SMOC: A Secure Mobile Cloud Computing Platform

Zijiang Hao, Yutao Tang, Yifan Zhang†, Ed Novak, Nancy Carter, Qun Li
{hebo, yytang, ejnovak, njcarter, liqun}@cs.wm.edu, †zhangy@binghamton.edu
College of William and Mary, Williamsburg, VA, USA
†State University of New York at Binghamton, Binghamton, NY, USA

***Abstract*—Mobile devices are now ubiquitous in the modern world. In this paper, we propose a novel and practical mobile-cloud platform for smart mobile devices. Our platform allows users to run the entire mobile device operating system and arbitrary applications on a cloud-based virtual machine. It has two design fundamentals. First, applications can freely migrate between the user's mobile device and a backend cloud server. We design a file system extension to enable this feature, so users can freely choose to run their applications either in the cloud (for high security guarantees), or on their local mobile device (for better user experience). Second, in order to protect user data on the smart mobile device, we leverage hardware virtualization technology, which isolates the data from the local mobile device operating system. We have implemented a prototype of our platform using off-the-shelf hardware, and performed an extensive evaluation of it. We show that our platform is efficient, practical, and secure.**

I. Introduction

Smart mobile devices are gradually becoming the dominant daily computing platform for many people [1], [2]. While many applications today run directly on individual mobile devices, we envision a mobile-cloud computing model emerging whereby individual devices run user interface software with the bulk of computation performed on a virtual machine (VM) running in a commodity cloud hardware environment [3]. In this paper, we aim to build a platform that supports free migration of apps between smart mobile devices and cloud based hardware.

In the mobile-cloud model, user input on the mobile device is transmitted to cloud processes running on the VM. Results from cloud processing are transformed into display content and then transmitted back to the mobile device. The mobile device and the cloud VM share functionality to meet user needs. By moving heavy computational processes from the mobile device to the cloud VM, the mobile-cloud model improves user response time and reduces device energy consumption. It also has security advantages, e.g., sensitive data can be stored in the cloud, safeguarded from a compromised mobile device OS or app, and also protected from exposure to a thief. A user could acquire a new smart mobile device, download the interface software, and resume arbitrary tasks from before the compromise or theft occurred.

There are already solutions for the mobile-cloud model in recent literature. We believe, however, that our platform has greater potential than what is immediately obvious and can achieve more than what existing solutions can do. There are two concepts underlying our platform that differentiate it. First, we are proposing a resource sharing platform in the sense that an app can freely change its executing location between the mobile device and the cloud. In contrast, existing solutions only allow apps to run in the cloud. Second, our platform provides security guarantees even when the mobile device operating system has been compromised, which is a feature that existing solutions cannot offer.

We achieve the first concept by running a VM in the cloud which has an execution environment compatible with that on the smart mobile device. Our platform shares resources between this cloud VM and the mobile device in both directions, i.e., the mobile device shares its files and I/O devices with the cloud VM when the app is running in the cloud, and the cloud VM shares its files with the mobile device when the app is running on the mobile device. The cloud VM does not need to share its (virtual) I/O devices with the mobile device in the latter case, because they are not involved in the app's execution. Our system currently only supports offline migration, i.e., when an app wants to change its executing location, our platform will cease its execution if it is running, transfer all the related data to the target location, and re-launch the app if necessary. We leave online migration to future work.

Considering the first concept, our platform is clearly different from most existing solutions, such as Chrome OS [4]. Like many others, Chrome OS follows a client-server computing model in which the cloud behaves as a server that hosts apps, while the smart mobile device behaves as a client that communicates with the cloud. This model fails to meet a wide range of user needs, because it only allows apps to run in the cloud. Our platform, on the other hand, is a resource sharing platform, in which apps can run on both locations and can freely change their executing location.

The second concept is achieved by leveraging the hardware virtualization functionality on the smart mobile device. To be more specific, we suppose that a hypervisor runs on the smart mobile device, which hosts a guest OS. The guest OS could be malicious, and may launch attacks on the apps that it hosts. The apps may also be malicious, compromised, or attacked by the guest OS. In any case, they may leak sensitive information stored on the smart mobile device or input by the user. However, we trust the hypervisor, because the hypervisor is always much smaller than the guest OS, and can be fully verified through formal verification or manual audit. Moreover, the hypervisor is unlikely to install any third-party applications or libraries, and thus gets rid of many potential risks. We also trust the cloud, under the assumption that it is established by a famous company with high concern for their reputation, such as Amazon, Google, Apple, and Microsoft. These companies

usually have the technical strength to protect their clouds, and are unlikely to intentionally compromise user privacy.

Under these assumptions, our platform offers security guarantees concerning an untrusted guest OS running untrusted apps. Our novel approach is to make the hypervisor responsible for sharing the smart mobile device's input interfaces with the cloud, and for blocking hardware input events from traveling to the guest OS. By doing this, we can guarantee that the guest OS can learn nothing about the user's input, while at the same time remain responsible for sharing the smart mobile device's output interfaces with the cloud. To the best of our knowledge, we are the first to leverage hardware virtualization to support security reinforcement for smart mobile devices.

The following example elaborates on this idea. Consider a smart mobile device, running a compromised guest OS, which executes a malicious background service that stealthily records the user's input through a software keyboard on the screen and sends it to a remote, malicious user. In this case, if the user runs a banking app directly on the smart mobile device, or through any existing mobile-cloud solution, her account and password are likely to be leaked to the malicious user. If the user runs the banking app on our platform, however, there is no such security concern. The hypervisor will capture the touch events from the software keyboard and forward them directly to the cloud VM, bypassing the guest OS entirely. Meanwhile, the banking app functions properly on the cloud VM, because it receives the user's input from the hypervisor.

To summarize, our contributions in this paper are three-fold.

- We propose a secure mobile-cloud computing platform which is designed as a resource sharing platform in the sense that apps running on it can freely change their executing location. This is a more flexible design than those of existing solutions, and it meets a wider range of user needs.
- We are the first to leverage hardware virtualization on the smart mobile device to provide security guarantees in the case that the smart mobile device's operating system is untrusted.
- We implement a prototype system following our platform design, and conduct several experiments on this system. The results demonstrate that our platform is efficient and practical.

II. Related Work

Mobile offloading. An increasing amount of attention has been invested in mobile offloading recently. CloneCloud [5], [6] partitions a mobile application automatically by using static analysis and dynamic profiling, such that part of the application can be offloaded to the cloud to achieve performance and an energy efficiency improvement. It also provides a runtime system to facilitate the offloading. MAUI [7] also approaches the topic of mobile offloading from the perspective of automatic mobile program partition. The main objective of the partition is to dynamically decide which part of the program should be offloaded to achieve maximum energy savings. COMET [8] implements a runtime system on top of the Dalvik Virtual Machine, the virtual machine used by Android, to allow for simultaneous executions of the same multi-threaded application in several machines. The enabling foundation is a distributed shared memory model. In contrast with these existing works, our platform is a resource sharing platform through which the user can run an app on either the cloud or the smart mobile device.

Thin client architecture in mobile computing. When a user decides to run a mobile application in the cloud with our system, we essentially turn the local mobile device into a thin client. MobiDesk [9] introduces a mobile virtual desktop hosting infrastructure that transparently decouples the display, operating system, and the network of a user's computing session from end-user mobile devices, and moves them to hosting providers. SmartVNC [10] is a system to port VNC, which is a remote computing solution, to smartphones while achieving the same level of user experience as on PCs. While work like MobiDesk and SmartVNC try to reduce workloads in mobile devices by turning the devices into thin clients, other works have also studied the energy consumption implication of applying the thin client architecture in mobile computing [11], [12]. Distinct from these works, our platform adopts a spilt client design, combined with hardware virtualization on the smart mobile device, to provide security guarantees even if the smart mobile device's guest OS is untrusted.

Exploiting hardware virtualization technology. Hardware virtualization technologies on the x86 architecture [13], [14] have long been used to develop solutions to protect system security. For example, SecVisor [15] utilizes AMD-V [14] (formally known as AMD Secure Virtual Machine (SVM)) to build a hypervisor with a small code base to ensure the code integrity of guest OS kernels. Specifically, the hypervisor by SecVisor is able to prevent code injection attacks by allowing only user-approved code to run in kernel mode. TrustVisor [16] also uses AMD-V to build a small hypervisor to ensure the guest OS's data integrity and secrecy, in addition to code integrity. Lares [17] exploits Intel VT-x [13] to achieve active security monitoring inside a virtual machine environment. In our platform, we choose to allow the mobile device OS and apps to run on VMs in the cloud. Part of the reason for this choice is to utilize these existing solutions to ensure we have a secure execution environment in the cloud. On the local device side, we exploit the newly developed ARM hardware virtualization technology [18] to protect the user's input and data collected by various on-board sensors, which can reveal sensitive information about the user. Although there are some efforts to use this technology to build general purpose hypervisors [19], [20], to the best of our knowledge, we are the first to utilize it to build a system specifically for security/privacy.

III. Platform Design

Our platform spans across the user's local mobile device and a remote cloud server. We believe that such a mobile-cloud computing platform is more capable than what people usually

think and can achieve more than what existing solutions can do. In this section, we describe our platform design in detail.

A. Design goals

The main design goal of our platform is to share the necessary resources between the smart mobile device and the cloud, such that an app can run either on the smart mobile device or in the cloud, arbitrarily. This design goal contains two aspects. First, the platform must be capable of executing an app in the cloud without any modifications to the app itself. This is possible only if the platform shares the smart mobile device's I/O interfaces with the cloud, such that an app running in the cloud can receive the user's input (e.g., from touch screen, sensors, keyboard, etc.) and render the output (e.g., display, sound, etc.) on the mobile device. Second, an app installed in the cloud should be able to be downloaded to the smart mobile device and work properly, as it does in the cloud. This can be achieved if the app can access the same resources regardless of its location. For example, files accessed by the app should be synchronized across both locations.

Running apps in the cloud is necessary for several reasons. First, the user may not trust the app (e.g., because the app has access to sensitive data) but may still want to run it. In this case, she may utilize our platform and run the app on a VM in the cloud. Even if the app compromises this VM, it cannot affect other apps and data on the user's local mobile device, and the user may simply delete the VM afterwards. In this way, even a malicious app is completely quarantined. Moreover, commercial clouds often provide powerful anti-virus services, and a malicious app is more likely to be caught if these services are implemented in the cloud used in our platform. Second, smart mobile devices are always resource constrained [21]–[23]. To solve this problem, our platform allows the user to offload apps from the smart mobile device to the cloud, which typically has abundant resources. Third, some organization or developer may want to publish an app without disclosing any proprietary secrets about it (e.g., the binary of the app might be reverse engineered to compromise some critical algorithms). In this case, the organization/developer may publish the app using our platform and only allow the app to run in the cloud. In this way, no malicious user can recover a complete binary file for further analysis.

After being deployed, our platform will have some apps installed in the cloud by default. However, the user may want to download an app from the cloud to the smart mobile device for some reason, for example, the latency between the smart mobile device and the cloud becomes unacceptable. This can be achieved transparently in our platform, because all resources, including files, are shared between the smart mobile device and the cloud.

The cloud exploited in our platform is considered a commercial cloud. However, it could also be a private cloud established by, and serving, only a single user. Consider the scenario where a user does not trust an app, but still wants to run it. She may build a private cloud which does not connect to the Internet, and deploy our platform on this cloud in conjunction with her smart mobile device through a local area network. She can then run the app in the private cloud to prevent her mobile device from being attacked by the app.

Another design goal of our platform is to provide a secure environment for the user to run apps. As we mentioned previously, the VM in the cloud is isolated from the smart mobile device, and thus a malicious app running on the VM cannot affect any other apps running on the smart mobile device. Moreover, commercial clouds often provide powerful anti-virus services, which can be utilized on our platform to defend against malicious apps. These are two cases in which our platform can provide strong security guarantees. We have yet to consider another important scenario; suppose a user wants to run a banking app on the smart mobile device, but the smart mobile device's operating system has been compromised and a malicious background service stealthily records the user's input and sends it to a malicious user. In this scenario, running the banking app on the smart mobile device is dangerous because the malicious user may learn the user's banking account and password via her input. Our platform provides a secure environment that defends against this attack by leveraging hardware virtualization on the smart mobile device.

B. Design details

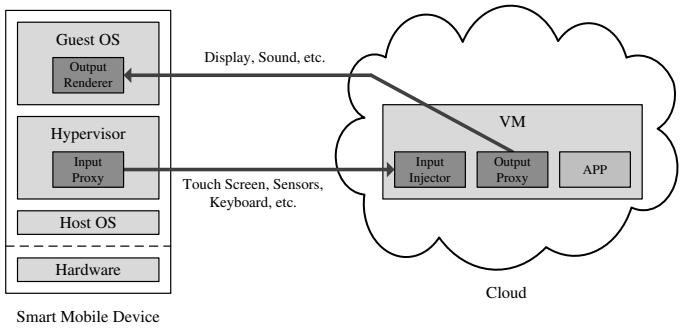

Fig. 1: An app is running in the cloud in our platform.

Fig. 1 demonstrates our design when an app is running in the cloud on our platform. The app is running in the cloud, hosted by a VM which has an execution environment compatible with that of the smart mobile device. This allows the app to run without requiring any changes. Hardware virtualization is enabled on the smart mobile device. A hypervisor running on the smart mobile device hosts one or more guest OSes in which the app can be executed.

As described in our main design goal, the smart mobile device needs to share I/O interfaces with the VM such that the app can work properly in the cloud. To achieve this, two client programs are provided on the smart mobile device. One of them, depicted as "Input Proxy" in Fig. 1, is responsible for capturing the user's input through the smart mobile device's input interfaces (e.g., touch screen, sensors, keyboard, etc.) and sending it to the VM. The VM will then emulate the user's input such that the app can receive it and work properly. On the other hand, the app generates output (e.g., display, sound,

etc.) when it is running in the cloud and sends the output to the VM's output interfaces. The VM will then transfer the app's output to the smart mobile device. The other program, depicted as "Output Renderer" in Fig. 1, is responsible for receiving this output and rendering it on the smart mobile device so the user can perceive it. By communicating in this way, the smart mobile device shares its I/O interfaces with the cloud VM such that the app can be executed in the cloud instead of on the smart mobile device.

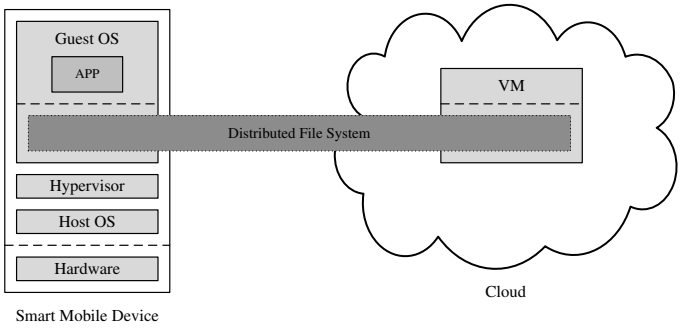

Fig. 2: An app installed in the cloud is downloaded to, and executed on, the smart mobile device in our platform.

Fig. 2 demonstrates our design in the case that an app installed in the cloud is downloaded to, and executed on, the smart mobile device. Our platform allows the user to either launch the app in the cloud or download it to the smart mobile device and execute it. When the user downloads the app to the smart mobile device, a distributed file system across the VM that hosts the app in the cloud and the guest OS on the smart mobile device will be automatically enabled. This distributed file system is needed in order for the VM to share resources with the smart mobile device, allowing the app to work properly without any modifications. Recall that the smart mobile device needs to share its I/O interfaces with the VM when the app is running in the cloud. As the app is now running on the smart mobile device, it can directly access the smart mobile device's I/O interfaces, so no I/O interface sharing is needed. However, the app, previously installed in the cloud, may need to access some resources (such as files) on the VM to work properly. For this reason, we design the distributed file system in our platform.

There are two ways to download an app from the cloud to the smart mobile device. The first way is offline downloading, in which the user downloads the app's binary file when the app is not running, and executes it on the smart mobile device thereafter. The second way is online downloading, in which the user migrates the app's process while the app is running, and continues its execution on the smart mobile device. For simplicity, our platform currently only supports offline downloading. However, conceptually, online downloading could also be achieved. We leave this to future work. After being downloaded to the smart mobile device, the app may need to access files on the VM that previously hosted it. Our distributed file system, spanning across the smart mobile device and the VM, allows the app to access such files after it has been downloaded. Therefore, the app can work properly on the smart mobile device without any modifications.

Combining these two design elements, illustrated by Fig. 1 and Fig. 2, our platform achieves its main goal. The second design goal can also be achieved if the hypervisor in Fig. 1 is trusted. This is a reasonable assumption, because the hypervisor is always much smaller than the guest OS, and can be fully verified through formal verification or manual audit. Moreover, the hypervisor is unlikely to install any third-party applications or libraries, and thus gets rid of many potential risks.

As we mentioned previously, the guest OS running on the smart mobile device is untrusted in our platform. It might be malicious and runs a stealthy key logger in the background. In this case, the user's private information, such as the banking account and password, could be compromised. To solve this problem, our platform leverages hardware virtualization on the smart mobile device and adopts a split design for the client program to share the smart mobile device's I/O interfaces. Fig. 1 demonstrates how our platform provides strong security guarantees in the case that the guest OS is untrusted. The guest OS runs on the trusted hypervisor, rather than directly on the smart mobile device's hardware. The app runs on a VM in the trusted cloud. The input proxy, residing in the hypervisor, traps the user's input such that it cannot be received by the guest OS, and transfers it to the VM. This is feasible because the hypervisior is the first layer on the smart mobile device at which hardware events (such as touch points) arrive, so the input proxy can process these events before passing them to the guest OS, or hiding them from the guest OS. Therefore, the app can work properly in the cloud, but the guest OS cannot learn anything about the user's input. On the other hand, the guest OS is the location where the user "logically" launches the app, so it should also be the location where the app's output should be rendered. For this reason, the output renderer is placed in the guest OS, responsible for rendering the app's output.

Our platform provides security guarantees even if the guest OS is untrusted, because the VM in the cloud takes the place of the untrusted guest OS. The app runs on the VM instead of the guest OS, and the user's input is sent to the VM but hidden from the guest OS. However, this implies that our platform only works if the cloud is trusted. We do trust the cloud in our platform, as we mentioned previously, by assuming that the cloud is provided by a famous company with high concern for their reputation, such as Amazon, Google, Apple, and Microsoft. These companies are unlikely to intentionally compromise their users' privacy. Moreover, their clouds often provide powerful anti-virus services, and thus a compromised VM or app running in the cloud will likely be caught by these anti-virus services.

It is also arguable that the user may delete the guest OS if she suspects the guest OS has been compromised, and installs a new one that is safe to execute apps. This solution may also solve the problem, but has some drawbacks. The main drawback results from the fact that a smart mobile device is usually resource

constrained and thus can support only a limited number of guest OSes. In this case, a guest OS on the smart mobile device may have many apps installed and most of them may have no security issues, even if they are executed in a compromised guest OS. Therefore, the user may be unwilling to delete this guest OS. Meanwhile, the limited resources on the smart mobile device prohibit the user from installing a new guest OS.

The second design goal can only be archived by the design illustrated in Fig. 1, which implies that the user needs to execute the app in the cloud to enjoy strong security guarantees. When the user downloads the app to the smart mobile device for some reason, e.g., the latency between the smart mobile device and the cloud becomes unacceptable, she implies that she wants to trade strong security guarantees for higher usability. Therefore, it is natural that our platform does not make the same security guarantees in this case.

IV. System Implementation

To prove that our platform is practical and can work well, we implement a prototype system, following the design outlined in the previous section. In this section, we elaborate on the implementation details of this prototype system.

A. Setup

The smart mobile device used in the prototype is a Samsung Chromebook. We choose this mobile device because it is one of a few that support hardware virtualization, which is required by our platform. The host OS installed on this Chromebook is Ubuntu Linux and the hypervisor running on this host OS is KVM plus QEMU. Both KVM and QEMU are customized for the Chromebook, as provided by [24], the authors of which also provide a bootloader to enable the hardware virtualization features of the Chromebook. QEMU emulates a Cortex-A15 VExpress hardware abstraction, and Android is installed on this hardware abstraction as the guest OS.

The cloud in the prototype system is established on a Lenovo laptop. The host OS on the cloud is Windows, and VMware Workstation is installed on the host OS as the hypervisor. A VM is created on the VMware Workstation with Android-x86 [25] installed as the guest OS.

B. Sharing I/O interfaces

Fig. 3 demonstrates how the components cooperate in the prototype system when an app is executed in the cloud. Darker blocks indicate the components where our implementation has been involved. As depicted in Fig. 3, four components are implemented to share the I/O interfaces across the smart mobile device and the cloud. They are input proxy, input injector, output proxy, and output renderer.

The input proxy works with the input injector to share the input interfaces across the smart mobile device and the cloud. As described in Section III-B, the input proxy resides in the hypervisor on the smart mobile device. We integrate the input proxy into QEMU in our implementation, because QEMU is part of the hypervisor. More specifically, we implement the input proxy's functionality in QEMU, such that QEMU can

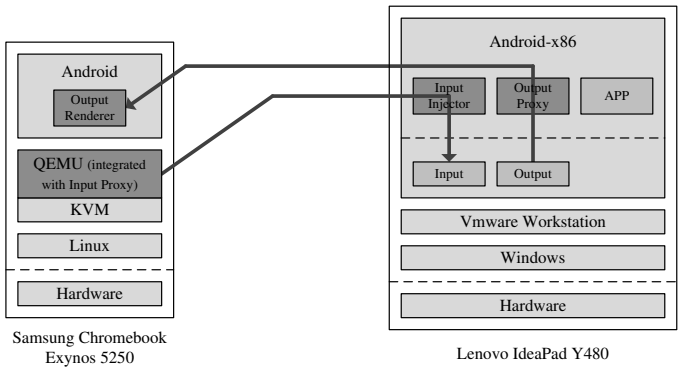

Fig. 3: An app is executed in the cloud in the prototype system. Darker blocks indicate the components where our implementation has been involved.

capture the user's input from the keyboard (i.e., the hardware keyboard) as well as an attached accelerometer (SEN-10537 Serial Accelerometer Dongle), and transfer it to the cloud. There is not much difference between the keyboard and the accelerometer from QEMU's point of view. Therefore, we only focus on the keyboard part in the following discussion.

As the input proxy is integrated in QEMU, we need to define a way by which QEMU can enter and exit the input proxy mode to emulate the input proxy's launch and termination. In our implementation, QEMU will enter the input proxy mode when it detects that the user has repeatedly pressed the "Caps Lock" key six times. After entering the input proxy mode, QEMU will continue monitoring the keyboard and exit this mode if it receives an ESC keystroke. When it is in the input proxy mode, QEMU maintains a connection to the cloud. Upon receiving a key event from the host OS, QEMU will encrypt the key code and send it to the cloud. It will not generate the virtual keyboard interrupt for the guest OS. Therefore, the guest OS will not know this key event and thus cannot learn the user's keyboard input, as described in Section III.

We have considered several places to implement the input proxy functionality. The first place we have considered is the host OS's keyboard driver. However, we have quickly found that this is not a good choice, because it affects all the programs residing in the host OS alongside the hypervisor (i.e., any program running on the host OS will not receive any input events from the keyboard). The second place is the guest OS's keyboard driver. We have also decided against this choice, because it violates our platform design by making the guest OS capable of learning the user's input when the app is running in the cloud. According to our design in Section III, the most natural place to implement this functionality is in the hypervisor. As QEMU is responsible for providing the hardware abstraction to the guest OS, we have decided to implement this functionality in QEMU.

The input injector in the cloud, implemented as an Android app on the Android-x86 VM, is responsible for maintaining the connection to QEMU. It also listens on this connection for encrypted key codes sent by QEMU. When an encrypted key code is received, the input injector will decrypt it, and inject

the key code into the VM through the Linux *sendevent* utility. Then the app running on the VM will be notified of the key event and receive the key code.

The output proxy works with the output renderer to share the output interfaces across the smart mobile device and the cloud. We only consider sharing the screen frames in our prototype system, as this is enough for the user to track the app's output on the mobile device in most cases. Therefore, the output proxy needs to take a screenshot of the Android-x86 VM periodically and transfer it to the output renderer, which will then render it on the smart mobile device's screen.

The output proxy is implemented as another Android app alongside the input injector on the Android-x86 VM. It is responsible for maintaining a connection to the output render, and for sending the VM's screenshots periodically, as fast as possible, through this connection. It captures the VM's screenshot by invoking the *screencap* program provided by Android-x86. A captured screenshot is then stored as a png image file on the local disk. Finally, the output proxy sends the file data to the output renderer, which will further display the screenshot on the smart mobile device's screen.

The output renderer is implemented as an Android app running in the Android guest OS on the smart mobile device. It connects to the output proxy when the app is launched in the cloud. After the connection has been established, the output renderer listens to this connection for the VM's screenshots sent by the output proxy. When a screenshot is received, the output renderer stores it as a local png image file and sets this file as the source of its `SurfaceView` component. The `SurfaceView` component then displays the image file on the screen. This process is done periodically, as fast as possible, such that the user will perceive the stream of the app's screen output as if the app were running locally.

C. Distributed file system

Fig. 4 demonstrates how the components cooperate in the prototype system when an app, installed in the cloud, is downloaded and executed on the smart mobile device. As discussed in Section III-B, we need to share files between the cloud VM and the smart mobile device to allow the app to work properly on the smart mobile device. To achieve this, we implement a distributed file system across the smart mobile device and the cloud in our prototype system.

The implementation of the distributed file system contains three parts, as illustrated in Fig. 4. First, the VFS in the Android guest OS is modified such that it communicates with a user-level program depicted as "file system client" in Fig. 4. This is achieved by leveraging the netlink socket mechanism provided by Linux, as Linux is the underlying kernel of Android. Through a netlink socket, a user-level program and the Linux kernel can communicate with each other in a straightforward manner. Second, the file system client is a user-level program running in the Android guest OS on the smart mobile device. This program is implemented as a Linux binary responsible for communicating with both the local Linux kernel and the cloud. Third, the "file system server" in Fig. 4 is a user-level

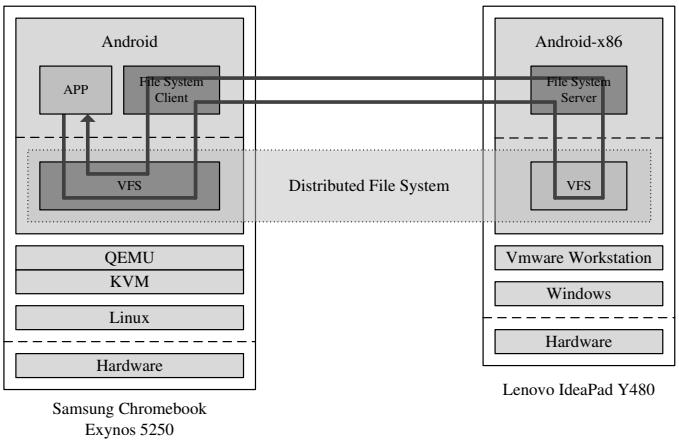

Fig. 4: An app, installed in the cloud, is downloaded and executed on the smart mobile device in the prototype system. Darker blocks indicate the components where our implementation has been involved.

program running on the Android-x86 VM in the cloud. This program is also implemented as a Linux binary, responsible for communicating with the file system client on the smart mobile device and executing file operations on the local file system.

As illustrated in Fig. 4, the smart mobile device and the cloud communicate with each other through a connection between the file system client and the file system server in the distributed file system. This can also be implemented in several ways, e.g., the VFS on the smart mobile device may directly communicate with the file system server, or even with the VFS on the VM, to avoid switching between the kernel and the user space. We choose to use user-level programs to manage this connection, because they are more robust and easier to configure, even though they introduce some overhead when interacting with the local VFS in the kernel.

When an app is executed in the cloud, it may only access local files. To access a local file, it will first invoke the system call *sys_open()* to communicate with the VFS, which will open the file and return the file descriptor to the app. Through this file descriptor, the app can read data from the file, write data to the file, or close the file by invoking the *sys_read()*, *sys_write()* and *sys_close()* system calls, respectively. When the app is downloaded from the cloud and executed on the smart mobile device, however, it may access remote files in the cloud. The distributed file system is implemented to support this kind of remote file access in the prototype system.

To implement this distributed file system, we begin with a simple solution. We redirect every *sys_open()*, *sys_read()*, *sys_write()* and *sys_close()* system call that is invoked on the smart mobile device's VFS to the VM's VFS running in the cloud. To be more specific, when an app running on the smart mobile device tries to open a file that is considered in the cloud (e.g., in a directory under */sdcard*), the smart mobile device's VFS will switch to user space by notifying the file system client of this event. Then the file system client will redirect the *sys_open()* system call to the file system server in

the cloud with exactly the same parameters. After receiving this redirected system call, the file system server invokes it on the VM's VFS and gets the corresponding file descriptor. It then returns this file descriptor to the file system client, which will switch back to the local VFS with the file descriptor it has received. The VFS logs this file descriptor as a remote one and returns it to the app. Then the app can use this file descriptor to read, write, and close the remote file. When the app reads the file through this file descriptor, the VFS will notice that this file descriptor actually refers to a remote file. It will then redirect the *sys_read()* system call to the VM's VFS similarly to when it redirects the *sys_open()* system call.

The solution described above is straightforward. However, it involves too many network communications between the smart mobile device and the cloud, and too frequent switching between the user space and the kernel on the smart mobile device. As a result, it has prohibitively low performance. To solve this problem, we implemented the distributed file system differently. Fig 4 illustrates part of this new solution, i.e., how it works when the app tries to open a remote file in the cloud. When receiving a *sys_open()* system call from the app and finding that it wants to open a file in the cloud, the smart mobile device's VFS will switch to user space by notifying the file system client of this event. The file system client then communicates with the file system server, which will send back the data of the corresponding file. The file system client then stores the file on the local disk as a cached file, with a temporary file path that is not likely to conflict with that of any other file. Then the smart mobile device's VFS opens this file instead of the remote file and returns its file descriptor to the app. The file descriptor is marked as "remote" and the remote file path is recorded in the file descriptor by using two fields that we have added in the file descriptor data structure. Then the app can use this file descriptor to read and write the cached file. When closing the file, the VFS will notice that it is a cached file. It will switch to user space by notifying the file system client of this event. The file system client then sends the cached file back to the file system server, which will overwrite its local file with the cached file in the cloud.

It is arguable that this solution may cause inconsistency when an app on the smart mobile device and an app in the cloud access the same file simultaneously. Nevertheless, it is rare that two apps access the same file simultaneously. Most mobile OSes, such as Android, forbid or at least recommend against an app accessing files belonging to another app, for security purposes. Furthermore, although it is not mandatory, a prudent way to use our platform is to create a separate VM in the cloud for every app. Two apps will never access the same file simultaneously in this case.

V. Discussion

Hypervisor in local mobile device. To allow for consistency with our system implementation, we present our system design in a way that used a "type 2" hypervisor (i.e., a hosted hypervisor that runs in a host OS) in the local mobile device to achieve high security and privacy guarantees. It is worth noting that it is not necessary to use type 2 hypervisor in our design. Actually, a type 1 hypervisor (i.e., bare-metal hypervisor that runs directly on top of the hardware) fits better into our goal of providing high security guarantees. This is because the main reason we use a hypervisor in local mobile device is to isolate the sensitive inputs (i.e., user taps on the touch screen, data collected by various on-board sensors) from local mobile device OS and apps. These components are greater risks if they are compromised. A type 1 hypervisor, with a small trusted computing base (TCB), can also fulfill our needs, but developing a new hypervisor from the ground up, based on the newly introduced ARM hardware virtualization, would require a lot of effort in engineering optimal parameter settings. Therefore, we opt to use KVM, which is a full-fledge hypervisor that has been recently ported to the ARM architecture, for a fast proof of concept demonstration of our system design. For future work, we plan to develop our own bare-metal hypervisor to further improve our system.

The use of Chromebook in the prototype system. We use a Samsung Chromebook as the mobile device in our prototype system implementation. The main reason for this choice is that it has a hardware configuration that supports ARM hardware virtualization, and it requires relatively less effort to enable and run the ARM based KVM, which is the base hypervisor for our prototype system. Although the Samsung Chromebook looks more like a regular laptop, it actually shares the hardware similar to most recent smartphones and tablets. For example, the Chromebook used in our system has a Samsung Exynox 5 Dual SoC, which is the same SoC found in the Google Nexus 10 tablet. The ARM Cortex-A15 MPcore processor contained in the Exynos 5 Dual SoC is also used in many other smartphones, like the Samsung Galaxy S4/S5 smartphones, the Galaxy Note 3 smartphone, and the Galaxy Tab Pro tablets. The 2 GB RAM capacity in our Chromebook is also the standard configuration commonly found in the latest smartphones. Therefore, our choice of using Chromebook as the mobile device can fit well into the smart mobile device world in terms of computational capability.

VI. Evaluation

In this section, we describe the real-world experiments we have conducted to evaluate the performance of our prototype system.

A. Experimental setup

Our prototype system consists of two parts, the smart mobile device and the cloud VM. The smart mobile device is a Samsung Chromebook featuring the Exynos 5 Dual SoC, 2 GB RAM and 16 GB SSD hard drive. The Exynos 5 Dual SoC is equipped with a 1.7 GHz dual-core ARM Cortex-A15 processor. The ARM Cortex-A15 processor has hardware virtualization support, which allows us to implement the proposed hypervisor in our prototype system. The Chromebook runs Linux (kernel version 3.13.0) as the host OS. On top of the hypervisor, we run Android Jelly Bean (Android version 4.1.1, Linux kernel

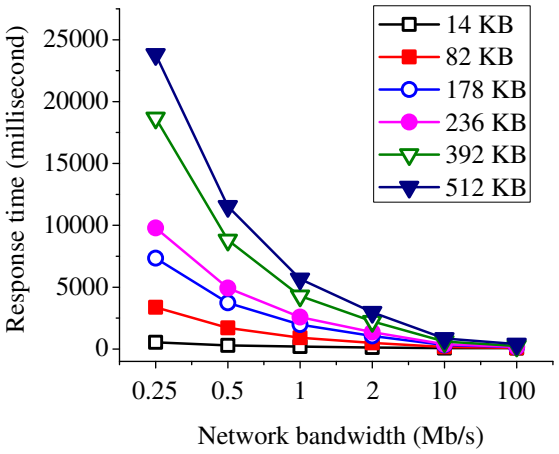

Fig. 5: App response time if running in the cloud: UI response time (Y-axis) of screenshots with different size under different network conditions (X-axis).

version 3.9.0) as the guest OS. The cloud VM runs Android-x86 [25] as the guest OS (Android version 4.3, Linux kernel version 3.10.2). It is hosted by the VMware workstation 10.0.1 virtual machine monitor (VMM) running on a host PC with an Intel Core i7 CPU (2.3 GHz) and 8 GB RAM. The smart mobile device and the cloud VM are connected through an 802.11n wireless router.

B. Running apps in the cloud: the response time

The most significant feature of our system is that we allow the entire mobile device OS and the apps to run in the cloud. Therefore, we first evaluate the app response time when the apps are running in the cloud. In our system, we send the output of an app running in the cloud back to the device by first taking screenshots of the app, and then transmitting them back to the device. Therefore, there are two major factors that can affect app response time, network bandwidth and the size of each screenshot. We design an experiment to evaluate the impacts of these two factors. In our experiment, we use an image viewer app to open different pictures. Remember that with our system, the app launch and the picture opening operations are triggered by the user on the mobile device side, and the actual operations are conducted on the cloud side. Once a picture is opened, our system takes a screenshot of it, and sends the screenshot back to the mobile device to display to the user. We measure the response time as the time difference between the point when the user triggers the picture opening action on the mobile device, and the point when the screenshot is sent back from the cloud and displayed. We choose different pictures, such that we have screenshots with different sizes (14 KB, 82 KB, 178 KB, 236 KB, 392 KB, and 512 KB are used in our experiment). We also configure the wireless router to achieve different network bandwidth rates (0.25 Mb/s, 0.5 Mb/s, 1 Mb/s, 2 Mb/s, 10 Mb/s, and 100 Mb/s are used in our experiment).

Fig. 5 depicts the experiment results. We can see that network bandwidth has a significant impact on app response time when the bandwidth is small. But this impact gradually diminishes as the available bandwidth increases. For example, when the network bandwidth is fixed at 0.25 Mb/s, it takes only 0.5 second to open a picture 14 KB in size. But it needs almost 24 seconds to get back a 512 KB screenshot. The response time ratio for these two screenshot sizes (i.e., $\frac{512\text{KB}}{14\text{KB}}$) is $\frac{24\text{second}}{0.5\text{second}} = 48$. But when the network bandwidth is set at 100 Mb/s, the response time ratio for the same screenshot sizes (i.e., $\frac{512\text{KB}}{14\text{KB}}$) is only $\frac{0.4\text{second}}{0.07\text{second}} = 5.7$. Fortunately, modern mobile data networks can support a very high transmission rate. The 4G LTE network has a peak download speed approaching 100 Mb/s [26], and on a typical day 4G download speeds can range from 2.8 Mb/s to 9.1 Mb/s, with an average value of 6.2 Mb/s [27]. With the average 4G download speed (6.2 Mb/s), the app response time of our prototype system for a screenshot of 512 KB is only about one second.

TABLE I: Hypervisor overhead

	Native OS	With unmodified hypervisor	With our hypervisor
Delay	4 ms	63 ms	65 ms

C. Performance of the hypervisor on the local device

Our system exploits the ARM hardware virtualization technology to achieve user/sensor input isolation from the mobile device OS. Specifically, when an app is running in the cloud, our hypervisor intercepts all the inputs from the user (e.g., touch screen, keyboard) and sensors, performs an encryption on them, and sends them to the cloud. We design an experiment to evaluate the overhead introduced by our hypervisor. In this experiment, we use keyboard input to evaluate the hypervisor overhead. We test three cases. In the first case, we use the keyboard to provide inputs for a user program running on the native OS of the mobile device. In the second case, we test the same scenario, except that the user program is running in the guest OS of the mobile device. In this case, the unmodified hypervisor will introduce some overhead to the keyboard input operation. The third test case shares the same setup as the second one, except that the hypervisor is the one used in our prototype system, which performs encryption on the intercepted keyboard inputs. Because we want to test the overhead caused by our hypervisor, we direct the encrypted input up to the user program in the guest OS, instead of redirecting it to the cloud. In all the three cases, we measure the time delay between the keyboard input interrupt and the time when the user program receives the input. We perform each test ten times, and report the average value here. Table I shows the results of the experiment. The results suggest that by running the user program in the guest OS, the time delay increases by one order of magnitude, compared to that when the user program is running in the guest OS. This is normal because the guest OS involves many switches between many entities including the guest OS, the hypervisor, the host OS, and the QEMU hardware emulator (required by KVM). It is worth noting that when comparing to the case with the unmodified hypervisor, our hypervisor only incurs a very small amount of additional delay (about 3%).

TABLE II: Performance of the distributed file system

	64 KB	256 KB	512 KB	1 MB	5 MB	10 MB
Open	36 ms	85 ms	115 ms	225 ms	827 ms	1324 ms
Read	1 ms	1 ms	5 ms	11 ms	20 ms	38 ms
Write	1 ms	3 ms	6 ms	13 ms	68 ms	128 ms
Close	39 ms	68 ms	116 ms	219 ms	680 ms	1337 ms

TABLE III: Performance of the native file system

	64 KB	256 KB	512 KB	1 MB	5 MB	10 MB
Open	1 ms	1 ms	2 ms	5 ms	14 ms	24 ms
Read	1 ms	1 ms	6 ms	11 ms	21 ms	40 ms
Write	1 ms	3 ms	6 ms	13 ms	67 ms	126 ms
Close	1 ms	1 ms	2 ms	3 ms	14 ms	27 ms

D. Performance of the distributed file system

The purpose of our distributed file system is to allow users to run apps locally on their mobile device. In this experiment, we evaluate the performance of file open, read, write, and close operations of our distributed file system. Each test is performed ten times. Table II shows the performance results of our distributed file system. As a comparison, Table III shows the results of the native file system. From these results we can see that our file system incurs a time overhead for the open and close operations. This is because when an app running in the mobile device tries to open a certain file that is not available in the local device, our file system transparently caches the file of interest from the cloud to the mobile device to allow the app to proceed. When the app finishes accessing and closes the file, our file system automatically writes the file back to the cloud. Therefore opening files is costly. But writing and reading them is much less so. Since we are using whole file caching in our current implementation, file size and network bandwidth have major impacts on the open/close delay. For file read/write, since our file system allows local access to the cached copy, it has the same performance as the read/write operations in a native file system.

VII. Conclusion

The mobile-cloud computing model will be the dominating trend in the future. However, existing solutions do not fully exploit the potential of this model. To this end, we aim at designing a solution that goes beyond the current state of the art. In this paper, we propose a novel mobile-cloud platform with two fundamental contributions. First, our platform allows users to freely choose to run their applications either in the cloud or on their local devices. We feel that this is a very useful and practical feature for users, and believe we are the first to consider this situation in a mobile-cloud platform of this kind. Second, our platform provides security guarantees against untrusted applications and an untrusted local device's operating system, by leveraging hardware virtualization technology. To the best of our knowledge, we are the first to utilize hardware virtualization to strengthen the security on mobile devices. Based on these design concepts, we build a prototype system on a Chromebook acting as the user's local mobile device, and a commodity x86 laptop PC, which acts as a cloud server. Our evaluation on the prototype system proves that our platform is useful and pragmatic.

Acknowledgment

The authors would like to thank all the reviewers for their helpful comments. This project was supported in part by US National Science Foundation grants CNS-1320453 and CNS-1117412.

References

[1] X. Ni, Z. Yang, X. Bai, A. C. Champion, and D. Xuan, "Diffuser: Differentiated user access control on smartphones," in *IEEE MASS*, 2009.
[2] J. Teng, B. Zhang, X. Li, X. Bai, and D. Xuan, "E-shadow: Lubricating social interaction using mobile phones," in *IEEE ICDCS*, 2011.
[3] Y. Cui, H. Wang, X. Cheng, and B. Chen, "Wireless data center networking," *IEEE Wireless Communications*, 2011.
[4] Wikipedia, "Chrome OS," http://en.wikipedia.org/wiki/Chrome_OS.
[5] B. Chun and P. Maniatis, "Augmented smartphone applications through clone cloud execution," in *USENIX HotOS*, 2009.
[6] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *ACM EuroSys*, 2011.
[7] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *ACM MobiSys*, 2010.
[8] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *USENIX OSDI*, 2012.
[9] R. A. Baratto, S. Potter, G. Su, and J. Nieh, "Mobidesk: Mobile virtual desktop computing," in *ACM MobiCom*, 2004.
[10] C. Tsao, S. Kakumanu, and R. Sivakumar, "Smartvnc: An effective remote computing solution for smartphones," in *ACM MobiCom*, 2011.
[11] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *USENIX HotCloud*, 2010.
[12] Y. Lin and M. D. Francesco, "Energy consumption of remote desktop access on mobile devices: An experimental study," in *IEEE CloudNet*, 2012.
[13] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, 2005.
[14] AMD, "AMD Virtualization," http://www.amd.com/en-us/solutions/servers/virtualization.
[15] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SOSP*, 2007.
[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *IEEE SP*, 2010.
[17] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE SP*, 2008.
[18] R. Mijat and A. Nightingale, "White paper: Virtualization is coming to a platform near you," 2011.
[19] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," in *ACM ASPLOS*, 2014.
[20] P. Varanasi and G. Heiser, "Hardware-supported virtualization on arm," in *ACM APSys*, 2011.
[21] H. Han, Y. Liu, G. Shen, Y. Zhang, and Q. Li, "Dozyap: Power-efficient wi-fi tethering," in *ACM MobiSys*, 2012.
[22] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li, "Optimizing background email sync on smartphones," in *ACM MobiSys*, 2013.
[23] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *USENIX NSDI*, 2013.
[24] "Virtual Open Systems," http://www.virtualopensystems.com/.
[25] "Android-x86 Project," http://www.android-x86.org/.
[26] Wikipedia, "4G," http://en.wikipedia.org/wiki/4G.
[27] Y. Zhang, C. Tan, and Q. Li, "Cachekeeper: A system-wide web caching service for smartphones," in *ACM UbiComp*, 2013.