Graphical User Interface for Virtualized Mobile Handsets

Janis Danisevskis, Michael Peter, Jan Nordholz, Matthias Petschick and Julian Vetter
Security in Telecommunications
Technische Universität Berlin
Email: {janis,peter,jnordholz,matthias,julian}@sec.t-labs.tu-berlin.de

***Abstract*—Type-1 hypervisors have been suggested as a solution to *bring your own device* (BYOD) for their ability to enforce strong isolation. However, the lack of graphics acceleration support, crucial to good user experience, in mobile virtual machines (VMs) has limited the appeal of such solutions. We present a system architecture for providing access to graphics acceleration hardware to mobile VMs as well as a trusted and identifiable input and output path between the user and a VM. We built a prototype based on a small type-1 hypervisor and determined the GPU virtualization penalty on the performance to be only 5 %. A small impact on the trusted computing base (TCB) makes our architecture a viable solution even for high security demands.**

I. Introduction

Bring your own device (BYOD) poses the challenge of consolidating two seemingly contradictory philosophies on a single mobile handset. Private users expect to be in full control of their mobile device, benefiting from the extensibility that makes their smartphone smart. On the other hand, enterprises that issue smartphones to their employees are interested in mitigating the threat of a malware infection [38], which comes with the extensibility and may eventually yield information leakage. To that end, they seek to enforce very strict policies, which may be perceived as overly restrictive by the device owner.

Virtualization has been suggested as a solution to this quandary [29], [13], [25]. Its formidable isolation properties make for a suitable platform for providing multiple mutually distrusting environments that can enforce individual policies. But more than in the desktop and server realm, the user experience of mobile devices hinges on their graphics output performance. The lack of virtualization solutions for graphics processing units (GPU) on mobile handsets, however, is an obstacle to the wide adoption of virtualization for BYOD.

We propose an architecture that allows for secure screen, input, and GPU sharing. Our approach allows to convey to the user the identity of the virtual machine (VM) in the foreground in a manner resilient to forgery. To show the viability of the approach, we implemented this architecture on the basis of L4Android [29]. We evaluated our prototype using off-the-shelf high-level benchmarks as well as specially tailored microbenchmarks. To quantify the GPU virtualization overhead of our prototype more precisely, we built a setup where a single VM was granted direct, non-virtualized access to the GPU. We measured a frame rate penalty of 5 % against this pass-through setup with a benchmark that drove the device into saturation. The frame rate penalty of the pass-through setup was 11 % versus a non virtualized environment. For typical graphics processing demands as required by the graphical user interface or mobile games, our prototype matches the native frame rate of 60 Hz with ease and little impact on the system load.

Our contributions are:

- We propose an architecture for efficient and secure switching of user input and output for virtualized environments on mobile handsets. Our approach counters spoofing and eavesdropping attacks on user input and output by providing a trusted and identifiable path between each virtual machine and the user.
- To the best of our knowledge, we present the first academic work on providing secure access to GPU computing resources to virtualized environments on mobile handsets.
- We implemented a prototype of the proposed architecture.
- We evaluated the virtualization overhead of the prototypical implementation of our proposed architecture.

The outline of this paper is as follows: We present the prerequisites, challenges and requirements for our proposed architecture in Section II. In Section III and IV, we present the architectural design of the secure I/O management and GPU-virtualization scheme, respectively, followed by a description of our prototypical implementation (Section V). In Section VI and VII we discuss the TCB impact of our design and present the performance evaluation, both based on our prototypical implementation. We put our work into context in Section VIII before we conclude with Section IX.

II. GUI for virtualized Handsets

Graphical user interfaces (GUI) for virtualized environments on mobile handsets differ from GUIs in the desktop domain. Handsets are constrained in memory, computational capabilities, energy consumption and screen size. Also, the concept of a windowing system has proven impractical on mobile devices. We therefore focus on an efficient scheme that allows one virtualized environment to take possession of the screen at a time. And we make great effort to keep performance impact to a minimum while making no compromises on isolation and thus security.

We now discuss the challenges in framebuffer and input handling as well as in providing efficient access to GPU

computing resources to the virtualized environments. We then pose the minimal requirements on the hardware and software that we determined as needed to build our architecture upon. But before we do so, we refine the BYOD use-case from the introduction and introduce an attack model.

A. BYOD Exemplary Use-Case

A mobile handset—smartphone or tablet—shall be used by a user both as personal and business phone. It therefore provides two environments, the business and the private environment. In the business environment, the user can access internal communication or documents that may be confidential. To protect corporate assets, a policy enforced in the corporate compartment may encompass the following provisions:

1) Restricted network access, e.g. all data communication must be routed through the corporate IT-infrastructure via a VPN.
2) Restricted extensibility, e.g. the user can install no or only a restricted set of applications.
3) No information flow to or from other environments on the same device.

The second environment is for personal use by the user and should be free of any restrictions. Therefore the rules 1 and 2 do not apply here. Policy 3 is redundant if the only other environment is the business environment, but it should be up to the user otherwise and may well be desirable for privacy reasons.

B. Attack Model

For the use case laid out in the previous section, we discuss our attack scenario. We assume that one of the environments has been infiltrated by an attacker. Because of its open and extensible nature, we deem the private environment more prone to being infiltrated, possibly by tricking the user into installing a Trojan horse. In any case, the attack model presumes one environment being under the control of an attacker, who is trying to subvert the policy of the other environment.

We address three categories of attacks that the attacker may mount on the non compromised environment, namely spoofing, eavesdropping and evasion of isolation.

Spoofing
: By spoofing we refer to two very similar attacks with a subtle distinction. The first is a social engineering technique also known as phishing. The attacker presents a dialog on the screen that is familiar to the user, thereby tricking the user into giving away confidential information such as login credentials. Felt and Wagner [20] investigate this technique on mobile handsets, but the problem extends also to virtualized mobile environments. With the second attack, the attacker also shows the victim familiar screen content but then uses the thus gained trust for feeding it false information.

Eavesdropping
: The attacker could try to eavesdrop on the user input and/or output of the other environment. Malware that eavesdrops on user input is also known as a keylogger. A prominent example on desktop systems —although not intended as malware—is the program *xeyes*. It shows two eyes following the mouse pointer traveling across the screen. It works even if the program is not in focus, which means it receives user input events (here mouse motion events) that were not intended for it. Eavesdropping on the output [30] can be worthwhile for an attacker indeed as every potentially confidential piece of information that is viewed by the user eventually passes trough the framebuffer as an output image. On mobile handsets with on-screen keyboards in particular the output response to user input allows logging of keystrokes [22].

Isolation evasion
: Evading the isolation of the virtualized environment by gaining code execution outside the protection domain is the ultimate goal for an attacker. When this happens, all isolation guarantees are null and void. Kortchinsky [26] presents an exploit of a weakness of a virtual VGA-adapter that results in code execution outside the attackers virtual machine. Danisevskis et al. [16] and Clark [10], [15] independently exploited two different bugs in two different mobile GPU drivers working by the same principle. Both manipulate the GPU's memory management and use the GPU as an agent for arbitrary memory access. In both cases, this leads to privilege escalation and code execution in the privileged domain. While their work is not related to virtualization, a similar flaw in a virtualizing GPU driver would have devastating effects on the isolation guarantees of the respective virtualization scheme.

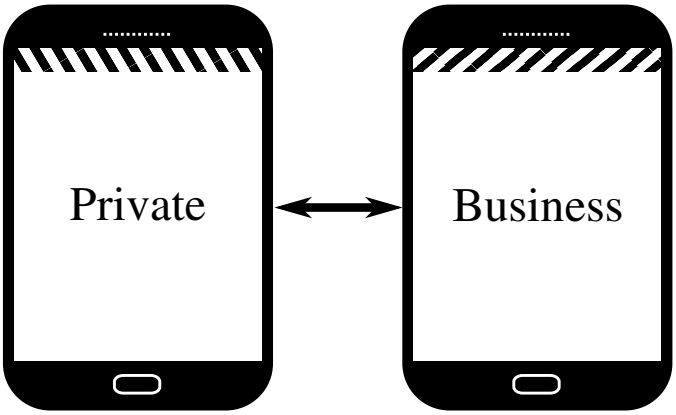

Fig. 1. The output of two different virtual machines on the same device. A label at the top of the screen, indicating the origin of the rendition in the client area below, is provided by the system and cannot be changed by the client VM.

C. Challenges in Input and Output Handling

A framebuffer is a region of memory that holds a bitmap of coded color values. A display controller scans such a

framebuffer periodically and sends the color values to the screen. Each VM has its own framebuffer, to which it renders its output, and it is the responsibility of the virtualization layer to arrange for the display controller to receive the output of the currently visible VM.

If multiple VMs share a screen, it may not be apparent to the user which one is currently visible. To counter spoofing attacks, we need a way to inform the user about which virtual environment she is interacting with. This could be done via special hardware such as a multicolor light-emitting diode (LED) or a label on the screen [28]. Figure 1 shows two possible states of the screen with two distinct labels at the top of the screen. The label could be color-coded, a string of characters, or a geometric pattern, but most importantly, it must never be forgeable or manipulatable by any of the virtual environments.

Preventing eavesdropping is more a technical rather than a user interface design issue. The system itself must be designed such that input events are only ever reported to the virtual environment indicated by the label rather than to anyone who requests them. The same goes for the output path. Crosstalk on the output path that could result from, e.g., sharing a common framebuffer must be eliminated at all cost.

It is no coincidence that what we described above reminds of the principals of *identifiability* and *trusted path* as defined by Yee [37] and which is summarized by the author as: "The system protects me from being fooled."

We see little potential for isolation evasion in the handling of framebuffer and input events because both related hardware components display controller and touchscreen do not typically use direct memory access (DMA) for writing memory. It is conceivable however that an attacker who gains control over the display controller redirects the scanout region such that memory that contains secret key material is sent to the screen where it can be recovered with a camera.

D. Challenges in GPU Virtualization

GPU virtualization is not classical device virtualization. From a virtualization point of view, GPUs behave much more like CPUs. They are highly programmable devices that fetch their programs from memory, which even in non-virtualized environments come from mutually distrusting entities. And they can read from and write to any region of the physical memory they can access, which, just like on a CPU, is governed by a memory management unit (MMU). Therefore these access restrictions must be administered with great care to prevent an attacker from evading a protection domain by means of the GPU.

When virtualizing CPUs without support for memory management virtualization (e.g. nested paging), one needs to resort to shadow paging [11]. This leads to considerable overhead because CPU page tables are highly dynamic and are usually populated lazily. As it turned out, the execution model of the GPU, which is job oriented without support for preemption, is better suited to virtualization.

GPU drivers assemble and lay out the jobs in memory and populate the corresponding context accordingly prior to starting the job on the GPU. Unlike process page tables, GPU page tables are not populated lazily. This static behavior is well suited for applying shadow paging to GPUs without the performance overhead that comes with handling dynamic CPU address spaces.

Now that we have motivated and presented the requirements for our architectural design, we discuss the minimal requirements, both for hardware and software, we impose on the target platform.

E. Hardware Model

We base our architecture on the following exemplary hardware model. We assume there are three main components. One is the CPU, or multiple CPUs for that matter. Furthermore there are a GPU and a display controller. All three have access to shared physical memory. Access to this memory by the CPUs and the GPU, at least, is mediated by individual memory management units (MMUs). The display controller must support at least two scanout regions or overlays that can be configured to service different parts of the screen.

F. System Model

An architecture such as the one we are proposing cannot stand without a surrounding infrastructure. Therefore we state the following minimal system requirements for our architecture. It goes without saying that being a virtualization system, the runtime environment shall support virtual machines (VM). The system must have a trusted boot procedure that gives the VMs integrity and the notion of a persistent identity. It shall also provide a means for inter-VM communication with low latency. The bandwidth of these channels does not need to be high. In fact, a low latency notification mechanism, such as virtual interrupts, suffices if shared memory regions across VMs are available.

Each VM is assigned a portion of physical memory, called *guest physical* memory. The virtualization layer maps each guest's physical memory to a region of host physical memory assigned to that guest. Depending on the platform's capabilities, the mapping is either performed in hardware (nested page tables) or software (shadow page tables).

III. Secure input and output

Smartphones typically have a touch screen and a few mechanical buttons. The screen is driven by a display controller. It scans a given region of memory, interprets the content as a map of color values and feeds them to the screen. We set out to building a dependable mechanism to share these resources between two or more VMs, thereby labeling the output at all times as depicted in Figure 1.

Our architecture comprises five components as depicted in Figure 3. Two device drivers, one driving the display controller and one driving the input devices (touchscreen and mechanical buttons). Next there are two switches for policy enforcement for graphical output and input events, respectively. Finally,

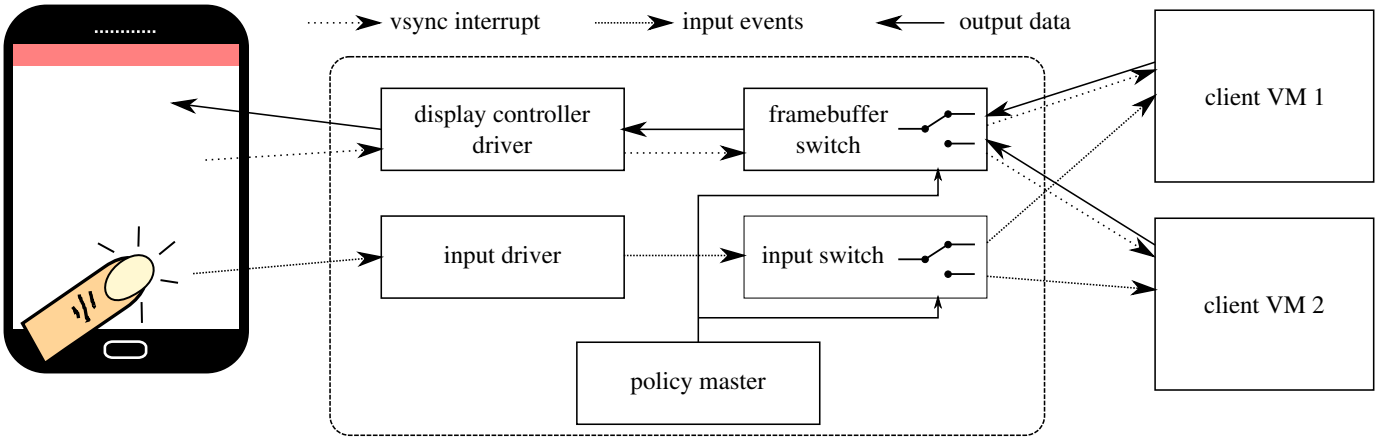

Fig. 3. The I/O switching architecture with all five components (middle) in the context of the (touch)screen as (input)output device (left) and two client VMs (right).

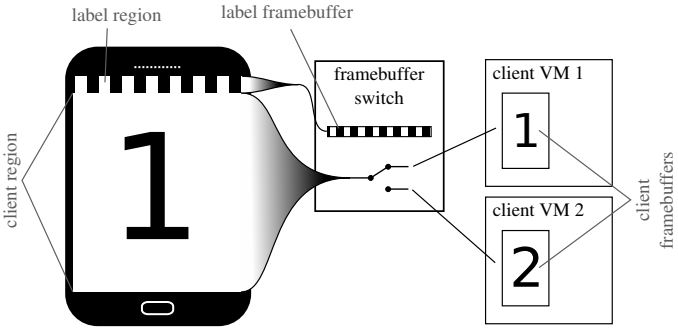

Fig. 2. A smartphone's screen partitioned into label and client region (left), the framebuffer switch (middle) and two client VMs (right).

there is a policy master. The switching components rely on the service of their respective drivers and, in turn, provide similar services to their clients, the virtual machines. Additionally, the switching components provide an interface to allow setting of the policy decision. It is important to note that the switchers do not make policy decisions; rather they guarantee their enforcement.

We now describe each of the components in terms of the services they need and provide.

A. Framebuffer Management

The display controller driver provides an abstraction of the screen. Thereby it partitions the screen into two logical regions, the label region and the client region, as shown in Figure 2. Using the display controller's support for multiple scan-out regions or overlays, each of the regions may be backed by different framebuffers. The driver offers a service to attach arbitrary buffers to the logical screen regions and to retrieve information about the region's geometry and pixel layout.

The framebuffer switch relies on the service of the display controller driver. It attaches a label framebuffer of its own to the label region (Figure 2). And it draws a label into this framebuffer, indicating the current policy decision.

To enforce policy, the framebuffer switch maintains a session associated with each client. One of these sessions may be active in accordance with the policy decision it receives via its policy interface. Clients use the service of the framebuffer switch to set their active framebuffer. Switching between multiple buffers is easily possible, allowing for the straightforward implementation of double buffering. To announce a buffer, clients communicate the guest physical address range of it to the framebuffer switch, which knows how to map it to a host physical address for a given client. The framebuffer switch can then by means of the display controller driver attach the active client's announced framebuffer to the client region.

The display controller's capability of laying out a limited set of buffers on the screen is exploited by Android for offloading compositing work from the GPU[1]. By using one of these overlays for a secure label, we reduce the number of those that the client can use by one. As Android cannot expect the same number of overlays to be supported on all hardware platforms, it can flexibly compensate for the loss by using the GPU. Possibly remaining overlays can be made available with our approach. We did not implement this feature in our prototype, though.

The display controller generates a *vsync* interrupt on the start of the vertical sync gap. The display controller driver, which owns the device and receives that interrupt, forwards it to the framebuffer switch which, in turn, passes it on to the active client. Making this information available to the client is important because some operating systems, such as Android, use it to coordinate its rendering activities.

We have accomplished two things now. First, due to the private nature of the client framebuffers, we have a trusted path from the VM to the screen. Second, due to the labeling mechanism, the user can identify the origin of the content. What is more, neither the driver nor the switcher needs access to the content of client framebuffer itself. There is no

[1]See: Hardware Composer at https://source.android.com/devices/graphics/index.html#hardware_composer

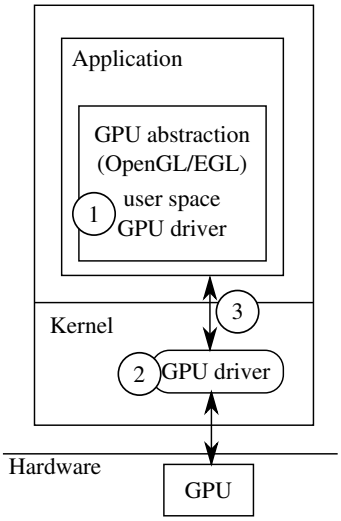

Fig. 4. A mobile GPU driver stack as found in a typical Android based handset.

expensive copying of bulk data, e.g. image data, necessary. Combining the label with the client output is done by the display controller without assistance from the CPU.

B. Input Management

Input driver and input switcher are designed analogous to the display controller driver and the framebuffer switch. The input driver drives the touchscreen and mechanical buttons of the device. In turn the switcher relies on the input driver's service to receive input events. It enforces a given policy by passing input events on to the client that is active according to the current policy decision.

The input switcher can raise a secure attention event when it detects a secure attention sequence [35] or a gesture [28] issued by the user. This event is never passed on to a client but rather to a policy decision maker. This policy decision maker could now switch input and output to the next VM or, instead, direct input and output to itself and present the user with a dialog. While our prototype has a policy decision maker, we deliberately keep it out of the scope of our architectural design.

We now provided a trusted path from the input devices to the VM. Identifiability for the input path relies on whether or not input switch and framebuffer switch received the same policy decision announcement. We therefore introduce the policy master that has the sole purpose of keeping both switchers in sync.

IV. Secure GPU multiplexing

In Section III, we presented our secure framebuffer switching scheme. The client VMs still need an efficient way to fill their framebuffers with content. Therefore they are given access to the computational resources of the GPU. To do this, we introduce a new component called the GPU server or *resource governor* (GPU-RG). Before we explain the workings of our proposed architecture, we present a typical driver stack of a mobile GPU.

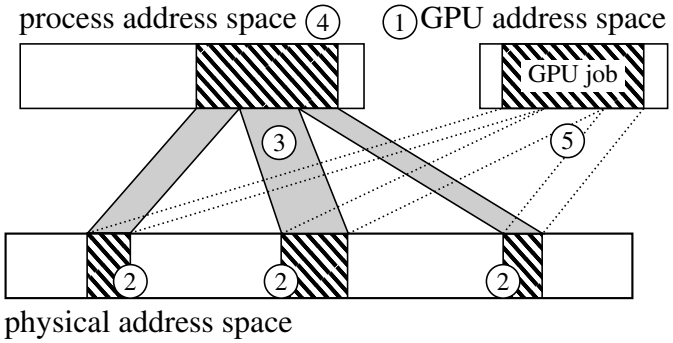

Fig. 5. GPU address space population with physical memory buffers and their relation to the process address space.

A. A Mobile GPU's Driver Stack

A typical mobile GPU driver stack comprises a user part (see Figure 4 ①) and a kernel part (②). The user part of the driver translates high-level API-calls—such as OpenGL—into the instruction set of the GPU. It uses the kernel/user interface (③) of the kernel part to populate a GPU context with a job for the GPU to run and, eventually, to start the execution of the job.

The context population is depicted in Figure 5. Every application requesting to use the GPU gets its own GPU address space (①) just as it gets its own process address space (④) on the CPU. The GPU address space is the environment where the GPU program is executed in. To populate this address space with actual physical memory, the application, by means of the kernel-user interface, allocates physical memory buffers (②) and maps them into its own process address space (③) as well as into the GPU address space (⑤). It then lays out the job in memory by means of the process mappings (③) such that it forms a consistent executable job in the GPU address space. To comply with the requests of the application, the kernel driver constructs a page table that expresses a mapping from the GPU address space of the application to physical addresses. When the application requests to start a job, the kernel driver activates the corresponding page table on the GPU's MMU and passes user supplied information about the job layout on to the GPU.

B. The GPU Ressource Governor

Figure 6 shows our approach to providing mobile virtual machines with access to graphics computing resources. Note that as the mobile operating system moved into a virtual machine, the kernel-user interface (③) remains unaltered and so does the user part (①) of the driver. The hardware-facing part of the kernel driver was replaced by a stub (④) that communicates with the GPU server (⑤) via a communication channel (⑥), which is controlled by the underlying hypervisor. The GPU server, which exercises full control over the GPU including its MMU, may be a native user space application—if the hypervisor supports those—or run inside a virtual machine.

The GPU server allows the creation of contexts just as the kernel driver did before. In fact, it allows the creation of

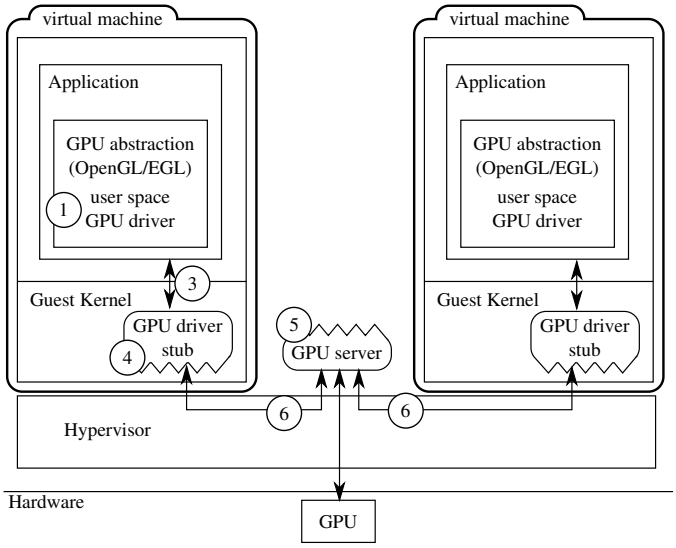

Fig. 6. The GPU server (middle) running as user space application serving two client VMs (left/right) on top of hypervisor or microkernel.

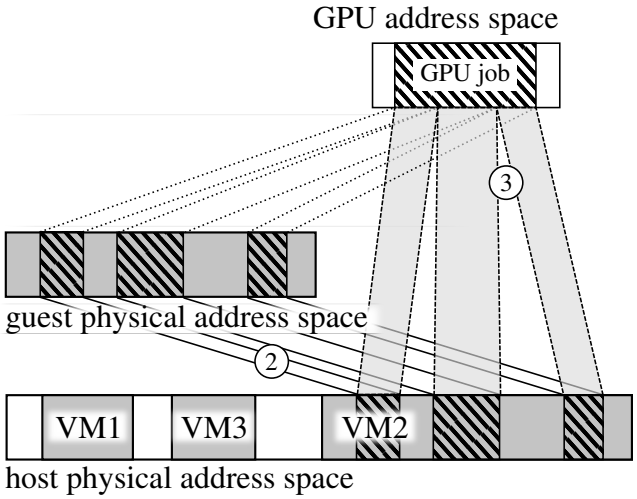

Fig. 7. Guest physical address space (gray, middle, left) and its representation in physical memory (gray, bottom, right). A GPU address space (top, left) in the context of two levels of translation due to virtualization.

multiple contexts per client, on which the guest-kernel drivers can map their client's contexts. For starting jobs, the stub driver simply passes the user supplied information on to the GPU server, which, in turn, activates the corresponding context and starts the job on the GPU. That is if the GPU is idle, otherwise the job is queued.

Complications arise when building the GPU context because what a virtual machine considers physical memory is in fact just another address space, the guest physical address space (see Figure 7), adding another level of translation (②). However, the MMU of the GPU does not have this expressive power. Therefore, the GPU server must construct a page table that maps directly from the GPU address space to the host physical address space (③). This technique is called shadow paging [11] and is widely applied in virtualization if the underlying hardware does not support multiple layers of translation. The mapping target is expressed in terms of guest physical addresses, and eventually the GPU server, which knows how guest physical addresses are mapped to host physical ones for a given client, constructs the effective page tables.

All information related to GPU jobs coming from the user space must be considered to be not trustworthy. Sanitizing this user supplied information, however, is not required as the GPU's MMU restricts memory accesses by the GPU.

V. Prototypical Implementation

Our prototype encompasses all of the components we described in Section III and Section IV: Two switches for framebuffers and input events, the input and display controller drivers, the policy master, and the GPU server. We implemented all of these components individually, each running inside their own protection domain. In this respect, our design is very flexible. It may even be beneficial to integrate switchers with their respective drivers to reduce communication overhead.

We built our prototype on a Samsung Galaxy SIII GT-I9300. It has an Exynos 4412 SoC[2] featuring four ARM Cortex A9 CPU cores running at a maximum of 1.4 GHz clock frequency and an ARM Mali 400MP GPU. The basis for our software stack is L4Android [29] consisting of the Fiasco.OC[3] microkernel with its complementary runtime environment L4Re[4] and L4Linux[5], a modified Linux kernel that runs as user space program on Fiasco.OC. We use Fiasco.OC and L4Re based on public revision 38 and L4Linux based on Linux[6] 3.0.101. Instead of the Android open source project (AOSP)[2], which is used by L4Android, we use Cyanogenmod [3] CM-10.1.3 as user facing component.

The GPU shadow paging implementation of our prototype has a noteworthy distinction. Unlike in typical shadow paging, we do not have the stub driver of the client create guest page tables; rather we let it forward every mapping request to the GPU server, which, in turn, constructs a page table for the GPU's MMU on the clients' behalf. This saves memory on the client side, but of course, the memory used for these page tables must be accounted for to prevent denial-of-service attacks. Therefore, we use a quota mechanism to restrict the amount of secondary memory that a client can use for page tables on the server side. We would like to emphasize that this shadow paging scheme is only concerned with the construction of GPU address spaces. The page table management for task address spaces is still performed by the microkernel, and its performance is not affected by the user-level handling of the GPU page tables.

An alternative implementation would be to maintain guest GPU page tables on the client side. The GPU server would

[2]System on Chip
[3]http://os.inf.tu-dresden.de/fiasco.
[4]http://l4re.org
[5]http://os.inf.tu-dresden.de/L4/LinuxOnL4/
[6]http://kernel.org

Module	SLOC
GPU server	2,679
display driver	2,382
frame-buffer switch	548
input driver	710
input switch	539
total	6,858

TABLE I
COMPONENT SIZES OF THE PROTOTYPE IN *source lines of code* (SLOC) MEASURED WITH DAVID A. WHEELER'S "SLOCCOUNT".

Module	SLOC
Fiasco.OC	28,943
moe	4,044
ned[7]	16,078
sigma0	1,023
io	12,864
total	62,952

TABLE II
COMPONENT SIZES OF THE ENVIRONMENT IN *source lines of code* (SLOC) MEASURED WITH DAVID A. WHEELER'S "SLOCCOUNT".

then derive the effective GPU page table for the next job from the corresponding guest GPU page table on demand. This scheme would restrict the secondary memory consumption on the server side to one page table but would also incur undesirable runtime overhead when switching GPU contexts.

VI. IMPACT ON TCB SIZE

Murray et al. [32] define the trusted computing base (TCB) as "the set of components which a subsystem S trusts not to violate[, although capable of doing so,] the security of S". The Fiasco.OC microkernel and the L4Re components *sigma0* (the root-pager), *moe* (the root-task), *ned* (the bootstrapper) and *io* (the device manager) constitute the TCB of our prototype. The subsystems we added in turn add to the TCB of the subsystems, such as the VMs, that depend on their services. Worse yet, some of these components such as the GPU server and the display controller driver control DMA capable devices, which gives them a means to violate the security of even those subsystems that do not depend on their services. The implications, however, depend on the capabilities of the controlled device in question.

Table I shows the sizes of the components we added in our implementation and Table II shows the sizes of the L4Re components and the Fiasco.OC microkernel for comparison. We can see that our additions are rather moderate. They appear outright minuscule when compared to other GPU virtualization techniques found in the desktop and server realm, which are typically based on *API-remoting*. API-remoting places the virtualization boundary at the application programming interface, such as OpenGL or Direct3D. This removes hardware dependency from the VMs at the cost of a larger TCB. Smowton [35], for example, suggest a GPU virtualization scheme for the Xen [9] hypervisor with an impact on the TCB of 80,000 SLOC even though his approach reduces TCB impact when compared to other API-forwarding techniques. To be fair, in Smowton's approach, the vitualization boundary is both hardware and API-independent, which makes it unique in the field of GPU-virtualization. Trading hardware independence for a tiny TCB is a fair trade-off for use-cases demanding high security.

VII. EXPERIMENTAL RESULTS

Lacking a comparable virtualized mobile system, we compare the graphics rendering performance of our prototype with a non-virtualized setup running on the same hardware platform. For better comparability, we fixed the clock frequencies of all CPU cores to 1.4 GHz and the GPU clock frequency to 266 MHz. To quantify the performance impact of our architecture as opposed to the impact suffered from moving Android/Linux into a VM, we devised an intermediate setup, where the resources of the GPU (memory mapped device registers and interrupts) are *passed through* to one client VM, which drives the GPU directly. Therefore benchmarks will suffer the same overhead from system calls and shadow paging as well as interrupt latency but without the anticipated communication overhead of our GPU virtualization scheme. It is important to note that the pass-through scenario is not fit for production for two reasons. First, the GPU cannot be shared by multiple VMs. Second, even more importantly, direct access to the GPU allows the guest kernel of the particular VM to break out of its isolation domain. Therefore it is but a vehicle for performance evaluation.

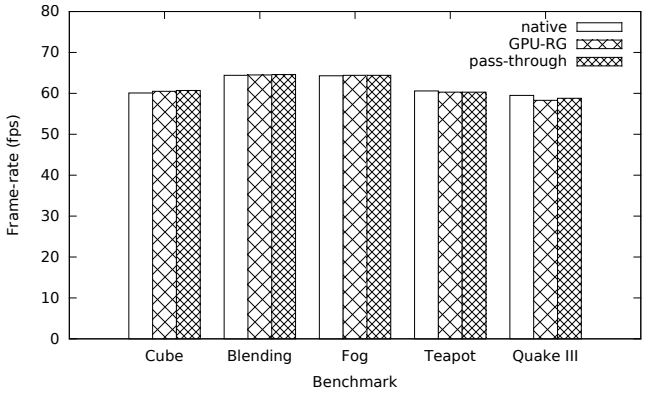

Fig. 8. High level 3D Benchmarks. Four benchmarks (Cube, Blending, Fog and Teapot) are part of the Android benchmark suite 0xbench by 0xlab. The fifth is the ioquake based Android port of QuakeIII Arena QIII4A.

First we measured the performance with a high-level 3D benchmark. Figure 8 shows the absolute results of these benchmarks. The first four benchmarks are part of the 0xbench [1] benchmark suite. The fifth benchmark is the `FOUR.DM_68` demo of QuakeIII Arena running on QIII4A [8], which is the Android port of the *ioquake3* [6] engine, in timedemo mode. The benchmarks use different techniques whereby they synchronize rendering to the display's refresh rate of 60 Hz. In any case, all benchmarks hit an artificial limit, which made them useless for characterizing the virtualization overhead.

[7]Includes a lua interpreter of size 14,124 SLOC.

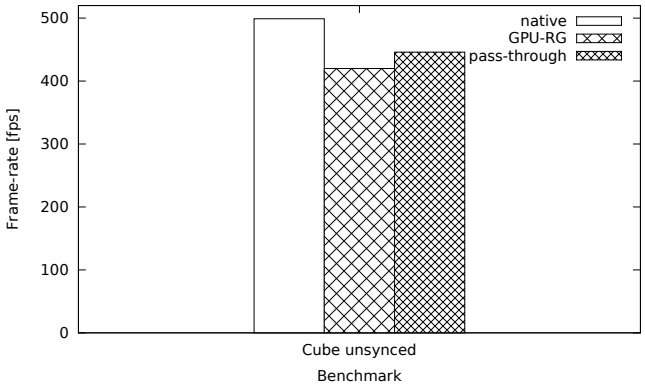

Fig. 9. Modified version of the 0xbench benchmark "Cube".

experiment			GP	PP
native	submit	[μs]	15.0	25.2
pass-through	submit	[μs]	22.1	34.9
	notify	[μs]	3.6	3.2
GPU-RG	submit	[μs]	47.3	67.5
	notify	[μs]	52.8	49.7

TABLE III
RESULTS OF THE JOB_TOOL BENCHMARK. THE TABLE SHOWS THE TIME IT TAKES TO SUBMIT A JOB TO AND RECEIVE A JOB COMPLETION NOTIFICATION FROM THE MALI MP400 GPU'S GEOMETRY PROCESSOR (GP) AND PIXEL PRESENTER (PP).

Therefore we released the handbrake of the "Cube" benchmark by rendering it off-screen, decoupling it from the artificial refresh rate limitation. The results for this benchmark are shown in Figure 9 as "Cube unsynced". The modified benchmark drives all scenarios into saturation at between seven and eight times the frame rate of the synchronized version. Compared to the non-virtualized scenario, we see a frame rate penalty of 11 % for the pass-through scenario and 16 % for the GPU-RG scenario.

Other GPU-virtualization approaches often incur massive CPU load and contention on the virtual network interconnect. Depending on the scenario, CPU overhead, even a significant one, may be acceptable in the server and desktop realm, because CPU cycles and energy are available in abundance, and the achievable graphics performance is still worth the cost. However, for mobile applications, where battery capacity is usually at a premium, GPU virtualization must not entail excessive CPU load. To measure the CPU load of our solution, we developed the microbenchmark `job_tool`.

The `job_tool` benchmark measures the time it takes to submit a precompiled GPU job.[8] Also, we measured the overhead when a job completion is signaled through an interrupt. To do so, we instrumented the Fiasco.OC kernel to log interrupt events and the guest kernel driver's top-half handler to log GPU interrupt events. [9] When `job_tool` gets notified of a job completion, it evaluates the trace buffer entries of the immediate past and determines the time it took for the event to travel from the Fiasco.OC kernel to guest kernel's top-half handler. Note that this delay does not exist in the native case as the interrupt registers directly in the Linux kernel.

Unlike most modern GPUs, the Mali 400MP does not have a unified shader design but rather two distinct programmable blocks for the geometry phase and the fragment phase, called geometry processor (GP) and pixel presenter (PP) (sometimes also pixel processor), respectively. The individual results are presented in Table III.

We can see that job submission times increase by 7.1 μs (9.7 μs) per GP (PP) job when the driver moves from bare metal (native) into a VM (pass-through). This is well within the range we would expect considering that system calls in an L4Linux VM incur the cost of two address space switches and four privilege level transitions rather than just two privilege level transitions. Moving the GPU driver into its own protection domain (GPU-RG) increases the cost of a job submission by another 25.2 μs (32.6 μs), owing to the additional context switches into the GPU-server.

Job completion notifications incur a cost of 49.2 μs (46.5 μs) when comparing GPU-RG with pass-through. To understand where this rather large difference came from, we discriminated different measurements by the number of context switches that occurred on the CPU handling the notification. We found that the measurements in the pass-through scenario were dominated by instances where no context switch happened between the occurrence of an interrupt and its delivery to the VM. Apparently the VM was already running most of the time when the interrupt occurred. In the cases where a context switch was necessary the delay rose from three microseconds to approximately 22 μs. With the interposition of GPU-RG, the server was almost never running when the interrupt occured, so at least two context switches where necessary: The first switched to GPU-RG, and the second switched to the designated client VM.

With this benchmark we measured the modified code paths involved in the dynamic behavior of our architecture, which is why we believe that this benchmark captures the extra CPU load incurred by the system. Using GPU-RG increases the time to handle a GPU job by 74.4 μs (GP) and 79.1 μs (PP) compared to the pass-through scenario. To complement these numbers, we determined that all of the high-level benchmarks issue one set of jobs—encompassing one GP and four[10] PP jobs—per frame and the compositor of Android adds one more set of jobs to produce the visible output. For a target frame rate of 60 Hz, this is equivalent to an extra load of 4.6 % of the CPU time incurred by one CPU core.

[8]We make sure that the job is submitted while the GPU is idle. Otherwise the benchmark measures the time it takes to queue a job in the guest driver which takes around 11 μs in both scenarios pass-through and GPU-RG.

[9]The trace buffer is a facility in the microkernel that allows logging with very low overhead. Each logged event is marked with a timestamp. The trace buffer can be made accessible in VM processes.

[10]The ARM Mali 400MP GPU can be implemented with varying numbers of PP-cores. The variant in our test device has four PP-cores and can thus execute four PP-jobs in parallel.

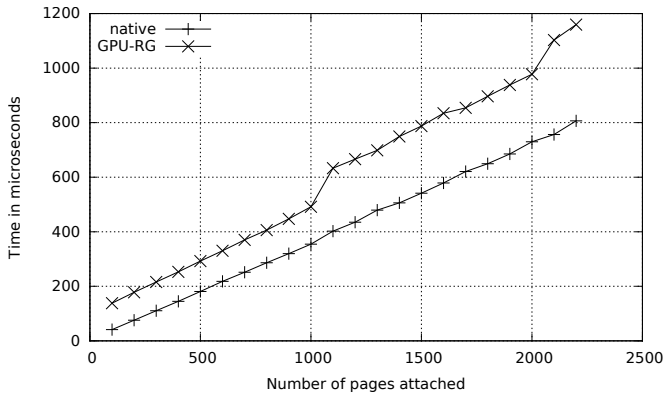

Fig. 10. The figure shows the time it takes to map pages into the GPU address space. Every data point is the average of 10,000 measurements. The numbers were generated with `map_tool`.

The time needed to build a GPU context is also influenced by our GPU server design. Because mapping request must now be communicated to the server, we expected a communication overhead. Context building happens infrequently, e.g., when an application is started or when the scene of a game changes. The performance of context building does not so much influence the framerate, rather it influences the load time of an application. We designed the `map_tool` microbenchmark to measure this overhead. It repeatedly attaches buffers of various sizes to a GPU address space and measures the time needed to complete the request. We show the results for buffer sizes from 100 to 2,200 pages[11] in Figure 10. Each data point is the average of 10,000 measurements. We can see a conservative 30 % overhead compared to the native scenario. This does not mean that load time of an application or a scene increases by this ratio because attaching buffers to a context is but a small fraction of this operation. Even for large buffers the absolute overhead is limited to few milliseconds and hardly noticeable by the user.

What is interesting about this plot is that a design decision we took when building the prototype manifests as a step between 1,000 and 1,100 pages and between 2,000 and 2,100 pages. As a performance optimization, we batched up to 1024 mapping requests. The steps in the plot constitute the points where the request buffer fills up and an extra IPC-rountrip is necessary to flush the buffer.

VIII. Related Work

We group the related work into three categories: BYOD solutions, secure graphical user interfaces and GPU virtualization techniques.

a) BYOD:

VMware MVP [13] is a mobile hosted hypervisor[12] solution for Android. It features isolated Android domains for the fast deployment of enterprise applications and policy enforcement. The authors acknowledge the onerosity of supporting a huge diversity of device drivers. Their solution is pragmatic in that they map a limited set of services to standard interfaces provided by the host leaving out more complicated devices such as the GPU, thereby trading user-experience for security and a clear preference for compatibility and manageability. As with all hosted hypervisors, the TCB of all VMs encompasses the host kernel and all privileged host processes. We believe that for applications with high security demands, the concept is not well suited because unable to provide the resources needed for leisure and gaming in a confined VM, it exposes the host system to the hostile world of malware.

Cells [12] uses Linux containers and a driver namespace concept to provide multiple runtime environments. It provides the user with multiple virtual phones by duplicating the user-space software stacks sharing a single underlying kernel. This allows Cells to reuse device drivers with little or no modifications and consequently provides near unhampered graphics performance. Cells' virtual phones have a large TCB due to the application of the Linux kernel. Furthermore, there is no secure label indicating the current environment to the user.

vNative [18] hosts multiple Android-based VMs on a Xen hypervisor for devices featuring an Intel Medfield SoC. Unlike MVP and Cells, it allows full device pass-through to all VMs. To tame DMA devices, vNative makes use of isolated memory regions (IMR), a proprietory feature of the Medfield SoC. The downside is that due to the pass-through architecture, only one so-called foreground VM can make progress at a time while background VMs are deprived of any computing resources. Due to the strong dependency on IMR, the concept is limited to a single platform. Also, there is no report of a graphical VM identity indicator.

b) Secure GUI:

The EROS Window System (EWS) [34] "provides robust traceability of user volition and is capable (with extension) of enforcing mandatory access controls." To that end, it features unforgeable window labels and window decorations as well as exclusive input event routing, the recipient window of which is indicated by dimming all other windows. EWS is explicitly not optimized for performance and there is no mention of accelerated graphics.

Nitpicker [21] generalizes from the underlying system, allowing legacy windowing systems running alongside other (native) applications. It introduces the X-ray mode, which changes the appearance of the windows to indicate identity and focus and can be activated by the user when in doubt. The authors state that a trusted boot process and client authentication is required to reliably report the identity of the clients (applications) to the user.

Crossover [28] transfers some of the ideas presented by EWS and Nitpicker, such as secure labeling and trusted path, to the mobile BYOD use-case. It is complementary to our work in that it has a strong emphasis on usability aspects. Besides providing for secure GPU sharing, we also improved on the input and output switching in that we provide vsync and

[11] Due to the page table format, a page of 4,096 bytes is the granularity for mapping requests.

[12] A hosted hypervisor is also referred to as Type-II hypervisor.

double buffering support, both of which are essential features. Crossover's architecture does not allow for a VM to use more than one hardware overlay. In contrast, our architecture reserves only one hardware overlay and leaves all others for the use by a guest VM. We have not implemented that feature in our prototype.

c) GPU Virtualization:
Depending on the virtualization boundary, GPU virtualization schemes can be grouped into the two categories: front-end and back-end virtualization. Front-end virtualization schemes use a high-level API, such as OpenGL, as the virtualization boundary. Back-end schemes place the virtualization boundary at or close to the hardware interface of the device.

Front-end virtualization schemes reuse commercially available drivers to provide a 3D rendering service. These considerably large drivers need to be deployed on the host side of the virtualization boundary and therefore inside the TCB. This technique suffers overhead from marshaling and auditing the remote API calls. Still, it holds appeal in that it hides the physical hardware from the guest VMs, which is desirable for heterogeneous environments where VMs are to be dynamically migrated.

VMGL [27] is a classical front-end virtualization scheme. A stub OpenGL library is deployed in the guest, marshaling and transmiting OpenGL calls to the host using WireGL [24] over a virtual network interconnect. It is complemented by VNC [33] to support input, 2D output, and seamless integration into the host's window management.

Blink [23] extends OpenGL (BlinkGL) with the goal to overcome the weaknesses of API forwarding. Versioned shared objects reduce the bandwidth demands by passing large objects, such as textures, by reference rather than sending a serialized version via network. Stored procedures reduce context switching overhead, e.g. for reacting to user input. A Just-In-Time compiler further improves the performance of the stored procedures. While Blink is compatible with legacy OpenGL programs, only client applications using the BlinkGL extension can benefit from the optimizations.

SVGA3D [19] emulates a SVGA adapter at the MMIO-register level. It extends the VGA adapter with a command set for 3D acceleration. On the host side this emulation is performed using a high level API.

Xen3D [35] is a split-driver approach based on the Gallium3D graphics driver framework. Gallium3D [5] comprises an API specific *State Tracker*, a hardware specific *Pipe Driver*, and a *Window System Driver*. All three parts are agnostic about the intricacies of the implementation specifics of the respective other parts, which requires generalized interfaces. Xen3D uses this generalized intermediate representation between State Tracker and Pipe Driver as the virtualization boundary. When compared with the other front-end virtualization schemes, this holds appeal in that it moves a great deal of the high-level API abstraction into the guest, thereby reducing the impact on the TCB size.

Back-end virtualization schemes forgo portability as they require hardware specific software in the guest. The gain, however, is performance and fidelity that is unmatched by front-end schemes. Moreover, the impact on the TCB size can be reduced by an order of magnitude.

Tian et al. [36] present a true back-end virtualization scheme. GPU commands are directly executed by the GPU rather then being emulated on the host side using a high-level API. So as to gain interposition, they partially emulate an Intel GPU on MMIO-register level. In contrast we use paravirtualization, which accepts guests modifications for better performance due to less interaction between the guest and the device virtualizer. Tian et al. have all guests share the lower half of the GPU address space. While conflicts are avoided by memory ballooning, there is no hardware-enforced isolation between guests in that region. Isolation is achieved through auditing command sequences for illegitimate memory accesses—that is, accesses to memory regions belonging to other guests. However, command streams need to be copied or made non-writable prior to auditing in order to prevent *time of check time of use* (TOCTOU) attacks. In contrast, in our solution GPU address spaces are never shared. Guests have their GPU jobs executed in an environment where access is confined to memory alloted to the corresponding guest, obviating the need for command stream auditing.

IX. Conclusion and Outlook

We proposed an architecture for building a hardware-accelerated graphical user interface for virtualized environments on mobile handsets. The feasibility of the approach was shown by means of a prototypical implementation serving as a testbed for a series of experiments. For GPU-intensive applications, the frame rate dropped by 5 % on top of an 11 % drop due to CPU virtualization overhead. Our implementation comprises less than 7,000 SLOC spread across five components of individual sizes ranging from 539 SLOC to 2679 SLOC. The components, some of which are critical to the system's correct behavior due to their role in GPU page table management, are small enough to allow for thorough auditing. Our approach counters spoofing and eavesdropping attacks on user input and output by providing a trusted and identifiable path between each virtual machine and the user.

There have been reports of information leakage via in-GPU memory and registers [31], [14], [17]. While we are not aware of comparable research on mobile GPUs, we cannot rule out the possibility of residual information remaining in the GPU for other clients to extract after a job completes. Other than providing a passive side channel, this could also be leveraged to establish a covert channel between two supposedly isolated domains. Unfortunately, an investigation of this matter requires intimate knowledge about the hardware in question, which manufacturers are reluctant to provide. Reverse engineering efforts [7], [4] can provide researchers with valuable information needed to address this issue in the future.

Acknowledgements

This work was partially supported by the EU FP7 Trustworthy ICT program (FP7-ICT-2011.1.4) under grant agreement

no. 317888 (project NEMESYS).

REFERENCES

[1] 0xbench. https://code.google.com/p/0xbench/.

[2] Android open source project. https://source.android.com/.

[3] Cyanogenmod. http://www.cyanogenmod.org/.

[4] Freedreno project. https://freedreno.github.io/.

[5] Gallium3d. http://www.freedesktop.org/wiki/Software/gallium/.

[6] ioquake3. http://ioquake3.org/.

[7] Lima driver project. http://limadriver.org.

[8] Qiii4a. https://play.google.com/store/apps/details?id=com.n0n3m4.QIII4A&hl=de.

[9] Xen. www.xenproject.org.

[10] Cve-2014-0972. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0972, 01 1014.

[11] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 2–13. ACM, 2006.

[12] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In T. Wobber and P. Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 173–187. ACM, 2011.

[13] K. C. Barr, P. P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *Operating Systems Review*, 44(4):124–135, 2010.

[14] S. Breß, S. Kiltz, and M. Schäler. Forensics on GPU coprocessing in databases - research challenges, first experiments, and countermeasures. In G. Saake, A. Henrich, W. Lehner, T. Neumann, and V. Köppen, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), - Workshopband, 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, volume 216 of *LNI*, pages 115–129. GI, 2013.

[15] R. Clark. Kilroy. https://github.com/robclark/kilroy.

[16] J. Danisevskis, M. Piekarska, and J. Seifert. Dark side of the shader: Mobile gpu-aided malware delivery. In H. Lee and D. Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 483–495. Springer, 2013.

[17] R. Di Pietro, F. Lombardi, and A. Villani. Cuda leaks: information leakage in gpu architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[18] Y. Dong, J. Mao, h. Guan, J. LI, and Y. Chen. A virtualization solution for byod with dynamic platform context switch. *Micro, IEEE*, PP(99):1–1, 2015.

[19] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. In *first USENIX Workshop on I/O Virtualization*, 2008.

[20] A. P. Felt and D. Wagner. Phishing on mobile devices. In *In W2SP*, 2011.

[21] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, 2005.

[22] T. Fiebig, J. Danisevskis, and M. Piekarska. A metric for the evaluation and comparison of keylogger performance. In C. Kanich and P. Lardieri, editors, *7th Workshop on Cyber Security Experimentation and Test, CSET '14, San Diego, CA, USA, August 18, 2014*. USENIX Association, 2014.

[23] J. G. Hansen. Blink: Advanced display multiplexing for virtualized applications. In *Proceedings of the 17th International workshop on Network and Operating Systems support for Digital Audio and Video*, 2007.

[24] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140. ACM, 2001.

[25] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.

[26] K. Kortchinsky. Cloudburst. *Black Hat USA June*, 2009.

[27] H. A. Lagar-cavilla and M. Satyanarayanan. Vmm-independent graphics acceleration. In *Proceedings of VEE 2007*. ACM Press, 2007.

[28] M. Lange and S. Liebergeld. Crossover: secure and usable user interface for mobile devices with multiple isolated OS personalities. In C. N. P. Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 249–257. ACM, 2013.

[29] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operation system framework for secure smartphones. In *SPSM*, 10 2011.

[30] C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*. The Internet Society, 2014.

[31] C. Maurice, C. Neumann, O. Heen, and A. Francillon. Confidentiality issues on a GPU in a virtualized environment. In N. Christin and R. Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2014.

[32] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, 2008.

[33] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33–38, Jan 1998.

[34] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the eros trusted window system. In *USENIX Security Symposium*, pages 165–178, 2004.

[35] C. Smowton. Secure 3d graphics for virtual machines. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 36–43, New York, NY, USA, 2009. ACM.

[36] K. Tian, Y. Dong, and D. Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 121–132. USENIX Association, 2014.

[37] K. Yee. User interaction design for secure systems. In R. H. Deng, S. Qing, F. Bao, and J. Zhou, editors, *Information and Communications Security, 4th International Conference, ICICS 2002, Singapore, December 9-12, 2002, Proceedings*, volume 2513 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 2002.

[38] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 5 2012.