.37 + 270	1697.28
An idle 512MB VM	858.37 + 402	975
A browser app running inside a 512MB VM	858.37 + 553.3	2263.56

TABLE I: The average run-time memory usage and power consumption of a Zenfone2 over a 10-minute interval when it runs vanilla Android, Android with KVM when no VM is active, a browser application on vanilla Android-x86+KVM, an idle 512MB VM on Android-x86+KVM, and an active VM on Android-x86+KVM that in turn runs a browser application

consumption measurement is taken over a 10-minute interval, and every reported number is an average of 3 measurements.

B. Virtualized Smartphone Results

1) *Virtualization Overhead*: *Brahma*'s virtualized smartphone is modified from Zenfone2 by adding the KVM module into the baseline Android-x86 kernel and the QEMU module into every process that hosts a virtual machine. To measure the additional run-time resource usage associated with virtualization, we measured five different scenarios, which correspond to the last five rows of Table I: a Zenfone2 running the vanilla Android kernel, a Zenfone2 running a modified Andorid-x86 kernel that includes the KVM module, a Zenfone2 running the KVM-included Android-x86 kernel and a native browser application on top of it, a Zenfone2 running the KVM-included Android-x86 kernel and an idle 512MB VM on top of it, and a Zenfone2 running the KVM-included Android-x86 kernel, and a 512MB VM on top of it, where a browser application runs inside the VM.

Even though the size of the KVM module is 896KB, inclusion of KVM only increases the Android-x86 kernel's image size from 12.8 MB to 13.0MB. The second and third rows of Table I show that when no virtual machine is running, the virtualized version of Zenfone2 incurs 3.5MB or 0.4% more memory and 5.3% more power consumption than the unmodified Zenfone2. Running an idle VM on the modified Android-x86 kernel incurs lower power consumption than running a browser application (975 mW vs. 1697.28 mW), and the power consumption of running a browser application inside a VM (2263.56 mW) inside a VM is substantially lower than the sum of the energy consumptions of running the same browser application and the VM separately (1697.28+975 mW). Because of thin provisioning, the memory consumption of an idle 512MB Android-86 VM costs only 402MB, which is less than 512MB. However, because of the addition of the QEMU module to each VM process, the memory consumption of a 512MB VM running an active browser application is 553.3MB, which is more than 512MB. The memory consumption of the QEMU module is approximately 32 MB.

No. of VMs	512MB	768MB	1024MB
1	1029.1+11.01	1029.1+10.69	1029.1+10.50
2	+11.01	+10.69	+11.07
3	+11.39	+10.87	+11.14
4	+11.39	+11.07	+11.07
5	+10.50	+10.75	+10.82

TABLE II: The disk space consumption in MB as an increasing number of VMs that are booted from the same kernel image and configured with 512MB, 768MB and 1024MB are created on a virtualized Zenfone2. The last four rows shows the additional disk space requirement as the N-th VM ($N = 2, 3, 4$ and 5) is created.

Because *Brahma* uses VM cloning to create new VMs that are based on the same kernel image, the total disk space requirement for the images of newly created VMs grows very slowly when these VMs are derived from the same kernel image. Table II shows that when new VMs are booted from the same kernel image, each newly created VM roughly imposes an additional 11MB disk space requirement, and moreover this additional disk space requirement is independent of the amount of physical memory configured with the new VM. The second row of Table II shows the base kernel image size and the additional disk space required when the first VM is created.

2) *Virtual Machine Cloning*: Section IV-D describes four different VM cloning designs. Table III shows the time taken to clone a live VM configured with 512MB, 768MB and 1024MB, when each of these four mechanisms is applied. In this study, the VM being cloned stays idle after its boot-up. The end-to-end VM cloning time is the sum of the times required for the three steps of cloning a VM: (I) stop the VM to be cloned, (II) prepare the image for the new VM via file copying, and (III) boot up the new VM.

In the first design, *All-in-one Image*, the bulk of the VM cloning time is spent on Step (II). Because the times for the three VM cloning steps do not vary much with the amount of configured physical memory of the VM being cloned, the VM cloning times for VMs of different physical memory configurations are largely the same.

In the second design, *Image and Overlay*, the time required for Step II is significantly reduced because the overlay file (11MB) is much smaller than the base image file (about 1GB). As a result, the end-to-end VM cloning time of the second design is lower than the first design by 15 seconds, which results mainly from the reduction in the image file copying time. For the same reason as the first design, the VM cloning times for VMs of different physical memory configurations under the second design also do not differ much from one another.

The third design, *Image and Overlay with Warm Boot*, replaces cold boot used in the first and second design with warm boot, which requires saving the source VM's memory state to the overlay file, makes a copy of the overlay file,

and boots up the new VM from the new copy of the overlay file rather than from the base kernel image. Moreover, this design uses `savevm` rather than `shutdown` to stop the VM being cloned. Compared with the second design, `savevm` plus warm boot together results in a factor of three reduction in the VM cloning time even though it involves an additional VM memory state saving step, which takes less than 3 seconds. The memory state copy times for different VMs with different amounts of configured memory are largely the same, because this copy time depends on the amount of memory actually provisioned for the VM being cloned rather than its originally configured amount. For example, because of thin provisioning, VMs that are originally configured with 512MB, 768MB and 1024MB are all provisioned with the same amount of physical memory, roughly 360MB. As a result, the memory state saving copy times for these VMs are largely the same.

The fourth design, *Image and Overlay and State with Warm Boot*, improves upon the third design by using `migrate` rather than `savevm` to save the source VM's state to a separate file, and eliminating the need to copy the saved memory state file of the source VM. Compared with the third design, the fourth design produces a 35% reduction in VM cloning time regardless of the physical memory configuration of the source VM. This cloning time reduction depends on the actual amount of memory provisioned for the source VM and is thus largely independent of its physical memory configuration. Note that the saved memory state file in the fourth design can be safely discarded after the new VM successfully boots up, and therefore does not constitute a long-term liability.

3) *Memory Deduplication*: To minimize the total physical memory usage of running multiple VMs that are booted up from the same kernel image on a smartphone, *Brahma* applies KSM to eliminate the redundancies among these VMs. Table IV shows the total physical memory usage of an increasing number of VMs running on a virtualized Zenfone2 that are booted up from the same kernel image but configured different amounts of physical memory, when KSM is turned off and on. These VMs stay idle after they are booted up. Without KSM, the total memory space requirement is roughly equal to the number of VMs multiplied by the amount of provisioned physical memory for a single VM.

When KSM is turned on, it could identify and remove intra-VM and inter-VM redundancies. The second row of Table IV (No. of VMs = 1) shows that the amount of physical memory provisioned for a VM is smaller than and independent of its physical memory configuration, because QEMU supports thin memory provisioning; moreover, the redundancy within the same VM is very small because the memory space requirement when KSM is turned on is very close to that when KSM is turned off.

Every new VM incurs between 60 to 70 MB of additional memory usage regardless of the VM's physical memory configuration. This result suggests that the memory state differences among these concurrently running VMs mainly result from some VM-specific state that has nothing to do with the VM's physical memory configuration. To determine in

Design	512MB VM				768MB VM				1024MB VM			
	Stop	Copy	Bootup	Total	Stop	Copy	Bootup	Total	Stop	Copy	Bootup	Total
1	14.2	16.7	24.5	55.4	14.1	16.9	24.2	55.2	14.0	16.9	24.3	55.2
2	14.0	0.08	26.8	40.9	13.9	0.08	26.5	40.5	14.0	0.08	26.7	40.8
3	5.4	2.8	4.7	12.9	5.6	2.9	5.0	13.5	6.0	2.8	4.5	13.3
4	2.8	0.08	5.4	8.3	3.0	0.07	5.6	8.7	3.1	0.8	5.4	8.6

TABLE III: The end-to-end VM cloning times in second of four different VM cloning designs for VMs that are configured with 512MB, 768MB and 1024MB. Design 1 is All-in-one Image, Design 2 is Image and Overlay, Design 3 is Image and Overlay with Warm Boot, and Design 4 is Image and Overlay and State with Warm Boot.

No. of VMs	512MB	768MB	1024MB
1	214/202	222/207	224/210
2	430/263	438/269	452/284
3	645/334	652/336	680/361
4	856/400	884/420	902/431
5	1070/469	1104/488	1116/493

TABLE IV: The total memory usage in MB as an increasing number of VMs that are booted from the same kernel image and configured with 512MB, 768MB and 1024MB are created and run on a virtualized Zenfone2, when KSM is turned off (left) and on (right)

VM deduplicated	512MB	768MB	1024MB
1st	317	326	327
2nd	993	996	1015
3rd	755	766	778
4th	911	921	947
5th	1076	1089	1123

TABLE V: The amount of time in msec required to deduplicate a new VM with a physical memory configuration of 512MB, 768MB and 1024MB, when there are a varying number of pre-existing VMs

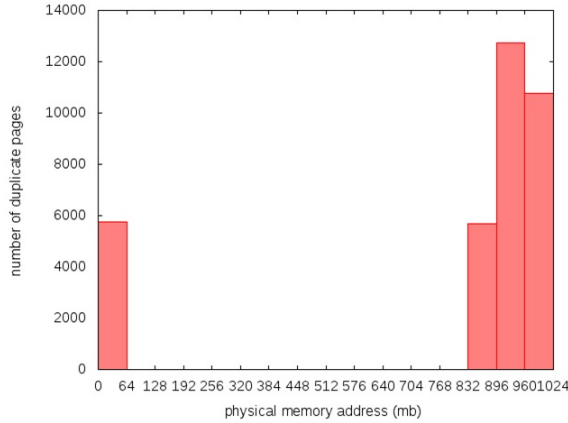


Fig. 4: The histogram of the number of duplicate pages over the physical address space of a VM with a physical memory configuration of 1024MB. Each bar represents the number of duplicate pages within a 64MB address space range.

which parts of a VM’s physical address space its VM-specific state lie, we count the number of pages in each 64MB region of a VM’s physical address space that are duplicates with respect to other concurrently running VMs, and the results are shown in Figure 4, which suggests that a VM’s VM-specific state mainly lies in the middle of the VM’s physical address space.

Table V shows the time required to deduplicate a new VM with a different physical memory configuration when the number of pre-existing VMs varies from 0 to 4. KSM uses two trees. The stable tree holds duplicate pages and is long-lived, whereas the unstable tree holds non-duplicate pages and is recreated repeatedly at run time. To deduplicate the memory

pages of a new VM, KSM (I) scans the non-duplicate memory pages of existing VMs to build up the *unstable tree*, and (II) checks the new VM’s pages against the stable and unstable trees to add each page to one of these two trees. To add a page to the stable tree, KSM needs special set-up such as copy-on-write for the page and thus incurs additional performance overhead. The time required for Step I is linearly proportional to the number of pre-existing VMs, and the time required for Step II is largely fixed and depends only on the amount of memory provisioned to the new VM. This is why the total memory deduplication time in Table V grows linearly with the total number of existing VMs, except when the second VM is added. When KSM deduplicates the first VM, initially there is no stable tree, and KVM builds up the stable tree from the duplicated pages within the first VM. This stable tree is small because the intra-VM redundancy is relatively rare. When KSM deduplicates the second VM, it first creates an unstable tree from the first VM’s non-duplicate pages, and checks the second VM’s pages against this unstable tree and the stable tree to grow the stable tree. After the second VM is deduplicated, the stable tree remains largely unchanged, because it already captures most of the duplicate pages among these VMs. That is, the bulk of the stable tree is formed when KSM deduplicates the second VM. That’s why the total deduplication time for the second VM is anomalously larger than the total deduplication times for other VMs.

C. Virtual Mobility Infrastructure Results

We set up a test-bed to compare three Android-x86 VMI execution platforms in terms of the touch input update rate, the video output frame rate, and the round-trip ping latency. The

Configuration	Frames per sec	Touch Updates per sec	Ping (msec)
Remote PM on Odroid	32	45	13.01
Remote VM on X86 PC	39	57	5.8
Local VM on Zenphone2	33	30	NA

TABLE VI: Comparison of the touch input rate, video output frame rate, and round-trip ping latency among three configurations: a remote Android-x86 physical machine based on Odroid-XU4, an Android-x86 virtual machine running on a X86 PC and a local Android-x86 virtual machine running on the user's Zenfone2

test-bed consists of a virtualized Zenfone2 that is connected to a 5GHz 234Mbps 802.11AC Wifi router, which in turn is connected via a Gigabit Ethernet switch to an Odroid SBC and an X86 PC. The spatial resolution of the display used in this study is 800 x 600 at 130 dpi and the temporal resolution is 91 fps.

Table VI shows the performance measurements of (I) an Android-x86 physical machine running on Odroid-XU, (II) an Android-x86 virtual machine running on a X86 PC and (III) a local Android-x86 virtual machine running on the user's Zenfone2. In general, Configuration (II) is the most performant, but Configuration (I) is pretty comparable to Configuration (II). Surprisingly, Configuration (III) performs worse than Configuration (I) even though the former does not involve networking. This may be because the hardware behind Zenphone2 is slightly less powerful than Odroid-XU. These results demonstrates that an Odroid cluster is a promising backend VMI execution platform.

To evaluate the effectiveness of the direct frame buffer write optimization, we measured the video output frame rate it is turned on and off. When it is off, the measured video frame rate is 17.5, and when it is turned on the measured video frame rate is 33, almost a factor of two improvement.

VII. CONCLUSION

In the era of consumerization of enterprise IT, the BYOD trend is inevitable, and the associated security problem poses a serious challenge to corporate IT departments. The thesis of this research is that smartphone virtualization could make an effective solution to the BYOD problem as long as it is proved to be practical and usable on commercial smartphones. This paper describes the design and implementation of a comprehensive smartphone virtualization solution called *Brahma*, which consists of a virtualized smartphone system and a virtual mobility infrastructure system. The first *Brahma* prototype is fully operational on Asus Zenfone2, and the initial performance results taken from this prototype look promising in terms of user interface fluency and additional memory resource usage and power consumption due to virtualization. Specifically, this work makes the following research contributions:

- A unified smartphone virtualization architecture that encompasses both virtualized smartphone and virtual mobility infrastructure by allowing virtual machines to run on a cloud backend or on a smartphone,
- A set of optimization mechanisms such as direct frame buffer access, VM cloning, memory de-duplication, etc. that collectively improve the user interface fluency and minimize the resource/performance overhead associated with virtualized smartphone, and
- A complete prototype implementation of the *Brahma* architecture on a commercial smartphone without any hardware changes and its detailed performance evaluation and analysis.

REFERENCES

- [1] Android on Intel Platforms. <https://01.org/android-ia/>. accessed Mar 11, 2016.
- [2] Android-x86 - Porting Android to x86. <http://www.android-x86.org/>. accessed Mar 11, 2016.
- [3] DevicePolicyManager APIs. <https://source.android.com/devices/tech/admin/managed-profiles.html#purpose>. accessed Mar 11, 2016.
- [4] How to use the Kernel Samepage Merging feature. <https://www.kernel.org/doc/Documentation/vm/ksm.txt>. accessed Mar 11, 2016.
- [5] Introduction to Android. <http://developer.android.com/guide/index.html>. accessed Mar 11, 2016.
- [6] Linaro ARM Hypervisor. <https://wiki.linaro.org/Core/Virtualization?action=show&redirect=Virtualization>. accessed Mar 11, 2016.
- [7] Multimedia Redirection. <https://github.com/FreeRDP/FreeRDP/wiki/Multimedia-Redirection>. accessed Mar 11, 2016.
- [8] ODDROID-XU4. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825. accessed Mar 11, 2016.
- [9] RED HAT ENTERPRISE VIRTUALIZATION: HYPERVISOR. <https://www.redhat.com/f/pdf/rhev/DOC076-RHEV-Hypervisor.pdf>. accessed Mar 11, 2016.
- [10] Supporting Multiple Users. <https://source.android.com/devices/tech/admin/multi-user.html>. accessed Mar 11, 2016.
- [11] Trebuchet Launcher - GitHub. https://github.com/CyanogenMod/android_packages_apps_Trebuchet. accessed Mar 11, 2016.
- [12] Virtual desktop infrastructure (VDI). <http://searchservervirtualization.techtarget.com/definition/virtual-desktop-infrastructure-VDI>. accessed Mar 11, 2016.
- [13] Virtual Smart Phones in the Cloud. <https://svmp.github.io/index.html>. accessed Mar 11, 2016.
- [14] WebRTC. <https://webrtc.org/>. accessed Mar 11, 2016.
- [15] 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*, pages 173–187. ACM, 2011.
- [16] 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? *ACM SIGOPS Operating Systems Review*, 44:124–135, 2010.
- [17] 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 *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.
- [18] D. Jaramillo, N. Katz, B. Bodin, W. Tworek, R. Smart, and T. Cook. Cooperative solutions for bring your own device (byod). *IBM Journal of Research and Development*, 57:5:1–5:11, 2013.
- [19] Mark McLoughlin. The qcw2 image format. <https://people.gnome.org/~markmc/qcow-image-format.html>. September, 2008.
- [20] Dan Meyer. Agawi puts touch-screens to the test. <http://www.rcrwireless.com/20140115/devices/agawi-puts-touch-screens-to-the-test>. January, 2014.
- [21] T. Zefferer and P. Teufl. Policy-based security assessment of mobile end-user devices an alternative to mobile device management solutions for android smartphones. In *Security and Cryptography (SECRYPT), 2013 International Conference on*. IEEE, 2013.