2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

# Decomposable Trust for Android Applications

Earlence Fernandes, Ajit Aluri*†, Alexander Crowell, Atul Prakash
Computer Science and Engineering, University of Michigan, Ann Arbor
{*earlence, acrowell, aprakash*}*@umich.edu*, †*aaluri@vmware.com*

***Abstract*—Current operating system designs require applications (apps) to implicitly place trust in a large amount of code. Taking Android as an example, apps must trust both the kernel as well as privileged userspace services that consist of hundreds of thousands of lines of code. Malware apps, on the other hand, aim to exploit any vulnerabilities in the above large trusted base to escalate their privileges. Once malware escalates its privileges, additional attacks become feasible, such as stealing credentials by scanning memory pages or intercepting user interactions of sensitive apps, e.g., those used for banking or health management. This paper introduces a novel mechanism, called Anception, that strategically deprivileges a significant portion of the kernel and system services, moving them to an untrusted container, thereby significantly reducing the attack surface for privilege escalation available to malware. Anception supports unmodified apps, running on a modified Android kernel. It achieves performance close to native Android on several popular macrobenchmarks and provides security against many types of known Android root exploits.**

***Keywords*—*Android, Virtualization, Root Exploits, Trust Decomposition***

## I. Introduction

Smartphones are characterized by an ecosystem of online app markets that enable developers (hobbyists, professionals, criminals) to reach large audiences. According to a recent Kaspersky report [30], 98.05% of known malware targets Android, a popular smartphone platform, to gain a foothold on the device. Once a foothold is acquired, malware escalates its privileges and then targets mobile banking and personal data [3], [2].

Android provides basic isolation among apps. For example, Android assigns a different Linux-UID (user ID) to each installed app. However, in practice, malware can still exploit vulnerabilities in system services or the operating system to escalate its privileges and break isolation [22], [31], [34], [37]. For example, Gingerbreak was an exploit on Android that exploited a vulnerability in the root-capability *vold* volume daemon on Android to escalate its privileges. Another attack vector for unprivileged malware is to exploit a kernel vulnerability. For example, on Linux, vulnerabilities continue to be found – CVE-2013-2094 local privilege escalation that exploits Performance Counters for Linux, CVE-2014-7145 in the Linux CIFS file system code, CVE-2014-6416 buffer flow vulnerability in the Linux network code [14].

Once an unprivileged malware exploits a privileged service or the kernel, it opens the door to further rootkit-style attacks, including tampering with the code of installed apps or system libraries, examining and tampering with virtual memory of other apps, monitoring their communications, etc. For example, in the *Man-in-the-Binder* attack, once malware gains root privilege on the OS it intercepts IPC communication between an app and the UI stack to steal all touch input data, that includes text input on the virtual keyboard such as userids and passwords [7].

In this paper, we aim to protect sensitive data in an app's virtual memory such as banking credentials, health data and corporate data. These apps authenticate the user via UI interaction and these interactions must be protected. While these apps may send data over the network over an authenticated and encrypted channel, the same data will reside in unencrypted form in virtual memory. Thus, virtual memory of these apps must also be protected. Currently, none of this data is protected if an unprivileged app exploits a kernel or privileged service bug and escalates its privileges.

The most secure solution today to address the problem for the user to use two physical devices, one for trustworthy apps and another for untrustworthy apps. Besides being inconvenient in terms of managing two devices, the user has to judge the trustworthiness of apps correctly. If the user is tricked even once into installing a malicious app on the same device as the one that contains trustworthy apps, security guarantees break down.

A similar solution to the two-device solution is to use a single device that is partitioned to provide multiple virtual devices. Cells [5] and systems based on Cells, e.g., Airbag [45], are examples of this. This reduces the inconvenience of carrying two physical devices, but retains the other disadvantages of a two-device solution. If a user is tricked into installing a malicious app on the virtual device that contains trusted apps, the confidentiality of data in trusted apps can be violated by privilege escalation attacks. Another system proposal is Overshadow [13] and similar mechanisms [47], [25] that introduce a memory cloaking primitive wherein an app's virtual memory is encrypted in a trusted layer (such as a hypervisor or hardware support [9]) upon a context switch to the untrusted OS (or another process). Unfortunately, the Android's UI stack resides within the untrusted OS and thus will remains vulnerable to malware – the design does not consider securing UI interactions – the primary method for sensitive data to flow between a user and the app.

This paper presents a novel solution for protecting apps from each other. Our design, called *Anception*, uses virtualization as a building block but does *not* require the user to make an *a priori* judgment on trustworthiness of apps. Instead, it deprivileges significant portions of the kernel and system services so that the attack path that is normally possible for a malware app for privilege escalation is blocked.

To achieve its security goals, Anception adapts the classical virtualization model and executes many system services as well as most system calls on a low-privilege container kernel.

*Ajit Aluri is now with VMware.

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.15

However, unlike the classical virtualization model, the container kernel does not have access to either the user-interface interactions of apps or to their virtual memory. This provides a foundation for building high assurance apps that can better protect themselves without requiring a shaky assumption that the user will never install malicious apps on the same system.

We prototyped Anception on Android 2.3 and 4.2 in the form of two loadable Linux kernel modules, consisting of approximately 5.2K lines of code. Anception does not require any modifications to the Android Framework or to Android applications. We make the following contributions:

- The notion of *trust decomposition* for Android apps running on a monolithic OS, wherein, the trust an app places in the OS is split between a smaller trusted host component and a larger untrusted component. The key security guarantee enabled is confidentiality of virtual memory and of UI interactions with a smaller trusted base.
- The design and implementation of Anception, a system architecture that deprivileges Android system services and kernel services and delegates their functionality to an unprivileged virtualized container (Sections III, IV). Anception is able to deprivilege 1.2M lines of code from the linux kernel and 108K lines of code from the privileged Android userspace (Section V)
- Detailed evaluation of the security decisions made during design (Section V). We analyzed 25 Android vulnerabilities related to privileged system services and kernel services from the past 4 years and determined that the reduced attack surface provided by Anception would have blocked 23 of them sufficiently to prevent significant attacks on both the host OS as well as other apps on the system; the remaining two attacks would have succeeded, but could have been detected and prevented with simple policy-based checks at the system-call interface on both standard Android and Anception-based Android.
- Performance evaluation of Anception-based Android (Section VI). While Anception-based Android suffered in performance on some microbenchmarks that involved system calls crossing boundaries between the trusted host and the untrusted container, the performance hit was relatively modest on I/O-based benchmarks and negligible on graphical and interactive macrobenchmarks.

## II. Threat Model and Security Guarantees

***Threat model.*** The attacker is a low assurance app downloaded from official and unofficial app stores and exploits vulnerabilities in the kernel services and privileged userspace services with the aim of corrupting and stealing information from high assurance apps. We assume that both high and low assurance apps are installed on the same operating system. Additionally, high assurance apps are well-designed – they use encryption for network communications and do not leave secrets in plain text anywhere except in virtual memory.

We do not prevent theft of secrets in an app's memory via covert channels, e.g., by observing memory and CPU usage patterns. We also do not prevent disruptions. A malicious app may be able to disrupt the running of other apps even without escalating privileges. Such attacks are likely to be noticed quickly by users and thus we consider them to be less effective.

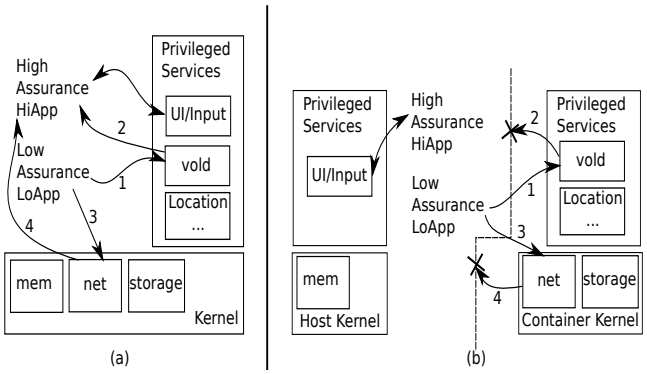


Fig. 1. Exploitation channels available. In (a), the low assurance app triggers a vulnerability in a privileged service (1) and uses the extra privileges to tamper with the high assurance HiApp (2). Similarly, LoApp uses an exploit in the network stack (arrows 3 and 4). On Anception (b), the compromised privileged service cannot directly access the state of HiApp (arrows 2 and 4 are blocked) because the services are delegated to the container.

The host operating system kernel must be trusted but normally presents a significant attack surface. Our system reduces the attack surface it presents to apps by running any privileged operations of an app in a virtualized environment.

***Security Guarantees.*** We provide confidentiality of an app's virtual memory in the presence of malicious apps that exploit kernel and privileged userspace services for privilege escalation. Additionally, we ensure that malware cannot eavesdrop on and tamper with the UI interactions of an app.

## III. Anception Design

### A. Overview

Consider two apps, HiApp and LoApp. HiApp is a high assurance app such as a banking or health management app and LoApp is a low assurance one such as a game or a calculator. Figure 1(a) shows the execution environment of an app on Android. The kernel provides the usual services of memory, storage, and networking, among others. The privileged services provide higher level functions such as data management, sensor management, location management, etc. Privileged services consist of large bodies of complex systems code and thus provide a large attack surface (on Android, the privileged services consist of approximately 180K lines of code). The same attack surface is exposed to both apps. Worryingly, LoApp can exploit any bugs in the large attack surface presented by the privileged services as well as the OS and then use their privileges to exploit and steal secrets contained in the memory of HiApp. Examples of possible exploits using previously vulnerabilities on Android include the following: (1) LoApp exploits a negative index vulnerability in the privileged `vold` service (Figure 1) and uses it to exploit HiApp[1]; and (2) LoApp triggers an exploit in the networking stack and uses kernel privileges to steal secrets from HiApp[2].

[1]This vulnerability existed on Android, and is known as GingerBreak.

[2]This too existed in the Linux kernel and is known as CVE-2009-2692.

The core of the matter is that high and low assurance apps share the same trusted, but likely buggy, base. It is perhaps reasonable to assume that high assurance apps and services do not exploit the bugs; however, no such assumption can be made of low assurance apps. Thus, we seek to split the trusted base such that low assurance apps cannot violate the confidentiality of high assurance apps by exploiting privileged services, even if these services are buggy. Our confidentiality guarantees are primarily confined to virtual memory of high assurance apps. As far as the confidentiality of file system data and network communication of high assurance apps is concerned, it can be relatively easily provided by using encryption (we do protect the keys that are used to bootstrap the encryption). Also, we focus on confidentiality rather than integrity. As we will see, the options available for violating integrity of a high assurance app will be severely limited and are likely to be detected. We assume that high assurance apps use end-to-end authentication and encryption when communicating over the network.

Our defense partitions the trusted base into a host kernel and a container kernel. The privileged services are carefully partitioned to run on these two kernels. An important question is deciding the partition in which each service executes. Figure 1(b) shows our decision. The trusted host contains the virtual memory of the app, its read-only code and the UI/Input service. The untrusted container has the storage, networking and other services, such as *vold* and location service. The goal is to run non-UI and non-memory services in the container to the extent feasible to reduce the available attack surface. The container itself is deprivileged *with respect to* the host kernel; however, services within the container maintain enough privilege to perform their assigned tasks. We measured the lines of code in privileged services on Android. We found that approximately 109K lines out of 181K are not connected with UI/Input management, suggesting that up to 59.9% of the trusted base in the form of privileged services can be reduced.

The reason for keeping the UI/Input service as part of the trusted host kernel is that fundamentally all sensitive interactive input from the end-user is obtained through the UI/Input service (e.g., passwords, touch inputs). If this service had been delegated to the untrusted container, a compromise within the container would have allowed stealing of user-provided sensitive information.

The virtual memory of an app also must necessarily contain sensitive state (e.g., passwords). Hence, we maintain virtual memory of all apps on the host. In our example, LoApp may compromise a privileged service, but this service resides in the container that is unprivileged with respect to the host kernel. Thus, the attacker cannot access HiApp's memory via the deprivileged service, thereby reducing the available attack surface.

Storage deserves special consideration. A typical storage stack is quite large (ext4 on Linux is approximately 26K lines of C, and all filesystem-related Linux code is approximately 725K lines of C). Therefore, it is desirable to delegate storage calls to a container. However, the storage stack is also used by trusted host services (e.g., package installer) and the HiApp's code must be protected from tampering by LoApp. To protect an app's code but still redirect all app-generated file system calls to the container, Anception keeps the read-only code of an app on the host and keeps all other data files of the app in the container. This design implies that the app's data could be stolen if a low assurance app compromises the container. Fortunately, this problem is easy to address by high assurance apps encrypting their writes to the container or by extending our file system implementation in the container to use known transparent encryption techniques [18] to secure the reads/writes when they cross the boundary from host kernel to the untrusted container kernel. Similarly, we assume the high assurance app encrypts network packets before utilizing container network services. We describe a detailed example of how to use Anception's architecture to construct a secure banking app later in this section.

Anception bridges IPC channels to function across the kernels. For example, our implementation supports shared memory and Android's custom Binder IPC. We also designed a system call bridge that transfers system calls destined for the container. We provide details later in this section and in Section IV.

Anception's design thus simplifies the ability of apps to protect themselves from malicious apps by reducing the attack surface that is available by both deprivileging services as well as by transferring the network and filesystem calls to an untrusted container kernel.

### B. Architecture

We utilize a deprivileged container to execute delegated privileged services. We refer to this as a container virtual machine (CVM). Anception's design is based on the following principles:

1) *Launch the app from the trusted host kernel:* The app's code is stored outside the CVM.
2) *Protect the UI/Input from the container's kernel:* Do not trust the container with UI related operations but handle them centrally on the host.
3) *Protect the app's virtual memory from the container's kernel:* Do not trust the container with virtual memory of applications. Instead, pages are managed by the trusted host outside the control of the container.
4) *Protect the host:* Protect the host system as much as possible under the constraint of the first three principles. To the extent feasible, privileged operations (system calls) invoked on the host kernel should instead run in the context of the CVM.

The above principles have to be achieved on an operating system designed for mobile devices where resources are limited. Since attacks on the kernel are part of the threat model, a possible solution is to give every app a guest kernel, but we assume that is not yet practical for both resource and performance reasons, and unlikely to be practical even with hardware advances as the power demands of an increased code base will outweigh any benefits.

We briefly describe why each of the above principles is necessary under our threat model in order to protect high assurance apps from untrusted apps. We assume that a low assurance app has compromised the CVM via the larger available attack surface (e.g., one of the privileged services) and has escalated its privileges. Consider a well-designed mobile banking app. Upon launching, the banking app acquires

a user ID and password via the touch-screen interface on the device.

Since the attacks we consider enable privilege escalation, the attacker can modify the app's code such that the banking app itself communicates the password to an attacker. On Anception, principle 1 ensures that the app code is stored on the host. As stated, all non-code state of apps are maintained in the CVM. Thus any attacker-issued writes are serviced in the CVM that does not contain app code.

Next, the compromised OS can intercept the user ID and password as it is entered by the user. Principle 2 ensures that the CVM will not have an opportunity to intercept that input since the UI stack does not run in the CVM.

Next, the app will store the user ID and password in its virtual memory. This provides another opportunity for the attacker to steal the information. For example, if there were to be a scheduler context-switch due to a timer, the kernel gets control and could read the app's memory. Principle 3 ensures that the CVM will not be able to read the user-level memory pages of the app.

Principle 4 is necessary since the host kernel must be trusted (that is true in all virtualization solutions). Thus, we minimize the attack surface of the host kernel, i.e., execute the system calls on the CVM's kernel when it is consistent with other principles.

We now show that above principles are sufficient to build a simple secure banking app that can protect its sensitive data from malware. Figure 2 depicts a banking app running on Anception's infrastructure. The app is launched from the host and its memory pages exist only on the host. A certificate to authenticate the bank's server is read from its code base and loaded into its virtual memory at launch time. It receives a user ID and password securely from the host-side display manager and stores them in its isolated memory pages. Using these two items, the bank app can communicate with the bank server over an end-to-end secure protocol such as TLS/SSL (session keys for this will get negotiated end-to-end and reside in the memory of the app). Communications go through the CVM, potentially running malicious apps, but the CVM is unable to read them. A bank server may also provide secure storage of persistent data through this secured network connection.

Note that writing to local storage is not necessary to build a fully-functional secure app that works with our threat model. In practice, however, most apps do take advantage of local storage. If that were to be the case, a compromised container can steal information from such storage. However, an app may store cryptographic keys in its code that is protected from other apps and from the CVM. Then, the app can encrypt any data written to storage in the CVM. We discuss a enhancement to the Anception design in Section VII to allow secure local storage transparently for apps.

***Launching apps securely.*** When an app is downloaded from an online store, it is installed on the host. Thus the app's code resides on the host. Figure 3 shows the banking app's code outside the CVM. The code is maintained in a permission protected directory such that only the app and the system may access the code. Android already supports this requirement. This achieves the first principle. Note that Anception provides

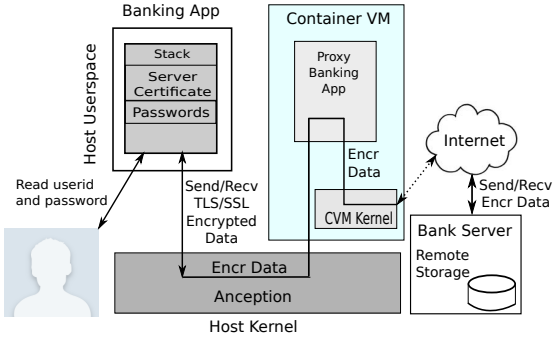


Fig. 2. Design of a secure banking app. The app maintains cryptographic keys in its read-only code in trusted storage on the host VM. This code is only accessible by the app and the system. Other apps cannot access the trusted storage.

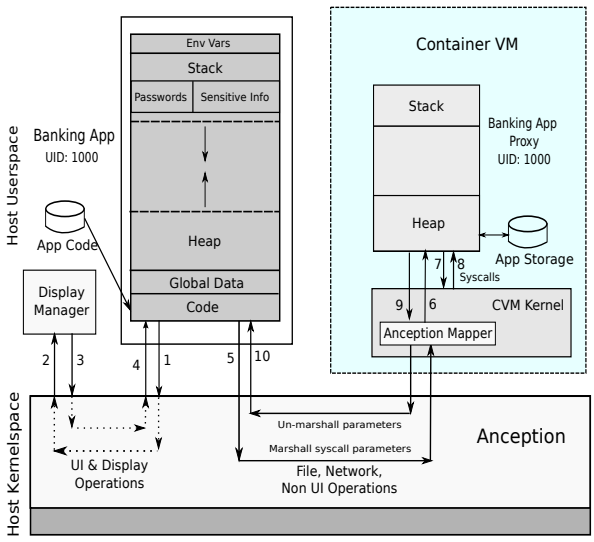


Fig. 3. High Level Architecture. A proxy process executes in the CVM whose purpose is to execute forwarded system calls. The proxy has the same security credentials of the host process *within* the CVM (same UID, set of permissions).

protection to only non-root user apps. Therefore, all launched apps have a non-zero UID. [3]

***Isolating and securing the UI/Input.*** On Android, if an app wishes to display output or receive input, it needs to create the appropriate UI elements in memory and then request the services of the display and input managers through an IPC. The IPC interface is well defined on smartphone operating systems, just as in standard Linux based systems (OpenGL, X). On Android, display-related operations occur through IPCs (specifically, a type of `ioctl` system call) on the *WindowManager*. Anception detects and identifies these operations at the system call interface and lets them pass through to the host OS (Figure 3). Any information collected through UI elements resides on the host only, hence obeying the second principle.

[3]If an app changes its UID after being launched, Anception will kill the app. Such changes are not permitted as per the Android security model and are made possible only on rooted devices, that are outside our threat model.

Servicing UI/Input on the host enables an important optimization. The CVM runs a headless Android stack, thus cutting its memory consumption. We have evaluated memory consumption (Section VI-C) and found that Anception runs an Android stack consuming around 64MB [4].

***Protecting User Pages of an App from the CVM.*** When the user launches the app, code is loaded into memory pages on the host. These pages are not visible to the CVM because the host exercises strict control over the pages available to the CVM, thus obeying the third principle. As we observe in Figure 3, sensitive data exists in pages outside the addressable region of the CVM.

***Reducing the Attack Surface of the host kernel.*** Anception creates a lightweight proxy for the app that executes *within* the CVM and has the same security credentials (UID, umask, directory structure) as that of its host counterpart *within* the CVM. When the app issues a system call, Anception's host-side redirection logic inspects the call and generally redirects them to the CVM. The proxy's purpose is to execute any forwarded system calls from the host kernel. The other advantage of using a proxy is that the permissions model of the host is transported to the CVM. The same permission checks that would have been executed on the host for a process are executed on the CVM. In Figure 3, Anception operates in host kernel space and serves to transfer such calls to the CVM kernel by performing the appropriate marshaling (including pointer translation if call arguments contain pointers) of data.

### C. Splitting the execution of Android apps

Anception services a subset of system calls on the CVM, which is lower privileged, while the process memory resides on the host. The other subset of calls is serviced on the host kernel. Consider the following cases:

- *An app that makes no system calls*: In this case, the app executes entirely on the host. Since it makes no system calls, the app is not going to be able to attack the system to escalate its privileges. Such apps are not likely to be very interesting, but they work correctly under Anception.
- *An app that makes only UI/Input system calls*: Anception recognizes UI/Input related calls and lets them execute on the host. This is possible because the UI operations are well-defined and easily identified. There is no overlap of UI-managed (e.g., framebuffer, touch screen) resources with other resources visible to an app (e.g. files); that is, UI related calls do not manipulate non-UI resources on Android.
- *An app that makes system calls that depend on an abstract handle*: As before, Anception recognizes the UI calls and service them on the host. Other calls that provide functionality through an abstract handle, such as the file descriptor, can be serviced in the CVM as long as the invariants provided by the handle in the form of its exposed functionality are maintained. Examples include the vast majority of file system, IPC, and networking related calls.

[4]For comparison, even an old version of Android (GingerBread) required atleast 256MB to run.

```
1  // obtain an FD for binder communication
2  binder = open('/dev/binder', 'rw');
3  // get bank server cert that came with the code
4  sockfd = socket(AF_INET, SOCK_STREAM, 0);
5  connect(sockfd, "bank.com", ...);
6
7
8  ioctl(binder, IOC_WAIT_INPUT_EVT, &in_buff);
9  if(in_buff.event == EVT_PWD_ENTERED)
10 {
11   // append the command for the bank server
12   in_buff.text += '_LOGIN_CMD';
13   // call a userspace encryption routine
14   // establish TLS connection and authenticate
15   // using the id/password to
16   // the bank
17   send(sockfd, outbuff, sizeof(outbuff));
18 }
19
20 close(binder);
21 close(sockfd);
```

Listing 1. Simplified benign code

```
1 sockfd = socket(PF_BLUETOOTH, SOCK_DGRAM, 0);
2 fdin = open('arbitrary.txt', 'rw');
3 sendfile(sockfd, fdin, NULL, PAGE_SIZE);
```

Listing 2. System calls executed to trigger null dereference in kernel space

Consider the (simplified for illustration) example app in Listing 1. The app obtains a handle to the binder device (Line 2), which is used to interact with the UI subsystem. Then, it opens a socket and connects to the banking server (Lines 4, 5) using a preloaded certificate (line 3) in the code and then waits for an input event by executing an `ioctl` IPC on the binder device (Line 8). When an input is delivered by the input subsystem, the `ioctl` returns with the input data. The app checks whether the data is from a password box, and then it performs an encryption operation using a userspace library function (Line 13-15). Finally, the app initiates a handshake with the banking server using the encrypted packet (Line 16).

On Anception, Line 2 is executed on the host as per the second principle. Lines 4 and 5 handle network communication and are serviced by the CVM. Line 8 executes on the host since the handle came from the host. As per the third principle, encryption happens in isolated virtual pages on the host (user-level library code). Finally, the send operation is serviced by the CVM (Line 16). The file handles are closed on the CVM (Lines 20-21).

Let us consider how Anception executes a malicious app. Listing 2 lists the set of system calls executed by CVE-2009-2692. This exploit works by invoking a null page dereference (line 3) from kernel space. Under Anception, we execute all 3 system calls in the CVM. The exploit specifies shellcode by asking the ELF-loader to load some code at the null page. As all memory pages are managed on the host, when the null dereference is triggered *inside* the CVM, the shellcode is not available and the exploit manages to only crash the CVM. The host OS remains protected.

Some system calls like memory mapping `mmap2()`, `fork()` and `sigaction()` do not use abstract handles to identify their services because they modify process state within

the host kernel. We discuss these next.

### D. Redirection Logic

As highlighted in the previous subsection, we want system calls to be executed on the CVM under the context of the proxy (e.g., file I/O, network, most IPCs), but some system calls must be executed on the host (e.g., UI-related). The redirection logic, which is part of the Anception kernel modifications to Android on the host, makes that decision. Fortunately, the execution environment of an Android app is well-defined in terms of the file, network, IPC and memory operations allowed. Developers use a specific API to ensure a well-behaved app according to the best practices[5]. Below, we consider major classes of system calls and how the redirection logic handles them, so as to provide correct API semantics to the apps.

***File I/O.*** The Android file-system is partitioned into a read-only part containing system code and a read-write part guarded by UID-based permissions for apps to use. Each app has its own directory (`/data/data/package.name`) on the read-write part and no one else may access the contents except the app itself[6]. On Anception, following Principle 1, we load the shared libraries from the host's read-only file system. Accesses to the app's data directory, on the other hand, are redirected to the CVM. If there is initial data packaged with the app, during installation this is unpacked to the app data directory. Anception copies over this data to the CVM as part of the app enrollment procedure. At runtime, any new files created or existing files that are modified exist only in the CVM due to redirection. Accesses to devices, with few exceptions such as handling interactions with the Window Manager via *binder*, are directed to the CVM.

***UI operations.*** Android apps request UI operations through an IPC on the *WindowManager* that is a centralized entity for frame buffer management. The requests are identified by inspecting the IPCs issued by an app. Anception services all such operations on the host.

***Network I/O.*** If an app wishes to perform network I/O (including local sockets), all such operations are serviced by the CVM. A socket open request results in a handle on the CVM, which is passed back to the host. Operations on that handle (e.g., send) are marshaled and passed back to the CVM. That also implies that the CVM's external connectivity can be controlled from the host by firewall rules, if desired.

***IPC.*** Android provides a custom capability-based binder IPC mechanism that simulates a synchronous procedure call across processes. Anception transparently bridges IPCs originating from the host destined for the CVM. Apps also use binder IPC to talk to other apps. We allow such IPCs to proceed on the host. Traditional IPC mechanisms such as unix domain sockets are supported similar to Network I/O.

***Memory-mapped files.*** Anception executes the memory mapping in the proxy's address space and temporarily pins those pages (after forcing read faults to ensure the data was demand paged). In the meantime, we perform a null mapping on the host by extending the *brk* of the app by the same amount of pages the memory mapping takes. We then copy page data from the proxy to the app via efficient remapping of pages. Write-back is used when data has to be synchronized with the CVM, for example, the `msync` operation. In this way, we avoid transferring every page fault back to the CVM.

***Fork/Clone and exec.*** Fork/Clone is replicated on the proxy as well since Anception maintains a one-to-one correspondence between host processes and proxy processes. When the fork/clone executes on the host, the child is assigned to the CVM too. An app cannot escape the CVM through fork/clone/exec calls.

Nothing special needs to be done to the proxy on an *exec* system call. The proxy continues to store the resource handles. The host process executes the new code. Code that is a system binary is simply executed on the host since the host's version is identical to the guest's. Code that is user-generated is first copied out from the guest to a special execution cache on the host that is not accessible to the untrusted app, and then executed from the execution cache. The reason for this is that we don't want the app to trick the system into copying an executable to a restricted location. Expressing the policy this way is much cleaner.

***System Management.*** Dangerous calls like `insmod`, `rmmod`, `shutdown`, `ptrace` [10] and others relating to whole system management are denied to applications because no user downloaded app should ever invoke these. Android security model denies them as well.

## IV. Implementation

Anception prototype runs on a Samsung Galaxy Tab with Android 4.2 and Linux Kernel 3.4. We added two new kernel modules, one each for the host and the guest. The lguest [40] hypervisor provides the virtualization technology, although other hypervisors can be used. The detailed architecture is shown in Figure 4. The CVM is a headless Android instance and it executes app proxies. Anception sits at the host kernel's system call interface and is implemented in approximately 5200 lines of code.

***1) Host-Guest Communication.*** Anception marshals system call data (including pointers) into a host kernel buffer. The marshaled data is copied over to a set of pages (Figure 4) that are remapped (using the `kmap` function) from the guest kernel space. Note that the guest, being unprivileged, cannot map memory outside the assigned region. The guest uses a hypercall mechanism to signal the host. The host injects interrupts into the guest kernel to signal the guest. Combining these two techniques, Anception implements a controlled communication channel between the CVM and the host. Our previous prototypes investigated other forms of communication such as sockets [16] and virtio [41] but they exhibited high overhead due to unnecessary data copy operations.

***2) Anception System Call Interception Method (ASIM).*** System call interception is used for a variety of purposes [23], [21]. Anception uses the technique to capture calls and forward them to the CVM. We investigated existing methods of interception such as ptrace, ftrace, dtrace and kprobes. Anception's first prototype used UML and ptrace but the overhead was grievous (upwards of 60x). kprobes is not ideal for our use-case because we are only interested in specific

[5] Available at http://developer.android.com/guide/index.html

[6] sharing can occur if apps share a UID

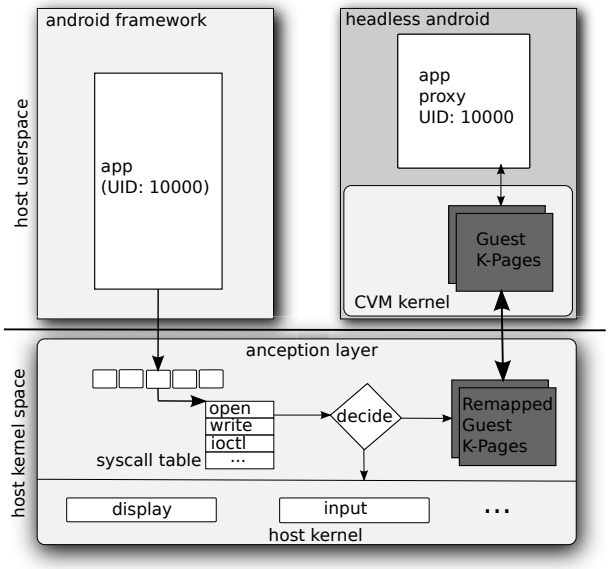


Fig. 4. Detailed Anception Architecture: Guest kernel pages are remapped into host kernel space for quick marshaling

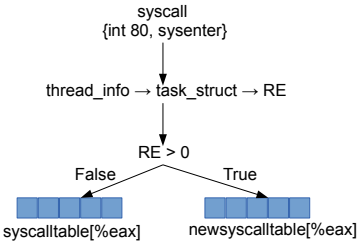


Fig. 5. Anception Syscall Interception Method: Uses a virtualization byte (Redirection Entry)

processes system calls and not the whole system. Instead, Anception introduces a one byte field in the `task_struct` of a process. We patch the system call handler (Figure 5) to inspect this field, known as `RE`, or the redirection entry. If its value is non-zero, we index an alternate system call table containing stubs that point to the CVM. The stubs perform marshaling, pointer translation and make use of the page-sharing scheme described above. Using a single byte, we can virtualize the *entire* system call table of a process with negligible overhead (Section VI-A).

***3) Efficient Call Execution.*** When a call is received by the guest kernel, we need to schedule it on the correct proxy for execution. Naively, one would notify a proxy and transfer the call data from guest kernel to the proxy userspace. Then the call would be issued, for a total of 4 context switches. Instead, Anception forces the proxy to be in an *interruptible sleeping wait* in guest kernel space. So, when a call comes, in, we post the call data (after rewriting pointers if necessary) to the proxy, that is waiting already in kernel space. It executes the call from its context, and returns the result, all in kernel space, thus saving 4 context switches.

***4) Headless Android.*** We built a headless Android version that runs in the CVM. The display devices were removed, along with the set of services related to the UI such as the *WindowManager* and *InputMethodManager*. This design reduces memory pressure on the host when running the CVM because no memory is reserved for framebuffers by the headless CVM instance.

***5) Device support.*** As a proof of concept, we implemented a virtual storage and network device within the guest. Additional device support can be added using existing virtualization techniques. An option is to build a virtual device node in the guest and a bridge-driver to the host similar to the virtual network device. Another option is to use Xen-style virtualization or device namespaces [5], [45] and directly assign the hardware device to the CVM. A third option is virtualization at the device file layer [4].

## V. Security Evaluation

We performed a qualitative and quantitative evaluation of the security guarantees provided by Anception. First, we analyzed previously reported Android vulnerabilities that do local privilege escalation to determine their effect under Anception. Second, we compute the number of system calls redirected to the CVM and quantify the reduction in attack surface of the host. Third, we quantify the amount of lines of code deprivileged by Anception's design. Lastly, we measure the size of Anception's trusted code base.

### A. Case Studies

We present examples of how Anception defeats different classes of privilege escalation attacks that aim to break app isolation.

***1) Kernel level bugs:*** These types of bugs provide kernel code and memory access to unprivileged processes. For instance, a buggy driver's device file could be left world-readable and world-writeable. Specifically, the *kernelchopper* exploit (CVE-2013-2596) [31] accessed kernel memory by mapping the `/dev/graphics/fb0` device node and then performed code injection into the mapped kernel memory. On Anception, the attempt to open the device node would be redirected to the CVM and then fail in the CVM because the CVM does not provide framebuffer access.

***2) Privileged userspace bugs.*** The userspace runs different privileged (uid 0) code and services that have been exploited by malware [22], [37]. An example is the *GingerBreak* exploit that targets the volume daemon (`vold`). The `vold` listens on a netlink socket whose permissions were configured incorrectly allowing anyone to send messages. A function in the code also has a overflow vulnerability that permits a local attacker to perform code injection. On Anception, the CVM provides an alternate execution environment. Therefore, communication of shellcode to the `vold` is redirected to the CVM environment and the exploit manages to perform a code injection on the CVM-version of `vold`. The host is safe at this point. A detailed walkthrough appears later in this section.

In the above attacks, it is possible that an exploit compromises the CVM. But, as discussed in Section III, stealing data from other well-designed applications running on the CVM is still difficult because the CVM does not have access to the memory pages of the apps or their UI events. Integrity,

however, is not guaranteed. For example, the CVM can return bad results from system calls [12].

If an exploit gains root access or control of the CVM, it is then restricted to the privilege of the CVM. It cannot map pages outside its assigned memory or even access a display device.

### B. Vulnerability Study

The CVE database for exploits involving Android contains a significant number of reported vulnerabilities that are root exploits or other serious attacks that attempted to gain control of the entire system. Of the 25 such vulnerabilities since 2010 that we located, three of them targeted privileged system services only to obtain root access and the remaining 22 targeted kernel interfaces to escalate their privileges. Out of 25, our detailed analysis of their attack vector showed that eight would have acquired root on the unprivileged CVM only (and thus not be able to read memory pages of other apps or monitor their UI interactions). Fifteen of these would have failed completely to compromise anything. One such example is the *Exploid* vulnerability that first creates a special file on the filesystem and then invokes the kernel hotplug functionality. With Anception, the file is created on the CVM, but the kernel hotplug executes on the host, thus the exploit fails to accomplish anything. Our findings were that only 2 out of 25 of the vulnerabilities would have resulted in root access on the host, but the exploits would have been easily detectable and thus preventable with simple checks at the system call interface on both Android and Anception (see Section V-A for further details). Due to space constraints, we document our analysis at http://goo.gl/brEbjW.

It is interesting to compare the above results with classical virtualization in which all apps run in an unprivileged guest. In that case, all of the above vulnerabilities could have ended up compromising the guest, but not the host OS. While this prevents host OS compromise, this would not have protected the virtual memory or UI interactions of other apps within the same guest. The key insight here is that *it is important to protect apps from each other with a smaller trusted base, not just the OS from the apps*.

### C. Example Exploit Walkthrough – Gingerbreak

Gingerbreak is a local privilege escalation based on a negative integer array access in `vold`, the volume manager on Android. It has been used by a number of malware applications [29] as a method to gain superuser privileges and nullify the Android security model. Below, we summarize the steps that a malicious app with Gingerbreak takes and the actions that occur when that app is running on Anception.

1) Gingerbreak starts out by making a copy of itself by reading `/proc/self/exe` and writing to the malicious app's private directory. With Anception, the write will be redirected to the app's private directory, which is an identically named and configured directory in the CVM. Thus, a copy of the exploit's executable will be made in the CVM.
2) The exploit then proceeds to its information gathering stage. The first step here is to find the `vold` daemon by its process identifier. It does this by opening `/proc/net/netlink`. With Anception, this `open` system call will be redirected to the CVM and the exploit will read the CVM's runtime information of the netlink environment. We have an identical environment in the container.
3) The exploit then searches `procfs` for `/system/bin/vold` and makes a note of the corresponding PID. Note that now, the Gingerbreak exploit executing on the host has obtained the PID of the `vold` executing inside the container.
4) The exploit proceeds to find the address of `system` and `strcmp` inside `/system/bin/libc.so`. As applications execute on the host, and any useful application will use *libc*, Anception simply allows opens and reads to execute on the host for such system code libraries.
5) The exploit, in the next stage of information gathering, attempts to find the Global Offset Table (GOT) start address of `vold`. The Gingerbreak exploit does this by opening the `vold` executable and using the ELF-32 API to parse it. The exploit then looks for the last piece of information to find the storage device that `vold` manages. System files are involved in the reading process and as per our rules, we let them go through on the host itself since these files are read-only.

Coming to the actual privilege escalation, Gingerbreak needs to find the negative index value to send to `vold` so as to achieve code execution. It uses a brute force approach by trying values in a range and then scanning the logcat crash logs for failed attempts. The exploit creates its own logcat log file (which is redirected and created in the app's container), kills logcat (which is mirrored in the app's container as well), and then restarts it by specifying its own file as the log file (also restarted in the app's container). Note that as per Anception's rules, when a fork/exec occurs, we simply let the fork happen on the host, but the new process is bound to the app's container; the sandbox is extended to the forked process. Since the new logcat is bound to app's container, it sends its output to a file that only exists in that container.

Once an index has been calculated, Gingerbreak forms a netlink message and uses socket calls to talk with the `vold` process. With Anception, Gingerbreak sends shell code with the negative index to `vold` inside the container. This causes `vold` to execute the exploit binary that was copied into the container. The exploit always checks on execution whether its uid is 0. Since `vold` started it the second time in the container, the root check succeeds and the exploit has succeeded *inside the container VM*. At this point, if the exploit tries to corrupt the virtual memory of an app, it will issue reads/writes on a target app's `/proc/pid/mem`. On Anception, the exploit ends up reading the memory of the proxy and not the real app.

### D. Attack Surface and TCB

***Host system call interface.*** Anception reduces the attack surface of the host kernel presented to apps by executing many of the system calls on the CVM. To quantify this, we analyzed 324 Linux system calls. Using our redirection logic, Anception redirects 70.7% (file, network, IPC) calls and executes 20.4% (process control, signal handlers) on the

host always. Anception executes part of the functionality of 6.5% of the system calls on both the host and the CVM (e.g., fork, mmap), as described in Section III-D. Finally, we block 2.1% (module insertion, shutdown) calls since they are outright malicious if executed by an app. This is more of an optimization to save round trips to the CVM, where, if redirected, these calls would have no effect.

***UI/Input system exploits.*** Anception trusts the UI system. In our case, the UI system's attack surface is the set of types of `ioctl` calls an application may invoke on the window managers. We have not found any attacks via these IPC calls to the UI/Input and lifecycle management services. We did find exploits involving direct access of the frame buffer [31]. As we have shown in Section V-A, such calls are redirected to the CVM.

***Anception runtime.*** A large percentage of Anception's code marshals and unmarshals data. Concretely, out of 5219 lines of C code (measured using sloccount), 2438 lines deal with marshaling and unmarshaling (46.7%). The remaining lines deals with bookkeeping such as maintaining process state and memory maps. Automated techniques can instead be used to generate the marshaling/unmarshaling code from an interface specification.

***Number of deprivileged lines of code.*** Anception deprivileges a significant portion of the kernel and framework services. We analyzed the Android framework and present a conservative lower bound on the number of lines deprivileged. All measurements were made on Android 4.2 and Linux kernel 3.4. Privileged framework services are 181,260 lines of code. Services related to UI, input and lifecycle management comprise 72,542 lines. Anception's current implementation deprivileges approximately 60% (108,718) of Android privileged service code. We also obtained rough estimates on the number of lines deprivileged within the linux kernel for the filesystem and the network. `fs/ext4` contains 26,451 lines of code, while `fs/` contains 725,466 lines of code. Similarly, `net/ipv4` contains 59,166 lines and `net/` contains 515,383 lines of code. Thus, Anception deprivileges approximately 1.2 million lines of kernel code related to the file system and network.

## VI. Performance Evaluation

We quantify performance using several popular benchmarks available from the Play Store. These benchmarks test Disk I/O, 2D/3D graphics, CPU and memory performance and are used by systems in the literature. We also execute microbenchmarks to quantify the overhead introduced by system call interception and host $\leftrightarrow$ guest context switching, which we term as *world switching*. All experiments are run on a Samsung Galaxy Tab 10.1 hosting Android 4.2 with 1GB of RAM, a 1.6GHz processor and 64MB physical memory assigned to the CVM. All readings are averaged over 5 runs of the benchmark unless stated otherwise. Each benchmark averages results internally as well. For example, our microbenchmarks allow sufficient time for the caches to warm up before making timing measurements.

**Active-set of apps.** The benchmarks reported below were run concurrently with the standard set of Android 4.2 apps that launched at boot. Based on the official Android source code (4.2), these apps are: home screen, launcher, contacts (and its provider process), photo gallery, dialer, MMS and settings. Vendors customize this list and add more apps. On our Samsung Galaxy tab, there were a total of 23 apps (including standard apps) in the active-set when our benchmarks were run. We did not kill the active-set since it closely resembles real world usage of the device were multiple apps are present in the executing state.

| syscall | Native | Anception |
|---|---|---|
| Null call – getpid | 0.76 $\mu s$ | 0.76 $\mu s$ |
| Filesystem – write (4096B) | 28.61 $\mu s$ | 384.45 $\mu s$ |
| Filesystem – read (4096B) | 6.51 $\mu s$ | 305.03 $\mu s$ |
| Binder IPC – ioctl (128B) | 12 ms | 31 ms |
| Binder IPC – ioctl (256B) | 12 ms | 31.3 ms |

TABLE I. ASIM LATENCY

### A. Microbenchmarks

We measured the overhead introduced by ASIM using the *getpid* syscall. For this purpose, getpid call executes on the host and does not involve any world switching. The first row of Table I shows that the ASIM is very efficient and introduces negligible overhead. We then measured the performance of read and write syscalls to quantify the overhead due to world switching. The benchmark writes (reads) 16 MB of data to (from) the internal storage of the device. The results are indicated in the Table I below. Apart from the world switching latency, part of the increase in latency is attributed to chunking behavior in the data transfer channel as we can only accommodate fixed sized buffers[7] for transfers between the host and guest. Our current configuration chunks data into 4096 byte packets.

A slowdown on a system call is experienced only when a call is serviced in the CVM; when an app is *not* making a system call, i.e., only running user-level application code, it runs at native speed. Furthermore, UI-related system calls, which all occur via ioctl() system call on Android, run at essentially native speed since they are not redirected because of security reasons. Not redirecting UI-related system calls also helps performance. Using ProfileDroid [44], we found that approximately 58.7% to 80.1% (average = 73.7) of system calls made by popular apps are *ioctl* calls. After performing an additional custom profiling of only *ioctl* calls, we found that 81.35% of such calls are UI-related and thus will run at native speed.

IPC and filesystem calls serviced in the CVM experience a slowdown, though the overheads appear acceptable. For example, an IPC call to get a GPS fix will return with an added latency of 19 *ms*. Regarding file I/O microbenchmarks, the additional latency appears to have minimal impact at the macro level, possibly because of extensive memory-buffering that occurs in filesystems. The macrobenchmarks below to evaluate SQLite database performance, which is used extensively in Android by many apps, validates this.

### B. Macrobenchmarks

We used AnTuTu [6] (v2.9.4) to evaluate Anception performance on app-oriented DB workloads (see Figure 6). Anception's score was only 3% lower than with native Android,

[7]We can increase buffer sizes based on profiling information

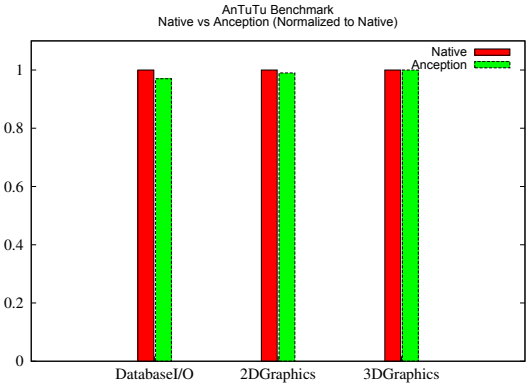


Fig. 6. Relative AnTuTu benchmark scores – Higher bar is better.

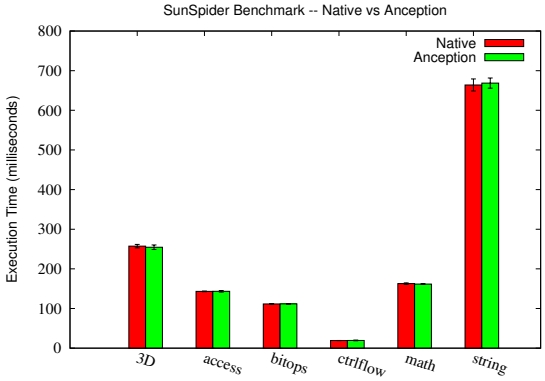


Fig. 7. SunSpider Benchmark illustrating Anception performance across several benchmarks.

suggesting that the I/O performance hit at the microbenchmark level is largely masked by memory buffering that occurs within the filesystem and within SQLite. Earlier studies of Android have shown that SQLite database I/O is typically the bulk of the filesystem I/O by apps on Android. So, we believe that deprivileging filesystem code by executing system calls on the guest is the correct design decision. If I/O latency were to matter in some context, one could choose to keep filesystem I/O on the host side (while still keeping rest of the code in the CVM deprivileged).

We also ran additional AnTuTu [6] macrobenchmarks as well as SunSpider [42] benchmarks, which test Anception's ability to run unmodified, graphics-rich applications. For the AnTuTu benchmark, Anception's overall score is 2.8% less than native Android. Detailed results on individual tests appear in Figure 6. Anception's performance was close to native on 2D and 3D tests (Figure 6). On the SunSpider benchmark Anception's performance was essentially indistinguishable from native Android (Figure 7).

We ran a sqlite benchmark that wrote 10,000 rows (each row is 26 bytes) of data within a transaction. Given that 90% of write requests on a smartphone are to a sqlite DB [28], and a further 64% of I/O operations less than 4KB in size [28], we feel this is a good characterization of normal smartphone file I/O workload. The time to execute the benchmark on Anception is 86.67 $\mu s$ (SD = 1.17) compared to 86.55 $\mu s$ (SD = 2.0) for native Android. Thus, Anception's performance is virtually indistinguishable from native.

### C. Memory Overhead

Anception's unique design enables us to execute headless Android within the CVM. We have found empirically that assigning 64MB to the CVM allows proper operation (typical Android devices have 1 – 4GB RAM). Note that the version of Android executing in the CVM is a stock version of Android minus the UI code. The active memory used is 25460 KB $\pm$ 524.54 KB out of 49228 KB available on average, i.e., almost 51% of assigned memory is available for use by proxy processes. A proxy process is much smaller than the actual process running on the host and thus Anception supports multiple proxies in the CVM comfortably.

## VII. Discussion

High assurance apps can store cryptographic keys in their read-only code that is protected by Anception from other apps and the CVM. However, Anception's architecture makes it possible to transparently provide encryption to any app. All that is required is to provide each app a transparent cryptographic file-system in the CVM. A per-app key for that filesystem would reside in the host. An implementation is to use *encfs* [18], a user-level encrypting file system based on FUSE [19]. The app's data directory would point to an encrypted file system. The reads/writes from the FUSE layer to the host OS are redirected as before. The net result is that the CVM only sees read/write calls that contain encrypted data, but does not have access to the keys (which reside on the host).

Anception does not rule out *Iago attacks* [12] on an application, say the banking app, from the compromised CVM. Iago attacks are carried out by returning bad system call results to an app and attempting to corrupt it at run-time and make it leak unencrypted sensitive data. These attacks are difficult to craft. The easiest attack vector for this kind of attack would be to modify the results from file read calls. Using an encryption wrapper for file system calls, as described earlier in this section, makes such attacks more difficult.

## VIII. Related Work

***OS Virtualization.*** Current research is bringing classical virtualization to smartphone hardware [27], [15], [17], [8]. Cells [5] is a lightweight (namespaces) approach to virtualization that uses a shared kernel for VMs and is vulnerable to kernel-level exploits. AirBag [45] builds upon Cells and uses a single Play Store instance to provide a seamless user experience. However, both systems run complete Android stacks in each VM, The key difference with Anception is that our design deprivileges a large chunk of the trusted base and delegates it to a container. As we run apps on the host, but delegate calls to services in the container, we are able to run a headless OS in the container, thus reducing memory consumption by design.

Overshadow [13], InkTag [25] and SP$^3$ [47] are systems aimed at providing virtual memory and disk data confidentiality when the host OS is malicious through the use of

encryption. Anception provides similar guarantees through memory isolation and split execution. Anception takes a finer grained view of the OS and provides confidentiality of UI interactions.

Library OSes such as DrawBridge for Windows [39] provide userspace linkable OSes that execute an app in an isolated environment with a separate kernel. An open research problem with such systems is providing high performance graphics. Android, natively provides several mechanisms that maintain maintain smooth UI performance. Anception's design leverages this existing code base to provide high performance graphics.

ExpressOS [35] is focused on running apps on a small verified kernel. However, ExpressOS requires that apps fully trust the data received from the UI stack that is not verified. Additionally, all privileged system services (UI/Input stack included) execute in the same VM, thus exposing sensitive user interactions to malware.

The basic mechanism of servicing system calls in different kernels has been used in Pods [32], VirtuOS [38], and ProxOS [43] for other purposes than in this paper, e.g., to improve server reliability, isolating services, and improving tolerance against driver bugs. Anception incorporates lessons on system call interposition tools, and pitfalls in doing that, as described in [21], [20].

Flicker [36] reduces the trusted base of apps drastically using AMD-specific processor support and requires developers to construct Pieces of Application Logic (PALs) that are secured using hardware primitives. While flicker does promote better modularity in apps, it requires that apps be modified to be protected. Anception works with unmodified apps.

***Microkernels for smartphones.*** L4Android is a microkernel based OS framework for Android built on the L4 microkernel [33]. While microkernels reduce the trusted base of the core kernel, there are other privileged userspace processes running that could be leveraged to attack trusted apps. Anception provides an architecture to further reduce this trusted base using the lightweight Anception container where even privileged system processes can be sandboxed away from trusted apps.

***App Sandboxing.*** Providing isolation on Android is currently focused on policy based approaches. TrustDroid [11] creates trust domains through framework modifications and IPC monitoring, AppFence [26] presents fake data to untrusted apps and Aurasium [46] performs bytecode rewriting to embed isolation policies in app code. Janus [23] was an early tool implementing policy based on system call interception, and seccomp [1] is a more recent version based on similar principles. These systems are vulnerable to privileged userspace bugs and kernel exploits. Additionally, they do not enable a framework for the design of secure apps. PREC [24] is targeted at foiling exploits by exponentially slowing down the execution of system calls from suspicious contexts and relies on building a profile of normal and abnormal behavior but does not protect against kernel level exploits and does not reduce the amount of system-code the app must trust.

***File System Isolation***. Android recently incorporated a multiuser feature that helps in setting up multiple user accounts and sharing of a single device. Each user is assigned a unique user ID and corresponding directory on the filesystem (`/data/users/ID`). When the device switches to a user, symbolic links are set up from an app's directory (`/data/data/APP.PKG`) to the private user directory. However, this design is not aimed at isolating malware that use privilege escalation attacks and does not provide memory isolation in the event of an OS compromise.

## IX. Conclusion

Modern operating systems such as Android provide malware a large attack surface running into hundreds of thousands of lines of code, consisting of both privileged services and the operating system kernel. This paper presents a system architecture called Anception for deprivileging both portions of the kernel services as well as several system services for Android with the goal of protecting UI input and virtual memory of high assurance apps from malware. To achieve that, we kept security-relevant portions of apps (in particular, UI-related services, virtual memory, code, and security-relevant keys) on the host and segregate rest of the app's functionality as well as many system services to a guest virtual machine container. Analysis showed that the Anception architecture moves significant chunks of privileged code to an unprivileged container and it would have blocked 23 out of 25 of the previously reported privilege-escalation vulnerabilities on Android. The penalty for deprivileging code with Anception is modest. While, as expected, I/O and cross-container IPCs take a performance hit on microbenchmarks, on macrobenchmarks and on real applications, the impact is minimal.

## X. Acknowledgments

We thank the reviewers for their thoughtful feedback. This material is based upon work supported by the National Science Foundation under Grant Numbers 093629 and 1318722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Google chrome security team. seccomp-sandbox. accessed 18 july 2013. http://code.google.com/p/seccompsandbox/.

[2] Kaspersky Labs. Targeted Trojan Attack stealing personal information. http://www.securelist.com/en/blog/208194186/Android_Trojan_Found_in_Targeted_Attack.

[3] McAfee Labs. Phishing Attack replaces Banking app with malware. http://blogs.mcafee.com/mcafee-labs/phishing-attack-replaces-android-banking-apps-with-malware.

[4] Amiri Sani, A., Boos, K., Yun, M. H., and Zhong, L. Rio: A system solution for sharing i/o between mobile systems. In *Proc. of the 12th Annual Int. Conf. on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 259–272.

[5] Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. Cells: a virtual mobile smartphone architecture. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 173–187.

[6] Antutu Benchmark for android. Accessed 18 July 2013. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark.

[7] Artenstein, N., and Revivo, I. Man-in-the-Binder: He who controls IPC, controls the Droid. In *Europe BlackHat Conf.* (Amsterdam, The Netherlands, 2014).

[8] BARR, K., BUNGALE, P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., AND ZOPPIS, B. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 124–135.

[9] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 267–283.

[10] BERNASCHI, M., GABRIELLI, E., AND MANCINI, L. V. Remus: a security-enhanced operating system. *ACM Trans. Inf. Syst. Secur. 5*, 1 (Feb. 2002), 36–61.

[11] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and lightweight domain isolation on Android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 51–62.

[12] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proc. of the 18th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 253–264.

[13] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 2–13.

[14] Linux Kernel vulnerabilities: CVE Database. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html.

[15] DALL, C., AND NIEH, J. KVM for ARM. In *Proc. of the 12th Annual Linux Symp.* (2010), OLS'10.

[16] DIKE, J. User-mode Linux. In *ALS '01: Proc. of the 5th Annual Linux Showcase & Conf.* (Berkeley, CA, USA, 2001), USENIX Association, p. 2.

[17] Dual Android using Xen. Efficient GPU virtualization. Samsung R&D UK. http://ftp.osuosl.org/pub/fosdem//2014/UD2120_Chavanne/Saturday/DualAndroid_on_Nexus_10_using_XEN.webm.

[18] EncFS. http://www.arg0.net/encfs.

[19] Filesystem in Userspace. http://fuse.sourceforge.net/.

[20] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symp.* (2003), pp. 163–176.

[21] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *IN NDSS* (2003).

[22] CVE-2011-1823. Gingerbreak. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823.

[23] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proc. of the 6th Conf. on USENIX Security Symp., Focusing on Applications of Cryptography - Volume 6* (Berkeley, CA, USA, 1996), SSYM'96, USENIX Association, pp. 1–1.

[24] HO, T.-H., DEAN, D., GU, X., AND ENCK, W. PREC: Practical Root Exploit Containment for Android Devices. In *4th ACM Conf. on Data and Application Security and Privacy* (San Antonio, TX, March 2014).

[25] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: secure applications on an untrusted operating system. In *Proc. of the 18th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 265–278.

[26] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proc. of the 18th ACM Conf. on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[27] HWANG, J. Y., SUH, S. B., HEO, S. K., PARK, C. J., RYU, J. M., PARK, S. Y., AND KIM, C. R. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. pp. 257–261.

[28] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/o stack optimization for smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 309–320.

[29] JIANG, X. GingerMaster malware on Android. http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/.

[30] Kaspersky Security Bulletin for 2013. https://www.securelist.com/en/analysis/204792318/Kaspersky_Security_Bulletin_2013_Overall_statistics_for_2013.

[31] Kernelchopper/Motochopper exploit. http://forum.xda-developers.com/showthread.php?t=2255491.

[32] LAADAN, O., AND NIEH, J. Operating system virtualization: practice and experience. In *Proc. of the 3rd Annual Haifa Experimental Systems Conf.* (New York, NY, USA, 2010), SYSTOR '10, ACM, pp. 17:1–17:12.

[33] LANGE, M., LIEBERGELD, S., LACKORZYNSKI, A., WARG, A., AND PETER, M. L4android: A generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 39–50.

[34] levitator. Jon Larimer and Jon Oberheide. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1352.

[35] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in ExpressOS. In *Proc. of the 18th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 293–304.

[36] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for tcb minimization. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 315–328.

[37] mempdroid exploit. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0056.

[38] NIKOLAEV, R., AND BACK, G. VirtuOS: an operating system with kernel virtualization. In *Proc. of the 24th ACM Symp. on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 116–132.

[39] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proc. of the 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 291–304.

[40] RUSSEL, R. lguest: Implementing the little Linux hypervisor. In *OLS '07: Proc. of the Linux Symp.* (June 2007), vol. 2, pp. 173–178.

[41] RUSSELL, R. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 95–103.

[42] SunSpider Benchmark for android. Accessed 18 July 2013. https://www.webkit.org/perf/sunspider/sunspider.html.

[43] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: making trust between applications and operating systems configurable. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 279–292.

[44] WEI, X., GOMEZ, L., NEAMTIU, I., AND FALOUTSOS, M. ProfileDroid: multi-layer profiling of Android applications. In *Proc. of the 18th Annual Int. Conf. on Mobile Computing and Networking* (New York, NY, USA, 2012), Mobicom '12, ACM, pp. 137–148.

[45] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., AND JIANG, X. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proc. of the Network and Distributed System Security Symp. (NDSS)* (San Diego, CA, February 2014).

[46] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical Policy Enforcement for Android Applications. In *Proc. of the 21st USENIX Conf. on Security Symp.* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 27–27.

[47] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. of the 4th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 71–80.