ACCELERATED SECURE GUI FOR VIRTUALIZED MOBILE HANDSETS

vorgelegt von

Dipl.-Ing.
Janis Danisevskis
geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 15. August 2017

Berlin 2017

Abstract

Mobile handsets, especially so-called smartphones, have become an indispensable commodity in day-to-day life. However, their growing versatility came at the cost of ever-increasing complexity, and this raises severe security concerns. This has come to be especially problematic for corporate IT infrastructures, because it is increasingly hard to reconcile personal user expectations with corporate security demands. A particular manifestation of this quandary is the bring-your-own-device (BYOD) application, where multiple mutually distrustful stakeholders maintain individual security interests.

A technique for safeguarding these interests is virtualization. Limited computational capabilities and battery capacity pose challenges to virtualization being a nascent discipline in the embedded computing realm. Graphical user interfaces of smartphones rely heavily on graphics acceleration hardware, a fact that makes moving a contemporary smartphone operating system into a virtual machine particularly challenging.

This work sets out to explore the threats that arise by sharing the graphical user interface infrastructure of a smartphone among multiple virtual machines. These findings have led to the development of a small and strongly compartmentalized secure graphical user interface infrastructure, which provides an identifiable and trusted path between the user and the virtual machine and which caters to the performance demands of contemporary mobile user interfaces by allowing shared access to graphics acceleration hardware. A thorough performance evaluation employing a series of specially tailored evaluation tools attests that the working prototype built and described as part of this work has outstanding performance, matching a non-virtualized smartphone, with only little impact on the system's load and latency.

Contents

Listing of figures

List of Tables

List of acronyms

ARM	Advanced RISC Machine
ASID	Address Space Identifier
BYOD	Bring Your Own Device
CI	Coordinating Instance
CP	Command Processor
CRT	Cathod Ray Tube
DBG-ID	Debug Identifier
DC	Display Controller
EWS	EROS Trusted Window System
FB	Framebuffer
FRC	Free Running Counter
GIC	Generic Interrupt Controller
GLSL	OpenGL Shading Language
GP	Geometry Processor
GPIO	General-purpose Input/Output
GPU	Graphics Processing Unit
GPUMMU	GPU Memory Management Unit
GPURG	GPU Recource Governor

GUI	Graphical User Interface
GVA	GPU Virtual Address
I^2C	Inter-Integrated Circuit
IPC	Inter Process Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
JIT	Just-In-Time
KIP	Kernel Information Page
MCT	Multi Core Timer
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
OS	Operating System
PMU	Power Management Unit
PP	Pixel Presenter
PTE	Page Table Entry
RAM	Random Access Memory
RDM	Routing Decision Maker
RISC	Reduced Instruction Set Computer
SLOC	Source Lines Of Code
SYSMMU	System MMU
TCB	Trusted Computing Base
TI	Texas Instruments
TLB	Translation Lookaside Buffer
TOCTOU	Time Of Check/Time Of Use

TTBR Translation Table Base Register

UI User Interface

UMP Unified Memory Provider

VBus Virtual Bus

VM Virtual Machine

VMM Virtal Machine Monitor

VSO Versioned Shared Object

VSYNC Vertical Syncronization

1 Introduction

The advent of Apple's iPhone in 2007 and Google's Android operating system in 2008 represented a leap in mobile computing experience, which eventually brought mobile hand-held computing to the masses. At the time, other contemporary systems appeared rather clumsy and were awkward to operate in comparison. The discrepancy in the usability then developed into a potential threat to the national security. Personnel in critical areas, such as government officials, despite being obliged to use specially certified devices based on older technologies, started to adopt the new devices, which were easy to use and rich in features. But versatility comes at the price of complexity, and complexity is prone to errors yielding security vulnerabilities. The history of security vulnerabilities found in the new operating systems would prove anyone right who warned about using smartphones in this field.

In this context, the SiMKo3 project was initiated. SiMKo is short for *"sichere mobile Kommunikation"* (German for: secure mobile communication), and its goal was to devise a smartphone certified for handling classified information and, at the same time, exhibiting the versatility and usability of a contemporary hand-held device. This author was among a team of researchers at Technische Universität Berlin, which set out to achieve this goal using virtualization. An off-the-shelf smartphone was to be amended with a trusted boot process aided by a secure element providing reliable key storage for disk and communication encryption as well as facilitating pre-boot authentication of the user. The network connectivity of the phone was to be tightly controlled by means of a mandatory VPN tunnel into the user's corporate or governmental infrastructure, and its configuration was to be outside the control of the user. In order to provide a small and customizable trusted computing base, the phone's software was to be based on a small microkernel with compartmentalized runtime environment. The high security requirements demanded a small trusted computing base for two reasons: The consensus was that a smaller code base yields fewer programming errors and thus less potential vulnerabilities. Moreover, it was agreed upon that presenting the feasibility of a compartmentalized solution with sufficiently small subsystems would pave the way for applying formal methods to the problem in the future. In order to accommodate contemporary usability demands, the microkernel was to host two instances of the Android operating system. One of the two Android instances, the secure compartment, was to be locked down, its versatility reduced to a fixed set of certified applications. The other, the open compartment, was to be left customizable at the discretions of the user.

This arrangement, in which two mutually distrusting compartments with different security classifications shared a common screen and common input devices, posed a challenge. The increasing popularity of the new mobile operating systems gave rise to a surge of malware. Typically, malware is designed to infect as many targets as possible, often by masquerading as legitimate applications that the user installs voluntarily (Trojan horse). Restricting the adaptability of the secure compartment by prohibiting the installation of third-party applications, however, does not remove this attack surface entirely. The open compartment is still prone to such malware, and in an ecosystem with high-profile users, it must be assumed that the user may be subjected to targeted attacks. An attacker could exploit the fact that the open and secure compartment do share one screen and common input devices and thus mount an impersonation attack with malware installed in the open compartment, thereby stealing credentials that must never leave the secure environment of the secure compartment or feeding false information to the user. It has been imperative that the user must be able, at any time, to make a well-informed decision about with which compartment he or she is interacting.

Virtualization is a technology well established in data center and desktop computing. Mobile embedded virtualization, however, has been a young discipline, where limited computing capabilities and energy supply posed new challenges to the operating systems engineer. Smartphones, in particular, were tedious to virtualize, due to the sheer number of integrated peripheral devices. Further, even the virtualization of devices, such as the GPU, which have been virtualized before, needed to be thought out anew under these new constraints. A survey of existing GPU virtualization solutions yielded that they either did not allow for sharing the resource, lacked interposition, bloated the trusted computing base, were wasteful with CPU cycles, or exhibited a combination of these drawbacks. It came without a question that the GPU needed to be shared, while at the same time, the strong isolation promised by the underlying microkernel was to be upheld. Eventually, avoiding superfluous CPU usage by the GPU virtualization scheme was the one requirement mostly owed to mobile computing constraints.

Scope and Goals

This work addresses the secure user interface for multi-stakeholder mobile handsets based on virtualization technologies on a system level. It is, therefore, immediately relevant to the operating system design of secure smartphones and tablets, but it is also relevant to other embedded applications where high assurance meets the need for well performing visualization. Graphical design, look and feel, and user guidance are not within the scope of this work. Where the user experience is concerned, this work addresses it indirectly by optimizing technically measurable quantities, such as frame rates and latency. The design was driven by the isolation requirements of integrating high security applications and infotainment, the physical user interface design of mobile handsets, and the limitations in available resources such as energy supply and computational capabilities.

The construction of the virtualized smartphone was a team effort and therefore posed a rare opportunity. Further, it also meant that the graphical user interface architecture presented in this work needed to be built within the constraints given by more general design decisions. Moreover, it was confined to the hardware chosen to be supported by the prototype. Within these constraints, the goal was to develop widely applicable models for a secure and hardware accelerated graphical user interface, supporting at least two virtual machines on a contemporary mobile handset as well as their prototypical implementation. The design should exhibit strong isolation with little impact on the trusted computing base. It was mandatory to provide a trusted and identifiable path between the virtual machines and the user. In addition, the performance was to be as close as possible to the performance of a non-virtualized device with as little overhead as possible.

Contributions

As a contribution, this work assesses the security implications of contemporary embedded graphics processing units (GPU) on virtualized and non-virtualized systems. A fully functional exploit is presented, substantiating the security implication claims. Two complementary subsystems were designed and implemented: one providing a trusted and identifiable path between the user and the VMs, with an emphasis on efficiency and low overhead, the other providing the VMs with an efficient and isolation-preserving means of tapping into the computational capabilities of the embedded GPU. The performance of the prototypical implementation was evaluated with respect to throughput, latency, CPU usage, and power consumption. The landscape of benchmarks and tools that was created for the purpose of this evaluation poses an auxiliary contribution in itself, and it has been applied to at least one other problem outside the scope of this work.

Organization

This work is organized as follows: Chapter 2 provides the background to this work. To this end, introductions are given in general system design and TCB construction as well as virtualization, leading up to a primer on the Fiasco.OC μ-kernel, its runtime environment L4Re, and L^{4}Linux, a rehosted version of the Linux operating system kernel running on top of Fiasco.OC. This is followed by adoptions of a typical mobile GPU driver architecture, GPU virtualization techniques, and secure GUI design. After setting the stage for the main act of this work, the potential threats to a system's integrity are assessed and illustrated through the dissection of two mobile GPU driver stacks in Chapter 3. With the lessons learned in the preceding chapters, Chapter 4 and Chapter 5 each present the design space and, subsequently, the prototypical implementation of two complementary and fully compartmentalized subsystems providing a secure and well-performing means for a trusted and identifiable user input/output path and graphics acceleration, respectively. Before this work concludes with Chapter 7, the prototypical implementation presented is evaluated in Chapter 6, with respect to performance, latency, power consumption, and impact on the trusted computing.

2 Background

The requirement of allowing a user to access classified material and at the same time indulge in the malware-ridden delights of mobile entertainment called for a high level of assurance. This assurance was achieved by drastically reducing the trusted computing base (TCB) and the attack surface on critical infrastructure, using the compartmentalization achieved through virtualization, where legacy operating system components were to be reused. This work aims at providing an efficient graphical user interface architecture adhering to the same principles, which warrants an introduction of the concepts and technologies that the secure smartphone is based. First, in Section 2.1, the concept of the TCB is introduced and illustrated by examining some of the existing smartphone architectures designed for comparable use cases. Section 2.2 gives a tour of the virtualization techniques and concepts for contemporary computer architectures, with a supplement covering the details of Fiasco.OC, L4Re, and L^{4}Linux, which are of special importance for understanding the prototypical implementations of this work. A typical driver stack of a mobile graphics processing unit is covered in Section 2.3. This is followed by a discussion of GPU virtualization techniques in Section 2.4, using a series of examples from contemporary literature. The last section of this chapter, Section 2.5, covers the principles of secure graphical user interfaces (GUIs).

System Architecture and TCB

Since the advent of computer systems, operating systems have come a long way. This work is only concerned with systems of the fourth generation as described by Tanenbaum [61]. That is, the processor is expected to support at least two privilege levels—user and supervisor—with well-defined entry vectors for system calls, exceptions, and interrupts, as well as the memory protection and access control to memory-mapped resources provided by means of a memory management unit (MMU). These primitives allow for running a fourth-generation operating system kernel in the supervisor mode, with other programs—processes—running in user mode isolated from the kernel and each other by time multiplexing and address space separation. The systems that can be built in this way exhibit great versatility and are vastly diverse. The applications span small embedded systems, desktop computers, data centers, and supercomputing clusters. The constraints exhibit the same diversity: Web servers need to service vast amounts of requests, and they thus need high throughput. Desktop computers have to meet low response times and soft real-time constraints to provide a pleasant user experience. Safety-critical systems may have hard real-time constraints. All of these constraints can be met by fourth generation architectures, but not all implementations of such are equally well suited for each task.

Linux, for example, has been embraced by a large open source community as well as by many commercial contributors. Found in almost all of the aforementioned applications, it shows great performance and has extraordinary driver support. However, when it comes to applications requiring a truly high level of assurance, one is wise to shy away from the vast code base that the Linux kernel has acquired over the years, which has recently exceeded 20 million lines of code. Moreover, while the consumer electronics industry cherishes the availability, versatility, and extensibility of the large and monolithic Linux kernel, there is a consensus in the operating systems and security research community that systems demanding high security and assurance should be designed as compartmentalized [22, 56, 58] and with a reduced trusted computing base.

> **Trusted Computing Base (TCB)** The totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.
>
> —"The Orange Book" p. 112 [44]

In UNIX-like multiuser systems, the same user ID and set of group IDs are usually assigned to all programs run by a given user, who is identified when logging into the system. File accesses rights in UNIX-like system are granted with user ID and group ID granularity, which means that all of the programs run by the user have access to the same set of files. This provides sufficient protection to the user's data under the assumption that all of these programs are trusted by the user and act solely on the behalf of the user. An assumption not held in the wild. A typical example is ransomware, which is malicious software encrypting all accessible files and holding them for ransom. One of the key features of smartphones is extensibility. Nonetheless, it is also one of their greatest problems, as not all third-party applications are legit [67]. The smartphone operating system Android, which, running on top of a Linux kernel, is a UNIX-like system, accounts for the mutual distrust of applications by assigning each application an individual user ID. This is possible under the assumption that a smartphone has only one user, the owner. It empowers the applications to guard the persistent data stored in a file against access by other applications. The Android middleware and the Linux kernel still have many channels through which malicious applications can steal or manipulate the user's data or credentials [20].

In an attempt to mitigate the crosstalk through the Android middleware, Cells [14] uses Linux containers and a driver namespace concept to provide multiple runtime environments duplicating the Android user-space middleware isolated by the Linux kernel. Nevertheless, as stated before, the Linux kernel is huge, and it has a large attack surface, one too large for a high assurance application.

VMware MVP [16] goes one step further by deploying instances of Android, including their underlying Linux kernel, in virtual machines (VM). VMware MVP is a hosted hypervisor solution, also known as Type-2 hypervisor, where an operating system kernel, here the Linux kernel, acts as hypervisor. The approach has some intriguing properties, such as compatibility and manageability; properties that are valued by administrators in the context of bring your own device (BYOD) applications. The idea behind the BYOD concept is that employees should use their own personal mobile phone, which presumably most of them have anyway, rather than being issued a company phone. However, incorporating third-party devices into the corporate IT raises a lot of questions as to the enforcement of corporate IT policies. To that end, the hosted hypervisor solution has some merits in that it allows business appliances, that were provisioned in advance, to be deployed on an employee's personal phone. Moreover, the virtualization interface reduces the attack surface on the acting hypervisor for attacks from inside the VM. But this compatibility comes at the cost of fidelity and performance. While the host operating system can tap into the full potential of the smartphone's hardware, the VMs are limited to the minimum essentials needed for business applications. In consequence, the host exposes the wide attack surface of the Linux kernel to the malware-ridden world of casual entertainment, while the business applications are strongly confined in their VMs. But with the Linux kernel, which has the role of the hypervisor, subverted, the protection of the VMs is penetrated from the outside. Figuratively speaking, it is like having a bouncer check on guests coming out of a club, rather than on those requesting admission. The TCB of the VMs consists of the acting hypervisor, which is the Linux kernel, as well as privileged user-level applications running on the host.

In contrast, the requirements of the secure smartphone dictated that the so-called open world, the one that was supposed to be freely customizable by the user with no restrictions as to which applications where allowed, was at least as confined as any other if not the most confined one. To that end, an architecture was chosen that was based on a small microkernel acting as hypervisor. Any user-facing world, or compartment as it was regularly called, was confined in individual VMs. [1] With around twenty-five thousand lines of code, the microkernel was by three orders of magnitude smaller than the Linux kernel in the hosted hypervisor solution. Therefore, the impact of the hypervisor on the TCB was greatly reduced. However, the microkernel, being limited in its functionality, needs the support by user-space servers to provide all the services that a VM requires. These servers, quite naturally, belong to the dependent VM's TCB as well. Above that, there are service providers that must be attributed to the TCB of a subsystem, even if the subsystem has no dependency, direct or indirect, on the provided service. The device drivers that control devices with extensive DMA capabilities are an example for such service providers [56]. Nonetheless, while any compromised subsystem of a monolithic kernel such as the Linux kernel will render all security assumptions of the whole system void due to the lack of separation, one must carefully discriminate among servers by reviewing the security properties affected when compromised. A compromised input device driver without DMA capabilities, for example, would be able to eavesdrop on, block, modify, or inject user input events. Thus, the security attributes affected would be the confidentiality, availability, and integrity of the user input events. The driver would belong to the TCB of the subsystems relying on these attributes. A VM's TCB can be constructed on a fine granularity. However, the question as to the fidelity and performance remains, and this is one of the driving questions behind this work.

Virtualization

The purpose of virtualization is to give users the illusion of multiple virtual machines (VMs) on a single physical device. Virtual machines are very versatile and find application in a large variety of fields. Besides the consolidation of multiple instances on a single machine, virtualization allows abstraction from the hardware to the extent where suspending and migrating running VMs becomes possible, even across heterogeneous hardware configurations. The later is an invaluable feature for efficiently managing resources in data centers. Virtual environments can perform logging outside the reach of an

[1] As for the discussion in Section 2.2, "virtual machine" is a bit of a stretch for the technology used for the secure smartphone. However, the architecture suggested in this work is equally applicable to full- or para-virtualized architectures. So the "virtual machine" terminology is used throughout this work.

attacker, thus safeguarding the logs from manipulation [19]. Alternatively, logging can be performed to the extent that a VM can be replayed, which is an invaluable tool for both debugging hard to trigger bugs and forensically analyzing an attack after it happened [19, 27]. Simple versatile interfaces resembling the machine interface promise strong and controllable isolation between subsystems deployed in different VMs. And, the deployment of applications in a VM, along with their own operating system (OS), allows for running legacy applications alongside newer ones or applications that where simply written for a different OS.

Nearly as diverse as the applications of VMs is the diversity of virtualization concepts. However, before virtualization technologies are discussed, the introduction of some virtualization-related terminology is warranted. "The heart of a VM system is the virtual machine monitor (VMM) software which transforms the single machine interface into the illusion of many"—Goldberg [34]. The term VMM, however, is not strictly defined and is often used interchangeably with the term *hypervisor*. And often enough, the notion of "something highly privileged determining the fate of a VM" is sufficient to describe a VMM or hypervisor. Depending on the context, however, these two can have somewhat different meanings that are not consistent throughout the literature. Agesen et al. [12] consider the VMM as provider of mechanisms to one individual VM. An operating system kernel underneath provides policy, such as scheduling and memory management. The operating system kernel, together with the VMM, forms the hypervisor of the corresponding VM. In contrast, Peter et al. [52] see the VMM as the component coordinating the execution of a VM and as provider of services beyond CPU and memory [47]. In this mindset, the hypervisor is but a concept providing mechanisms, such as the construction of protection domains and the controlled communication across protection domain boundaries; and its role is taken by a microkernel, which implements no policy with the exception of scheduling in this particular case.

From a purely theoretical perspective, any universal Turing machine can simulate any other Turing machine—that is, leaving the memory constraints aside. But this goes only for the correctness of the program execution. When timing and performance enter the picture, the equivalence of real and virtual machine vanishes [34]. Nevertheless, the term *full virtualization* is commonly used for techniques simulating a machine to the extent where it allows running unmodified guest code. In contrast, *para-virtualization* provides interfaces with machine-equivalent functionality, which require adjustments in the guest's code. Full virtualization can naturally be achieved by full simulation of the desired machine architecture. For early stage kernel development, the benefits of this approach outweigh the considerable performance cost. But when it comes to productivity, as in the field of data center consolidation, the incurred overhead is unacceptable. When Popek and Goldberg set forth the "Formal Requirements for Virtualizable Third Generation Architectures", they explicitly required that "programs run in this [virtual] environment show at worst only minor decreases in speed"—[53].

Intel's x86 architecture, which dominated the end-user computing market over the past decades like no other, does not meet these requirements. Virtualization requires the operating system kernel, which normally runs in the privileged mode, to run non-privileged as a guest, and the x86 instruction set architecture defines instructions that behave differently when executed in different privilege levels. If such an instruction traps, that is, it causes the CPU to drop into privileged mode and commence execution at a predefined position, then its behavior can be emulated. This, however, does not apply to all of these *sensitive* instructions of the x86 architecture, a circumstance, which renders this architecture non-virtualizable by trap-and-emulate [53, 55]. The technique of binary translation was introduced [11, 12] in order to perform full virtualization on x86 with decent performance. For this, the kernel code of the guest operating system is parsed, and sensitive instructions are replaced with equivalent code at, or immediately before runtime. This translation is done only once for every chunk of code, when executing it for the first time, rather than each time it is executed, as it would be in the case of machine simulation. This allows for the deployment of unmodified operating systems in a VM, with the virtualization system generating equivalent and high performing code transparently.

Another technique to overcome the non-virtualizability of CPU architectures, such as x86 and ARM, is para-virtualization [15]. Here, the requirement to present a guest of the virtual machine with an interface identical to that of a real machine is relaxed. The guest kernel is required to adjust to the changes in the machine interface by replacing sensitive instructions with equivalent code or hypercalls. Hypercalls relate to guest kernels and the VMM or the hypervisor, like system calls relate to processes and the operating system kernel. A virtualization-aware guest kernel may use hypercalls to call into the underlying VMM and access machine interfaces with equivalent functionality. The intrusiveness of these modifications can vary strongly, from the replacement of sensitive instructions to the replacement of sensitive operating system subsystems. VMware [18] coined the terms shallow and deep para-virtualization for these two strategies.

Although Shapiro contends that it is not para-virtualization [58], rehosting can be considered a form of deep para-virtualization. It definitely poses one extreme in a spectrum of intrusiveness of virtualization solutions. Rehosting [36, 45, 46] requires modifications of the guest to the extent that it can run on the kernel-user interface of a microkernel, providing threads, protection domains or tasks, and inter process communication. With the advent of the vCPUs [41] this microkernel-user interface, in fact, drew a little closer to the machine interface. Whether or not this technique can be considered para-virtualization, it shares with virtualization the ability to run multiple guest operating system kernels, which in turn can run complex setups of unmodified applications. This gives the user the impression of a virtual machine system. Sleek, virtualization-friendly interfaces make a good argument for this technique's security properties, such as the memory protection and isolation. The prototypical implementations in this work are based on L^{4}Linux, a Linux kernel rehosted onto the Fiasco.OC microkernel, which in fact implements vCPUs. Section 2.2.3 gives a more thorough introduction into this combination and into L4Re, Fiasco.OC's complimentary runtime environment.

Memory virtualization

So far, different virtualization strategies were discussed in terms of the machine interface in general and in terms of sensitive instructions in particular. A particularly interesting and performance-relevant aspect of the machine interface is the translation lookaside buffer (TLB), which is key to memory management and protection. The TLB allows for efficient translation from virtual addresses to physical addresses. By controlling the allowed translations, operating systems can implement protection domains with different mappings, thereby controlling whether or not memory resources can be addressed and therefore accessed by a thread executing in a particular protection domain. Because the TLB is a rather small associative memory, it can only hold but a subset of the translations of a protection domain. It acts as a cache for the page tables, data structures residing in memory and holding the mapping information of a protection domain. The page tables are consulted by the page table walker should the TLB be unable to comply with a request. The page table walker and the TLB together form the memory management unit (MMU). [2] Due to their role in memory protection, the page tables must be thoroughly protected from unauthorized access, which is why they are managed by the operating system kernel running in the privileged mode of the processor. When moved into a VM, the guest operating system kernel is no longer authoritative for all of the physical memory resources. Rather it shall only access those resources assigned to it by a higher authority, that is, the hypervisor. With respect to the page tables, this means that it can no longer be in control of the effective page tables, because this would allow it to create mappings to resources that are off-limits. To run an unmodified guest regardless, the technique of shadow paging was introduced [11], a technique whereby the guest populates its page table with mappings from virtual addresses to physical memory, or rather to guest physical memory, which is what the guest "perceives" as physical. These guest page tables, however, are no longer consulted by the MMU directly; rather, the hypervisor constructs the so-called shadow page tables holding the actual effective mappings between guest virtual and host physical addresses. Shadow paging concerns the proper maintenance of the shadow page table. To perform this maintenance, the hypervisor must become aware of when the active guest page table within a VM is switched; it must be aware of the current guest-physical to host-physical mapping of the VMs as well as changes to these mappings.

To populate the shadow page table, the hypervisor's page fault handler queries the guest's active page table and uses the knowledge about the guest-physical to host-physical mapping to create a valid shadow-page table entry. That is, unless it does not find a valid entry in the guest page table, in which case it injects a page fault into the VM. Accordingly, the hypervisor must modify the shadow page table when mappings are removed from the guest page tables. This is done by revoking the guest's access rights to the guest

[2]This refers to the x86 and ARM CPU architectures discussed here. Architectures with software loaded TLBs exist but are not considered here.

page tables. Subsequent accesses to the guest page table result in a page fault, which can be used by the hypervisor to track page table modifications. This technique allows full compatibility with existing operating systems; however, it is rather costly in terms of CPU cycle usage. Further, extra page tables are needed, thus increasing memory usage. The para-virtualization approach of the Xen hypervisor [15], also known as *direct paging*, relaxes the requirements on the machine interface. The guest page table is used directly, a technique obviating the need for extra shadow page tables. Obviously, the guest can no longer modify its page tables at will. Rather, it registers, through a hypercall interface, pages of memory for the use as page tables or page directories. The guest may modify these pages only through hypercalls, allowing the hypervisor to intercept and sanitize the mapping requests. An elaborate page type system is required to assure the invariant that a page either can be used as a guest page table or is writable by the guest but not both at the same time.

If the hardware architecture supports multiple levels of translation, one speaks of nested paging or second level address translation. With support for nested paging, the notion of guest physical addresses becomes an architectural entity. Guest page tables can be used directly by the hardware, and the intermediate result of the first level of translation is subjected to a second level of translation, which is under the control of the hypervisor. Hardware supporting nested paging also exposes a virtual page table base register (e.g. CR3 on X86 or TTBR0/TTBR1 on ARM); page faults are directly injected into a VM without hypervisor interaction, reducing the performance impact on the guests.

Device virtualization

The virtualization of CPU and memory is only one-half of the story. Computers that cannot communicate with the outside world are as useful as a brick and not even this in the case of virtual computers. Communication with the outside world requires peripheral devices, such as network adapters, screens, and keyboards. Other peripheral devices provide mass storage, sensory information, or special purpose data processing acceleration, such as the graphics processing unit (GPU), which is of particular interest for this work. Once again, there are different strategies by which virtual devices can be presented to a VM. Full virtualization of a device allows the guest to run an unmodified device driver. This requires a device interface identical to the interface of the actual physical device. A memory mapped IO interface, for example, can be emulated by intercepting virtual register accesses in the page fault handler. This technique can be very wasteful, as shown in the following example: Consider a device where most registers hold configuration data; only one register actually triggers an operation, such as data being send on a peripheral bus, when accessed. This access is the only time the VMM needs to be notified, while all other accesses could be held in memory until an action is required. The granularity of the paging mechanism, however, would not allow for selective register interception. With para-virtualization, the device interface can be augmented with hypercalls, potentially reducing guest VMM interaction dramatically. Naturally, this requires the installation of an appropriate device driver in the guest. Devices are very diverse in their requirements regarding latency, bandwidth, and CPU interaction. Occasionally, devices are abstracted to the extent that their services can be provided via a network connection. Device virtualization is discussed in more detail and by example of GPU virtualization in Section 2.4.

Fiasco.OC, L4Re, and L^{4}Linux

As hinted at before, this work is based on an operating system rehosting solution, composed of Fiasco.OC, L4Re, and L^{4}Linux. Fiasco.OC is a microkernel that provides the mechanisms for the construction of protection domains, the controlled communication between them, as well as scheduling. L4Re complements the microkernel with services for resource management and application support libraries. L^{4}Linux is a rehosted version of the Linux kernel. In the following, these components are introduced and illustrated by an example of how to bootstrap a system containing these components.

Fiasco.OC

The μ-kernel chosen as the basis for the virtualized smartphone is Fiasco.OC, a member of the L4-family [29, 48] of μ-kernels. Fiasco.OC abstracts from the basic CPU functionality, the timers, the interrupt controller, and the memory management unit. It performs scheduling and provides means for interprocess communication, but it leaves memory management as well as cache maintenance and device interrupt handling to the user space by exposing abstraction primitives accordingly. The “OC” suffix stands for **o**bject **c**apabilities, which is owed to the access model of Fiasco.OC. All abstractions and services are accessed through capabilities that must be held by the accessor. Here, the capabilities are kernel-protected pointers to kernel objects. Holding a capability constitutes sufficient proof of authority over the referenced object. However, before discussing the primitives exposed by Fiasco.OC, the debug features of Fiasco.OC need some attention as well, as they are going to be used for evaluation in Chapter 6. They pose an exception in more than one way: First off, there is a device driver for serial communication built into the kernel; it is used for the output of error messages and warnings. Further, if the extensive debug features are enabled at compile time, then a very versatile in-kernel debugger can be accessed through this serial line. The debug features also expose an interface to the user space, which does not succumb to the capability-based access regime. Another interface exposed to the user space is the kernel info page (KIP). This page is immutable by user-space applications and holds some general information about the platform. It is mentioned here because it also holds a field called `kclock`, which will be referred to later. This field is periodically updated, usually with a timer frequency of 1 KHz, providing a time source to user-space applications.

Fiasco.OC exposes eight different primitives to the user. They can be grouped as follows: *Threads* and *tasks* are the units of temporal and spatial isolation, respectively. The inter-process communication (IPC) primitives are *IRQ* and *IPC-Gate* for asynchronous and synchronous IPC, respectively. The *factory* allows for the creation of instances of all of the previously mentioned types of kernel objects, including itself, thereby providing a means to subject kernel memory consumption to a quota mechanism. Using the *scheduler* object, scheduling parameters of a thread, such as the CPU affinity and priority, can be manipulated. The interrupt control unit, *ICU*, allows for the registering of hardware interrupts. In addition, the *VLog* interface exposes the serial input and output device driven by the kernel, allowing for output, logging, and user input. The latter three objects are singletons and cannot be created using a factory object. Their protocol, however, can be implemented by user-space servers to provide similar services, such as a virtual ICU or an output multiplexer for interleaved output by multiple applications through VLog.

Instances of these kernel object types are exposed to the user by capabilities that can be invoked through a unified interface. A thread stores the parameters that it wishes to pass to the invoked object in its associated user thread control block or *UTCB*. The UTCB is a preallocated portion of the memory associated with each thread. It contains, among other items, sixty-three machine word-sized message registers holding parameters before and return values after capability invocation. Although threads invoke capabilities, they cannot own or hold capabilities. Instead, running threads are, at any time, associated with a task.

Fiasco.OC follows a hierarchical memory model, similar to the one described by Liedtke [48]. In Fiasco.OC, however, address spaces are represented by tasks. With the *map* and *unmap*[3] operations, memory and resources can be delegated and revoked to and from a task. Tasks also represent a capability space holding references to kernel objects. Unlike the memory model, the capability model is not hierarchical, that is, capabilities can be passed to other tasks using the *map* operation, but they cannot be revoked. A capability holder, however, can choose to destroy an object, provided its capability is powerful enough. Capabilities can have different qualities, such as the right to destroy the object that it refers to. A subset of these qualities can be passed on to the receiver of a capability when it is transferred, either by the above-mentioned map operation or, also possible, by trading it via a previously established communications channel, that is, an IPC-gate. Tasks are thus able to hold resources such as memory and kernel objects, and a thread can access the memory and invoke the capabilities held by the associated task.

To establish a communication channel, both communication partners must be in possession of a capability referring to the same IPC-Gate. One of the partners, the server, registers with the IPC-Gate, thereby informing the kernel that invocations of the IPC-Gate are to be forwarded to it, the registree. This registration, again, requires the capability to have the appropriate quality, which is not required by the client placing a request. Communication through an IPC-Gate is synchronous, that is, the server must be ready to receive whenever a client calls. Otherwise, the calling client blocks until the server becomes ready or returns with a timeout error

[3]The unmap operation corresponds to the flush operation as of Liedtke [48]—a grant operation does not exist.

code, depending on the timeout parameter specified by the client. Servers can provide arbitrary user-defined services through IPC-Gates. Notably, a server can, due to the unified interface, also mimic kernel-defined objects, such as the ICU or the VLog.

Asynchronous communication is possible through the IRQ object. Analogous to the IPC Gate, both communication partners need to be in possession of a capability to the IRQ object, and one communication partner, the one that wishes to be notified, registers with the IRQ object. Contrary to the IPC Gate, invocation of a valid IRQ object never blocks and always succeeds. The receiver is notified as soon as it is or becomes ready to receive. Also, the IRQ object is counting, which means that the receiver is guaranteed to be notified once for every time the IRQ object is triggered. Besides being used as a means for asynchronous communication between applications, the IRQ object serves as transport for device interrupts. For this purpose, a device driver registers with an IRQ object and binds the same object to a pin number of the ICU. In this arrangement, the IRQ object is triggered by the interrupt handler of the kernel, thus notifying the driver of an event.

The readiness to receive is a precondition for receiving a notification or a synchronous IPC call. Typically, a thread enters the state of readiness by issuing a system call. There is, however, another mode of operation that a thread can transition to, which expands the functionality of the thread—the vCPU mode. Once transitioned to the vCPU mode, the thread can still block and wait for IPC, but it can also indicate its readiness to receive, by a flag in the vCPU state area, which is an amendment to the UTCB. The flag works analogous to the interrupt flag of a physical CPU and, if set, allows for asynchronous interruption of the vCPU's execution. To facilitate this asynchronous interruption, the thread also stores pointers to a handler function and a stack in the vCPU state area. The same mechanism can be used to reflect exceptions and page faults into the vCPU; both can be selectively toggled on or off by flags in the vCPU state area. The vCPU mode also allows a thread to transition to a different task, a *secondary task*. As a task does not only represent an address space but also a set of privileges in the form of capabilities, this task transition allows the vCPU to temporarily give up its privileges. Upon an asynchronous notification, an exception, or a page-fault, the kernel also stores the architectural state of the vCPU to the vCPU state area and drops into the primary task, that is, if it happened to execute in a secondary one. Therefore, the vCPU mode allows for user-level scheduling and the transition of privilege levels, which strongly aids the para-virtualization of legacy operating systems.

L4Re

The eight kernel objects provided by Fiasco.OC are supplemented by a series of concepts, libraries, and user-defined protocols, which constitute Fiasco.OC's complementary runtime environment, L4Re. The following introduction to L4Re does not claim to be comprehensive, and it is limited to those elements of L4Re that are vital to understanding the following elucidations. It covers the principles of memory management, which, as was mentioned earlier, is left to the user space. Further, it will cover two protocols related to graphics output and user input.

The key elements of memory management in L4Re are the *dataspace* protocol and the *region manager*. A dataspace constitutes an abstract memory object managed with a given semantic. In L4Re, memory is typically allocated by requesting a dataspace from a dataspace provider. The application then associates a part of the dataspace with a part of its virtual address space by means of the region manager, which keeps track of the task's virtual memory layout. A memory access by the application then results in a page fault handled by the region manager. The region manager, knowing which dataspace corresponds to the faulting region, asks the corresponding dataspace provider for the actual physical memory pages and, subsequently, maps them into the application's address space. Memory allocation is but one typical way dataspaces and region managers interact; another is setting up shared memory regions. A dataspace can also denote actual physical memory, depending on the dataspace provider, which may or may not be contiguous, or a region of memory-mapped IO registers. Obviously, the latter case plays an important role in granting drivers access to devices' IO resources, while the former finds application in allocating immovable buffers for DMA or for the use as a framebuffer.

Another user-defined protocol of L4Re is the *goos* protocol. It is an integral part of L4Re's GUI architecture. The *goos* protocol uses dataspaces for passing framebuffers from a goos-server to a client. It adds meta information about the screen, views, and buffers. The relationships among

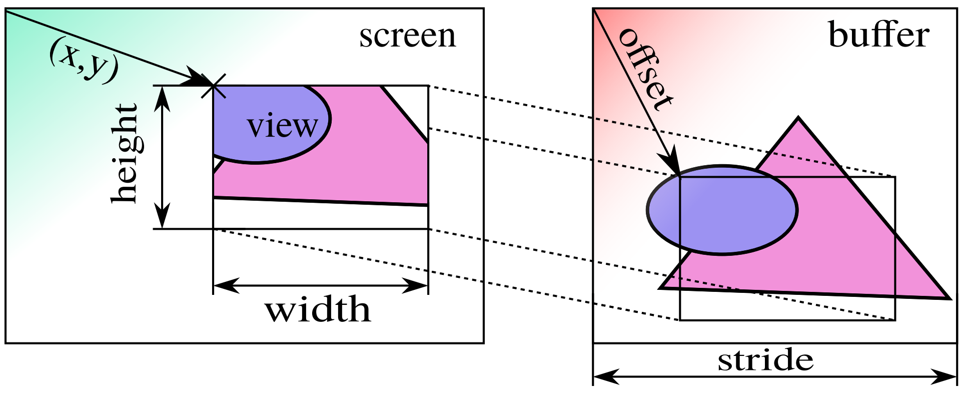


Figure 2.2.1: Goos protocol primitives. The screen is a physical output device, and a buffer is a memory region holding an output image. The view, defined by its width, height and position on the screen (x,y), denotes a region on the screen. A buffer can be attached to a view, indication that the view's screen region shall show a portion of the buffer, denoted by an offset and the view's dimensions.

these primitives are depicted in Figure 2.2.1. The protocol allows the handling of the physical resources, such as memory buffers, and one or more screens; the view is an abstract construct mapping a portion of a buffer to an equally sized portion of the screen. L4Re includes multiple implementations of the goos protocol: `Fb-drv` is a low-level display controller driver, which uses the goos protocol to expose the visible framebuffer to its clients. `Mag` is a window manager, exhibiting the most complete implementation of the goos protocol known to this author. While making use of a subset of this protocol, the purpose of this work required amendments to goos as will be discussed in Section 4.5. The *mag* window manager also implements the *Event* protocol, which is used to transmit input events. It uses IRQ-objects for asynchronous signaling in conjunction with a shared memory ring-buffer holding the event information. Events are encoded according to the Linux input event specification.[4]

Another important subsystem provided by L4Re is `io`. Evaluating a user-supplied[5] configuration, it groups resources of peripherals into so-called virtual buses (VBus). These VBuses can be assigned to other subsystems through IPC-gates, effectively granting access to the resources grouped in the corresponding VBus. Any user space driver in an L4Re-based system

[4]See `https://www.kernel.org/doc/Documentation/input/input.txt` and `https://www.kernel.org/doc/Documentation/input/event-codes.txt`.

[5]Here, the user is in fact a system integrator.

acquires access rights to MMIO resources and interrupts through `io`.[6]

Bootstrapping a minimal L4Re-based System

When Fiasco.OC boots, it expects a certain memory layout, in which it finds the binaries of the first two subsystems to run in its user space. One is the root pager, and the other the root task. L4Re provides a root pager by the name of `sigma0` and a root task by the name of `moe`. `Sigma0` owes its name to the concept of the hierarchical address space, where address spaces are denoted by the Greek letter σ, and the index zero denotes the root of the hierarchy [48]. The name `moe` is borrowed from the animated comedy series "The Simpsons", as are the names of other subsystems[7] of L4Re. Once the kernel has started these two subsystems, the user space takes control. Having all the privileges that the user space of Fiasco.OC can have, that is, the user memory by means of `sigma0`, the kernel memory by means of an unrestricted factory, and the singleton kernel objects ICU, VLOG, and Scheduler, the root task `moe` runs a helping subsystem, typically the Lua interpreter `ned`, to bootstrap the rest of the system. By evaluating a script that is provided by the system's architect and by harnessing the services provided by `moe`, `ned` begins setting up other subsystems. In doing so, `ned` ideally delegates just enough of the resources available to `moe` for these subsystems to accomplish their assignments, thus adhering to the principle of least authority. In all but the simplest "Hello World" setups, the first server started by `ned` is usually the resource manager `io`.

L^{4}Linux

Generally, L^{4}Linux [36] is a modified Linux kernel running on top of an L4 μ-kernel. L^{4}Linux comes in various flavors, depending on which μ-kernel it is running on. But, whenever there is a reference to L^{4}Linux in this work, the Fiasco.OC-based variant is meant. L^{4}Linux is modified to run on one or multiple threads in vCPU mode. The task of distributing the time quota available to these vCPUs among the guest's threads is left to L^{4}Linux.

[6] Depending on the platform architecture and maturity of platform support, this also goes for IO ports (x86) and individual general purpose input and output (GPIO) pins.

[7] A connection between the named characters and the functionality of the corresponding subsystems, however, is not known to the author.

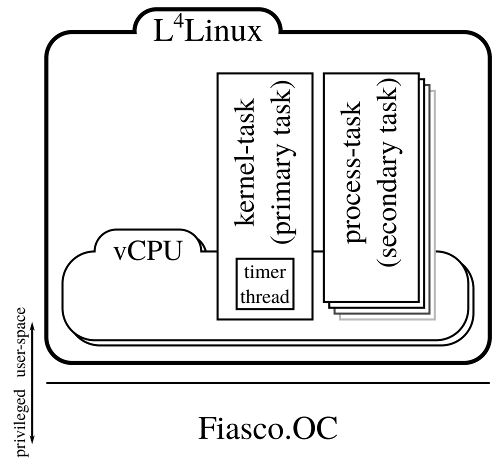


Figure 2.2.2: L^{4}Linux is a Linux kernel running in the user-space of an L4-family μ-kernel, here Fiasco.OC. An L^{4}Linux instance is a conglomerate of Fiasco.OC's primitives: One or more vCPUs which can transition between a kernel-task, which is the residence of the L^{4}Linux guest kernel, and several process tasks confining the guest processes. A timer-tread running in the kernel-task provides the instance with a time base, e.g., for scheduling.

Address space separation between the guest kernel and the individual guest processes is facilitated by tasks and the vCPU's capability to transition between them. The guest kernel resides in the primary, or kernel, task. Guest processes are implemented using secondary tasks. The guest kernel may instruct the vCPU to commence execution in a secondary task, and it regains control in the event of an exception, such as a page fault or a system call, or by an external event, such as an interrupt. These events are delivered to the guest kernel by the vCPU's upcall mechanism. Every transition between the tasks, naturally, causes an address space switch, which makes system calls and exception handling in L^{4}Linux inherently slow. In a way, L^{4}Linux uses tasks as shadow page tables. In this respect, the shadow paging approach used by L^{4}Linux differs from the way shadow paging was explained earlier. The guest page tables are not interpreted by the hypervisor, but rather by L^{4}Linux itself; shadow page table entries are made by the mapping mechanism of the task object. For example, when a page fault occurs while the vCPU is executing in a secondary task, a page fault is injected into L^{4}Linux by the upcall mechanism of the vCPU. This handler walks the guest page table. If it finds a valid entry, it maps memory resources into the faulting secondary task accordingly. Otherwise, it starts the regular Linux page-fault handling procedure, which, if successful, forwards the mapping to the microkernel as it resumes the execution of the faulting process. L^{4}Linux gets its memory portion, which it treats as physical memory, in an early boot phase by requesting it in the form of a dataspace from its assigned factory. This seemingly trivial detail will become important in the discussion of trust in resource management in Chapter 5.

There is basic virtual device support for L^{4}Linux in the form of paravirtualized device drivers: The `shm_net` driver, for example, allows point-to-point network connections between two L^{4}Linux instances. More related to this work is the `l4_fb` driver, a framebuffer driver using the aforementioned goos protocol as back end. This driver also provides an event interface based on the aforementioned event protocol. The reason these two seemingly unrelated functions have been consolidated into one driver is that L4Re's window manager `mag` and the screen multiplexer `con` combine both services in one communication channel.

GPU Driver Stack

Device drivers are both hardware-specific and operating system-specific. They communicate directly with a device and provide a mostly standardized interface for user applications. Thereby, device drivers abstract from the specifics of the hardware and provide application programmers with a unified interface for a certain class of device, regardless of the actual implementation of the device.

GPU drivers are no exception, but they are exceptional in their inherent complexity, which is required to provide the desired abstraction. OpenGL ES [7] is the standard API for GPU access on mobile handsets. This interface allows the application programmer to manipulate the abstract rendering pipeline that OpenGL provides. Shader programs written by the application programmer become part of the rendering pipeline. These programs are supplied in a language that is agnostic to the underlying hardware, the OpenGL ES shading language (GLSL). GPUs are designed to support an OpenGL rendering pipeline, but they do not understand OpenGL ES or GLSL as provided by the programmer directly. A GPU is a computing device with its own instruction set architecture (ISA). And while the mobile handset market is dominated by CPUs with ARM architecture, mobile GPU architectures are quite diverse (Mali, Adreno, Tegra, VideoCore, PowerVR, Vivante).

For providing the desired abstraction, a GPU driver stack provides many functions, some of which are typical for operating systems, such as memory management and scheduling. But some of the functions, such as compilers and linkers, are more commonly found in a development rather than a runtime environment. These compilers and linkers are needed to translate the abstract operations of the rendering pipeline, along with the shader programs, into an executable that can run on the GPU. As this executable consists of user-supplied code, it must be considered untrusted. At the same time, the GPU is a powerful DMA device. In order to confine the influence of this untrusted code to the system's memory, it must be subjected to some form of memory management.

Memory management has the following three aspects: The first is the accounting of physical memory. The second is the management of the virtual address space. Finally, there is the protection of physical memory resources. The systems addressed here, especially for the latter two aspects, are very tightly coupled. This is due to the use of memory management units (MMUs). MMUs translate bus requests issued by a bus master (e.g., the CPU) transparently from one address space into another by consulting a lookup table, the translation lookaside buffer (TLB). The TLB caches translations, and if a request cannot be complied—that is, a TLB-miss occurs—a page table that resides in the memory is consulted. [8] In performing this translation, the MMU both provides the bus masters with a virtual address space and provides for memory protection. The latter derives from the fact that a bus master cannot access physical resources for which no corresponding virtual address exists. These capabilities, virtual address space definition and memory protection, are quite naturally passed on to the software controlling the page tables.

Just as the CPU, the GPU in embedded devices and mobile handsets is a bus master, and its bus accesses are mediated by an MMU. So in the context of Android smartphones, the software controlling the page tables of this GPUMMU is the Linux kernel, specifically, the GPU kernel driver. The

[8]There are implementations of software-loaded TLBs, that is, upon a TLB miss an exception is issued, allowing a software routine to resolve the issue. The systems addressed here, however, perform TLB-miss resolution in hardware and thus have a fixed page table format.

GPU kernel driver offers a device interface that is by no means agnostic to hardware. This interface is used by a user-level driver, e.g., a shared library, which in turn provides the OpenGL/EGL API abstraction. It can now be delineated where each of the aspects of memory management concerning the GPU is situated. The physical memory is accounted for by the kernel. There are multiple interfaces provided by the Linux kernel, some of which have been introduced by GPU manufacturers, that allow user space applications to allocate physical memory resources for the use with the GPU (UMP, ION, dma_buff, ...). Some implementations may carve out a portion of physical memory for the sole use as graphics memory, but in principle, the user can allocate arbitrary amounts of physical memory for this purpose. The kernel merely tracks the ownership of the allocated resources. The user-space part of the GPU driver is entrusted with the layout of the virtual address space as "seen" by the GPU. But because of the entanglement of memory protection and virtual address space layout, the user-space program cannot be given direct control over the GPU's page tables. Instead, the GPU kernel driver offers an interface that allows attaching physical memory resources to the virtual address space of the GPU, a virtual address space that is private to the calling application. The kernel can exert memory protection by sanitizing theses attachment requests, thereby asserting that only those memory resources are attached that are genuinely under the control of the calling application.

Figure 2.3.1 shows the bookkeeping of the context layout in the user-space part of the driver. Additionally, there are the shader compilers as well as the linker. With these functional blocks, the user-space driver is fully capable of creating an executable with all related input data that can be executed by the GPU autonomously.

What remains is how a job is submitted to the GPU. This is dependent on the GPU's architecture. The GPU needs to know where to find all the parts of the job, such as shader programs, textures, and attribute lists, where to write the result and in which format. As the user driver assembles all of these bits, this information must be passed on to the kernel driver, which can in turn configure the GPU. Some GPUs have some sort of a command processor. In this case, all configuration data can be put into a command stream integrated into the job executable layout. This makes it sufficient for

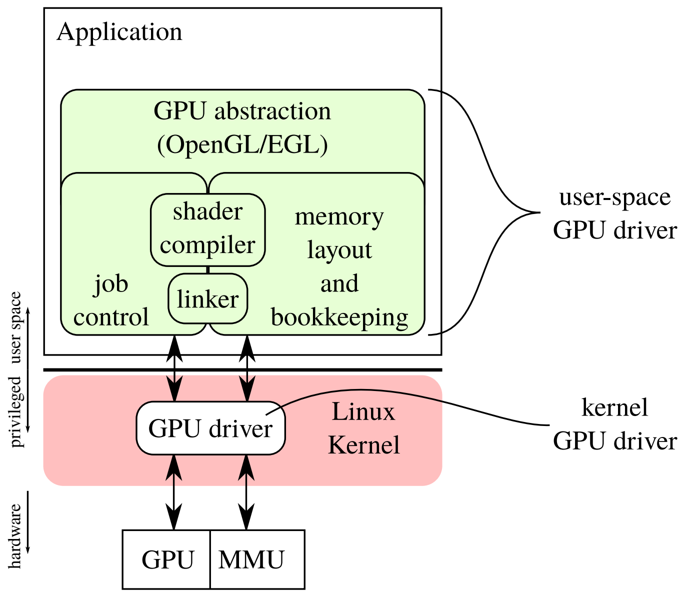


Figure 2.3.1: Components of a GPU driver stack found in a typical mobile handset.

the user driver to pass only the entry point of this command stream on to the kernel driver. Once the kernel driver has all the information, it can schedule the job to run as soon as the GPU becomes idle. It is in the responsibility of the kernel driver to enforce proper memory isolation. Therefore, it must assure that the correct context is active—that is, the correct page table is configured on the GPU's MMU—throughout the run-time of a given job.

GPU Virtualization

GPU virtualization is often categorized into front-end and back-end virtualization, depending on where the virtualization boundary is drawn. Typically, a virtualization scheme is referred to as front end if a high-level API serves as the virtualization boundary. Conversely, it is referred to as back end if the boundary is drawn at the device interface level. This is a very coarse-grained classification, and a closer look at existing GPU virtualization schemes reveals that each one has a rather unique way of placing the virtualization boundary and the driver components involved. It must be noted that none of the discussed GPU virtualization schemes address mobile-embedded GPUs, as this has not received academic attention until now.

VMGL [42] is a classical front-end virtualization scheme, with OpenGL as the virtualization boundary. It attempts to be independent of the underlying virtual machine monitor (VMM)—implementations for Xen and VMware exist—as well as agnostic to the underlying hardware; it supports ATI, Nvidia, and Intel GPUs. It deploys an OpenGL abstraction library in the guest, which marshals OpenGL commands and sends them via a network transport to the host. An X-Server extension communicates window size changes to the host; 2D output as well as user-input events are communicated using VNC [54]. One VMGL stub runs on the host for each guest application passing the marshaled commands on to the native OpenGL software stack. A VNC viewer interacts with this stub, composing the stub's 3D output with the client's 2D renditions. VMGL uses WireGL [38] as wire protocol. WireGL performs optimizations on the command streams to reduce network bandwidth usage. Still, while VMGL shows reasonable frame rates in 3D gaming benchmarks, it does so at the cost of massive CPU utilization and virtual network bandwidth consumption. VMGL goes to great length to support suspending guests. In doing so, it introduces quite some complexity into the guests, complexity required to track the OpenGL state. Being able to suspend a VM, however, allows the VM's migration. This is a great feature, but it comes at a price. To be able to migrate the VMs across hardware platforms, both source and destination must support the same subset of the OpenGL API. OpenGL extensions only available for one of the platforms can therefore not be supported on either. This limits the fidelity of the approach.

Blink [35] is a GPU virtualization scheme for Xen. Blink consists of a server and an API called BlinkGL, which is a superset of OpenGL. The Blink server runs as a user-space application inside a virtual machine that has direct access to the graphics acceleration hardware. It receives serialized BlinkGL commands from Blink clients via a shared memory-based transport. The Blink server executes these commands using commercially available OpenGL drivers corresponding to the underlying graphics hardware. The Blink scheme introduces stored procedures. These sequences of BlinkGL commands are used, e.g., to react to user interaction without the need to switch the context back to the corresponding client. They are translated into machine code using a Just-In-Time (JIT) compiler integrated into the Blink server. The JIT compiler is supposed to sanitize the BlinkGL command stream and perform global optimizations, exploiting "advanced knowledge of client behavior" [35]. Another optimization is concerned with the transfer of high-bandwidth data, such as for textures and framebuffer objects. Blink introduces a concept called Versioned Shared Objects (VSOs). This concept allows passing high-bandwidth data by reference, so that they can be "DMA'ed" directly into video memory. Using these optimizations, Blink can achieve quite impressive performance results, which are close to the native performance; provided that the client uses BlinkGL natively, allowing for the use of stored procedures. Legacy OpenGL applications, which are also supported by Blink, take quite a performance hit. Blink makes some interesting choices in overcoming typical problems with API remoting, such as the communication overhead via a wire protocol. Performance considerations aside, it suffers from a massively bloated TCB, consisting of a full GPU driver and OpenGL abstraction stack. In addition, there is the complexity of the JIT compiler, which raises questions about the expressiveness of BlinkGL and subsequently regarding the potential for an attack by injecting code into the Blink server.

Dowty and Sugerman [26] present another GPU virtualization scheme, SVGA3D. At first glance, it is not obvious whether to characterize this scheme as a front end or back end. Instead of using a wire protocol over a virtual network, as VMGL does, it emulates a virtual PCI SVGA adapter towards the guest. This adapter is supplemented by a new set of drawing commands introduced by SVGA3D. In a way, SVGA3D presents a new virtual GPU architecture, for which a complete driver stack is required within the guest VMs. So, from the perspective of a VM, this scheme looks like back-end virtualization. On the host side, however, the graphics adapter is emulated using a full OpenGL stack, which is a characteristic of a front-end virtualization. This makes the scheme agnostic to the API used on the client side, as long as a driver exists that can translate the user's API calls into SVGA3D commands. SVGA3D deploys an elaborate mechanism to share guest memory with the host's 3D driver. The authors claim: "This virtual DMA model has the potential to far outperform a pure API remoting approach like VMGL […]"—Dowty and Sugerman [26]. However, the benchmark results they present show frame rates that are inferior to those of VMGL when comparing native to virtualized frame rate ratios.

Smowton [60] presents Xen3D, which thrives on the merits of the Gallium [4] driver stack. This driver introduces a split into three components:

A state tracker, a pipe driver, and a window system driver. The state tracker provides an API abstraction, like OpenGL, and translates API calls into so-called pipe commands. The pipe commands, in turn, are interpreted by the pipe driver, which drives the underlying hardware. Xen3D conveniently uses these commands as a virtualization boundary, for they are agnostic to both the high-level API and the underlying hardware. It uses a shared memory transport to communicate these commands to the host side. For optimization, high-bandwidth data, such as textures, surfaces, and vertex buffers, are passed by reference, using safe handles that can be checked and translated into local pointers on the host side. The same goes for implicit pointers, such as line strides. Xen3D does not remote a high-level API, such as OpenGL, but rather an intermediate representation. This allows for good portability, as the virtualization boundary is agnostic to both API and hardware. Rather than using a full OpenGL graphics stack on the host side, Xen3D moves the state tracker, and therefore, the code that provides the high level API abstraction, into the guest, leaving only the pipe driver inside the TCB. This is in contrast to the other front-end virtualization schemes, which duplicate the API abstraction layer on the host side. An EGL-based compositor removes the necessity for a large windowing system, such as Xorg, on the host side, further decreasing the TCB impact. The author does not present any performance results because allegedly no pipe drivers for actual hardware were compatible with Xen at the time. However, he uses the size of an existing pipe driver for Intel's i915 graphics chipset and estimates the impact on the TCB by the pipe driver to be around 80,000 lines of code.

Like SVGA3D, gVirt [62] presents a graphics adapter to the VMs. But in contrast, gVirt emulates this graphics adapter only partially. The VMs can run an almost unmodified driver of an Intel Integrated Graphics GPU. The emulation is partial because native GPU commands are passed on from the guest driver to the hardware, thus avoiding emulation of the GPU rendering engine. Tian et al. [62] identified the command buffer and the framebuffer as the resources most critical to performance. Note that the authors use the term "framebuffer" for not only a buffer holding a rendered image but also, more generally, for denoting graphics-related buffers holding items such as vertex lists, textures, and shader programs. Therefore, the VMs are granted direct access to these resources, while MMIO register and page table accesses are emulated through trap and pass-through, and shadow paging, respectively.

Notably, there is only one shadow page table for all guest VMs, creating a single graphics address space. To share this address space among the guests, while at the same time providing isolation between guests, gVirt deploys various mechanisms: The address space is partitioned; the GPU commands submitted by the guests are sanitized for out-of-bounds memory accesses. These mechanisms call for a number of secondary mechanisms for performance optimization and protection. The authors observe, for example, that the partitioning leads to an inconsistent view of the graphics memory in the guest with respect to the host (or how the GPU "sees" it), which would require a mediator to translate the references in the command stream accordingly. To remove this requirement, the authors deploy memory space ballooning to create a congruent, even if holey, view of the graphics memory in the guests. Another measure stems from the command auditing. To prevent a time of check time of use (TOCTOU) attack on the command stream, the commands must be safeguarded against modification once the sanitizing commences. Two mechanisms are used to provide this protection, rooting in the way the commands were submitted. Commands can be submitted through a special ring buffer or as batches in plain memory. In the former case, the commands are copied into a shadow ring buffer, where they are inaccessible by the guest; in the latter case, the corresponding memory is made write-protected.

All of these mechanisms surely add complexity to the approach; and the TOCTOU prevention mechanisms seem to eat away some of the performance gains promised by having direct access to the command buffer. Eventually, all of these mechanisms have their roots in the design decision of having a single graphics address space, which is partitioned.

The authors make a case for having a split CPU/GPU scheduler. This author sees no incentive for scheduling CPU and GPU in unison. But apparently, they see a problem arising from this design choice:

> The split scheduling mechanism leads to the requirement of concurrent accesses to the resources from both the CPU and the GPU. For example, while the CPU is accessing the graphics memory of VM1, the GPU may be accessing the graphics memory of VM2. —Tian et al. [62]

This author does not see how this is a problem if the CPU cores and the GPU have individual MMUs. But the authors go on:

> gVirt runs the native graphics driver inside a VM, which directly accesses a portion of the performance-critical resources, with privileged operations emulated by the mediator. The split scheduling mechanism leads to the resource partitioning design [...]. —Tian et al. [62]

Here the author disagrees: Not the split scheduler calls for the resource partitioning but rather the fixation on a single graphics address space.

If the authors had allowed for one shadow page table per VM, at least, all of the various mechanism discussed above would be superfluous: The memory space ballooning would not be needed, as consistent views of the memory could be created by mapping the resources into the VM's and the GPU's space accordingly; command auditing would not be needed because there would be no foreign mappings in the shadow page tables, that is, any memory accessible would belong to the VM issuing the GPU commands; where there are no checks necessary, there can be no TOCTOU attacks. It would be the MMU that would perform the check at the time of use.

It was mentioned in the beginning of this section that gVirt allows running an almost unmodified device driver in the VM. In fact, the modifications are also due to the decision of having the common but partitioned graphics memory space, which, above all, allegedly sparked a modification in the hardware specification of the GPU in question:

> To support resource partitioning better, gVirt reserves a Memory-Mapped I/O (MMIO) register window, called gVirt_info, to convey the resource partitioning information to the VM. Note that the location and definition of gVirt_info has been pushed to the hardware specification as a virtualization extension, so the graphics driver must handle the extension natively, and future GPU generations must follow the specification for backward compatibility. —Tian et al. [62]

Secure GUI

A graphical user interfaces (GUI) constitutes the interface between a human user and a computer system, program, or application. GUIs are facilitated by graphical display devices for output and input devices, such as a keyboard, mouse, or touchscreen. Through the GUI, an application provides the user with information, such as text or imagery according to the applications designation, and through interactive elements, such as buttons or text fields, the user passes information to the application. But what constitutes a secure GUI?

It is easy to imagine that the information given to the user or passed to the application by the user is confidential. Possibly, this information is the content of an email or credentials for a web service. A secure GUI must ensure that this information stays confidential. Solving this goal by software, as attempted in this work for mobile virtual machines, has its limitations. Obviously, the information is presented on the screen to anyone who can gaze upon it. And, not so obviously, it has been shown that a technique called Van-Eck-Phreaking [28], by which the content of a computer screen can be recovered from the electro-magnetic emissions over a rather large distance, can be applied, though with limited range, to tablets [37]. Countering these kind of attacks is clearly out of the scope of this work. There are, however, user interface (UI) confusion attacks, such as phishing, and attacks on the availability, such as ransomware, that can be addressed by a proper GUI design.

Input events passing through the GUI also constitute authoritative actions, e.g., the setting and revoking of file permissions or the confirmation of an order. A secure GUI must ensure the integrity of input events. Ensuring the authenticity of physical key strokes, to the extent that the identity of the user is always assured, is out of the scope of this work. A proper GUI design, however, can prevent the forgery or injection of events by an untrusted third party, e.g., a compromised open compartment in the context of a virtualized secure smartphone. It can also provide the user with a trusted path to the TCB of the system, effectively countering availability attacks as they are performed by ransomware. Similarly, it must ensure the integrity of application output, preventing the forgery of information fed to the user by malware. The consequence of failing this integrity assurance is almost always the user being tricked into unintended actions, such as performing authoritative actions inadvertently, or giving away sensitive information, such as login credentials.

To be clear about terminology: In this context, the term *GUI design* is unrelated to the graphical design of user interfaces, which is also outside the scope of this work. To illustrate the meaning of the term GUI design, consider the path of a user-input event: Before an input event reaches an application, it first registers on an input peripheral that is connected to the computer through a bus system, such as USB[9] or IIC.[10] A corresponding bus controller notifies the CPU through an interrupt, whereupon the CPU runs a driver, which, in turn, queries the controller for input data. The user input now resides in memory and is easily accessible by the CPU, but the journey for the input event continues through more program logic, e.g., deciding which of the applications is the righteous recipient. It is this program logic in conjunction with the driver constituting the input path and the output path analogously, which is considered the GUI, the design of which is meant by GUI design.

The X-server in conjunction with the underlying kernel is an example of such program logic. It provides windows as abstractions of the screen and forwards input events to any program that requests them. Shapiro et al. criticize: "The most obvious security problem in X is the absence of policy of any sort"—[59]. And further: "X assumes not only that applications are cooperative, but that their actions reflect the volition of the user. In a world of increasingly hostile applications this trust assumption has become an unsupportable luxury"—[59].

Yee [65] formulates several principles for the input and output of secure systems. The most relevant for this work are the principle of the *trusted path* and the principle of *identifiability.* Regarding the trusted path, Yee states, "The user must have an unspoofable and incorruptible channel to any entity trusted to manipulate authorities on the user's behalf"—[65]. As an example, he presents the Ctrl+Alt+Del sequence of Microsoft Windows, which unspoofably triggers the display of the login prompt. The authoritative action performed by means of this trusted path is the identification of the user. After this action, the assumption is that all input represents the intent of the identified user and that this user is the recipient of the output. Any program started by the user's input is attributed to the user; therefore, the program inherits the privileges attributed to this user, and it is assumed that all programs share the screen gracefully and cooperatively. This is ex-

[9]`http://www.usb.org`

[10]Specifications available at `http://nxp.com`

actly the kind of trust model that Shapiro et al. call an “unsupportable luxury”. In contrast, they advocate a model by which the display server provides a trusted path to all applications, which in turn, run with individual sets of privileges. Their EROS Trusted Window System (EWS) [59] aims at providing a trusted path to every application. It does so by delivering input events exclusively to the window in focus. And, mandatory window decorations with unforgeable labeling as well as the prohibition of invisible windows mitigate, if not prevent, GUI confusion attacks [13, 17, 49, 50]. For the same purpose, Feske and Helmuth [31] introduce the X-ray mode into their Nitpicker window manager. A secure attention event, which just like the Ctrl+Alt+Del sequence, is never delivered to a user application, toggles the X-ray mode. When active, all but the window in focus are dimmed, decorations are displayed, and floating labels identify the origin of each window. Lange and Liebergeld [43], who investigated usability aspects of a secure GUI on smartphones, suggest a two-finger swipe gesture for this purpose. In all cases, the unforgeable labels fulfill the principle of identifiability, provided that a trusted boot process, in conjunction with a trusted loader, provides the labeling information [31]. The secure GUI, by design, must prevent applications from tempering with labels, especially their own.

In the context of the virtualized secure smartphone, the VMs and any native application[11] using the device’s display for a GUI are identified by a trusted bootstrapping procedure using cryptographic signatures for validation. These entities are to be identified by the labeling mechanism of a secure GUI, and to which a trusted path is to be provided. Applications inside the VMs are not under the scrutiny of this mechanism for lack of reliable labeling information. Mechanisms for labeling Android applications, which are built into the Android middleware, however, have been presented before [17, 30]. In a first pragmatic approach, the secure GUI of the secure smartphone was accomplished by the window manager `mag` (see Section 2.2.3). Like EWS and Nitpicker, `mag` features exclusive input event routing and window labeling and is a viable implementation of a secure GUI service provider. But, it is ill suited for the deployment on a mobile handset in several ways. First, the concept of window management has not caught on for mobile devices. Mobile applications nearly exclusively monopolize the

[11]Native applications are those that have been written for, and run directly on, the microkernel’s interface, rather than inside a VM.

screen. This is not a serious problem for the functionality, as `mag` is flexible enough to be configured appropriately. Leaving idle the many window management features implemented by `mag`, in this configuration, however, `mag` was unnecessarily complex and thus bloated the TCB. Second, the compositing technology used by `mag` is very demanding in terms of CPU cycles, to the extent that it neither achieves the required performance and fidelity nor meets the required power efficiency.

3 Threat Assessment

The GPU is a very powerful DMA-capable device, and it is well known that DMA devices pose a threat to systems, the security guaranties of which depend on strong memory protection domains [56]. Surely, the secure smartphone is such a system. Thus, if the computational powers of the GPU shall be harnessed for the sake of GUI efficiency, the GPU's DMA capabilities must be confined carefully. In the course of this chapter, a model of the GPU as an agent for copying memory content is presented, substantiating the rather unspecified threat of the GPU as a DMA device. Subsequently, this model is put to use on two bugs that existed in the GPU drivers of two successful smartphone SoCs deployed in the field, with the result of subverting the integrity of the entire system. Finally, the more subtle failings of one of the driver stacks are discussed, failings that potentially undermine the secure GUI principles introduced in Section 2.5. The chapter concludes with the lessons learned as well as the phrasing of design paradigms that shall be adhered to in the following chapters about the architectural design proposed in this work.

It is a well-known fact that DMA devices pose a threat to memory isolation and that they must be handled with great care [56]. What does that mean for the GPU being a DMA-capable device? The GPU's memory accesses are supposed to be tamed by an MMU, which is under the control of the kernel driver. In the course of this work, a bug was found in the GPU's kernel driver of a popular smartphone model, which allowed a malevolent user to gain control over this memory access regime. In an attempt to illustrate the consequences, a model of the GPU as a memory-copying agent was devised, which consequently, led to the design of an exploit.[1] The design and principle of the exploit is presented in Section 3.1. Shortly after the GPU-based privilege escalation attack was published [25], Clark [9, 21] published a bug in a different GPU driver that allowed the manipulation of the GPU's active page table with equally devastating effects on the system's integrity. In Section 3.2.2, Clark's findings are presented and explored further.

As it turned out, even if the buggy interface that allowed the first attack was configured as intended, the potential for exploitation would only degrade from privilege escalation to a privacy violation in that it leaked vital information. This is discussed in Section 3.3. The apparent fix to both the privilege escalation and privacy violation issue was to disable the buggy feature and to use one of the buffer-handling APIs that was mentioned earlier in Section 2.3. However, the UMP module which supplements the GPU driver, in particular, was found to be buggy in that it had too little entropy in the global handles used to identify buffers. This lead to a key logger that was evaluated by Fiebig et al. [33], and which is also discussed in Section 3.3. The chapter concludes with the statement of a threat model and the security requirements for the architectural design that this work suggests to which to adhere.

The GPU as an Agent to Copy Memory

For the sake of this discussion, it shall be assumed that a mobile GPU has unrestricted access to main memory. It is understood that the GPU can perform quite intricate computations from the realm of computer graphics on behalf of an application. But what makes the GPU a versatile tool for an adversary? This author aspired a model that was intuitive and simple to

[1]The exploit was implemented by Martha Piekarska as part of her master's thesis, and the design and evaluation was subsequently published in 2013 [25].

implement and powerful at the same time. The line of thought was simple: If a GPU could be used to display an image, e.g. a digital photograph, on the screen, it somehow copies the decoded color values from a region of memory to the framebuffer. If there is no transformation involved, that is, if the parameters such as the dimensions (width and height in pixels) and the coding of the pixel's color of the source image match those of the output region in the framebuffer, this amounts to an exact copy operation. Given that memory access is unrestricted, this allows an adversarial application to copy arbitrary information between portions of the physical memory, whether ordinarily accessible to that application or not. This notion of *identical scene rendering* is key to the adversarial power of a graphics processor. While more versatile GPUs may be leveraged by a sufficiently skilled graphics programmer to stage more sophisticated attacks, such as, e.g., scanning for key material, even the simplest graphics processor or bit-blitting engine capable of rendering the identical scene can be used for the attack described.

The rendering pipeline of OpenGL ES 2.0 works as follows: The input of a rendering program contains a grid of vertices, which constitutes the geometry of the scene to be rendered, where the vertices are points of that scene, described by a set of **attributes**. The semantic of an attribute is completely user-defined. However, one of the vertex' attributes is usually a position of some sort. In a first step, the geometry of the scene is fed into the geometry phase of the rendering pipeline. In this phase, the **vertex shader**, a user-supplied program, is executed once for each vertex. By interpreting the vertex attributes, the vertex shader program gives meaning to the attributes and at least derives a position on a two dimensional plane from them. The shader program may have additional output, which is stored in **varyings**. The output of the geometry phase is a set of polygons, the **primitives**, the corners of which are described by a position and an optional set of varyings. In the next step, the primitives are rasterized. That is, a rectangular section of the two-dimensional plane in which the polygons reside is sampled in a grid-like fashion to produce **fragments**. Thereby, the varyings are being interpolated according to the position of the fragment relative to the enclosing polygon's corner points. Before the result of the rendering pipeline can be written back to the output buffer, a color for each fragment must be determined. The second programmable stage performs this task by running the user supplied **fragment shader** program once for each fragment. This pro-

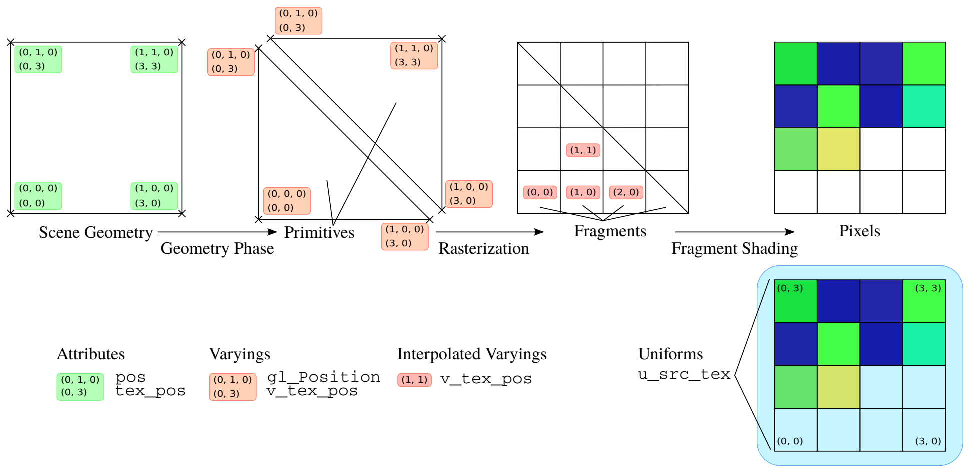


Figure 3.1.1: Copying a 64-byte memory region in the form of a 4×4 pixel texture using OpenGL ES 2.0. The figure shows how the input data—attributes and uniforms—propagate through the OpenGL ES rendering pipeline.

gram receives the interpolated varyings as input. There is one more kind of input to a rendering program, namely **uniforms**. Unlike attributes, which are vertex-specific, and varyings, which are fragment-specific, uniforms are literally uniform across all invocations of the vertex and shader programs involved in rendering a single frame. A typical uniform is a transformation matrix, which may be modified between two subsequent frames to change the perspective of the scene. Textures are also uniforms, which for example, may be sampled by the fragment shader.

To render the identical scene using OpenGL ES 2.0, one first needs a scene geometry. This geometry shall be a rectangle congruent with the view port, the portion of the scene that is going to be rendered. The top left diagram in Figure 3.1.1 shows such a geometry. This is given by four vertices with the corresponding attribute `pos`. The source buffer of the copy operation is given as the texture `u_src_tex`. For the fragment shader to be able to map the texture's color values correctly to the fragments, a second attribute `tex_pos` is given as input to the rendering pipeline. The second attribute is propagated by the vertex shader as a varying `v_tex_pos`, interpolated in the rasterization phase, and finally used for sampling the texture by the fragment shader.

This rendering pipeline may produce an output that looks like the input texture to the human eye. However, for it to perform an exact bitwise copy, some additional constraints must be applied. First, the output buffer must have the same dimensions (width and height) in pixels as the texture. After all, the copy would not be very exact if its size is not equal to the original number of bytes. Second, the pixel format of the texture must match the output format. A suitable format that is widely supported is `RGBA8888`. A pixel of this format is represented by a 32-bit word, every octet of which represents one of the channels **r**ed, **g**reen, **b**lue, and **a**lpha. And every value 0 through 255 of each octet is a valid code. The latter is important, because if there were unreachable bit patterns, this would not be a generic copy mechanism. Finally, both pixel and line stride must match, and to copy a continuous region of memory, they must be equal to the pixel width and line width, respectively.

The copy operation that was just described can be implemented easily using OpenGL ES. However, the OpenGL API, in conjunction with the EGL API, provides a means to allocate buffers for textures and output, thereby abstracting from memory handling details. In order to set the input and output regions to the exact locations, it is necessary for an adversary to work around these abstractions. While the identical scene rendering by itself can be expressed very generically, working around the memory abstractions is very implementation dependent. Here is the general problem:

Through the OpenGL/EGL API, the memory locations of buffers can only be specified through symbolic names generated and passed to the user at the time of their allocation. The abstraction layer, at some point, replaces the symbolic names with actual memory addresses. Thus, even if the GPU has unrestricted memory access, the attacker cannot simply specify the source and destination address of the copy operation through the API. Instead, either the attacker must acquire a symbolic name for the source or destination address, or he must patch the program before it is submitted to the GPU. From an isolation point of view, this is not a problem. The program is compiled and linked by the user-space part of the GPU driver, which means that patching it is as simple as any other memory access. Finding the location of machine word holding the address to be adjusted, however, may prove cumbersome, because the internal layout of the driver's data structures is hidden by the implementation, especially so if the driver is provided as a binary library, which is commonly the case. During the course of this work, appropriately patched programs were constructed, using two different techniques. For one set of experiments, an emerging open-source driver [6] was used, which was easy to adjust appropriately. For another set of experiments, the changing memory layout of the applications was observed through the Linux `proc` file system (`/proc/<pid>/maps`) during GPU program construction. This gave away where location of the artifacts that would form the GPU program within the attacker's application address space. Using the information about the structure of these artifacts from another open source driver effort [3], the address to be modified could be quickly found.

Unrestricted DMA through Driver Bugs

In the previous sections, it was established how a GPU can be used as an agent for copying data from one memory location to another. Naturally, if the GPU has unrestricted access to the main memory, this poses a serious risk, even if masked by user-level API. However, as discussed in Section 2.3, it can be assumed that the GPU's memory accesses are mediated by an MMU preventing illegitimate memory accesses—that is, if it is configured correctly. As stated in Section 2.3, the responsibility for the correct configuration of the MMU was located in the kernel driver of the GPU driver stack. These drivers can be buggy. In this section, two bugs are presented and examined, both prevalent in very popular smartphone models. The first one is rooted in an unsafe memory management interface provided by the kernel driver. This bug, which is present in the driver for the ARM Mali 400MP as found in the devices Samsung Galaxy SII and SIII equipped with Exynos4 series SoCs,[2] was found by the author. The second one is rooted in a misconfiguration of the GPU and allowed the user to submit unsafe commands to the GPU. It was present in the kernel driver of the Adreno GPU, which is part of Qualcomm's Snapdragon SoCs, found in smartphones of various vendors. This bug was discovered by Clark [9, 21].

Unsafe memory management interface

The bug discussed here is specific to a particular instance of a driver, namely, the Linux kernel driver for the ARM Mali 400MP, as found in the devices

[2]Samsungs Galaxy SII and SIII phones came with a variety of different SoCs. The outer casing being identical, the different flavors were only distinguishable by the model number. The vulnerable models with Exynos4 SoCs are GT-I9100 and GT-I9300.

Samsung Galaxy SII and SIII equipped with Exynos4 series SoCs. Throughout the rest of this section, it will be simply referred to as "the driver".

The driver provides its services to the user through the device node `/dev/mali`. Its interface can be loosely grouped into four categories. Considered *auxiliary* are the API calls that allow querying version of the driver, its API, and the underlying hardware as well as parameters such as the number of computational cores. The category of *development* constitutes API calls, such as for dumping the GPU's page table and for querying performance profiling information. Then, there is the *scheduling* category, which comprises calls for submitting jobs to the GPU and receiving notifications on job completions. Finally, there is the category *memory management*, which includes all API calls related to manipulating the GPU's virtual address space (GVA). The latter is of particular interest for this bug.

When an application opens the driver's device node, a session is created on behalf of the application and bound to the returned file descriptor `mali_fd`. Such a session includes a private page table for the use with the GPU's MMU. The driver's memory management API has three methods by which a user can allocate physical memory recourses and attach them to the GVAS. The latter amounts to a manipulation of the session's page table.

mmap The first method is the invocation of the Linux system call `mmap` on the file descriptor `mali_fd`. The prototype of `mmap` is as follows:

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

Invoking `mmap` performs three operations. It allocates a portion of physical memory with the size `length`. It maps this physical memory into the GVA at the location given by the `offset` parameter. It maps the same memory into the calling process' address space, the location of this mapping being returned to the caller. Note that the caller cannot influence the physical memory addresses, and all three operations appear to be atomic to the user. The effect of `mmap`, as usual, can be undone using `munmap`

UMP The Mali kernel driver is accompanied by a Linux kernel module called universal memory provider (UMP). It provides its services through the device node `/dev/ump`. Its API allows the allocation and, subsequently, the freeing up of physical memory resources. A thus allocated buffer can be referred to by using a numeric identifier. Using this Identifier as `offset` parameter to an `mmap` invocation on `/dev/ump`, a caller can map the buffer into its process' address space. The Mali kernel driver, in turn, offers an API call through the `ioctl` system call, which allows attaching UMP-buffers to a session's GVA. Again, the user has no influence on the physical memory addresses of the supplied buffers, as the buffers are referred to by a so-called "secure id".

MEM_MAP_EXT The `ioctl` call pair `MALI_IOC_MEM_(UN)MAP_EXT` offered by the Mali kernel driver constitute the third method. Its purpose is to allow attaching framebuffer memory for direct rendering. The argument of the `ioctl` call `MALI_IOC_MEM_MAP_EXT` includes the following triplet:

```
u32 phys_addr;
u32 size;
u32 mali_address;
```

It allows the mapping of a physical memory region of size `size`, starting at the physical address `phys_addr` into the GVA at the virtual address `mali_address`. This method is supplemented by a mechanism called `MEM_VALIDATION`, which checks the parameter `phys_addr` and `size` against hard-coded bounds. These bounds are to be set by the integrator of the driver to encompass the physical framebuffer.

The driver under investigation had the `MEM_VALIDATION` bounds set in such a way that it "limited" access to all of the available physical memory. This means that the `MEM_MAP_EXT` method could be used to map arbitrary regions of memory at arbitrary positions into the GVA, which amounts to the premise of unrestricted memory access by the GPU, as postulated in the previous sections. To show that this bug could be exploited using the identical scene rendering technique, a privilege escalation attack was implemented.[3] To that end, a payload (see Listing 3.1) was "rendered" into the text section of the Linux kernel, thereby patching the reboot system call to set the caller's user ID 0 (root) rather than rebooting the system. A fun

[3]The exploit was implemented and evaluated by Marta Piekarska as part of her master's thesis.

```
 0:     0xe52de004 |  push {lr}
 4:     0xe3a00000 |  mov  r0, #0
 8:     0xe59f1010 |  ldr  r1, [pc, #16] ; 20
 c:     0xe12fff31 |  blx  r1
10:     0xe59f100c |  ldr  r1, [pc, #12] ; 24
14:     0xe12fff31 |  blx  r1
18:     0xe3a00000 |  mov  r0, #0
1c:     0xe49df004 |  pop  {pc}
20:     0xc038dc44 |  prepare_kernel_cred
24:     0xc038ddd8 |  commit_cred
```

Listing 3.1: The payload rendered into the kernel text section in hexadecimal representation (left) and corresponding assembly language (right). It amounts to the equivalent C code: `commit_creds(prepare_kernel_cred(NULL));` (A similar version of this listing was previously published at ICISC2013 [25]).

fact: The texture in Figure 3.1.1 shows the exact payload used by the attack, interpreted as color codes.

Unsafe command interpreter

The second bug addressed here was found in the kernel driver of the Adreno GPU. Having a sound memory management interface that properly sanitizes the GPU's page tables is important, as was demonstrated in the previous section. But page table construction is only part of a proper MMU configuration. Another one is activating the page table at the right time and keeping it active as long as the GPU executes unprivileged code. In Section 2.3, this responsibility was bestowed on the privileged kernel driver for good reason. The particular driver under consideration, however, delegated this task to the GPU itself, likely, to reduce CPU-GPU interaction. This implies that the GPU can run in a controlled privileged mode, which is where the buggy version of the driver failed.

Adreno Mobile GPU Security Features

The Adreno GPU[4] features a command processor (CP). This CP can autonomously execute command streams issued to an in-memory ring buffer. The instruction set of the CP allows for indirect branches, which direct control flow to secondary command streams outside the ring buffer. Users would layout a GPU job in memory and then submit an indirect branch address to the kernel driver, which in turn, appends a branch instruction to the ring buffer accordingly.

Two security mechanism are deployed: A SYSMMU restricts memory accesses by the GPU to the bus system—that is, access to main memory as well as access to MMIO-resources, including that of the GPU itself, is mediated by the MMU. A *protected mode* mechanism allows configuring a set of GPU register ranges, rendering the specified registers inaccessible by the CP while the protected mode is activ. The rationale behind this seems to be that the CP needs to access the GPUs MMIO-resources to reconfigure itself. However, some registers have implications on the integrity of the system, and they must thus be safeguarded against manipulation by the user through an unaudited CP command stream.

Naturally, for the MMU to be effective, the correct page table must be loaded when a user-supplied job is started. Following the agenda of reducing CPU-GPU interaction, the driver's designers allowed the CP to perform the MMU manipulation. That is, the branch instruction that is appended to the ring buffer is prefixed by a series of commands performing a page table switch. What prevents the user to issue the same set of commands and switch to a set of page tables under the user's control is a second, "privileged" page table. ARM's SYSMMUs can handle multiple streams, each associated with a different device, and a different effective page table. The CP can switch the stream ID of the GPU, making the preconfigured secondary page table the authoritative page table for the GPU. The MMU's control registers are only mapped in this secondary page table and are thus unreachable for unprivileged code. The privilege transition, that is, the switch of the stream ID, was a command that could also be issued by unprivileged code. But

[4]The experiments discussed here where conducted on a Samsung Galaxy S4 with a Snapdragon 600 chipset (APQ8064T) featuring an Adreno 320 GPU. Judging from the genericness of the driver analyzed, the author has good reason to believe that the issues discussed here are not limited to this particular chipset; however, this was not explored further.

should the user issue this command inside a submitted indirect buffer, the privileged page table, where this particular buffer is not mapped, would become active. Consequently, the CP would not be able to fetch any further instructions. Instead, it would cause a page-fault and hang. Legitimate address space switches can only be performed through instructions on the ring buffer, which is mapped into the user and the privileged page tables alike.

Adreno mobile GPU exploitation

In 2014, Clark [9, 21] reported and exploited a flaw in the security mechanism of the Adreno GPU. Clark found out that, not only is the ring buffer mapped in both page tables, but also a couple of other buffers, one of which—called `setstate`—was writable by the GPU. He used CP commands to write code into the `setstate` buffer, which would activate a new page directory, one that was under his own control. Subsequently, he instructed the CP to branch into the `setstate` buffer, so that it executed the malicious code. Thus, memory protection was broken.

In response to Clark's exploit, a patch was released that made two modifications to the kernel driver: The `setstate` buffer was made non-writable to the GPU; and the MMIO-resources for switching the GPU's stream ID were subjected to the protected mode mechanism. The first measure effectively prevents Clark's approach. But in an effort to break the patched version of the driver again, two techniques were devised by the author to enable the exploit, even without the extra `setstate` buffer. These techniques will now be presented, even though no method has yet been found to circumvent the second measure introduced by the patch.

The first attempt started on the premise that the user cannot issue the command sequence required to switch the stream ID and subsequently modify the SYSMMU's translation table base register (TTBR) without causing a page-fault. So an alternative for the `setstate` buffer was required. But as for the patch, there were no writable buffers left to exploit. After all, there was still the ring buffer that was mapped in both page tables, and above all, it already contained the code for switching address spaces. The only problem was that the physical base address of the page directory that it would switch to was hard-coded into the command stream on the ring buffer; it was not writable by the user. So instead of changing the hard-coded base address on the ring buffer, which proved impossible, the attempt was made to bring the memory that it pointed to under the control of the user.

Assuming the user creates a context, issues a rendering job, and subsequently destroys the context; then a sequence on the ring buffer that activates the page table corresponds to the context the user created. But the context has been destroyed, and the physical memory backing the page directory of the context has been freed. This means that it has returned to the pool of unassigned memory pages. Subsequent memory allocations may yield the exact page. Once the page directory is under the user's control, the user can issue a job instructing the CP to indirectly branch to the old command sequence still lingering on the ring buffer, thereby activating the malicious page table. Two questions remain: How does the user get to know the physical address of the lingering page directory? How does the user know where to on the ring buffer to jump, exactly? The answer to both question lies with the identical scene copying technique. The ring buffer must be readable by the GPU; otherwise, it would not be able to execute the commands in it. [5] Thus, by patching an identical scene program to read from the ring buffer, the content of the ring buffer can be copied to a location accessible by the user applications. The application can then analyze the content of the ring buffer and find both candidates for lingering page directories and the corresponding jump offset on the ring buffer. Experiments[6] that showed the feasibility of the approach were performed with increasing success rate until an alternative approach was found.

The second approach was much less elaborate. Yet, it was only possible after learning more about the CP and its command set. Apparently, the CP had a command prefetcher, and indirect branches to user-supplied code would disable this command prefetcher. By performing a second indirect branch using a command that enables the prefetcher, it was possible to issue user code that would toggle the stream ID, thereby enabling the privileged page table, and then execute just enough prefetched commands without triggering a page-fault to modify the SYSMMU's TTBR and switch back to the now malicious page table.

Both techniques still relied on the ability to toggle the effective page table.

[5] Technically, executability does not imply readability, but the hardware under discussion does not make this distinction.

[6] The experiments were conducted by Felizitas Hetzelt.

However, this function was put under the reign of the so-called protected mode by the patch mentioned earlier. While the protected mode could be toggled by the CP with a command that, in principle, could be issued by a user, this command is only meaningful if the CP is not executing an indirect buffer. Experiments showed that as soon as the CP jumped into an indirect buffer, the command toggling the protected mode became a no-op, and subsequent attempts to access registers marked as protected would result in a protected mode violation and fail. User supplied code, however, is always executed as though supplied as an indirect buffer, and this even applies to the case where the user instructs the CP to jump back into the ring buffer.

Thus, the deployment of the protected mode, in conjunction with the indirect branch mechanism and the privileged page table, amounts to a privileged mode of execution, which was postulated as a requirement for delegating MMU configuration to the GPU. This protected mode mechanism separates the MMIO registers of the Adreno GPU into those that are critical to the integrity of the system and those that are not. The weakness discovered by Clark was due to a single register that was misjudged with respect to its criticality. The Adreno GPU, however, has many MMIO registers. Thus, for an outsider who has no access to the GPU's exact specifications, a moment of doubt remains as to whether the privileged mode of the GPU's command processor is now sound, or more misjudged registers lurk to be discovered by security researchers or, worse yet, attackers. The wary designer of systems may choose to treat the GPU as a computing resource running untrusted code and ensure that only rendering-related buffers are accessible to the GPU when it is executing autonomously. In this case, MMU access would be kept solely under the control of a trusted entity running on the application processor, where security mechanisms are well understood, and documentation is publicly available.

Unsafe Buffer Sharing

One of the main concerns of this work is the mobile GPU. It is key to rendering appealing GUIs and game content efficiently. But the GPU is not the only device involved in providing a pleasing user experience. Video decoders and cameras produce high-bandwidth video material, and there would be no GUI without the display controller sending video material to the screen. To the user space, all of these peripherals are individual devices. If two of them are to interact, buffers must be shared between the involved parties, e.g., when the display controller shall display a frame rendered by the GPU, or when one interacts with the user space, e.g., an application takes a snapshot from the camera. In Linux based systems, multiple interfaces[7] exist allowing the allocation of such buffers as well as their attachment to various devices and the user space. Special attention was given to two mechanisms already mentioned in Section 3.2.1, the `MAP_EXT_MEM` mechanism and the UMP interface.

Before revisiting the `MAP_EXT_MEM`, one of Android's permissions is introduced. In order to prevent applications from spying on other applications, the Android middleware entertains a permission called `READ_FRAMEBUFFER`. This permission is not available to third-party applications with good reason: It allows for taking screen shots, and thereby it exposes other applications' user output to the holder of the permission. The Android middleware, therefore, sets the access permissions to the framebuffer's device node (e.g., `/dev/fb0`) facilitating direct but very restrictive access to the framebuffer, in the belief that mediating access to the device node is mediating access to the framebuffer's content. But this is not so with the GPU driver investigated in Section 3.2.1. Considering the `MEM_VALIDATION` mechanism from Section 3.2.1, and assuming it was deployed as intended, that is, with proper range checking in place, an adversary application can still use the `MEM_MAP_EXT` interface. Clearly, the adversarial application does not have the luxury of unrestricted memory access any more. But it can still map the framebuffer into its GVA, even without proper permissions to the framebuffer's interface. In fact, checking mapping requests for legitimate ranges guides the adversary to the physical position of the framebuffer, which would be hard to guess otherwise. Once the framebuffer is successfully mapped, the technique introduced in Section 3.1 can be used to copy the framebuffer content into memory that is accessible by the adversary for further evaluation. Instead of merely copying the content, hypothetically, it could apply GPU-accelerated feature extraction on the way. This completely defeats the purpose of the `READ_FRAMEBUFFER` permission; user output can easily be

[7]OEMs tended to introduce their own frameworks: NVMAP (NVidia), CMEM (TI), PMEM (Qualcomm), and UMP (ARM). After consolidation, two frameworks prevailed: ION [66] and DMABUF [57].

acquired by malicious applications running on the same device. The implications are even more dire. By means of the GPU, any adversarial application has unrestricted access to the framebuffer. This means not only can any application read and modify the framebuffer content unchecked, but two applications can also set up a high-bandwidth communication channel. And if this was not bad enough, it further defeats the secure image-based trusted visual path mechanisms [17, 30], by which a user-chosen image provides proof of an application's trustworthiness.

The UMP mechanism was similarly flawed. The framebuffer itself is not the only buffer that holds user output. Textures, attributes, uniforms, and intermediate rendering artifacts must be stored somewhere in memory. And they must be accessible to the GPU; however, if these artifacts fall into the wrong hands, much can be inferred from them. Therefore, an application might expect to be provided with a mechanism that provides it with buffers solely accessible by itself and mappable into its GVA. For this purpose, and for buffer sharing, as discussed earlier, the Mali GPU's user-space driver used the UMP mechanism. This allows the allocation of buffers just as required, with the exception that the buffers are accessible to any application, being aware of a numeric handle generated upon allocation—the "secure ID". This would be sufficient if the secure ID was chosen from a large range and with an adequate source of entropy. Where the assessed implementation is concerned, the ID was chosen from a range of only 2^{24} elements, and worse yet—always the smallest unassigned ID was selected. Experiments showed that the IDs hardly ever exceed a value of 30. It was very easy to skim the ID space for valid entries and access the corresponding buffers. An application of this side channel was a key-logger. It was based on that, Android's on-screen keyboard typically draws the invoked letter on the screen as shown in Figure 3.3.1. This letter was placed into a separate buffer before further compositing. The key-logger sampled the ID space periodically. The appearance of a buffer with appropriate size in the ID space signaled a key press, and its content gave away the pressed key. Naturally, this method can also be used to spy on text documents or photos, in case they are held in buffers thus allocated. There are implementations that make the framebuffer driver UMP-aware and export the framebuffer by a UMP ID. While this was not verified experimentally, it is conceivable that in those cases, applications can circumvent the `READ_FRAMEBUFFER` permission discussed earlier by guessing

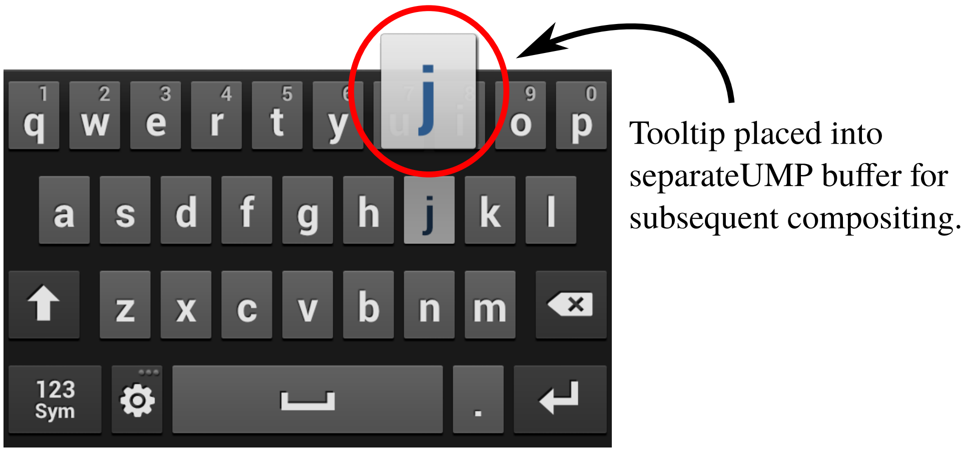

Tooltip placed into separateUMP buffer for subsequent compositing.
1 q 2 w 3 e 4 r 5 t 6 y u j i 9 o 0 p
a s d f g h j k l
z x c v b n m
123 Sym
.

Figure 3.3.1: Screenshot of an on-screen keyboard with activated letter tool tip (see annotation).

the UMP ID of the framebuffer. And just as with the `MAP_EXT_MEM` mechanism, these buffers pose unchecked implicit shared memory regions. Only this time, no detour through elaborate GPU programming is required; only using the plain kernel interface suffices.

Summary

The driver deficiencies discussed in this chapter are by no means architectural flaws inherent to mobile GPU driver stacks. All can be fixed, even if the fix requires a thorough overhaul of the driver stack on some occasions. Nevertheless, the lesson that can be taken away from this discussion is: Graphics drivers are complex. Interfaces are only as secure as the model they represent. If the model does not account for spatial isolation, or if functionally is carelessly added while not abiding by the model, e.g., for the sake of smooth integration, then the resulting interface is flawed with respect to security. It no longer provides the primitives required by higher layers to enforce a security policy.

Therefore, the following guidelines are framed and shall be adhered to throughout the following design chapters of this work.

Simple device model: To achieve the goals set for the secure smartphone, namely fidelity and high performance without compromising the isolation between the VMs, a device model is required, which is simple enough to reason about implications of memory isolation. The preferred model is that of an unprivileged computing device, much like the CPU when run in user mode, memory accesses of which are mediated by an MMU. From a software perspective, the mechanism enforcing the memory access protection boils down to the administration of page tables, which is well understood.

Minimal functionality: The kernel user split of the GPU driver stack already reduces the amount of complexity introduced to the kernel. In the context of the secure smartphone, however, some functionality, such as the dynamic buffer allocation, is not required to be part of the TCB. In consequence, the functionality can be reduced even further.

No redundant interfaces: To keep the complexity of the TCB at a minimum, the functionality implemented by the TCB shall be exposed by a single and redundancy-free interface.

No implicitly shared memory: Where shared memory regions are to be established, they must adhere to a higher-level policy. Two cases are of special interest: One, an untrusted party shall get access to a buffer that it does not yet control. This can only happen at the discretion of the policy-enforcing code of the TCB. Two, the TCB shall access a buffer on behalf of an untrusted party. In this case, the untrusted party must present some form of evidence of ownership, proving that it is already in control of the memory in question.

4
Secure GUI

The system protects me from being fooled.
—Yee [65] summarizing trusted path and identifiability.

A secure GUI provides a reliable means of communication between an application and a user, undisturbed by third subsystems outside the TCB. In Section 2.5, the principles of the trusted path and identifiability were introduced, and implementation examples were given. Furthermore, these principles were put into the context of mobile handset computing, and the existing implementation of the secure smartphone was introduced and criticized for lacking performance, fidelity, and power efficiency. This chapter develops a secure and compartmentalized input and output system design. First, the output and input path are separately discussed in Section 4.1 and Section 4.2, respectively. It follows an intermission on compartmentalized low-level drivers in Section 4.3, and on separated policy decision-making code in Section 4.4. Before this chapter concludes with a summary, the prototypical implementation is presented in Section 4.5.

Framebuffer Handling

One goal of this work was to replace the ponderous and overly complex framebuffer compositing of the window manager `mag` (see Section 2.2.3 page 21 and Section 2.5) with a lightweight and well-performing solution. This cannot be said without so much as a smirk, as where it comes to window managers, `mag` is pretty small. But it is what it is, a window manager; therefore, it provides more functionality than needed in this context. Nevertheless, the security properties of `mag` were to be upheld. As such, there must be no crosstalk in the output path leaking output between clients. Excessive CPU cycles consumption, due to spurious copying of framebuffer content, was to be avoided at all cost—short of sacrificing security properties. The following section will serve as an introduction to the underlying hardware model assumed, upon which the suggested architecture is to be built. In the course of discussing the output path security in Section 4.1.2, the framebuffer interposition techniques, compositing, mapping, and referencing, are introduced and their pros and cons are discussed. The section concludes with a discussion of how unforgeable labeling can be integrated into the approaches. The implications of the interposition techniques on the performance optimizations double buffering and hardware compositing are discussed in Section 4.1.4.

Hardware Model

The hardware components involved in the output path are depicted in Figure 4.1.1. These are the central processing unit (CPU), the display controller (DC), and the main memory, which is random access memory (RAM). A bus interconnect allows the bus masters, here the CPU and the DC, to access memory mapped resources, such as the RAM and the DC's control registers, through a unified address space, the physical address space. The CPU's bus access is mediated by an MMU. Moreover, a generic interrupt controller (GIC) receives asynchronous notifications and interrupts the CPU's execution. The display controller is of particular interest. It periodically reads a region of the RAM, the content of which it interprets as a matrix of color values, which it, in turn, sends to a connected display device. Thereby, it makes the digital representation of an image visible to a human user. The display controller itself has a memory-mapped interface through which an-

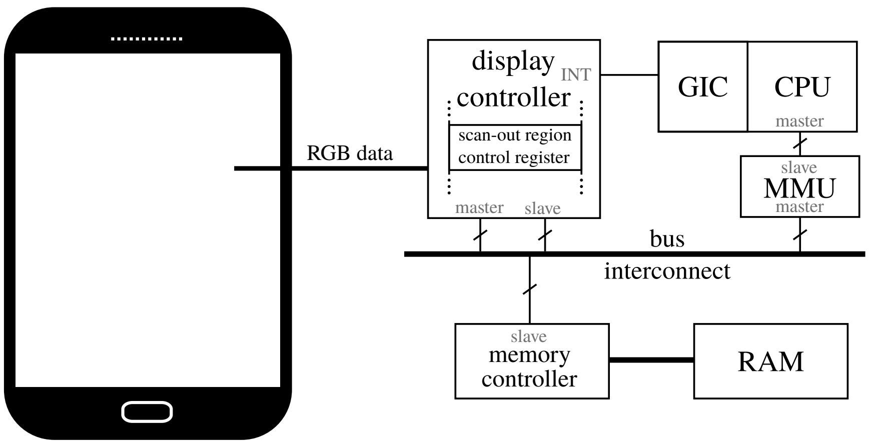


Figure 4.1.1: Hardware involved in graphic output. The display controller reads data from memory through its master interface, converts it accordingly, and sends it to the display device (left). The display controller is controlled through its slave interface by another bus master, here the CPU (right).

other agent, such as the CPU, can control its operation. This model assumes that, through this interface, the region of memory that is periodically read by the DC—the *scan-out region*—can be freely chosen. Moreover, it is assumed that the display controller can have more than one scan-out region, the content of which it can combine to produce output on the display device. Each scan-out region may need to be continuous in physical memory, a requirement that can be relaxed if memory access is mediated by an optional MMU.

Output path security

In the previous section, the scan-out-region was already introduced. Before exploring the design space of the output path, a couple of terms and concepts need to be introduced: the *framebuffer*, what it means if the framebuffer is *visible*, as well as the *on-screen* buffer. To facilitate a GUI, applications need to render meaningful graphical output. One such rendition is called a *frame*, and a memory buffer storing such a frame is called *framebuffer*. If the physical memory backing such a framebuffer coincides with the scan-out region of the display controller, the framebuffer is considered *visible*. In contrast, the *on-screen* buffer is thought of as holding the output of an application, which—or the content of which—may be made visible eventually, either by

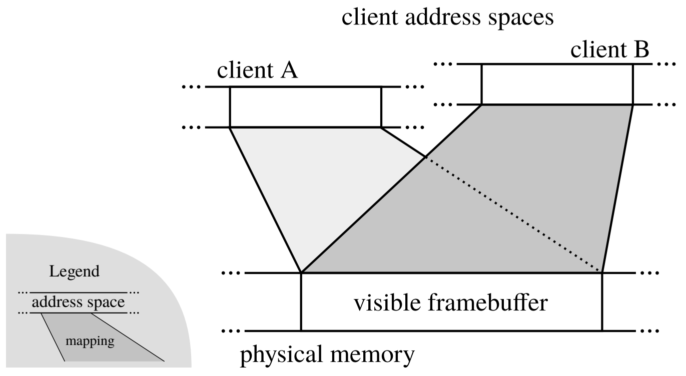


Figure 4.1.2: Naive sharing of the visible framebuffer between two clients. The visible in physical memory (bottom) is mapped in the virtual address space of client A and client B at the same time.

changing the scan-out region of the display controller, thereby making an on-screen buffer a visible framebuffer, or by copying its content into the actual visible framebuffer.

The most naive approach of sharing the visible framebuffer among several VMs is to give all VMs access to it by mapping it into the VMs' address spaces, as depicted in Figure 4.1.2. Clearly, this allows each VM to violate all of the security requirements postulated for a secure GUI in Section 2.5. The parties could spy on each other's output, as well as arbitrarily overwrite it, violating the confidentiality and integrity requirement. There is no way for the user to tell which party is responsible for the output, which is a deficiency undermining the identifiability principle. Leaving all security concerns aside, this would only work if all parties would cooperate, gracefully agreeing on the assignment of the framebuffer resource.

The given constraints of this work, however, dictate that the clients are mutually distrusting and potentially hostile. Therefore, some form of coordinating instance (CI) is required to orchestrate the usage of the visible framebuffer by multiple tenants. There are multiple strategies for the CI to accomplish this task:

Compositing

In the compositing scheme, every client, e.g. a VM, has its own private on-screen buffer. This buffer is only accessible to the corresponding client

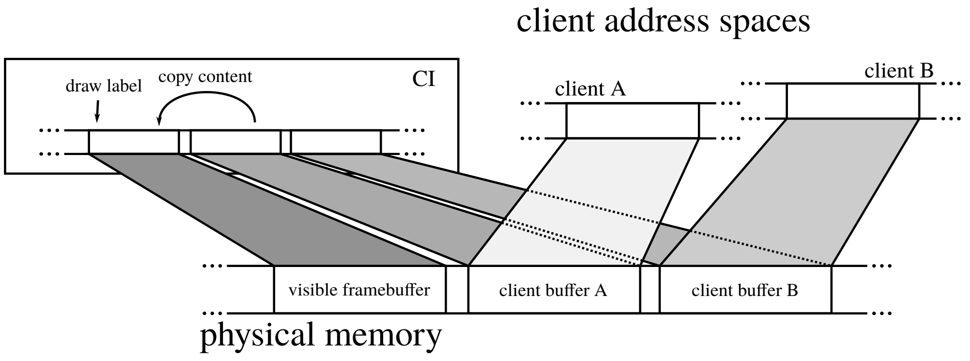


Figure 4.1.3: Compositing approach for framebuffer interposition. The visible framebuffer is only accessible by a coordinating instance (CI), as are the private framebuffers of the clients A and B. Accordingly, the CI decides which output is visible by copying the private framebuffer contents. It also draws a non-forgeable label, indicating the decision to the user.

and the CI as depicted in Figure 4.1.3. The CI may now choose to copy the content of an on-screen buffer into the visible framebuffer. Thereby, the CI can decide which client's output is currently visible. This method gives the CI a lot of freedom where to place the clients output on the screen, possibly scaling, or otherwise transforming, the content. The drawback of this method is, however, that a large amount of data needs to be copied by the CI. In addition, every change in the on-screen buffer needs to be reported to the CI. If the clients are virtual machines, the consequences of the latter differ depending on the mode of operation that the VM assumes. If, supposedly, the framebuffer were thought of as being visible, then every write to the framebuffer would have an immediate effect. No further actions would be required in the non-virtualized case; the driver might not even be aware that the framebuffer has changed. If moved to a VM, the same driver would need to report these occasions to the CI. But it cannot, because it is oblivious of them. Consequently, the CI would need to copy the client's buffer periodically, which leads to spurious CPU and bus activity whenever no actual updates have occurred. A modification of the client VM's user space, to that it reports updated regions of the on-screen buffer, could reduce the copying overhead; but it requires the client's cooperation. The compositing approach is performed by `mag` in the original implementation of the virtualized secure smartphone. Moreover, the most modern window servers use compositing, usually GPU-accelerated; this, however, requires an OpenGL stack in the TCB, thus bloating the TCB drastically.

Mapping

Besides copying framebuffer content, it is possible to give the clients exclusive, but temporary, access to the visible framebuffer [23] through the memory protection mechanism, the MMU. Here, the on-screen buffer can coincide with the visible framebuffer. This does away with copying and obviates the need for reporting updates. Transforming the client's content, however, is no longer possible. The implications of the complexity of implementing such a scheme depend on the encompassing system—especially when virtual machines are involved. Access restrictions in this scheme are eventually enforced through page tables. When revoking access to the visual framebuffer, the CI must remove all entries that refer to this framebuffer memory from the client's page table(s) as well as invalidate the corresponding TLB entries. In the context of virtual machines, this requires two different operations, depending on the hardware's support for virtual guest memory. If nested paging is supported, it suffices to remove the entries referencing the framebuffer memory from the client's host page table, the one translating from guest physical to host physical addresses. If shadow paging is used instead, the framebuffer memory mappings need to be tracked and removed from all of the client's shadow page tables upon revocation. To assign the visible framebuffer to a different client, the CI reinstates the formerly removed page table entries, or it lazily modifies memory management structures and lets a page fault handler do the work. While a client would most certainly notice that access has been revoked—because it would receive page faults upon its framebuffer access attempts—it needs to be notified when access to the visible framebuffer has been restored.

Notably, after the switch, the visible framebuffer still contains content drawn by the previous owner. To prevent the leakage of information, multiple strategies are conceivable. The visible framebuffer could be blanked before assigning it to the new owner. This leaves the screen blank until the new owner chooses to redraw the screen; triggering a redraw requires the client's cooperation. A slight modification makes this scheme transparent to the clients and prevents information leakage. Instead of revoking access to the visible framebuffer, access can be switched over to a shadow framebuffer, which is private to the client. Figure 4.1.4 depicts this scheme. The client can thus keep rendering, even though its renditions are not being displayed.

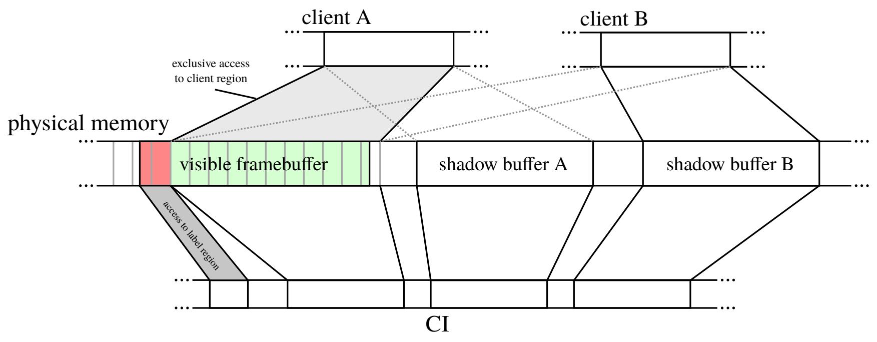


Figure 4.1.4: Mapping approach for framebuffer interposition. Client A has access to the client region of the visible framebuffer while Client B can render only into a shadow buffer. The CI can control the mappings and is able to swap this arrangement. It has access to all buffers, allowing it to update the visible framebuffer upon a switch. A label is drawn by the CI through framebuffer partitioning (see Section 4.1.3).

To prevent information leakage, the visible framebuffer can be overwritten with the content of the shadow buffer, which holds the latest output of the client being switched to. This strategy is used by Cells [23], where it revokes and remaps the framebuffers to alternating domains.

Referencing

In the previous schemes, the visible framebuffer resides at a static physical location. If this premise is relaxed and if every client has a private framebuffer, the CI can simply make a client's framebuffer visible by reconfiguring the scan-out region of the display controller (see Figure 4.1.5). For a private buffer to be made visible, however, it must be backed by a continuous memory; that is, unless the display controller's memory accesses are mediated by an MMU, which would allow for scattered buffers as well. Neither does this scheme incur any copying overhead, nor does it suffer the complexity that comes with revoking and reassigning memory access rights. Still, it is completely transparent to the clients. Implicitly shared memory, if time-multiplexed, as the visual framebuffer is in the mapping approach, where it needs special handling such as blanking, is not required. Another noteworthy observation is that the CI is not required to have access to the private framebuffers. Thus, in a way, a separation into control and data plane is achieved. This means that it can no longer violate the confidentiality

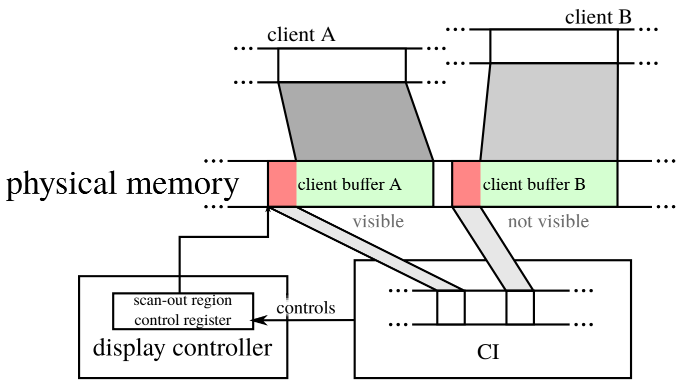


Figure 4.1.5: Referencing approach for framebuffer interposition. Clients A and B have exclusive access to their private framebuffers. The CI can only make one or the other framebuffer visible by manipulating the location of the scan-out region of the display controller. Labels are added through framebuffer partitioning (see Section 4.1.3).

requirement if compromised. It could, however, still violate the integrity requirement, e.g., by switching the scan-out region to a buffer accessible to the CI. A further separation of the CI into a low-level driver and a policy enforcing subsystem—a framebuffer switcher as it will be called later—could even mitigate this risk by further reducing the complexity of the low-level driver, which, in turn, restricts the space from which the framebuffer switcher can choose the visible framebuffer.

Labeling

So far, this section's concern was only with the confidentiality and integrity of the output path. However, an integral part of a secure GUI is the identifiability concept. To that end, the origin of the frames currently visible on the screen must be indicated to the user. As discussed earlier, it is assumed that the screen is used exclusively by one tenant at a given time. So this indication may be performed with specialized hardware. For example, a secondary device, such as a 7-segment display or a multicolor LED, could serve the purpose for a small number of tenants. If no such device is available, or if the number of tenants is extensive or very dynamic, the indicator must be drawn on the screen. In this case, care must be taken that the indicator cannot be forged by a tenant that can also modify screen content. Two established methods exist: The label can be displayed at a given and immutable position on the screen where no tenant is allowed to draw. Or,

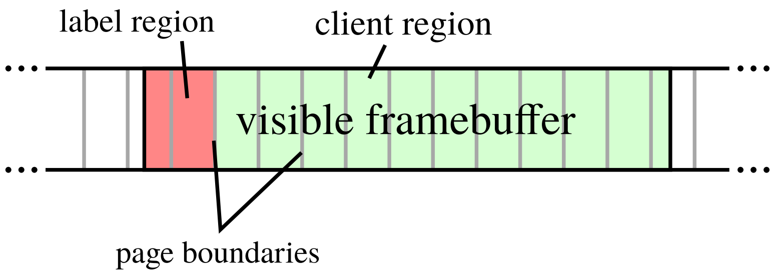


Figure 4.1.6: Partitioning the visible framebuffer on a page boundary into label region and client region

the label is displayed at a given time chosen by the user and is unpredictable by a tenant. [1] In any case, the label drawn must reflect the state of the output routing. This means that the label must be drawn either by the CI or by a trusted third using the CI to enforce an output routing decision. In the following, it will be discussed how a label can be drawn efficiently by the CI for the designs discussed in the previous section.

Drawing a label in the compositing approach is straightforward. During the copying of the client's screen content, the CI can simply alter a portion of the screen to display a label indicating the content's origin. It is not so easy in the context of the mapping scheme, where the CI can no longer reliably draw a label of origin onto the client's rendition. If it tried to alter the screen content, there would be a race between the client's framebuffer modification and the CI's label drawing. The "finish line" of this race is the scanning of the visible framebuffer by the display controller. While the odds can be changed in favor of the CI by syncing drawing to the end of the vertical sync gap,[2] there remains an uncertainty. This uncertainty may lead to flickering, if not to forgery; of which the latter is a plain violation of the secure GUI requirements. In the context of this technique, the said uncertainty can be removed by partitioning the visible framebuffer into a label region and a client region. The buffer must be aligned in memory so that the boundary of the partitions coincides with a page boundary[3] as depicted in Figure 4.1.6.

[1]Compare X-ray mode [31].

[2]Historically, the vertical sync gap is the period of time in which the electron beam of a cathode ray tube (CRT) is reset vertically to the beginning of the screen, e.g., from bottom to top. The vertical sync gap still exists because screen content is still read and presented sequentially, and it marks the time span between reading and transmitting two successive frames to the screen. The subject of VSYNC will be taken up again in Section 4.1.4.

Exclusive access can now be delegated to the client's region alone (see Figure 4.1.4), rather than to the full buffer removing the possibility of a drawing race and thus preventing flickering and, more importantly, forgery.

This method of drawing a label restricts the position and shape of the label. While using compositing, the label could be positioned anywhere on the screen. It is now restricted to being bar-shaped at one of the screens edges: top, bottom, left, or right, depending on the framebuffer's geometry.

The referencing scheme raises the same challenge to labeling the output as the mapping approach. Once the display controller's scan-out region is configured to scan the client's framebuffer, there is no more intermediation by the CI. Any attempt to add a label to the visible framebuffer yields the same race as in the previously discussed mapping approach. The same framebuffer partitioning technique can solve this problem; but in contrast, each private framebuffer must be partitioned in this way. This yields the positive side effect that the label needs to be drawn only once, rather than on every switch with the cost of slightly increased memory consumption (compare Figure 4.1.4 and Figure 4.1.5). The same restrictions apply for the shape and position of the label.

It was already mentioned that a system MMU, mediating the display controller's memory access, allows framebuffers scattered in physical memory. It would make the framebuffer partitioning more flexible because it would allow combining private framebuffers with a single label buffer. The page boundary alignment requirement remains, however, and with that the shape and position restrictions. If the display controller supports multiple scan-out regions (or overlays), as suggested in Section 4.1.1, the label drawing becomes more flexible. Depending on the capabilities of the display controller's overlay feature, the label's form and position can be chosen more freely. Support for an alpha mask or channel, which holds the opacity level of every single pixel of a frame, allows for arbitrarily shaped labels at arbitrary positions on the screen. Also, the buffer holding the label's bitmap can be chosen independently from any other framebuffer, as depicted in Figure 4.1.7, which lifts any possible alignment constraints, such as the one imposed by the framebuffer partitioning technique.

[3] Rumor has it that at least one version of the screen multiplexer `con`, which is a module of L4Re, has used this framebuffer partitioning technique.

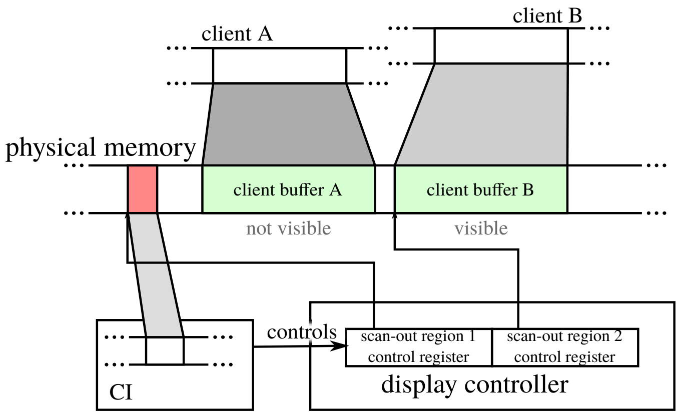


Figure 4.1.7: Referencing approach for framebuffer interposition with hardware overlay for labeling. The clients have access to their respective framebuffers. The CI has access to a label buffer and controls the scan-out regions of the display controller. Label region and client region are combined by the display controller to form the visible output.

Performance optimization

In the previous section, the design space of the output path was discussed with an emphasis on security and identifiability. The driving forces in this exploration, however, were performance and versatility. This section addresses the performance optimizations already existing and commonly used in contemporary smartphones. In order to build a competitive design, these optimizations must also be discussed in the context of the aforementioned design space. The first technique covered is double buffering, which, as can be seen later, has quite an impact on the prototypical implementation. The second technique is the use of framebuffer overlaying as a form of hardware support for framebuffer compositing.

Double buffering is a technique used to prevent rendering artifacts. Rendering artifacts occur when the display controller reads an image from the framebuffer, which the renderer has not finished rendering. A typical artifact is "tearing", which looks like a tear across the screen.[4] It occurs when the part on one side of the tear has already been updated while the other still holds an older rendition. To prevent these artifacts, the renderer could sync itself to the vertical sync (vsync) gap. This gap is the period that begins

[4]There can be other artifacts, depending on the workings of the render. If, e.g., the renderer is tile-based, such as the GPU discussed in the next chapter, the rendering artifacts may assume a checkerboard pattern.

after the display controller has finished reading one frame and ends when it starts reading the next. However, double buffering is commonly deployed if the duration of rendering is too long or unpredictable, in other words, when it cannot be guaranteed that rendering finishes before the vsync gap ends. As the name implies, double buffering makes use of two[5] buffers. One is an on-screen buffer; the other is a back buffer. Rendering is only ever performed into the back buffer, while the on-screen buffer can be scanned by the display controller. Once rendering into the back buffer has completed, the display controller is notified and instructed to scan the back buffer, which thereby becomes the on-screen buffer. Some display controllers support this technique, in that they allow two buffers to be configured; the active one—the one that is being scanned—is to be selected by setting a bit or an index in a control register. Hypothetically, if this change in configuration were to take effect immediately, it would quite naturally result in tearing artifacts, which were supposed to be avoided. Thus either the display controller is smart enough to postpone the effect until after the next vsync gap, or the controlling software must synchronize the reconfiguration of the display controller to the next vsync. In any case, there must be a back channel, by which the renderer can be notified that the back-buffer/on-screen buffer switch has completed. Otherwise, the renderer might start clobbering a framebuffer that is still actively scanned, causing visible artifacts.

The double buffering technique can be applied in all of the schemes discussed earlier. Similarly, the back channel requirement applies to all of them, even to the compositing scheme, where the CI assumes the role of a virtual display controller scanning the client's framebuffer. As a side effect, the compositing CI can use the buffer-swapping request as an indicator that updates have occurred, which in any case must quite naturally be forwarded to the CI; this allows the CI to avoid spurious copying. In the framebuffer-remapping scheme, what has been discussed earlier simply applies to two framebuffers that are visible alternatingly. The referencing scheme, in fact,

[5]There are other variants with more than two buffers, such as triple buffering. These variants also avoid rendering artifacts, but they also serve a different purpose. It allows the renderer to render into alternating back-buffers as fast as possible. As a result, when the next vsync gap ends, the most recent rendition can be selected. This improves gaming experience by reducing the latency between the game's world model and its rendered representation. It is, however, a wasteful technique in that resources are spent on renditions that are never viewed by a user. In the mobile computing realm, such waste of GPU cycles and thus of power is out of the question.

can be considered a special form of multi-buffering. Whether the CI switches to the back buffer of its current client or to a buffer of a different client, it yields the same operation at the display controller driver level of the CI.

In the course of this research, the main designated client system, Android, has evolved quite a bit. One evolutionary development was the ever-stronger dependency on graphics acceleration, which is addressed in the next chapter. Another was the utilization of hardware features such as the display controller's overlay capabilities to offload simple composing tasks off the CPU as well off as the GPU. This feature, if supported by a device, is exposed to the user space, e.g., through the framebuffer interface (`/dev/fb`), and abstracted from by the hardware composer library, which is part of Android's hardware abstraction layer. This is an example of device capabilities reaching into the user-space, despite the abstraction efforts of the kernel. Essentially, it allows the higher levels of the Android middleware to render into separate framebuffers, which are then combined into a single screen output, e.g., combining the status bar with the output of an application. To remain compatible with this hardware abstraction layer when changing the guest kernel driver and the architecture below, the composition features must be implemented at some level below the framebuffer interface. This could be in the guest kernel driver or in the CI. But composition in software is expensive, and before doing it in on the CPU, it should rather be offloaded to the GPU. Using the GPU in the lower layers of the stack, however, was ruled out earlier in order to avoid TCB bloating. A viable solution is to have the CI use the overlay capabilities of the display controller to perform the compositing on the client's behalf. It is but a matter of protocol definition to have the CI expose this feature to the clients. However, there is a catch: If the overlay feature is used by the CI for drawing the label, the number of overlay regions available to the clients is obviously reduced by one.

Summary

The virtualized secure smartphone with trusted and identifiable paths to identifiable entities, here VMs, requires an output path that must fulfill the integrity and confidentiality requirement. Through individual and private framebuffers, both can be achieved. A separation of control and data plane can further strengthen the security argument of the secure GUI architecture. The output path must not only meet the security requirements, but also the identifiability criterion. To that end, labels may be drawn to inform the user about the current renderer's identity, which must be provided through a trusted bootstrapping procedure. However, implementation details can have significant implications on the runtime performance and implementation complexity, which have potential user-space compatibility implications. Further, the design space covered in this section provides the foundation for the design decisions made in the prototypical implementation. The discussion of the prototypical implementation also puts the proposed architecture into the context of the underlying virtualization and compartmentalization technology. Before the prototypical implementation can be discussed, however, the equally important input path needs some attention.

Input Handling

The requirements regarding integrity and confidentiality apply to the input path, just as they did to the output path of the secure GUI. However, the nature of input data is fundamentally different from the output data. While graphics output is a rather high-bandwidth channel, the bandwidth requirements of the input channel is considerably lower. Further, to achieve a good user experience, the latency by which input events are delivered must be as small as possible. Moreover, the typical device model of input devices is fundamentally different to that of a display controller, as will be discussed in Section 4.2.1.

Hardware model

The hardware model of the input path depicted in Figure 4.2.1 is much more heterogeneous than that of the output path. A typical mobile handset has a touch screen and a varying number of mechanical buttons. The number of mechanical buttons can range from a very small set, such as power button, home and volume, to a full mechanical keyboard. The way these input peripherals are connected to the handset's SoC is just as diverse. Three options are introduced at this point, all of which were actually present in the implementation of the secure smartphone. The first option is a typical capacitive touch panel, connected to the SoC via a peripheral bus such as IIC (also I^2C) or SPI, and a dedicated interrupt line. Upon a recognized touch event, the panel raises an interrupt, and a driver on the CPU uses

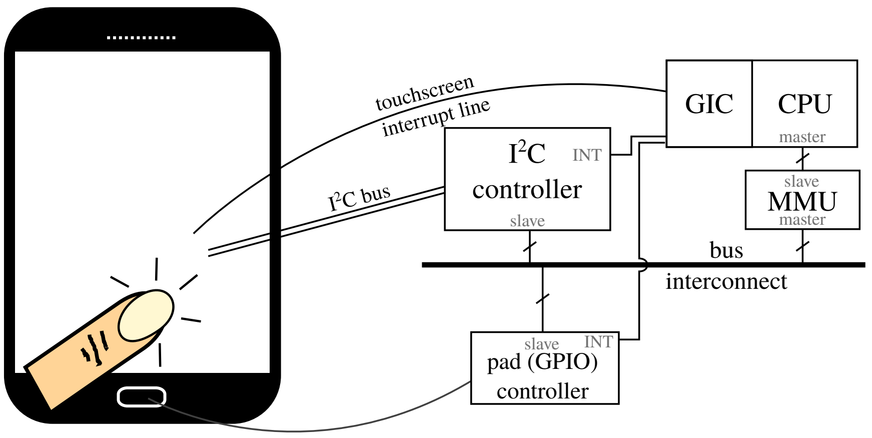


Figure 4.2.1: Hardware model of the input path. A touch screen device is connected to the SoC via a I^{2}C bus through an I^{2}C controller and an interrupt line connected to the GIC. A mechanical button is connected to an edge sensitive GPIO pin on the pad controller of the SoC.

the peripheral bus controller to fetch the data from the touch panel. The second option, typically used for simple mechanical buttons, is the use of edge-sensitive general-purpose IO (GPIO) pins. When a button is pressed or released, a switch is closed or opened, causing an edge signal on the IO pin, which causes the GPIO controller to raise an interrupt. A driver on the CPU can then read the level on the input pin to determine whether the button is pressed or not. The third option is a somewhat makeshift solution, which however, is quite commonly used. In this case, an input peripheral[6] with a peripheral bus interface is connected to GPIO pins of the SoC, rather than to a peripheral bus controller of the SoC. Like in the first case, the device raises an interrupt, notifying a driver on the CPU. However, the driver must then simulate the bus protocol by controlling the level of the corresponding GPIO pins through the GPIO controller. This technique is commonly known as bitbanging, and it adds considerable complexity to the driver, drastically increasing the interaction between the CPU and the GPIO controller. [7]

[6]In case of the secure smartphone platform, a capacitive button group surrounding the mechanical home button is connected to the SoC in this way.

[7]For example, transmitting a single bit over an I^{2}C bus using bitbanging requires no less than three MMIO accesses, that is, one for asserting the correct level on the data line and two more for generating two edges on the clock line.

Input path security

Before discussing the input path of the secure GUI, a recap of the output path is warranted. Taking a close look at the naive approach, the compositing approach, and even the mapping approach, reveals that there is in fact no device interaction necessary. All of these approaches assume a static visible framebuffer, which for example, could have been set up at an early stage during boot time by, say, a bootloader stage. Only in the referencing approach and when double buffering or hardware compositing comes into the picture, is limited device interaction necessary, that is, toggling the scan-out region as well as configuring, acknowledging, and handling the VSYNC-interrupt. Because of the hardware model of the input path, a driverless configuration such as this is not possible. Giving a driver inside a client VM a direct and exclusive access to the input hardware would be the closest equivalent to the framebuffer mapping approach. And, it would definitively fulfill the confidentiality and integrity requirement.

However, it is impractical for multiple reasons. Revoking and reassigning access is cumbersome, due to the state space that the device may assume, and which the individual client drivers may need to handle, or from which to recover. While this can in principle be overcome by an engineering effort, MMU-based IO access granting hits a brick wall when faced with devices that are accessible through GPIO pins. Page sizes are simply of too coarse a granularity to pass control over a small set of pins to a client; in fact, it is more likely that they share a single word of MMIO resources with other pins designated for completely unrelated functions. Other methods of interposing GPIO access could be applied, which will be discussed later in the context of the prototypical implementation.

Besides the practical implications of the exclusive access remapping approach, there are very serious implications on the availability of not only the input path of the secure GUI. If the input path is exclusively controlled by one tenant, how can the user reliably express to the TCB the intention to toggle input and output routing to a different tenant? A misbehaving tenant can naturally refrain from giving up the input path. As long as the labeling mechanism properly reflects the state of the input and output routing, such behavior does not imply a threat as regards to the secure GUI's threat model; however, it renders other tenants inaccessible. Interposition in the

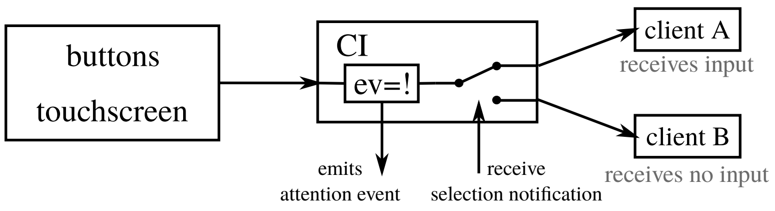


Figure 4.2.2: The coordinating instance (CI) of the input path, along with input devices (left) and two clients (right). Input events enter the CI, which forwards them to a selected client. It thereby enforces a policy decision that it receives via a separate interface from a decision-making module. The CI can emit secure attention events on receiving special user input, which it never forwards to a client.

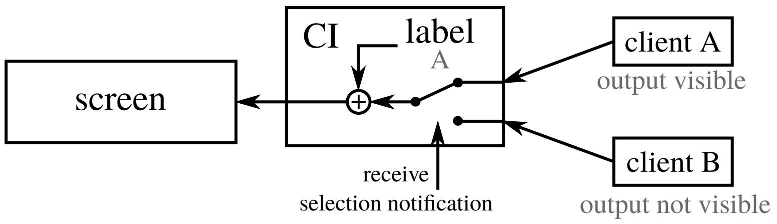


Figure 4.2.3: The coordinating instance (CI) of the output path, along with the screen (left) and two clients (right). It takes a policy decision via a special interface from a decision-making module. The CI enforces the policy decision that the designated client can draw on the screen and ensures that the current policy decision is apparent to the user by combining the output with an unforgeable label.

input path, which implies the capability of detecting a secure attention event within the TCB, therefore, becomes a matter of availability rather than a security requirement.

Once more, the concept of the coordinating instance (CI) is stressed as depicted in Figure 4.2.2. Events from the input devices enter the CI; the CI filters the input for a secure attention event, which it emits via a dedicated communication channel; all other events are forwarded exclusively to a selected client. Provided that the encompassing system facilitates confidentiality preserving channels, trusted paths can be established between the user and the clients. This implies the existence of a driver component that drives the input devices and forwards all input events to the CI. But the question remains unanswered as to the architecture of the input device drivers and their role as part of the TCB. In fact, the same goes for the framebuffer driver.

Compartmentalized Low-Level Drivers

In the previous section, the question arose as to the role of the input and output drivers as part of the TCB. In a monolithic kernel, such as the Linux kernel, the answer to this question is straightforward. Naturally, the drivers are part of the kernel, where they have access to all of the hardware resources; they must be considered part of the TCB without any limitations. But when attempting to build a compartmentalized system architecture, questions arise as to where to draw meaningful boundaries, and how to enforce them, given a certain hardware architecture. The design paradigm followed in this work was to provide servers, each abstracting from a single device and providing this abstraction as a service to higher layers. Adhering to the principle of least authority, this driver server shall only have access to the single device from which it abstracts. Especially the latter requirement is easier to abide by for some devices than it is for others. Whether or not the principle of least authority can be upheld efficiently by using hardware capabilities depends on the interaction model between the device and the SoC. In the previous section, some of these models have already been discussed. Now, a more systematic view is warranted. Many devices are accessible through MMIO. In mobile chip sets, this is typically true for devices that have been integrated into the SoC. With paging enabled, a driver running on the CPU can only access these devices when the corresponding MMIO region is mapped into the address space of the driver. While this is equally true for a monolithic kernel where all drivers share a common virtual address space, this mechanism can be used to give compartmentalized drivers access to a single device, that is, as long as the MMIO regions of unrelated devices are aligned to the granularity at which virtual address spaces can be constructed, e.g., the minimal page size. Unfortunately, violations to this rule are not unheard of.[8] In such case, devices are not separable by the memory protection mechanism. But it is not the only arrangement where hardware mechanisms fail to provide adequate abstractions for compartmentalized drivers. Peripheral devices connected to the SoC through peripheral buses pose their own challenges. Supposing a device is connected to a peripheral bus mastered by a controller integrated into the SoC, then

[8]For example, the Allwinner Technology A10 SoC has the MMIO regions of many integrated devices mapped with 1024-byte alignment, whereas the page size is 4096 bytes.

the combination of device and peripheral controller can be thought of as a single device; and the corresponding driver can drive the bus as well as the connected device. If, however, a second device is connected to the same bus, the same dilemma is encountered; that is, the devices are inseparable by hardware mechanisms. A particularly nasty instance of this problem is the case of GPIO-connected peripherals. When device access is inseparable by hardware, software must deal with the issue. One way to deal with this issue is to declare the driver component “trusted enough” to be granted more privileges than required to drive the device in question. This option is most efficient at the cost of watering down the principle of least authority. The second option is to intercept and filter the MMIO access, presenting a fully virtualized hardware to the driver. The third option is to introduce more levels of abstraction. As an illustration, IIC bitbanging through GPIO pins shall be considered. A driver driving the GPIO controller can abstract from the MMIO interface and provide a service allowing a client to control a selected group of pins. The client can then perform bitbanging and drive a device on the IIC bus. Other than placing a boundary between the GPIO driver and the bitbanging code, a boundary can also be placed between the bitbanging code and the device driver or both.

The drivers thus compartmentalized are then no longer unconditionally part of the system TCB. They must instead be characterized by the service they provide and by the side effects the driven device can have on the system, to determine whether it belongs to the TCB of a subsystem or not. Also, it can be expected that the additional protection domain boundaries induce a communication overhead in the system. Both effects will be discussed in Chapter 6 by the example of the prototypical implementation.

Routing Decision Making

Figure 4.2.2 and Figure 4.2.3 imply a yet unspecified communication partner, one that is to be notified of the secure attention event and that emits selection notifications. As of this work, this communication partner will remain largely unspecified, and its design shall be left to the system integrator. However, this component, which will be called routing decision maker (RDM), is critical to the integrity of the secure GUI, which is why some requirements need to be specified. Because of the compartmentalized nature of the architecture, the CIs of the input path and of the output path are disjunct and cannot communicate with each other. Therefore, it is the responsibility of the RDM to keep the routing of both paths in sync. This requirement implies the existence of two possibly disjunct namespaces for the clients, shared between the RDM and the individual CIs, as well as the knowledge about which client goes by which name in which namespace, present in the RDM.

The simplest implementation of a RDM might simply toggle through the available clients in a round-robin fashion upon a secure attention event. A more sophisticated one may be a client to the secure GUI in its own right, directing the input/output routing to itself upon a secure attention event and presenting a dialog allowing the user to select one of the other available clients. Additionally, the RDM may also receive notifications from unprivileged clients expressing the wish to give up the input/output focus voluntarily.

Implementation Details

This section covers the prototypical implementation of the improved and compartmentalized secure GUI of the secure smartphone. It starts with the rationales for choosing a particular design from the design space discussed previously. And, before the actual implementation is discussed, the relevant details of the underlying hardware platform are introduced. The actual implementation was done in the context of Fiasco.OC, L4Re, and L^{4}Linux, which means that a good understanding of the terms and concepts introduced in Section 2.2.3 is helpful to follow in the following presentation. The implementation details will be presented as a series of incremental steps, starting from the original solution based on `mag`.

The output path of the `mag`-based solution followed the compositing scheme. As stated above, it fulfilled all the security requirements; it had private framebuffers preventing eavesdropping, as well as reliable labels allowing for identifiability. In terms of performance, however, this approach was less than ideal. Update requests were implemented using synchronous IPC, thus blocking until the update was performed. This had a devastating effect on the clients' response time. But regardless of the request notification method, copying the framebuffer content induced considerable load on the CPU, prohibiting acceptable frame rates, as can be seen in Chapter 6. The remaining solutions, mapping and referencing, do not have this copying overhead. Eventually, the choice was made in favor of the referencing approach, in conjunction with the overlay labeling method, for its flexibility and low implementation complexity. The full architecture of the output path consisted of a display controller driver and a output switching component, which was also responsible for drawing the label residing in individual protection domains. For the input path, the author intended a very similar solution based on a driver and a switcher component. However, for pragmatic reasons, two deviating solution saw the light of day. One was small, but it had input driver and switcher integrated into one protection domain. The other one was large, because it reused a Linux driver running inside an instance of L^{4}Linux and providing input events as a service. But it was compartmentalized, as the driver and the switcher resided in disjunct protection domains. In sum, both exhibited all the intended aspects.

The prototype was built upon two very similar smartphone platforms, the Samsung Galaxy SII GT-I9100 and the Samsung Galaxy SIII GT-I9300. These platforms have the following hardware details relevant to the secure GUI architecture. Where deviant, the specifications for the older GT-I9100 are given in parentheses: The platform is based on an Samsung Electronics Exynos 4412 SOC (Exynos 4210) featuring four (two) Cortex A9 ARM CPU cores running at a maximum of 1.4 GHz (1.2 GHz) and a Samsung Electronics FIMD[9] display controller. The SoC is paired with 1 GB of main memory. As input devices, the platform has a Melfas mms-100 (Atmel mx224) series capacitive touchscreen connected via IIC bus to the SoC, a Cypress CapSense touchkey (Melfas touchkey)-based module with two buttons connected via a IIC bus (bitbanging in the case of SII), and four GPIO based mechanical buttons, which were the power and home buttons, and the two rocker switch positions volume up and volume down.

The input and output paths were implemented separately. The first impulse for this decision, in fact, was pure pragmatism, pragmatism that was necessary because the object capability model [63] of Fiasco.OC/L4Re did not account for the diamond problem[10] of multiple inheritance. As in-

[9]Fully Interactive Mobile Display

[10]A class D inheriting from two base classes, B and C, which both in turn inherit from a common-base class A. The resulting inheritance graph is diamond shaped, thus the name "diamond problem".

troduced in Section 2.2.3, there are but a few real object types known to the Fiasco.OC microkernel. Many more types are expressed through user-defined protocols, and access control is exerted through the IPC-gate object, which constitutes a communication channel with guaranteed endpoints. To some extent, L4Re also supported the concept of inheritance by facilitating, for example, the simple combination of shared memory with ICU semantics, which constitutes L4Re's event protocol used for communicating input events. Essentially, it means that two or more protocols are conveyed across a single communication channel—here an IPC-gate. Screen multiplexers, such as `con` and `mag`, went one step further and combined the event protocol with the *goos* protocol, consolidating the input and output path on a single channel. This worked well until the requirement for VSYNC interrupt delivery arose. The goos protocol needed to be amended with ICU semantics, which, considering that the event protocol already had ICU semantics, led to an instance of the diamond problem. While many workarounds would have been thinkable, this dilemma sparked the idea of not only splitting the input and output path back into separate communication channels, but to take the concept of compartmentalization even further and have the input and output path serviced by mutually isolated servers. The task of routing input and output, formally taken on by `mag`, was moved into two new components: `fjug`[11], servicing the output path and `ijug`[12], servicing the input path. To account for the new split, the paravirtualized framebuffer driver of L^{4}Linux, `l4fb`, needed to be adjusted so that it would expect to receive the input events and the framebuffer service via two distinct channels, rather than via a combined one. Further, support for receiving VSYNC events was added to `l4fb`. On the opposite end of the output path was the L4Re server `fb-drv`. As it was, it set up a visible framebuffer and exported it to a client—here `mag`—via a very basic implementation of the goos protocol. This driver server needs multiple amendments. First, support for requesting VSYNC notifications was built in, which, as discussed above, sparked the amendment of the goos protocol. In order to support private buffers, the second amendment allowed the dynamic creation of multiple rather than one buffer. The third amendment allowed the abstraction from overlay windows as provided by the display controller and their export as views (Figure 2.2.1). The

[11]Short for framebuffer juggler
[12]Short for input juggler

latter two amendments were kept flexible enough to allow for the arbitrary association of buffers with views.

Naturally, `fjug`, the framebuffer switcher, was the new client to this enhanced service. Through this new service, `fjug` was able to provide its clients with private framebuffers as well as to allocate a framebuffer on its own behalf for the purpose of label drawing, which it then attached to a view covering the label region. The latter operation was executed by the `fb-drv` server through setting the scan-out region of the corresponding overlay window to the designated label buffer. Analogously, `fjug` toggled the output routing by attaching the private framebuffer of the selected client to a view covering the client region on the screen. Double buffering was realized by allocating private framebuffers of twice the size needed to back the screen. The clients were then allowed to select either the top or the bottom half of the buffer for being made visible. This request was forwarded by `fjug` to `fb-drv`, and vice versa, VSYNC events were forwarded from `fb-drv` to the currently active client.

As mentioned earlier, the input path was implemented in two different ways. On the guest side, the event driver, which was part of the `l4fb` driver, was used indifferently in both of the cases. The variations were on the opposite end of the stack. The first variant was an input driver `input-drv`,[13] which ran as a native Fiasco.OC application and contained not only the drivers for the three different types of input devices, but also the `ijug`[14] module. The drivers were derived from Linux driver code that had found its way into the `mag` component earlier. The difficulties mentioned earlier of abiding by the principle of least authority, which came with trying to grant access on a per GPIO pin granularity, were solved by a GPIO service[15] built into the resource server `io`. With the amendment of this protocol, it was possible to construct a VBus that comprised the MMIO resources of the IIC controller connected to the touch screen as well as the GPIO pins connected to the capacitive button element and the mechanical buttons. When later the architecture was ported to the Samsung Galaxy SIII,[16] the `ijug` module moved into its own protection domain while a headless L^{4}Linux instance took the role of the input driver.

[13]The original `input-drv` was built by Jan Nordholz

[14]Jon Tapsel implemented the `ijug` module according to the author's instructions.

[15]This service was designed and built into `io` by Alexander Warg.

[16]This porting effort was done be the company Trust2Core GmbH founded in 2012.

In summary, the secure GUI architecture consisted of the driver servers, driving the display controller and the input devices, and the two switchers `fjug` and `ijug` enforcing the routing of framebuffer output and input events as well as displaying the routing decision to the user and detecting the secure attention event. A vital aspect of the implementation has still not been discussed. It is the matter of belief that all of the components are actually communicating with whom they think they are and the trust that no untrusted third party can eavesdrop on these communication channels. This also relates to the correctness of the RDM's model of the setup. The responsibility for a sane setup lies with the bootstrapping procedure. As introduced in Section 2.2.3, L4Re based systems are bootstrapped by a Lua interpreter called `ned`. It may be assumed that the boot process is trusted. That is, the integrity of the TCB running, which includes the kernel, the root task, the root pager, `ned`, and the script it is executing, as well as the integrity of the to-be-loaded binaries belonging to the secure GUI, has been asserted appropriately. There are two typical ways in which `ned` establishes a communication between two subsystems: In the first, it creates a new IPC-gate and assigns it to one capability slot for each of the communication partners, where typically one offers a service used by another. In the second, it assigns the newly created IPC-gate to a server; however, it then expects the server to offer a factory service. This service is then used by `ned` to create one or more sessions, represented by IPC-gates that the server creates on `ned`'s behalf and which it then assigns to one or more clients.

Setting up the secure GUI, `ned` connected the drivers to the `io` server in the first manner. By these connections, the drivers were now able to request the resources required to drive the corresponding device, where the VBus definitions (see Section 2.2.3) supplied by the system's architect restricted resource access accordingly. The same mechanism was used to connect the `fb-drv` server with `fjug` and, where applicable, the input driver with `ijug`. Both `ijug` and `fjug` offered a factory service, which `ned` used to create the client sessions for the VMs. In making these connections, `ned` gathered all the information required by the RDM, that is, the information matching the `ijug` and `fjug` sessions to the corresponding VMs.

Summary

In this chapter, the security of the input and output paths of the secure GUI was discussed. It was done under the premise of hardware architectures found in contemporary smartphones. The discussion was supplemented by an exploration of compartmentalized drivers and the intricacies one faces trying to uphold their isolation. The resulting compartmentalized architecture was focused on providing mechanisms only, mechanisms that allow for an efficient implementation of a trusted and identifiable path between clients—here VMs—and the user, and the enforcement of a security policy. Policy, however, was excluded from the architecture, and a routing decision-making (RDM) module was postulated as being home to a policy. Eventually, the developed concepts were put to use in the presentation of the prototypical implementation.

The implementation of the prototype was described as is. In retrospect, the principle of least authority could have been abided by more strictly. For example, the `fb-drv` server was still allocating the resources for the framebuffers; it did so on behalf of the `fjug` server by exposing the allocation as a service. But strictly speaking, it did not even need the authority to allocate memory at all. Instead, all that it required was a description of the framebuffers, by physical address ranges, and possibly, a symbolic name for the framebuffers, so that it can offer a service allowing the activation of the buffers. The resources might as well have been allocated by the bootstrapping process, which is omnipotent anyway. Consequently, the bootstrapper—here `ned`—could have, in order to set up a session, allocated the framebuffer, allowed the corresponding VM to access the memory, passed a symbolic name to `fjug`, and passed the same symbolic name along with the physical address range to `fb-drv`. Going even further, the allocation of framebuffers could have been left to the VMs using a portion of their allotted main memory quota. This could have been done by means of the mechanisms used in the context of GPU virtualization, which will be discussed in the following chapter. These modifications would only alter the setup phase, which is why it may be assumed that, for evaluating the runtime behavior and performance, the prototype is good enough.

5
Mobile GPU Para-Virtualization

The purpose of graphics acceleration hardware is to provide higher performance than would be possible using software alone.
—Dowty and Sugerman [26]

Agreeable as this quote might be, it is not good enough for a virtualized secure smartphone. Not only must an accelerator, that is a GPU, provide good performance, it must also relieve the CPU of the burden of costly computations. As discussed in Section 2.4, many GPU virtualization schemes achieve good performance, but at the cost of high CPU load. This chapter develops a GPU-server architecture that facilitates graphics acceleration in the VMs of a secure smartphone. In order to cater to the limited power supply inherent to mobile devices, it does so without introducing significant CPU load. Besides the performance requirements, the design of this architecture was driven by the paradigms framed in Chapter 3, that is, a simple device model shall be assumed, the implementation of which shall have minimal functionality and a non-redundant interface, and it shall prohibit implicit memory sharing. To that end, this chapter starts with the assumed general hardware and programming model of a mobile GPU, resulting in a

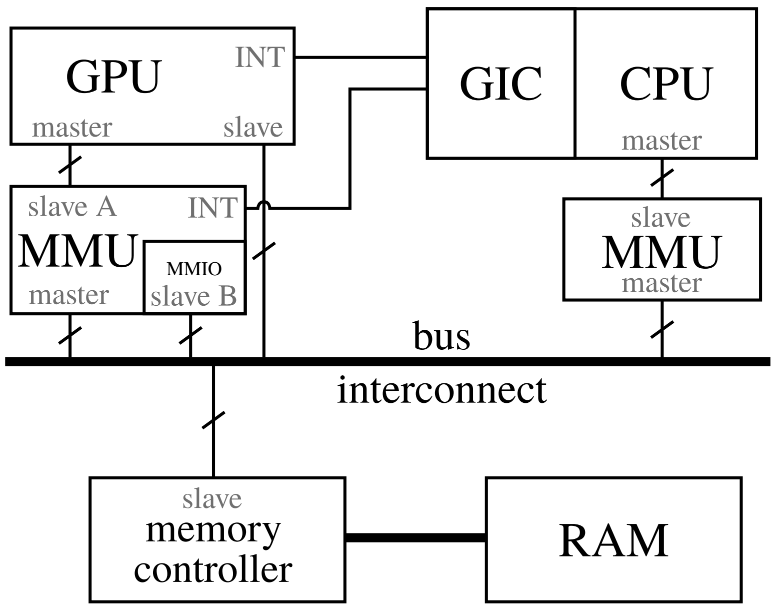


Figure 5.1.1: Model of hardware components involved in graphics acceleration. The CPU and the GPU share a common main memory. Access to the main memory via the common bus interconnect is mediated by individual MMUs.

discussion of actual functionality required in the TCB. The possible design space of this functionality is then discussed in Section 5.2 and Section 5.3. Eventually, the implementation details of the prototypical implementations are presented, and its integration with the secure GUI infrastructure is described.

Hardware Model and Programming Model

The hardware components involved in graphics acceleration are the CPU, the GPU, and the main memory, as depicted in Figure 5.1.1. A bus interconnect allows for the communication between these components. Accesses by the bus masters, here the CPU and the GPU, are mediated by individual MMUs. The GPU as well as the GPU's MMU also act as slaves, which allow the CPU to control these devices through an MMIO interface. Both the GPU and the GPU's MMU may raise interrupts on the CPU's generic interrupt controller (GIC), notifying the CPU of actions required. As a bus master, the GPU is expected to access the main memory, e.g., in order to read commands and data, such as shader programs, textures, and attribute lists, or to render into framebuffers. But this model shall also cover the possibility of the GPU accessing the MMIO resources of itself, its MMU, or other devices connected to the bus interconnect, that is, if its MMU is configured—or misconfigured—accordingly.

The programming model of the GPU is assumed job oriented. That is, a client using the GPU creates a rendering job, which it lays out in memory, then submits it to the GPU and expects a notification upon completion. Notably, by this model, GPU jobs cannot be preempted and resumed later. Further, it is assumed that the submitted job is executed by the GPU as is; that is, jobs are not modified by driver code. Consequently, the code executed by the GPU must be considered untrusted. So in order to uphold spatial isolation, the memory made accessible to the GPU must be a subset of the memory already controlled by the client. Here, the suggested GPU server therefore implements and exposes two basic mechanisms. It controls the GPU's MMU and exposes a memory management interface, sanitizing all requests with respect to the memory subset requirement. To cater to multiple isolated clients, it controls the GPU and exposes a job submission interface, scheduling jobs and activating the corresponding address space.

Memory Management

The architecture of a mobile GPU driver stack, as it is typically found in Android-based devices, was introduced in Section 2.3. Special attention was given to three aspects of memory management, and the locations of these aspects' representations in the GPU driver stack were identified. Recalling from Section 2.3, it was the responsibility of the kernel to account for physical resources and to exert proper memory protection. The third aspect, accounting of the virtual addresses and the virtual address space layout was performed by a user space agent. This proved sufficient to isolate multiple processes that make use of one another's services of the GPU—that is, of course, unless there are bugs in the driver. The consequences of such bugs, discussed in Chapter 3, are the same consequences on a system's integrity, as suffered by a virtualizing system if this software stack moves into a virtual machine unaltered. Even if the driver in the virtual machine is sound, it gives the guest kernel the power to evade memory isolation by means of the GPU. These concerns are valid, even if the guest kernel is well behaved and its integrity is intact, because the integrity of a system hosting virtual machines must not depend on the integrity of the guest operating systems. Consequently, the memory protection aspect of the GPU driver's memory management, with respect to the whole system, cannot reside within a potentially hostile VM. The matter of memory protection must be moved out of the VM. It can be incorporated into the hypervisor, into the kernel, or into a specially privileged and trustworthy VM. In the spirit of building a minimal and compartmentalized TCB, the choice here was to place this matter into a server running natively on the underlying microkernel—the GPU resource governor (GPURG). To meet its responsibilities, GPURG had to perform similar tasks as a hypervisor, providing the guest with virtual memory. Notably, the GPURG component did not account for physical resources, however. In the following, the concepts of shadow paging and nested paging, which were introduced in Section 2.2.1, are discussed in the context of the GPURG component and GPU virtualization.

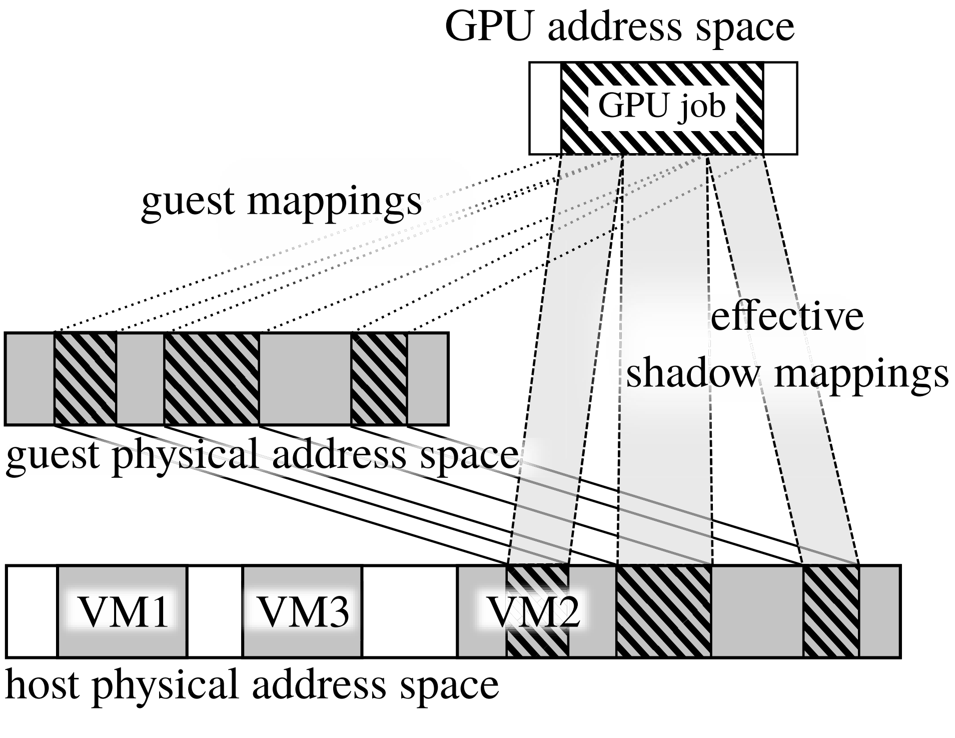

GPU address space
GPU job
guest mappings
effective shadow mappings
guest physical address space
VM1
VM3
VM2
host physical address space

Figure 5.2.1: The relationship between the GPU's address space, the guest physical address space, and the host physical address space; and the role of the shadow page tables in the context thereof. (This figure was published before by the author in the slides of a talk given at MOST15 [24].)

Shadow paging

The principles of shadow paging have been introduced in Section 2.2.1. And it is assumed that the GPU driver follows the same model as described in Section 2.3. However, the driver is now running as part of a guest operating system kernel inside a VM. Therefore, when it creates page tables expressing a GPU address space on behalf of its processes, these page tables map onto guest physical memory rather than actual physical memory as depicted in Figure 5.2.1. To perform shadow paging, GPURG needs access to the client's guest memory, and it must employ some form of page table base register through which it can be informed by the client about the active page table. To make sense of the client's GPU page tables, GPURG must also have knowledge about the client's guest-to-host mappings. It is assumed, unless stated otherwise, that the guest-to-host mappings remain static throughout the lifetime of a VM—that is, from the time of boot to the time of shutdown.

With these facilities, the GPU server can construct a shadow page table on the fly. When the GPU's MMU issues a page fault, the GPU server queries the corresponding guest's active GPU page table and uses the guest-to-host mapping of the particular VM to create a valid shadow page table entry. This, however, does not reflect the programming model of the GPU, by which the interaction between the CPU and the GPU shall be kept to a minimum, as it produces page faults with high frequency and thwarts the performance and offloading efficiency of the GPU.

To better cater to the GPU's programming model, the changes to the GPU address space can be pushed to the GPU server as they are issued by the client's applications. This requires a modification in the guest's kernel driver. Now, the shadow page tables are constructed when the corresponding GPU's page table is set up, which happens before a rendering task is started. The downside of this approach is that now every GPU page table is duplicated in the GPU server, and so is the page table memory consumption. Note, however, that the shadow page tables are still transient, and they can be reproduced from the guest page tables; keeping duplicates is only a performance optimization.

In a next step the guest's GPU page tables can be dropped altogether; this reduces the memory consumption in the guest VMs. The run time complexity involved in page table management is removed from the guest's driver. But now the shadow page tables are not transient anymore: Mind that if the GPU server discards its page tables, the information about the GPU address spaces is lost; therefore, it is not stateless anymore. In consequence, the GPU server must allow for the creation of one shadow page table per client session. While this has only positive implications on the runtime performance, there are some drawbacks in terms of memory consumption, which are discussed in Section 5.2.3, and on manageability, which are discussed in Section 5.5 in the context of suspending VMs.

Nested Paging

It is becoming increasingly common that system MMUs (SYSMMUs) are deployed in smartphone SoCs. If a SYSMMU mediates memory accesses of a GPU that already has its own MMU (GPUMMU), then nested paging can be performed for the GPU. For this to work, the GPU server needs to control both the SYSMMU and the GPUMMU. Consider, though, that in this model, no virtual page table base register exists as it does in CPUs with support for nested paging; it must be emulated or provided by a paravirtualization API, just as in the shadow paging approach discussed earlier. Through the emulated page table base register, the GPU server gets informed about the active guest GPU page table. It still has knowledge about the guest physical-to-host physical mapping as postulated earlier; and it uses this knowledge to set up a SYSMMU page table reflecting this mapping. When the GPU server schedules a rendering job on a client's behalf, it activates the client's GPU page table as well as the corresponding SYSMMU page table.

Resource management

Constructing page tables on a client's behalf consumes memory. This secondary memory consumption must be attributed to the corresponding client, or the system is at the risk to be forced into a denial of service situation by a misbehaving client. In the case of shadow paging, the secondary memory consumption is bounded by the memory needed for one fully populated page directory. When considering the paravirtualized nontransient scheme, the secondary memory consumption becomes unbounded, because the client can request the creation of an arbitrary amount of page tables. In this case, a per client quota is required to prevent crosstalk between the clients as well as denial of service.

Sharing

The previous section addressed how memory protection between VMs can be expanded to the GPU by means of a GPU server as a trusted system component. To recap, the GPU server exerts memory protection on the virtualizing system's behalf using the GPU's MMU or a combination of GPUMMU and SYSMMU using shadow paging or nested paging respectively. The system can let a guest run arbitrary rendering tasks on the GPU most assuredly that the GPU can only ever accesses memory already under that guest's control. But to allow sharing the GPU resource it must be addressed how guests can submit tasks to the GPU and how guests are notified about the completion of a job.

Following the driver model discussed in Section 2.3, a user space driver lays out the rendering job's executable in memory; a description of the job, which may at best be only an entry point, is passed on by the kernel driver to the GPU. Upon completion, the GPU signals the CPU with an interrupt; the kernel driver handling the interrupt in turn notifies the user space application about the completion. This narrow interface makes for an excellent virtualization boundary: Running inside a VM, the guest kernel GPU driver passes this job description on to the GPU server. It does this through an emulated set of MMIO registers or a para-virtualization interface. The GPU server can then schedule the job on the client's behalf, and when the job completes, notify the client, whereby it relies on a signaling mechanism provided by the underlying hypervisor or microkernel.

With such an interface and a memory protection mechanism in place, the GPU server is ready to accept job requests from arbitrary clients or guests; it can run these jobs safely while incurring no overhead from either copying high-bandwidth data or auditing submitted jobs.

Scheduling

A GPU scheduler should strike a balance between utilization, fairness, fidelity, and covert channel freedom. It must do so under a number of constraints: A target frame rate imposes periodic deadlines on the schedule; the execution model is job-oriented without the possibility of preemption; and the programming model allows for unbounded jobs.

A first-come-first-serve scheduler can easily keep the CPU busy, ensuring good utilization, but it is easy for one misbehaving client to starve others. In fact, due to the lack of preemption and the unboundedness of the jobs, this goes for any scheduling strategy. Therefore, a mechanism must be put in place "to determine when a shader program has run for an intolerably long time and abort processing"—Smowton [60]. Assuming such a mechanism exists, a round-robin scheduler can at least assure that every client makes progress. While a round-robin scheduler assures that each client is scheduled equally often, fairness is not established because the time available to each client depends on the runtime of the jobs issued by the clients. If two clients are involved that are subject to a strict security policy prohibiting information flow between them, then this yields a new risk: One client, the sensing client, could issue, with high frequeny, jobs that have a short runtime, the duration of which is known in advance. Knowing that the scheduler will interleave the jobs of other clients it can now measure its own jobs' latencies and thereby infer the runtime of the other clients' jobs. This is a side channel leaking state information of the other clients [20], and if shader programs with content sensitive runtime are involved, it may leak content [40]. A cooperating client may even actively modulate a signal onto the jobs' runtimes, establishing a covert channel with the sensing client.

A measure for countering covert channels is the use of slotted execution. Slotted execution assigns each tenant a fixed time quantum, effectively distributing the available time among the clients. But it also confines the runtime of the each tenant's job to an immovable time slice, or slot. The former quality establishes fairness. The latter quality is required for the absence of covert channels because deviations from the time slot boundaries can be measured and can therefore carry information. One downside of slotted execution is that the time allotted to but not used by one tenant is lost and cannot be assigned to another, thus slotted execution impairs utilization. Another downside manifests in conjunction with the assumed execution model of the GPU, where jobs can only be canceled but not preempted for later resumption: The slot duration restricts the maximum job length; longer running jobs fail. And, the target frame rate dictates a scheduling period within which all tenants must have been scheduled. For example: A target frame rate of f yields a slot duration of $\frac{1}{fN}$, where N is the number of tenants. Consequently, when the number of tenants N rises, the slotted execution scheduler can choose to degrade the frame rate f, or reduce the scheduling period and therefore the maximum job length. Either way, the fidelity of the system suffers.

The secure GUI architecture discussed in the previous chapter favors exclusive usage of the screen. Therefore, it was considered to give the client in focus also exclusive access to the GPU, a strategy unfair but faithful, and obviating covert channel countermeasures. However, it clashed with the expectation of the VMs, or rather with the expectations of the Android middle ware inside the VMs. Not servicing the rendering request of a background VM caused repeated timeouts, which prompted Android to adopt desperate measures, that is, restarting the whole service layer.

Under the given constraints, it is far from trivial to balance the four requirements stated earlier. It is up to the system's designer to choose one feature over the other, and occasionally intrusive changes to the guests may be required. Eventually, a round-robin scheduler made it into the prototypical implementation, thereby choosing fidelity over covert channel freedom and avoiding the guests' desperation. Cold comfort can be drawn from that the underlying virtualizing system was not free from covert channels after all [51].

Suspend and Resume

L^{4}Linux is not known for its ability to suspend or even migrate a running VM. Nevertheless, it is worth exploring the possibility of suspending a client to this GPU virtualization scheme. The only state held in the GPU server concerning a client is the corresponding job-queue and GPU's address spaces. The job-queue, consisting at most of one job per client and per processing core, can be flushed; the GPU server can refuse to take on more jobs from a client that is suspending. Thereby, an invariant of an empty job queue for suspended clients is created. If the page tables corresponding to the client's contexts are transient, as was discussed earlier in Section 5.2.1, they can simply be discarded; upon resume, they can be reconstructed from the guest's page tables. Otherwise, they must be transformed into a format that takes into account the guest physical layout and be stored as an amendment to the VM's state.

The fact that the entire state of an OpenGL rendering pipeline is stored in guest memory makes suspending GPU server clients very easy, compared to API remoting schemes. Migration across different platforms is a different story though. The hardware dependency is fused into the guest clients' applications by means of the GPU dependent user level drivers, which makes it next to impossible to migrate the VM to a different GPU.

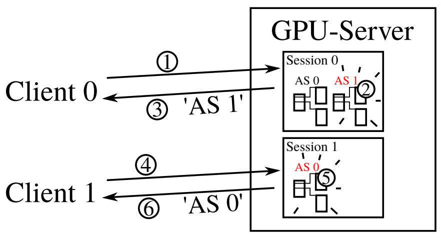
Implementation Details

The base system of the prototypical implementation is already known from Section 4.5 where the implementation details of the secure GUI were presented. In addition to what was presented earlier, both SoCs of the employed smartphones feature an ARM Mali400 MP4 GPU. It has one geometry processor (GP) and four pixel processors (PPs). Each of these processing cores has its own MMU; the MMUs share one commonly used second-level cache with the processing cores.

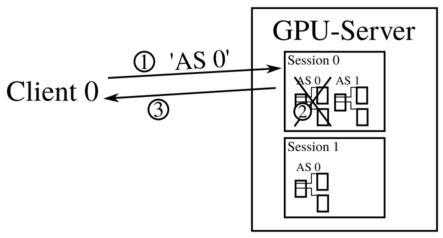
The prototypical implementation comprises a GPU server and a paravirtualization interface that defines the communication between the clients. The clients, here instances of L^{4}Linux, were adjusted to use this paravirtualization interface instead of driving the GPU directly, that is, through the GPU's MMIO interface. This modification was restricted to the lower end of the Linux kernel's GPU driver, leaving the kernel user interface unaltered. As a result, the GPU-specific user level drivers were deployed unaltered. The rest of the section covers the paravirtualization interface, the internal GPU-server structure, and the bootstrapping phase of the system that is essential for establishing assurance. Eventually, the integration with the secure GUI architecture is discussed.

Paravirtualization Interface

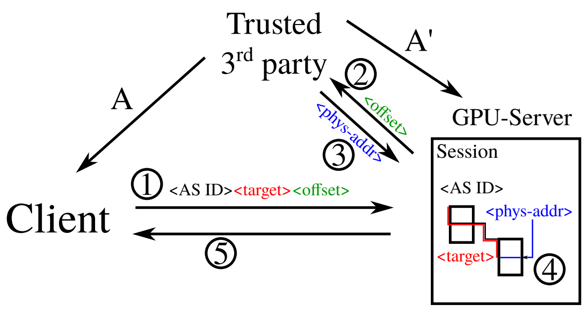
The para-virtualization interface between the GPU server and the guest kernel driver is composed of two functional groups: One functional group is concerned with creating, destroying, and populating GPU address spaces. The other is concerned with submitting jobs to and receiving job completion notifications from the GPU server.



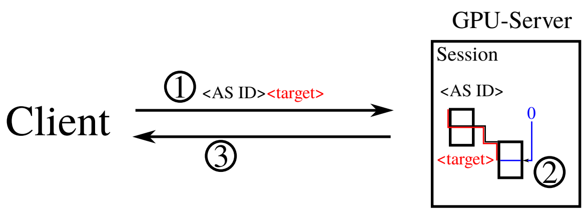
(a) GVA construction.



(b) GVA destruction.



(c) GVA mapping.



(d) GVA unmapping.

Figure 5.6.1: The four GPU-Server protocol functions of the memory management functional group are: (a) the construction of a GPU virtual address-space (GVA), (b) the destruction of a GVA, (c) the mapping of memory resources into a GVA, and (d) the releasing of memory resources from a GVA.

The first group comprises four functions: two for creating and destroying an address space and two for creating and deleting a mapping. To create a new address space, as depicted in Figure 5.6.1a, the client invokes the corresponding function ①. The GPU server, in turn, allocates a new page directory and assigns a new address space ID ② and—upon success—returns a the symbolic identifier ③. Internally, the GPU server administers one session data structure per client. These sessions are identified through the communication channel by which the function was invoked. Each of the sessions and therefore each client, has its own namespace of address space identifiers; consider the example of Figure 5.6.1a where in Step ⑤ the address space identifier 0 is assigned in Session 1, although Session 0 already has an address space ID of 0. Figure 5.6.1b depicts the destruction of an address space. Here the client invokes the destruction function and specifies the address space to be destroyed by using the previously assigned identifier ①. Subsequently, the GPU server destroys the given address space ② and returns control to the client ③. The unforgeablility of the communication channels and their identifiability are essential for the GPU server to provide isolation between the clients.

The address space identifiers are not the only entities with session-local namespaces. The same principle applies to the physical memory resources used by the clients as well. In order to make a mapping request, as depicted in Figure 5.6.1c, the client issues a triplet comprising an address space identifier (AS ID), a target address, and an offset. The client's session is again denoted by the channel used. Within the session, the address space identifier denotes the page directory to be modified. The target address denotes the index of the entry to be modified. The content of the new entry, however, must be sanitized very rigorously because failing to do so leads to evasion of isolation, as was established in Chapter 3. To that end, the physical address to be mapped is specified by the offset field. The offset specifies a chunk of physical memory within an abstract range of physical memory assigned to the client,[1] but it is not a physical address. The GPU server must use a trusted service (Figure 5.6.1c②) to translate the offset into an actual physical address ③. The trusted service may respond with an error code if the given offset does not yield a valid pysical address. And, for a given client, the translation can only yield physical addresses that are already under the control of that client. The assurance of the latter requirement will be covered later in Section 5.6.3 when the bootstrapping of the prototype is discussed. After translating the offset into a physical address, the entry can finally be made ④, and control can be returned to the client ⑤. If eventually, the client decides to remove a mapping from an address space, it invokes the function depicted in Figure 5.6.1d. Here, it suffices to specify the address space identifier and the target address ①. The latter is used by the GPU server to invalidate the page-table entry ② of the address space specified by the former. After that, control is returned to the client.

This first functional group is rather generic and can be used for any GPU

[1]See also "dataspace" in Section 2.2.3

that has its access to the main memory mediated by an MMU. In fact, this applies to any DMA-capable peripheral device with an MMU. The second functional group is concerned with the submission of rendering jobs. This

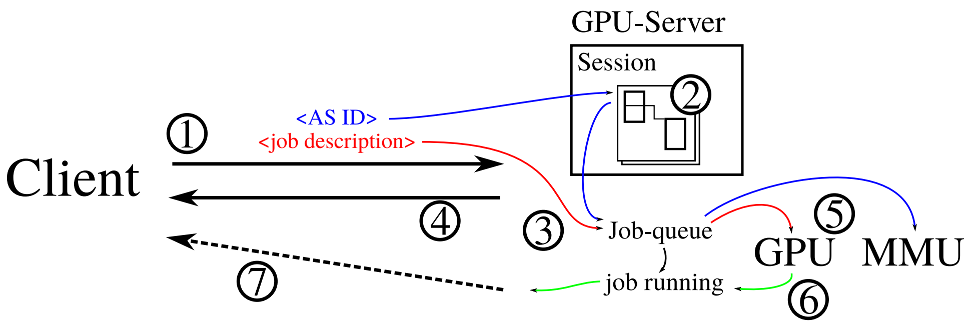


Figure 5.6.2: GPU job submission protocol flow. Upon job submission request ① by a client, the server selects the corresponding page directory ② and places both the job description and the page directory into a job queue ③. Then control is returned to the client ④. The GPU server activates the corresponding page directory and starts the job ⑤ once the GPU is ③ or becomes ⑥ idle and the job is marked running. When the job completes ⑥, a notification is sent to the client asynchronously ⑦.

function is depicted in Figure 5.6.2. To submit a rendering job to the GPU server, the client specifies a job description and an address space identifier ①. As discussed before, the job description is but an entry point to an executable residing in memory, or a small set of parameters for the GPU; it varies with the GPU architecture. The address space identifier selects a page directory ② from the session's set of page directories, where the session is selected, once more, through the communication channel by which the operation was invoked. The tuple of address space and the job description is appended to the job-queue ③. If the GPU is busy, the client-server interaction ends here, and control is returned to the client ④. Otherwise, the page directory is activated on the GPU's MMU, and the job is started on the GPU ⑤ while the job is marked running before control is returned to the client ④. Once the GPU completes the job, it raises an interrupt and notifies the GPU server ⑥. The GPU server, in turn, checks the job-queue for subsequent job requests and starts the next one if present. In any case, it notifies the client ⑦ about the job completion before it blocks and waits for further client requests and GPU interrupts.

The MMU may raise an interrupt in the event of a page fault. This condition, however, is considered a mistake in preparation of a rendering job and thus an unrecoverable failure. In such a case, the GPU server resets the GPU and reports a failure to render the job to the client. If the GPU has a command processor, the principle client-GPU-server interaction remains the same. The program flow of the GPU server and the amount of CPU-GPU interaction differ, however. For example, Step ⑤ could be concurrently performed by the command processor of the GPU.

GPU server

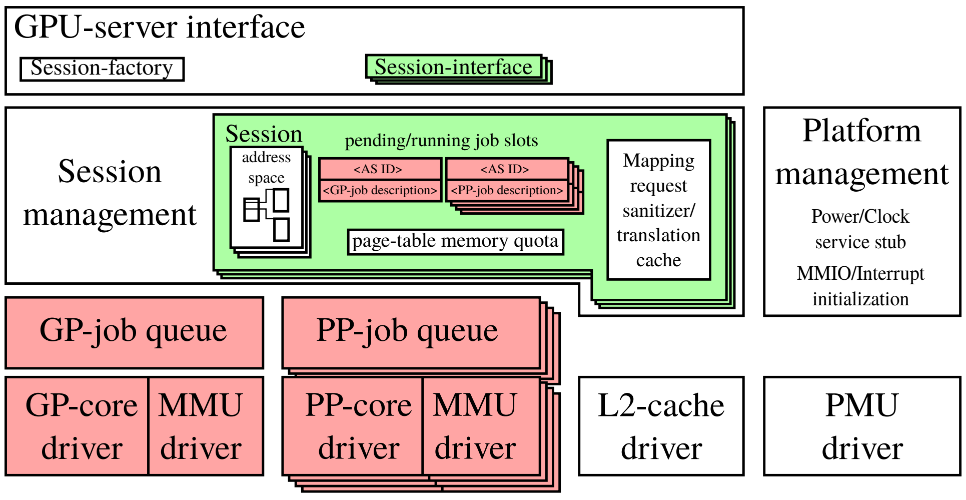


Figure 5.6.3: From top to bottom: The GPU server exposes a session-factory service and one session service via individual communication channels. A session management layer keeps track of the session-specific data structure comprising page directories, slots for pending jobs, mapping request sanitizer information, and page-table quota information. It is complemented by a platform management module. Drivers for the computational cores, the per core MMUs, the L2-cache, and the power management unit (PMU) form the lowest level of the GPU server.

While the GPU server interface discussed in the previous section is still generic, the architecture of the prototypical GPU server implementation is less so. This is owed to the architectural distinctiveness of the mobile GPUs and the runtime environment, which is here Fiasco.OC with L4Re. A glance at Figure 5.6.3 reveals that the GPU server exposes, typical for an L4Re server, a session-factory service and one session interface for every created session. The latter expose an interface adhering to the protocol discussed in the preceding section, and each of these services is exposed via an individual IPC-gate serving as unforgeable communication channel. Above that, sessions implement the ICU protocol to allow a client to register IRQ-objects for asynchronous job completion notifications. A shared memory buffer is established between the client and the server to accommodate batches of mapping requests, job descriptions, and a status field. The latter is a back channel to report the success or the failure of a rendering job to the client. The session-management layer administers one session data structure per session. Each may hold a set of page directories as requested by the client. Because the page directories can grow indefinitely at the whim of the client, page directory—and page-table—allocation is governed by a per session quota mechanism. Otherwise, the resource consumption of the GPU server is only linear with the number of session, which is not under the control of a client but rather of a trusted third. To that end, the GPU server has a constant number of pending job slots per session—here, one job to be queued or running per computational core. The information required to sanitize mapping request is supplied at session creation time by the trusted third, as indicated by the symbol A' in Figure 5.6.1c. The trusted third will be discussed in more detail in Section 5.6.3. The session-management module is complemented by a platform management module. The platform management module allows the GPU server to request the power supply and the clock signal of the GPU to be switched on demand. With this facility, the prototype can power down the GPU when it is idle, reducing the power consumption by roughly 10 mW (see Section 6.4.5). The same module also initializes the GPU server by requesting the required MMIO resources and interrupts. Drivers for the computational cores, their corresponding MMUs, the shared L2-cache, and the power management unit (PMU) form the lowest layer of the GPU server, interfacing with the hardware through the GPU's MMIO interface. Notably, the core and MMU drivers are instantiated once for each physical core, and so are the corresponding job queues, which can hold one job per session; this corresponds to the number of job slots available per session.

Bootstrapping

In the preceding sections, there were occasional references to a trusted third as an essential player in the game of sanitizing mapping requests. In this sec-

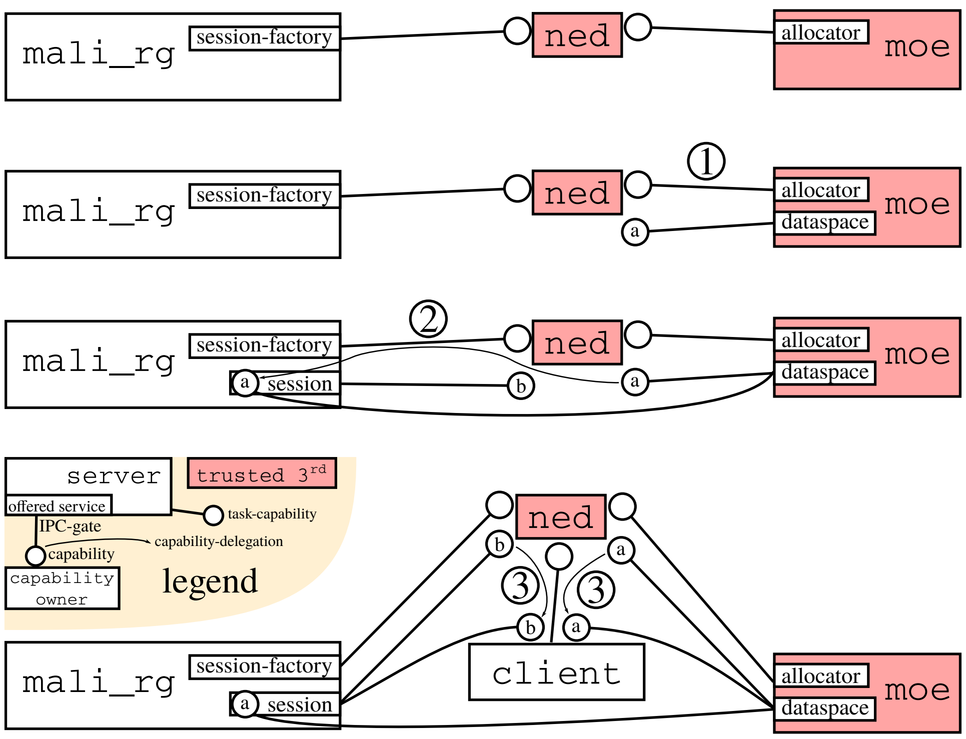


Figure 5.6.4: Bootstrapping of a new client in the context of Fiasco.OC and L4Re. Using `moe`'s allocator service, `ned` allocates memory ① destined to be used as graphics memory. With `mali_rg`'s factory service, it creates a new GPU-server session and delegates the newly allocated memory to the new session ② by means of capability a. When finally creating the new `client`, it delegates both the new GPU-server session and the graphics memory, by means of capability b and capability a, respectively, to the new `client` ③.

tion, the trusted third is introduced as it appears in an L4Re-based system. To that end, a closer look at the bootstrapping procedure of a new GPU client is warranted. A minimal L4Re-based system has already been discussed in Section 2.2.3 on Page 23. Starting from here, it shall be assumed that `ned` started an additional server, the GPU server `mali_rg`. Further, `ned` shall have access to the session-factory service of `mali_rg`. This situation is depicted at the top of Figure 5.6.4. First, `ned` allocates a region of memory, which shall be usable for rendering related operations ①. This memory is represented by a new instance of a dataspace service exposed by means of an IPC-gate capability ⓐ. In Step ②, the bootstrapper invokes the session factory and creates a new session ⓑ. While doing so, it delegates the previously acquired capability ⓐ to the GPU server, which associates it with the new session. Finally, in Step ③, `ned` creates the new client and delegates both the new dataspace service and the new session to it. The client can now use the memory represented by ⓐ and issue a request to map the memory into a GPU address space using ⓑ. Physical memory resources are specified in terms of dataspace offsets, and the GPU server can translate those into host physical addresses by means of the capability ⓐ. The contract that `ned` made with `mali_rg` in Step ② can be stated as follows: If a mapping request received via ⓑ can be translated into a physical address using the service behind ⓐ, then the request is legitimate. This is sufficient proof that the client was already in control of the physical memory resources affected. The bootstrapper `ned` and the root task `moe` vouch for this contract and thus constitute the trusted third.

In the prototypical implementation, L^{4}Linux was patched to receive all of its memory assignment through the bootstrapper, rather than requesting it from the root task on its own behalf. This enabled the L^{4}Linux instances to use all of its memory for rendering with the GPU. As a side effect, the GPU server was given access to all of the client's memory. This, however, is not an elevation of privileges, as the GPU server, by means of the DMA capabilities of the GPU, can already access all of the physical memory. It is already trusted with upholding the spatial isolation of the clients from one another as well as that of any other subsystem from its clients.

Integration with secure GUI

The same mechanism that was introduced in the previous section was used to associate the private framebuffers of the clients to the corresponding client session of the GPU server. Consider Capability ⓓ in Figure 5.6.5. It represents the private framebuffer of the client. With this capability, the client and the `mali_rg` server had a common namespace for the framebuffer resource. Consequently, clients were enabled to map their framebuffers into the GPU's address space, which allowed for seamless direct rendering. The `fb-drv` server, in this case, acted as dataspace provider. With two dataspaces associated with a client's session, the dataspace offset was no longer unique. Therefore, the otherwise unused lower bits of the dataspace offset were used for disambiguation in mapping requests issued by the client.

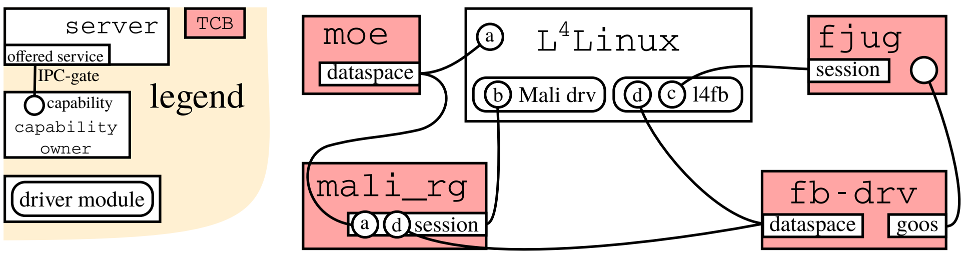


Figure 5.6.5: Communication channels after bootstrapping the GPU and secure GUI (output) subsystems omitting the bootstrapper `ned`. The left hand side is analogous to the outcome of Figure 5.6.4 with an instance of L^{4}Linux as client. Capability ⓐ represents the main memory of the L^{4}Linux instance, shared with `mali_rg`, and ⓑ represents the session interface by which the client can issue requests to the GPU server. The right hand side depicts the framebuffer infrastructure from Chapter 4. Capability ⓒ represents the session interface by which the client can issue buffer switch requests and receive VSYNC events. Capability ⓓ represents the framebuffer shared between the client and `mali_rg`, just as the main memory is by capability ⓐ.

Summary

In this chapter, a very compact GPU server design was presented that upholds memory isolation among GPU clients as well as between GPU clients and the TCB, that is, the system services and the underlying kernel. By concentrating on the most essential feature required to confine a DMA device, that is, memory isolation, and by placing the virtualization boundary at a very low level, the GPU server design became small, and at the same time, it exhibited great versatility and fidelity. Notably the GPU server waives memory allocation and assignment, services that are typically found in kernel GPU drivers. Instead, whenever clients wish to have memory resources accessed by the GPU, the GPU server requires a proof of ownership, here provided by a namespace for memory resources that is local to the GPU server, the corresponding clients, and a trusted system service, and that only includes the memory resources that the corresponding client can control. In this makeup, the GPU server makes heavy use of an MMU mediating the GPU's memory access and otherwise stays out of the way of any high-bandwidth memory transfers.

6 Evaluation

Whoever has tried to play video games inside a VM on their desktop computer knows that it is no fun. It is either slow and laggy or, if not both, the cooling system screams indicating hig heat development and therefore high power consumption due to the high system load. Mobile systems can afford neither. Therefore, the development of the secure GUI and GPU virtualization architecture was accompanied by a thorough performance evaluation, which led to various optimizations. The organization of this chapter is as follows: In Section 6.1, the methodology for this evaluation is presented, followed by a description of the experiments under evaluation in Section 6.2. The benchmarks used, some of which were specially devised, are described in Section 6.3. Subsequently, their results are discussed in Section 6.4. In the context of the results, the evolution of the prototypical implementation is discussed, with occasional excursions into performance implications of the underlying virtualizing system, that is, Fiasco.OC, L4Re, and L^{4}Linux. To top up this chapter, there is a discussion of the architecture's impact on the TCB and an assessment of the implications on the system's security if either of the subsystems is compromised in Section 6.5.

Methodology

For evaluating the runtime behavior of the prototype, off-the-shelf high-level benchmarks were used, as well as a series of custom-made microbenchmarks. The results of the former are presented "as is", with some exception discussed in the corresponding section. For the custom benchmarks, the infrastructure provided by the underlying device and software was used. This, however, needed some adjustments to reduce the measurement overhead and increase accuracy. This section discusses the facilities used and the modifications made.

Enhanced timer access

All of the custom benchmarks described in this chapter use the system call `gettimeofday` for generating time stamps, that is, where they do not evaluate trace buffer events, which will be discussed in Section 6.1.2. The default implementation of L^{4}Linux uses the `kclock` field of the KIP[1]. The `kclock` field is updated upon each timer tick, typically every millisecond. This granularity was to coarse for the purpose of evaluation. Instead, the free running counter (FRC) was used, which is part of the SoC's multi-core timer (MCT) functional block. It runs at 24 MHz clock frequency and therefore provides sub microsecond granularity. First, the FRC was exposed to the user space through an extension of the Fisaco.OC's debug interface. But, because this caused `gettimeofday` to incur an additional round trip to the Fiasco.OC kernel, the MCT's MMIO resources were exposed to the user space directly. The implementation of `gettimeofday` could thus read the FRC directly, with slightly reduced latency. Eventually, the runtime of a `gettimeofday` was determined to be .6 μs in the native case and 3.8 μs on L^{4}Linux. Exposing the MCT interface is, however, potentially dangerous and should not be done in a production system.

The Fiasco.OC trace buffer

Among the set of debugging tools with which the Fiasco.OC is equipped, is the trace buffer. The trace buffer is a region of pinned memory designated to hold events recorded with low latency. Trace-buffer events have a fixed

[1]See Section 2.2.3 on Page 18.

size of 64 octets; the first 34 contain the event's header including, among other items, the type of the event, a time stamp, and the registering CPU. The remaining 30 octets are the payload of the event, of which the layout depends on the event's type. Fiasco.OC supports a variety of event types, of which the recording can be activated through the kernel's debug interface. The event-recording feature is highly optimized; notably the activation of event recording is done through binary patching to spare the cost of an additional conditional branch.

The trace buffer was used for three different purposes: First, for gaining insight into the runtime behavior of the evaluated system. For this purpose, a tool for visualizing the trace-buffer was developed. Second, trace-buffer events were used as measuring points whenever the start and end of a duration were not within the same protection domain. For example: When measuring the duration between an interrupt registering in the kernel and a corresponding action taking place in a user-space application, both occurrences were instrumented to emit a trace-buffer event for later evaluation. Third, the trace buffer was used for discriminating measurements with respect to system noise sources, such as the occurrence of context switches within the measurement period.

Visualization

For visualization of the trace-buffer, two event types were the most important: Context switch events and vCPU events. The context switch event, if enabled, marks the occurrence of a context switch; it is emitted by the Fiasco.OC kernel. The vCPU events can also be enabled selectively and, if enabled, are emitted by the Fiasco.OC kernel. They mark five different conditions related to a vCPU's operation (Note that the numbers in the following enumeration where not chosen arbitrarily, but reflect the numerical representation internal to Fiasco.OC and thus the coding in the visualizations.):

0. **Resume**: This event is emitted immediately before the vCPU enters user space after the vCPU has invoked the resume operation. The resume operation is invoked when the vCPU wishes to resume execution in a secondary task. Furthermore, the vCPU may change its architectural state atomically. In the context of L^{4}Linux, this event marks the

point when the vCPU transitions from executing in the guest kernel to executing in a guest process.

1. **Asynchronous IPC (IRQ-object trigger)**: This event is emitted if an IRQ object that was registered with the vCPU was triggered, and the vCPU has interrupts enabled. When the vCPU resumes execution, it does so in the primary (or kernel) task at a predefined entry vector.

2. **Exception**: This event is emitted immediately before the vCPU is entered after an exception occurred while the vCPU was running with exceptions enabled. The vCPU resumes execution in the primary task at a predefined entry vector. In the context of L^{4}Linux, most of the time, this event type marks system calls issued by guest's user processes. But it can also mark other exceptions, such as the occurrence of an undefined instruction.

3. **Page fault**: This event is emitted immediately before the vCPU is entered after a page fault has occurred while the vCPU was running with page faults enabled. The vCPU resumes execution in the primary task at a predefined entry vector. In the context of L^{4}Linux, this means a page fault occurred while executing in a guest process, which is then reflected into the guest kernel for handling.

4. **Synchronous IPC (IPC-gate invocation)**: This event is emitted if an IPC-gate that was registered with the vCPU, was invoked and the vCPU was ready to receive. This mode of operation has no correspondence in a physical CPU interface. In the context of L^{4}Linux, this only occurs if the instance is acting as an L4Re server, e.g., offering driver services, such as driving the input devices, as discussed in Chapter 4.

With the context switch events, one graph per CPU was drawn over time, showing the currently executing context. A second graph was drawn showing the active protection domain (task/page table). Using the vCPU events, the second graph was extended to show switches to secondary tasks, which in this case represent switches to and from user level processes inside the VMs. These secondary task switches were annotated using the numbers described above. As a rule of thumb, a '0' can be read as a switch to a secondary task, whereas any other number indicates a switch back to the primary task with the reason encoded in the number. These two graphs give a good understanding about what is going on in the system. To reason about why the system behaved in a certain way, the graphs where augmented with other events, such as IRQ object triggers, IPC gate invocations, timer ticks, and a variety of custom events. With this augmentation, the communication between the subsystems could be observed; also, it allowed reasoning about scheduling decisions.

Modification

Fiasco.OC used the `kclock` for time stamping of trace-buffer events. Once again, the inherent precision of this method was to coarse for the intended measurements. Therefore, the free running counter (FRC) of the SoC's multi core timer (MCT) was used instead. The FRC was configured to run at 24 MHz, providing a sub microsecond granularity.

Another more subtle detail about trace-buffer events was the way in which numeric type identifiers were assigned. As Fiasco.OC advanced, its developers chose to use a dynamic assignment scheme for more and more event types. In consequence, the identifiers for a specific event could vary from build to build. This being unacceptable for offline analysis and visualization, static predefined identifiers for most event types were reintroduced.

Tracebuffer Extension

To extract trace-buffer information, a driver by the name `l4_ktrace` was built into L^{4}Linux that exposed the trace buffer to the guest's user space. A complementary tool (`dktrace`) was created, which could read the trace buffer from the guest kernel and dump it into a file. To generate a meaningful—that is, consistent as well as human readable—visualization, however, some extensions were needed, exposing meta-information about the traced objects. The traced objects, contexts, and tasks appear in the trace buffer either by their debug identifier (DBG-ID) or by a pointer. Both are not easy to map to specific applications by the user. Above this, pointers may refer to different objects over time. This happens when an object is destroyed and its resources are reused. Therefore, an extension was built into Fiasco.OC to expose a list of objects mapping DBG-ID and pointer to a human readable string identifying the resource. However, by the time a trace-buffer was dumped, objects that appear within the trace may have been destroyed; they would therefore not show up in this object list. Therefore, special events carrying the missing information were inserted into the trace upon object destruction. From these additional events and the object list, a lookup-table was generated off-line, mapping either DBG-ID or pointer in conjunction with the time of occurrence in the trace to a specific object.

Tracing in native Linux

To provide comparable measurements on a native Linux, a similar tracing mechanism was built into Linux as a kernel module. The module allocated a trace buffer and a structure holding the position-independent geometry of the buffer as well as the current write pointer. Both were exposed to the user in a read-only fashion through the invocation of the mmap mechanism on the device node `/dev/jd_trace`. An IOCTL interface allowed the user to emit custom events comprising a short string and three word-sized integers.

Measurement overhead

Trace-buffer events, although much cheaper than formatted output instrumentations, are not without cost in terms of the runtime overhead incurred. Moreover, the cost differs depending on from where the context trace events are being issued. Five contexts were evaluated: the Fiasco.OC kernel, the L^4Linux guest kernel, the L^4Linux user-space process, the native Linux kernel, and the native Linux user-space process. An event emitted by the kernel amounts to a memory access, whereas an event emitted from user space must first transition into the kernel. The cost was measured by issuing multiple events in a tight loop and by evaluating the advance of the events' timestamps. For events issued from the user space, this was straightforward. In order to perform the same experiment for the respective kernels and guest kernels, the event emitting loop was built into them, and a trigger was exposed to the their respective user spaces through interfaces as follows: Fiasco.OC was amended with a new extension to the debug interface allowing to trigger a burst of trace buffer entries. The L^4Linux driver `l4_ktrace` was amended with an IOCTL call forwarding the request from its user space to the underlying μ-kernel. Similar IOCTL calls were added to `jd_trace` and `l4_ktrace` to trigger event bursts in the native Linux kernel and L^4Linux

Issuer of trace buffer event		Linux	Fiasco.OC
kernel	[μs]	0.85	1.2
user space / guest kernel	[μs]	1.5	1.3
guest user space	[μs]	NA	5.6

Table 6.1.1: Cost of issuing a trace-buffer event in microseconds. The measurements represent the time between two successive events issued in a tight loop by any of the issuers in the left column. Each value is the mean of at least ten thousand samples.

respectively. The outcome of this experiment is presented in Table 6.1.1.

Fine grained CPU time accounting

For accounting CPU time, Fiasco.OC uses the `kclock` with a typical granularity of one millisecond. This granularity is too coarse to capture short IO-prone workloads. In consequence, IO-prone loads, such as driver threads, were often not accounted for. More often, the idle thread was accounted for the CPU time in such cases. An off-line evaluation of the trace-buffer, using both the coarse-grained method of the Fiasco.OC kernel and a more finely grained method, showed that the discrepancies in the measured system load accumulated to between two and five percentage points, with the coarse-grained method favoring the idle thread.

Measuring a lower-than-true system load naturally skews the comparison unfairly. Furthermore, power management can hypothetically benefit from measurements that are more precise, because it can react quicker to a beginning user interaction and scale up CPU speed. Therefore, Fiasco.OC was patched with a more fine-grained CPU time accounting, using the FRC instead of `kclock`.

Experiments

The prototype as described in Chapter 4 and Chapter 5 was compared against a native setup of Cyanogenmod 10.2 and two intermediate stages of prototypical implementation: Namely *pass-through* and *copying*. In the pass-through scenario, the GPU is driven directly by a L^{4}Linux compartment. That is, the MMIO-resources were passed through to the unmodified Linux driver, and the corresponding interrupts were delivered through IRQ-objects. This is inherently insecure because the corresponding guest kernel can easily evade isolation by means of the GPU with the techniques described in Chapter 3. However, it makes for a good performance evaluation vehicle because it incurs the CPU virtualization overhead, such as system call, shadow paging, and notification overhead, without the expected overhead incurred by gaining GPU interposition through GPURG. The second intermediate scenario, *copying*, uses the compositing approach in the framebuffer output path as described in Section 4.1.2.

The GPURG scenario underwent a series of evolutionary steps in the course of its development and evaluation. In the beginning, mapping requests issued by the user were broken down into the individual pages, and one request per page was forwarded to the GPU server at a time. The GPU server, in turn, issued one translation request to the corresponding dataspace provider to learn the corresponding host physical address and sanitize the request at the same time. Both steps induced massive overhead, and so the guest-physical to host-physical translations were cached by the GPU server and mapping requests were accumulated and sent to the GPU server in batches. The batch buffer was flushed when full or before control returned to the user so as to exhibit a behavior consistent with the user's expectations. Besides these optimizations, several cache policies were investigated for handling page-table memory in the GPU server. However, at this point, the evaluation uncovered that the largest portion of the overhead incurred during context population was not to be found in the GPURG architecture, but rather in the way L^{4}Linux populates process address spaces. All of these measures lead to enhancements in GPU context building speed, which influences the time that an application requires to load, or for a game to change a scene. And even though it was somewhat out of the scope of this work, attempts were made to lower the process population overhead. The context building overhead was measured with the `map_tool` benchmark discussed in Section 6.3.3, with its results presented in Section 6.4.3.

It was expected that the GPURG interposition also induces a cost in the GPU job submission and the completion notification mechanisms. To quantify this cost, a very short-running rendering job was devised, which was precompiled and run many times to gather samples of the quantities desired. Measurements were taken in the native, pass-through, and GPURG scenarios where applicable. The `job_tool` benchmark discussed in Section 6.3.2

generated the corresponding results, which are presented in Section 6.4.2.

Besides the GPU interposition cost, the cost incurred on the input event path and in framebuffer handling needed to be taken into account as well. The input path was evaluated with respect to the input event latency. The output path was evaluated with respect to the VSYNC notification latency and the cost of a buffer-swapping request. All three were evaluated in the context of the native and the GPURG scenarios. The path-through scenario does not differ from the GPURG scenario with respect to the input and output interposition and therefore is not considered here. The input and framebuffer interposition costs were measured using the `inputer` and `vsyncer` benchmarks, respectively. The former is presented in Section 6.3.5, and its results are discussed in Section 6.4.4; the latter is presented in Section 6.3.4, and its results are discussed in Section 6.4.4.

The design of the secure GUI was driven by security requirements. But as the window manager `mag` already met these security requirements, the motivation for the architecture presented here was performance and power efficiency. It remains to be shown that this architecture not only performs better in terms of achievable frame rate, but also in terms of power consumption. Therefore, in Section 6.3.6 an experimental setup for measuring the power consumption of the secure smartphone is presented; and, in Section 6.4.5, the results are discussed.

Benchmarks

In the previous section, the experiments evaluated were introduced, and their rationales were discussed. In this section, the details of the benchmark tools used to evaluate the experiments are presented.

High-level 3D benchmarks were used to gain a general impression of the graphics performance that the prototype was capable of; the microbenchmark `job_tool` was devised to measure the extra load induced on the CPU by measuring job submission and notification delay; the microbenchmark `map_tool` measures the overhead incurred due to the GPU shadow paging scheme; the microbenchmark `vsyncer` measures the latency incurred by the VSYNC notification and the buffer swapping request, and the microbenchmark `inputer` measures the latency incurred on the input path. Eventually, an experiment was set up for measuring the power consumption of the smartphone.

Highlevel 3D benchmarks

The benchmarks *OpenGLCube*, *OpenGLBlending*, *OpenGLFog*, and *FlyingTeapot* are part of the benchmark suite 0xbench [1]. All four are written in Java, and they use Android's OpenGL ES Java API. The 0xbench benchmark suite was modified to measure the system's utilization during each benchmark. To that end, Linux's `/proc/stat` interface was evaluated for CPU idle time. In order to capture the global system's CPU idle time in the virtualized case, this interface was amended with the idle time as measured by Fiasco.OC.

In addition, the popular first-person shooter Quake III Arena was used for benchmarking. To that end, the demo `FOUR.DM_68` was run in timedemo mode. It was run on QIII4A [8], the Android port of the *ioquake3* [5] engine. Two modification were applied to the engine, the source code of which is publicly available. It was modified to always run in timedemo mode and to use the Android logging API for printing messages, including the benchmark results.

Microbenchmark `job_tool`

The `job_tool` benchmark issues a short-running program to the GPU. This program was precompiled and linked to cut the CPU-bound load out of the measurement. The benchmark operates under the assumption that this job always completes in same period of time because the GPU hardware is the same and fixed to the same clock frequency in all scenarios. The jobs are issued by a user-space application either running on the Linux kernel (in the native case) or running on the L^{4}Linux guest kernel (in all other scenarios). To issue the jobs, the user-space application calls the ioctl calls `MALI_IOC_GP2_START_JOB` and `MALI_IOC_PP_START_JOB` to start the GP and PP jobs, respectively.[2] It then calls `MALI_IOC_WAIT_FOR_NOTIFICATION` to wait until the corresponding job has finished. This measures the corresponding cost when taking timestamps before and after the job submission. For measuring the notification cost, the trace buffer was evaluated after

[2]The ARM Mali GPU evaluated here has special computational cores for vertex shading (GP) and fragment shading (PP). The structure of the GPU was introduced in Section 5.6.

MALI_IOC_WAIT_FOR_NOTIFICATION returned. The IRQ-object trigger event served as a starting point for the measurement; in addition, the top-half interrupt handler of the guest kernel's GPU stub driver was instrumented to emit a trace-buffer event, which served as the endpoint. As the top-half handler is the first point at which this interrupt registers in the native Linux kernel, this measurement has no counterpart in the native scenario.

The trace-buffer also served as a source of information for reasoning about outliers. With the trace-buffer, measurements were discriminated with respect to the number of context switches occurring during the measurement period. This gave a good indication of whether the measurement had been skewed by system noise.

Microbenchmark `map_tool`

The `map_tool` benchmark performs a sequence of five operations, which are related to populating a GPU context with memory, and further measures their cost. After allocating a physical memory buffer, it (a) attaches it to the GPU address space, (b) attaches it to its own process address space, (c) touches every page by writing to the first word of that page, (d) releases it from the GPU context, and (e) releases it from the process address space. The cost of two of these operations, (a) and (d), is significantly influenced by the interposition of the GPU server. The others (b), (c), and (e) are not influenced by this interposition; however, they illustrate the impact of the para-virtualization of L^{4}Linux.

In the following, every operation is broken down into steps illustrating where the individual experiments differ. This information is vital to understanding the results discussed in Section 6.4.3 below and may be used as a reference.

a) **Attach to GPU address space:** The `map_tool` application uses the ioctl call `MALI_IOC_MEM_ATTACH_UMP` of the Mali user-kernel interface. In turn, the unmodified kernel driver—this applies to the scenarios "native" and "pass-through"—allocates page tables for the GPU's MMU as needed and creates the page table entries accordingly. In the para-virtualized case with GPU server interposition, the guest kernel driver breaks the request down into pages; each page's base address is then translated into a dataspace offset, which, together with the mapping target and address space identifier, is sent to the GPU server; which allocates the page tables on the client's behalf and creates the page table entries after translating the dataspace offsets into host physical addresses.

b) **Attach to process address space:** The `map_tool` application calls `mmap` on the UMP user-kernel interface to map the allocated buffer into its process address space. In turn, the (guest) kernel driver uses the Linux-API call `remap_pfn_range` to map the already allocated physical tiles into the process address space eagerly. There are subtle differences between the implementations of the native and the virtualized experiments. Both implementations modify their page-tables directly, however, only the native case cleans the cache line using `DCCMVAC`[3] on the new page table entry. This cache maintenance operation is not necessary in the virtualized case because the guest's page tables are never walked by MMU hardware, but rather by a software page-table walker. Furthermore, the ARM flavor of the Linux kernel maintains duplicated page tables. One duplicate is for the MMU hardware to walk, whereas the other accommodates additional flags, which the memory management subsystem of Linux needs but which cannot be absorbed by the architectural page tables. L^{4}Linux dropped the former.

c) **Touch pages:** In this phase, the application performs a single write access on every page of the attached buffer. While in the native case the effective page tables are already populated at this point, the shadow page tables in the virtualized cases are not. This means that in the native case this access amounts to a TLB miss followed by a page-table lookup and a subsequent cache line allocation. In the virtualized cases, however, a page fault occurs which is reflected into the guest kernel, causing a task switch; the guest kernel resolves the page fault by walking its page table corresponding to the current process and resumes the process, causing another task switch; in resuming the process it passes the result of the page fault resolution down to the hypervisor, which updates its shadow page table accordingly. Now that the effective page table is populated, the remainder of the operation commences as in the native case.

[3]See ARM Architecture Reference manual Section B4.2.1 for more details [2].

d) Release from GPU address space: To release the memory from the GPU address space, the application calls the ioctl `MALI_IOC_MEM_RELEASE_UMP`. The unmodified kernel driver removes the corresponding mappings from the GPU MMU's page tables and frees unused page tables accordingly. In the para-virtualized case with GPU server interposition, the guest kernel driver again breaks the release request down into pages; each page is specified by its mapping target, that is, its GPU virtual address, and the address space identifier; the GPU server removes the specified entries from the corresponding page tables and frees the unused page tables accordingly.

e) Release from process address space: The application calls `munmap` on the UMP user-kernel interface to release the memory from the process address space. There is no special implementation of `munmap` provided by the UMP driver; instead, the default implementation of the Linux kernel is executed. This implementation differs in the virtualized case from the native case only at the lowest level. Where the native Linux code would remove an entry from its page table, followed by appropriate cache and TLB maintenance operations, the L^{4}Linux code updates its own page table and issues an appropriate request to the hypervisor, which in turn updates the shadow page tables and performs the cache and TLB maintenance accordingly.

Microbenchmark `vsyncer`

The `vsyncer` application measures the latency of the vertical synchronization (VSYNC) notification event as it builds up traveling through the system. It does so by periodically evaluating the trace-buffer, which is filled with events recorded as the VSYNC notification passes through the various subsystems. In the virtualized case, this path is as follows: The beginning of the VSYNC-notification path is denoted by the IRQ-object trigger event recorded by the Fiasco.OC kernel. The next waypoint is the display controller driver, which forwards the event to the framebuffer switch `fjug`. After recording, an event from the framebuffer switch notifies the currently active client, the interrupt handler of which records yet another time stamp. The last event is recorded by an Android service, and it denotes the moment the VSYNC event finally reaches the user space. In the native case, only the interrupt handler of Linux, which is the closest equivalent to the IRQ-object trigger event in Fiasco.OC, and the Android service are instrumented.

Another quantity that the `vsyncer` application measures is the cost of swapping the visible framebuffer with the back-buffer. To that end, the components involved in this operation were instrumented, thus emitting tracebuffer events. Then, while the `vsyncer` tool gathered samples, buffer swaps were provoked by running graphics benchmarks and through user interactions. In the different experiments, the paths are composed as follows:

The native Linux driver accesses the display controller directly. It reconfigures the display controller to scan the back-buffer, instead of the visible framebuffer, as soon as it starts scanning the buffer the next time. Thereby, the back-buffer becomes the new visible framebuffer and vice versa. In the virtualized experiments, the panning request is forwarded first to the framebuffer switch `fjug` and then to the display controller driver `fb-drv`, which performs the display controller access on the clients' behalf. There is a subtle difference in how the display controller is manipulated. The display controller has shadow registers for preconfiguring up to three scan-out regions per overlay. The native driver uses two of these, one for each of the possible buffers. Causing a swap of the preconfigured buffer amounts to a mere flip of a bit in a different control register, indicating that the respective other buffer shall be scanned. In the virtualized case, there is an indefinite number of possible buffers, depending on the number of existing clients. Here, the framebuffer switch decides which framebuffer to display next. Consequently, the display controller configures an inactive scan-out region slot with the next visible framebuffer before arming the device for the framebuffer switch. In the "copying" scenario, the panning request is used as a signal to copy the content of the client's framebuffer into the visible framebuffer, which is never changed. The copying is performed by the framebuffer switch, and no further communication with the display controller driver is necessary.

Microbenchmark `inputer`

Just like the VSYNC notification and the swap buffer request, input events travel through multiple subsystems. As input events visit the various stations, they accumulate latency. To measure these latencies, the subsystems involved were instrumented to emit the tracebuffer events. And analogous to the `vsyncer` tool, the `inputer` benchmark periodically collects samples by evaluating the tracebuffer.

Only events generated by the touch screen were evaluated. The first station on the path of an input event, as can be reflected by the tracebuffer, is when an interrupt registers in the respective kernel (`input_kernel`). In the native case, only Android's EventHub was instrumented, marking the moment when the event finally reaches the user space. In the virtualized case, the kernel then notifies the input driver, which fetches the input event from the device via an I^2C bus. Once the event has been processed, the input switch (ijug) is notified, which is when the second event (`input_drv`) is emitted. The third event (`input_ijug`) occurs once the destination for the event was determined, and the guest is being notified. When a virtual input interrupt is processed by the guest kernel, a fourth event (`input_l4lx`) is written to the tracebuffer. Finally, the EventHub of the Android middleware emits an event (`input_hub`) whenever it is notified of an input event.

When the `inputer` benchmark awakes, it evaluates the immediate past as it is recorded in the tracebuffer, seeking events that where issued by the EventHub. Once such an event was found, it keeps searching backwards in time until it finds the corresponding `input_l4lx` event and then the corresponding `input_ijug`, `input_drv`, and `input_kernel` events. It only records the sample path if the tracebuffer is in a sane state, that is: No wraparound has occurred during sampling, and the same path has not been sampled in an earlier round.

Power measurement setup

The experimental setup for the power consumption measurement is depicted in Figure 6.3.1. The power consumption was measured with a Hardkernel[4] ODROID Smart Power device and a Samsung Anyway S101 Jig, which connects to a Samsung Galaxy SIII by a special test cable. At a first glance the cable's connector looks like a normal micro USB connector; however, it has six extra pins. Through this special connector, the Anyway Jig can simulate a battery, which allows the SIII to be operated without a battery installed. Thus, all power consumed by the SIII device is supplied by the ODROID Smart Power, which at the same time, measures the power

[4] `www.hardkernel.com`

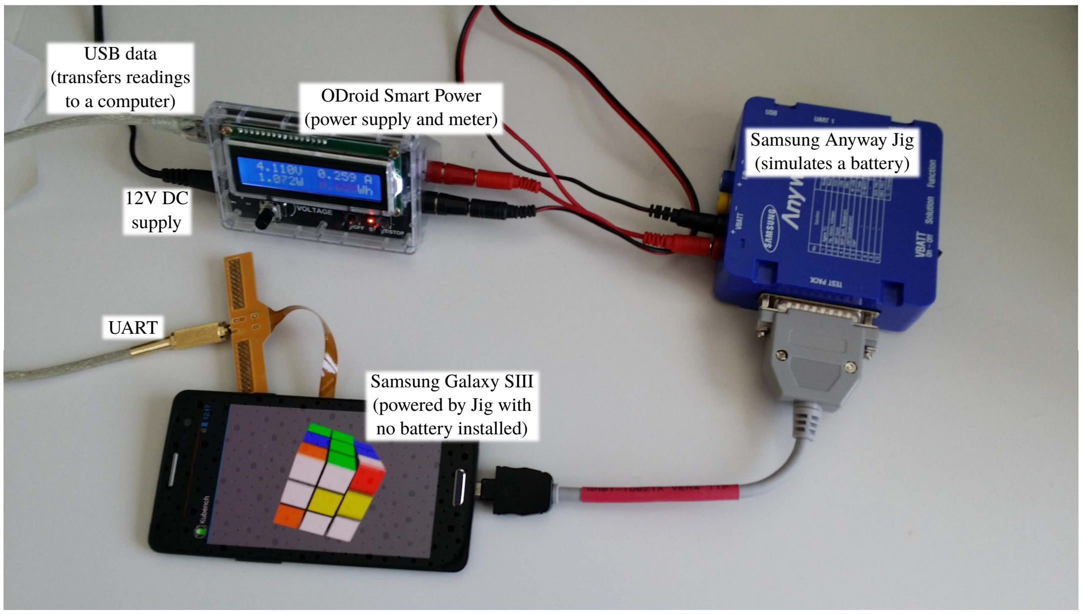


Figure 6.3.1: Power measurement setup: The device under test is connected to a Samsung Anyway Jig, allowing operating the device without a battery. An ODroid Smart Power serves as both power supply and meter. It is connected to a computer via USB for the purpose of recording measurements.

consumption. The ODROID Smart Power sends readings to a connected computer at 100 ms intervals, or with a 10 Hz frequency. However, the IC that takes the readings inside the ODROID Smart Power—a Texas Instruments INA231—is configured to produce a sample only every 263.8 ms. It produces samples that are averaged over 16 internal samples at a little under 4 Hz.

Results

In the previous sections, the foundations were laid for the evaluation results presented in this section. The benchmark results are presented in the same order that the benchmarks were introduced. So without further ado, here are the results of the high-level 3D benchmarks:

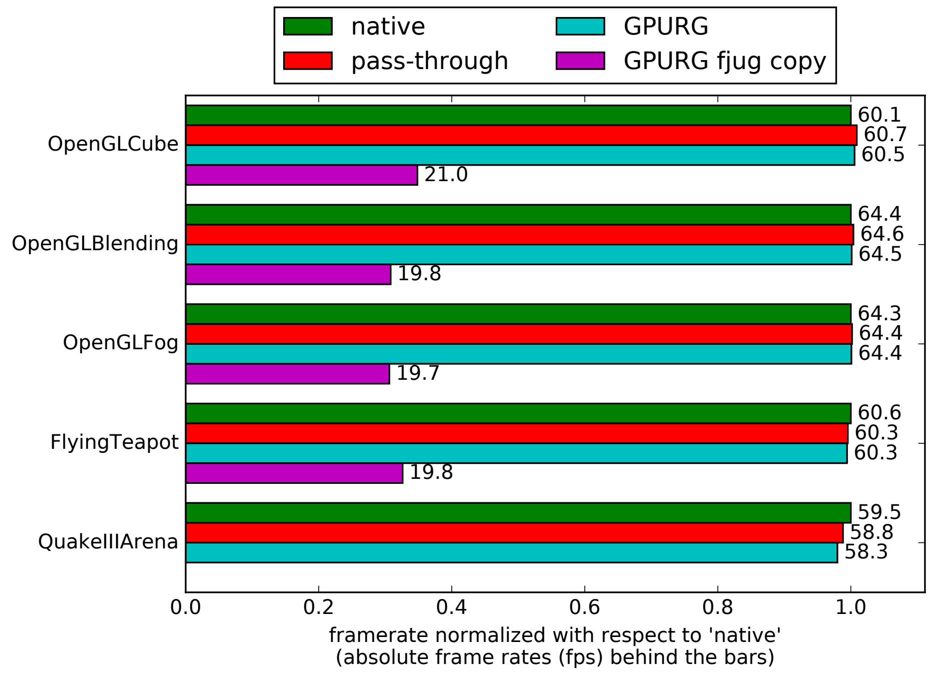


Figure 6.4.1: High-level 3D benchmarks. OpenGLCube, OpenGLBlending, OpenGLFog, and FlyingTeapot are benchmarks of the 0xbench benchmark suite [1]. The fifth benchmark is the `FOUR.DM_68` demo of QuakeIII Arena run in timedemo mode.

High-level 3D benchmarks

The results of the high-level 3D benchmarks are presented in Figure 6.4.1. Leaving the copying experiment aside, all virtualized scenarios achieve native performance in all benchmarks. Satisfying as this may be, this result does not characterize the expected virtualization and interposition overhead very well. The benchmarks are synchronized with the display controller's refresh rate of 60 Hz. More interesting was the investment in terms of CPU cycles needed to achieve this. To that end, the system utilization during the benchmarks was measured. Figure 6.4.2 depicts these results in terms of relative idle time,

$$T_I = \frac{\sum_{n=0}^{N-1} \Delta t_{\text{idle}}^n}{N \Delta t_{\text{wc}}} \tag{6.1}$$

where N is the number of CPUs in the system, and Δt_{idle}^n is the increase in idle time of the CPU n in the elapsed wall clock time period Δt_{wc}. The benchmarks all induce different CPU utilizations in the system. OpenGLCube and

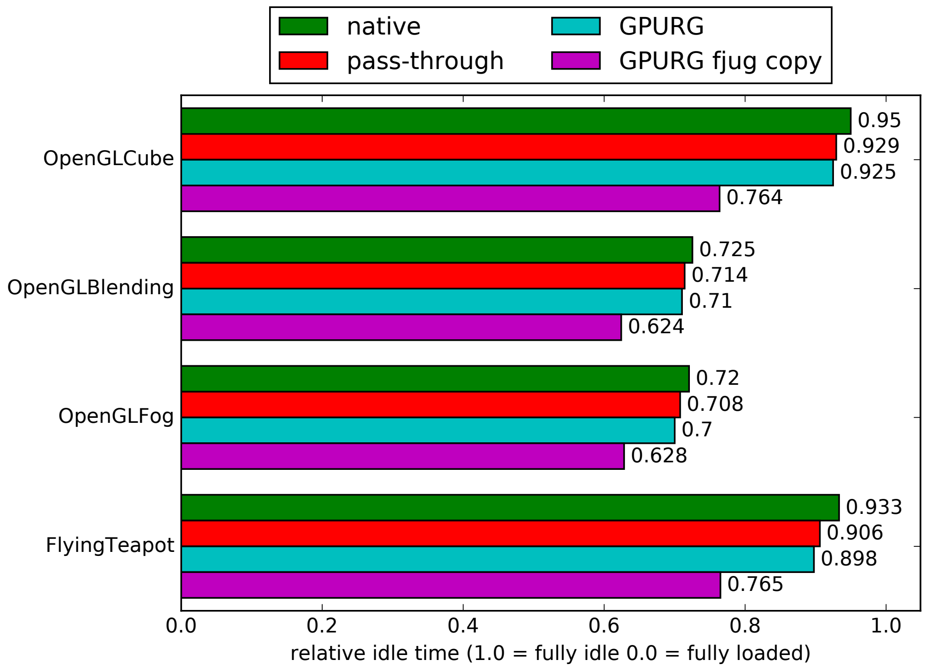


Figure 6.4.2: High-level 3D benchmarks. OpenGLCube, OpenGLBlending, OpenGLFog, and FlyingTeapot are benchmarks of the 0xbench benchmark suit [1]. This figure shows the relative CPU idle time during the runtime of the respective benchmarks.

Flying Teapot leave the system mostly idle, while OpenGLBlending and OpenGLFog drive the equivalent of one CPU[5] into saturation. The comparison between the experiments, native and pass-through, nicely illustrates the CPU virtualization overhead, whereas the comparison of pass-through and GPURG shows the additional cost of GPU interposition. It can be seen that the bulk of the cost is to be found in the former with only a little extra cost due to the GPU interposition.

These results in themselves represent the success of achieving the set goal of providing a secure GUI with graphics acceleration to the virtualized secure smartphone, wasting as few CPU cycles as possible. However, when taking into account the copying experiment that mimics the compositing approach of `mag`, it becomes apparent how large the improvement is, with respect to from where this work started. The achievable frame rate with software

[5]The system discussed here has four CPU cores, that is, one saturated core amounts to 25% CPU utilization.

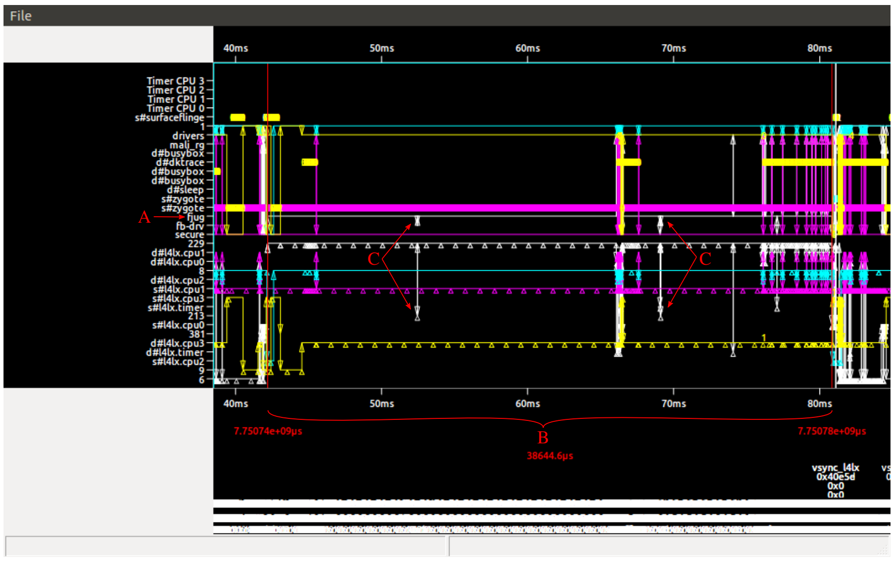


Figure 6.4.3: Visualized trace of `fjug` (A) copying a frame to the visible framebuffer. It takes nearly 40 ms (B) to complete, missing two VSYNC interrupts (C) on the way.

compositing is a third of the target rate of 60 Hz, that is, the refresh rate of the display controller, at a greatly increased system load. Figure 6.4.3 illustrates this by visualizing a trace of the `fjug` server copying a single frame for nearly 40 milliseconds, missing two VSYNC interrupts on the way. Here, the copying is single-threaded. It is certainly conceivable to increase the achievable frame rate by parallelizing the copying. But, provided one does no sooner run into bus contention, this would only drive up the load on the system.

Microbenchmark `job_tool`

Table 6.4.1 shows the results of the job tool benchmark. Notably, the submission times of the PP are considerably higher than of the GP in all of the experiments. This can be explained by the makeup of the corresponding job descriptions. For starting a PP job, more registers need to be configured, and consequently, their values need to be propagated through the driver stack. Directing the attention to the job submission time differences across the experiments, one can see, unsurprisingly, that the cost increases with the introduction of virtualization (pass-through) and again with the introduction of GPU interposition (GPURG). The former introduces roughly 40 % overhead, a little more for GP and a little less for PP jobs, and the latter nearly triples the native toll. Looking only at the ratios, these numbers look terrifying. But did not the high-level benchmarks tell a very different story? Indeed, one must put the absolute numbers into context. Considering a target frame rate of 60 Hz there is 16.6 ms time to render a frame. Of this time, 32.3 μs (42.3 μs) is lost to GPU interposition in the submission of a GP (PP) job. Under the assumption that one frame can be rendered with one set of jobs, where a set of jobs comprises one GP and four PP jobs, the overhead amounts to 1.2 % of the available time. The notification overhead adds another 1.5 % totaling an increase in system utilization of 2.7 %, which suddenly does not look so bad after all. This overhead, of course, scales with the number of jobs issued, that is, the smaller the number of jobs, the less the impact on the performance due to GPU interposition. This is well in line with the Mali GPU optimization guide [10], which advises to keep the number of draw calls—which translate into job submissions—at a minimum (see [10] Section 10.1).

Microbenchmark `map_tool`

In this section, the results of the `map_tool` benchmark are presented and discussed with respect to each of the steps (a) through (e) as described in Section 6.3.3. A brief survey of usage patterns, including web browsing, 3D benchmarking, and video gaming, showed that typical mapping sizes range from 30 to 1,800 pages, with a median of 900 pages. With the `map_tool` benchmark, measurements were taken for 100 through 2,200 pages at increments of 100. Each of the plotted data points represents the median of ten

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

Table 6.4.1: 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) in the different experimental configurations, native, pass-through, and GPURG.

thousand measurements.

GPU interposition

Figure 6.4.4 shows two evolutionary steps the architecture underwent. It shows the attach Step (a) of the context building procedure, measured with different optimizations enabled. The first attempts were terribly slow (none). Batching the mapping requests improved the performance enormously (map batching). Additionally caching the guest-to-host physical address translation in the GPU server improves the performance by yet another order of magnitude (both). The release operation is not influenced by either optimization. This is due to that for the release operation, only the GPU address range needs to be specified. Neither is guest to host translation required, which would benefit from caching, nor are the regions to be released scattered in the GPU address space, which would benefit from batching.

A more subtle matter is the choice of the right cache policy for the memory designated for the use as GPU page tables. The cache attributes of the corresponding memory regions must be thoroughly coordinated with the cache maintenance operations. Moreover, this is highly dependent on the SoC's implementation. In the system discussed here, the GPU and the CPU share solely the main memory (L3). Higher levels of caches (L1 and L2) are neither shared nor snooped. This means that all modifications performed by one of these agents need to affect L3 before the other may observe it. Likewise, any cache of the latter agent needs to be invalidated so as not to augment the view on L3. The GPU's MMUs, being part of the GPU's functional block, succumb to the same regime, or more specifically, their

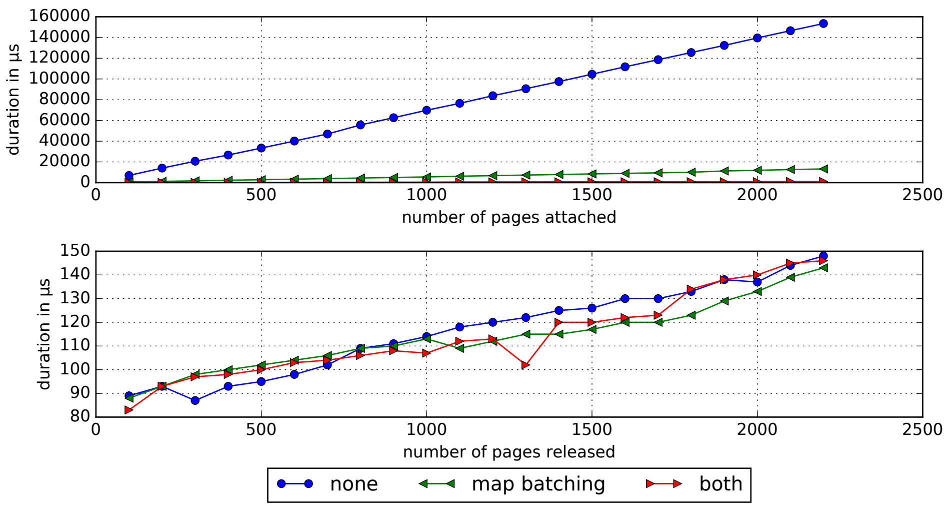


Figure 6.4.4: These graphs compare the two optimizations map batching and translation caching. Plotted is the time it takes to attach (top) and release (bottom) buffers of increasing sizes given in pages of 4 KiB to and from a graphics address space. The measurements refer to the GPURG architecture with no optimization (none), with map batching, with map batching and translation caching (both). The plotted values were determined as the median values of 10,000 measurements.

page tables do.

To choose a suitable cache policy for the memory pages destined to serve as page tables and page directories of the GPU, three experiments were implemented and tried. The three strategies were: Uncached, Buffered, and Cached. The Uncached strategy causes the CPU to stall until a memory access (read or write) has taken effect in L3; this strategy requires no further maintenance on the CPU side.

The Buffered strategy allows the CPU to use the store buffers of the underlying caches, which allows the CPU to proceed as long as the store buffers can accommodate any write access. As no cache lines are allocated, read access propagates all the way to L3. The store buffers are constantly drained;[6] so modifications propagate to L3 in a timely manner regardless of the maintenance steps taken. However, to be certain that a modification has taken effect at a given time, a synchronizing operation must be performed. This is done through a data sync barrier (`dsb`) instruction for L1, and a cache

[6]The Technical Reference Manual of the ARM L2C-310 [39] specifies nine circumstances that cause the store buffer to drain. The ninth reason is that a store buffer slot ages beyond 256 cycles. Other circumstances are strictly ordered memory access and hazards, such as a cacheable read at an address matching the address of the data in the store buffer.

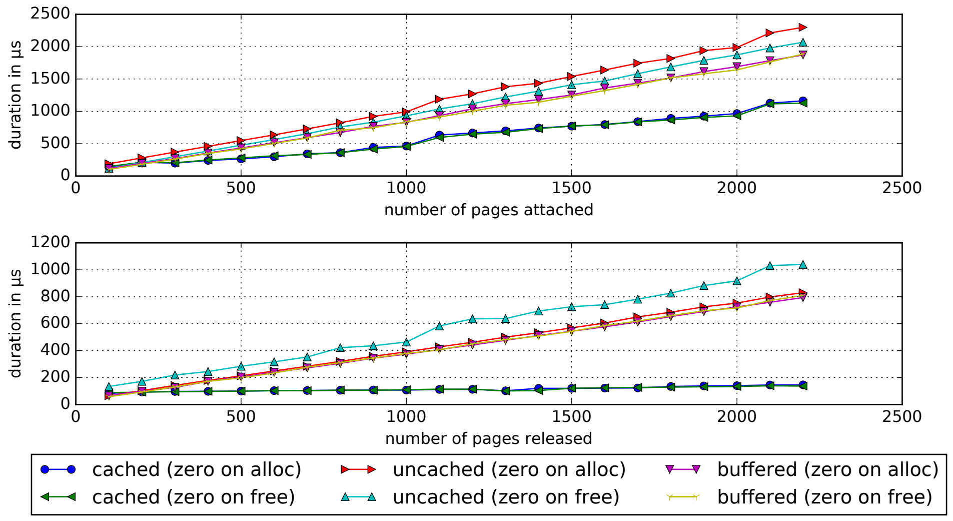


Figure 6.4.5: These graphs compare three caching strategies for GPU page tables, uncached, buffered, and cached and two page blanking strategies, zero on alloc and zero on free. Plotted is the time it takes to attach (top) and release (bottom) buffers of increasing sizes given in pages of 4 KiB to and from a graphics address space. All measurements where taken in the GPURG experiment with the optimizations map batching and translation caching enabled, and all values were determined as the median of 10,000 measurements.

sync operation for L2, which is initiated through the MMIO interface of the L2-cache (here ARM L2C-310 [39]). Consider though that these operations do not necessarily trigger the draining; rather they stall the CPU until the draining has completed.

The Cached strategy allows the CPU to make use of the cache RAM, with cache lines being allocated upon read or write access; therefore, modifications can linger in the caches indefinitely. To propagate modifications to L3, the dirty cache lines need to be cleared and thus written back to the lower levels.

Figure 6.4.5 illustrates the cost of the different strategies. The slowest of the strategies is clearly Uncached followed by Buffered. The Cached strategy beats both by far, for allocations of 300 and more pages. However, it may induce secondary cost by evicting other tasks' working sets from the cache.

Another matter, which is illustrated in Figure 6.4.5, concerns when page table pages are zeroed. Two strategies were implemented: zero-on-allocation and zero-on-free. By the former, pages destined for the use as page tables

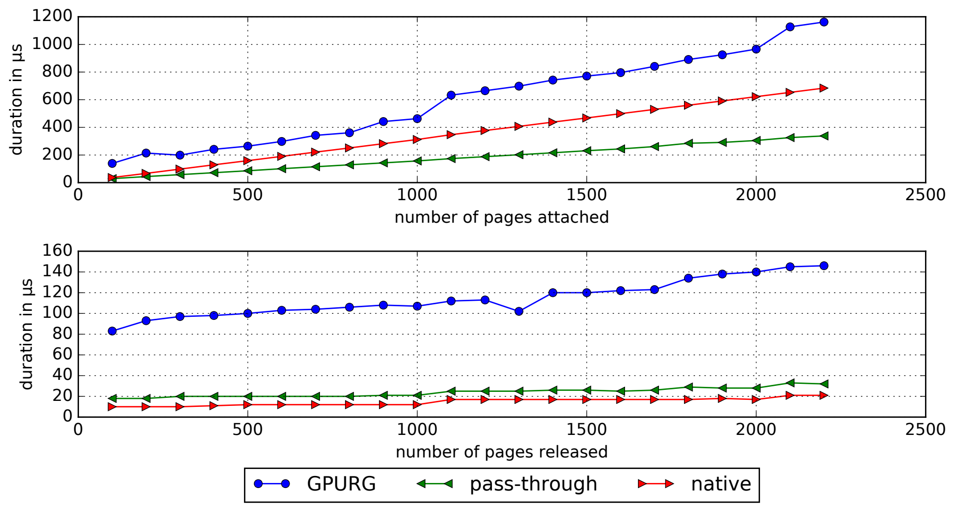


Figure 6.4.6: These graphs compare the three experiments, GPURG, pass-through, and native, with one another. Plotted is the time it takes to attach (top) and release (bottom) buffers of increasing sizes given in pages of 4 KiB to and from a graphics address space. All values were determined as the median of 10,000 measurements.

were zeroed when fetched from the page pool. By the latter, zeroing was performed upon freeing a page so that the page pool holds only zeroed pages invariably. The choice that is inherent to this design decision is whether the cost of zeroing a page table page is incurred at load time of an application or when it is torn down. This cost can be made out clearly in the Uncached strategy. However, for the Buffered and Cached strategies, it turned out to be insignificant.

Comparing the experiments, “native” and “pass-through”, with “GPURG”, with respect to the attach operation (a) cost, yields some interesting results. First, the GPURG architecture yields an overhead of about 30 % versus the “native” experiment. Interestingly, the pass-through experiment performed even better than the native one. An investigation of this matter yielded that the L2 sync operation discussed earlier was simply not implemented in L^{4}Linux. At this point, it must be interposed that the original GPU driver uses the Buffered strategy for GPU page table access. Thus, hypothetically, there was a race between the page table manipulation and the page-table walk of the GPU’s MMU. In practice, however, this race never manifested; probably, this was because the grace period between the page table manipulation and the activation of the GPU was generally to long.

Out of curiosity, an experiment was conducted during which the concluding L2 sync operation was left out in the GPU Server. While this showed no sign of malfunction, no perceivable gain in performance was measured as well. This was odd because a significant speedup was expected. The solution to this quandary was that the original driver very aggressively performed cache sync operations after modifying each page table entry. In an experiment—modifying the original driver—that postponed the cache sync operation until after the last modification of a bunch yielded roughly the same performance as the pass-through experiment, without, however, exhibiting the hypothetical race discussed earlier.

So inadvertently, the original driver was improved, increasing the gap between the performance of the GPURG and the native architecture. However, considering context building, being a rather infrequent operation, the results were still very satisfactory; spending one or the other millisecond on context creation occasionally is more than bearable. To find the by far larger portion of the context building cost, one must take a close look at the shadow paging scheme of L^{4}Linux on Fiasco.OC, which is done in the following section.

CPU interposition

The operations map (b), touch (c), and unmap (e) measured by the `map_tool` benchmark are indifferent with respect to the GPU interposition. They shed some light, however, on the shadow paging of L^{4}Linux.

The first thing that strikes one as odd, looking at Figure 6.4.7, is that L^{4}Linux apparently outperforms native Linux in the map operation. In both cases, the page tables are manipulated immediately. However, in the native case, those page tables are walked by the CPU's MMU, which, to be able to observe the modification, requires the corresponding L1 cache line to be flushed after a page table entry (PTE) was modified. In the paravirtualized case, this is not necessary because the guest page tables are never walked by the MMU hardware, but rather by the CPU itself. Pinpointing the cost to the cache maintenance operation happened somewhat anecdotal at first. It was observed that the mapping cost in the native case seemed to oscillate, and sometimes it was almost as low as in the virtualized case. As it turned out, the cost was low if only one of the CPUs was on-line and high if more than one was on-line. Experiments showed that the cost for the cache maintenance operation in question increased tenfold (from ∼30 ns

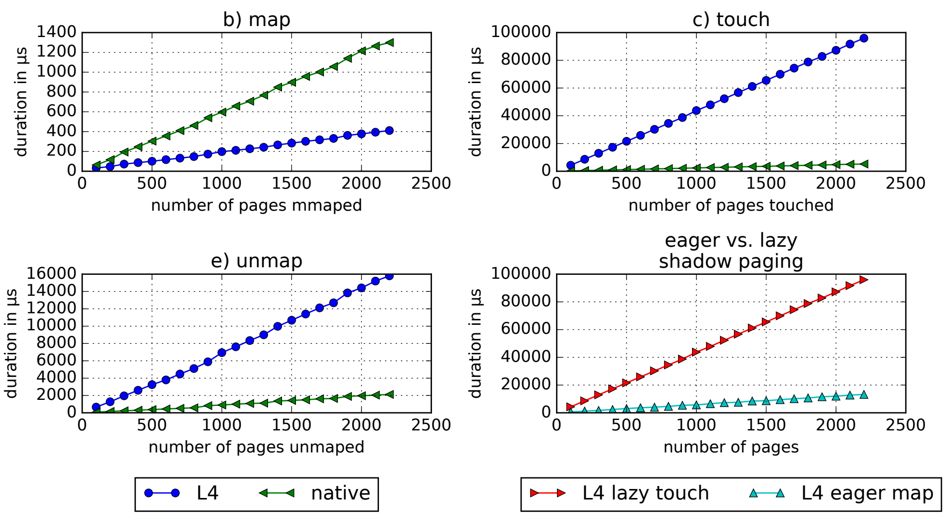


Figure 6.4.7: These graphs compare the experiments GPURG (L4) with native (b, c, and e), and two different shadow page table population strategies (bottom right). Plotted is the time it takes to map, touch, and unmap buffers of increasing sizes given in pages of 4 KiB. All values were determined as the median of 10,000 measurements.

to ∼300 ns) when more than one CPU was online. Another speedup stems from that Linux—on the ARM architecture—keeps duplicated page table records: One for the hardware, and one to accommodate additional status flags, which the architectural page tables cannot absorb. The former is not needed by L^{4}Linux.

Whatever head start L^{4}Linux gained in the mapping phase, it cannot make up for the terrible performance it shows when actually using these mappings by accessing the corresponding memory. Figure 6.4.7 c) shows that in L^{4}Linux, the cost of lazily populating the process address space through page faults is about 18 times as high as in the native case. Worse yet, in the native case, most of the cost can be avoided with the following little tweak. Indeed, in the native case, the page tables are modified during the map operation, and the page faults only occur because they are mapped read-only. This is done to keep track of whether or not the page is dirty and must be saved when swapped out to disk, either by the paging mechanism or due to the file cache mechanism. For mappings that are not backed by a file, however, Android does not perform swapping; rather, processes are being killed instead, whenever the system runs low on memory. Also, memory used by the GPU should be available as soon as a GPU job is started, to avoid GPU page-faults, let alone major[7] faults. Thus, if it is agreed upon that it is futile to flag a page of this particular mapping as dirty only on a write access, then it can be flagged dirty immediately and mapped writable, avoiding the subsequent write-page-fault. This reduces the cost of a write access by an order of magnitude, e.g., from $\sim$5 ms to less than $\sim$0.5 ms for 2,000 pages. This little tweak, however, does not work with L^{4}Linux because only the guest page tables are populated eagerly. And whether or not they are marked writable, a page fault occurs upon any kind of access, because the shadow page tables are not populated due to the map operation. While optimizing shadow paging mechanisms is out of the scope of this work, a naive attempt was made to reduce the cost of eager mappings, that is, the ones under discussion. During the mapping phase, the shadow pages were updated immediately, yielding two privilege level transitions per PTE rather than four privilege level transitions and two address space switches, which is the cost of a page-fault in L^{4}Linux. This pushed most of the cost into the mapping phase, yielding a reduction in cumulated mapping and accessing overhead of nearly an order of magnitude, as can be seen in the lower right graph of Figure 6.4.7, in which the cost of touching with lazy population is plotted together with the cost of mapping with eager population. This rather clumsy experiment indicates that there is room for optimization in the Fiasco.OC/L^{4}Linux shadow-paging scheme, in certain corner cases at least. Moreover, significant performance gains can be expected from hardware with support for nested paging, which is gaining traction in the market for ARM based embedded systems.

Framebuffer overhead

In this section, the results of the `vsyncer` benchmark are discussed. The measurement was performed by gathering ten thousand samples through periodic evaluation of the trace buffer, as described in Section 6.3.4, while operating the device. Figure 6.4.8 shows the latency of the VSYNC event as it builds up traveling through the system on native Linux as well as on Fiasco.OC. It shows that in the para-virtualized setup, there is a 16% overhead when compared to the native case. In absolute numbers, however,

[7]Faults that involve copying data in from slow mass storage

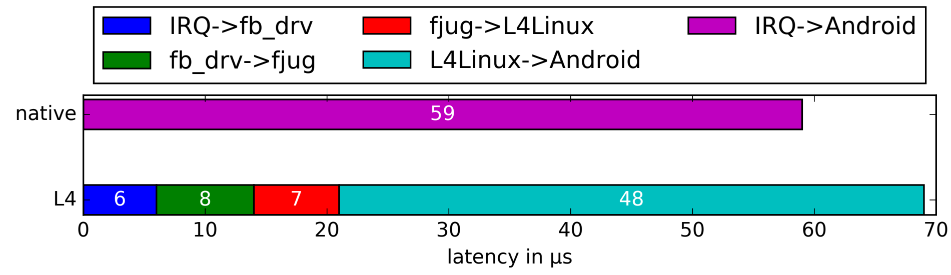


Figure 6.4.8: The graphs show the time it takes for a VSYNC event to travel from the initial interrupt handler to the Android user space system. The top graph shows this time span for the native experiment. The bottom graph shows this time span for the virtualized L4 experiment, split into the legs that the VSYNC event walks when traveling through the system.

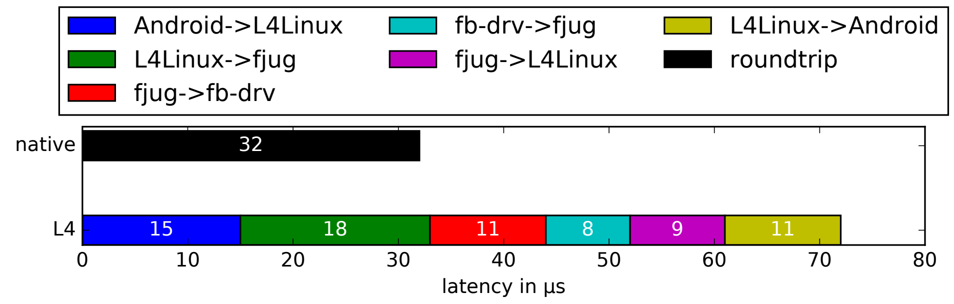


Figure 6.4.9: The graphs show the time it takes for a buffer swap request and its reply to travel through the system and back. The top graphs show this time for the native experiment. The bottom graph shows this time span for the virtualized L4 experiment, split into the legs that the request and the reply walk when traveling through the system.

this is mere 10 μs of extra CPU time spent, which, at a target frame rate of 60 Hz, amounts to less than a thousandth of the CPU cycles available—on one of the CPUs, that is.

The second value that the `vsyncer` application measured is the buffer swapping overhead. Figure 6.4.9 depicts the time spent passing the request and its reply from subsystem to subsystem. A 125% overhead or 40 μs in absolute numbers can be seen, which amounts to 0.24 % of extra CPU utilization given a target frame rate of 60 Hz.

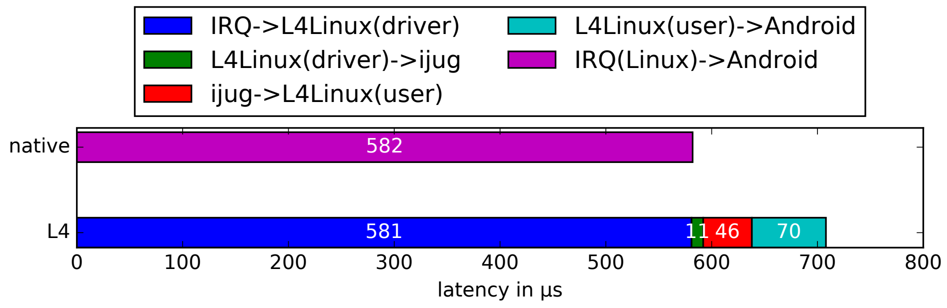


Figure 6.4.10: The graphs show the time it takes for and input event to travel through the system. The top graph depicts this time span for the native experiment, and the bottom graph depicts the L4 experiment split into legs representing the subsystems through which the event travels.

Input latency

Analogous to how the VSYNC latency was measured, the `inputer` benchmark evaluates the trace buffer periodically and thereby while the device is being operated, gathers samples. The touch screen, when operated, produces samples at a rate of 100 Hz, that is, every ten milliseconds. It is connected to the SoC via an I^2C bus and an interrupt line. The considerably large latency of ~500 μs, which can be seen in Figure 6.4.10 and is present in both experiments, is due to the transfer of the touch event data via I^2C and the protocol between the device and the SoC. What follows is the transfer of the event data through the input path, gathering more latency on the way. The input path's protocol is a little bit more elaborate than the simple notification passing, which happens in the VSYNC case. First, the event source notifies the sink of pending events. The sink subsequently fetches the events from the source. With an added 126 μs, or an overhead of 22%, this amounts to 1.25% of extra CPU utilization and, therefore, yields the largest optimization potential of the secure GUI infrastructure. During this work, however, input latency was not subject to performance optimization.

Power consumption

Figure 6.4.12 shows three sequences of readings, "referencing", "copying", and "native". The top graph shows the raw data, whereas the bottom graph

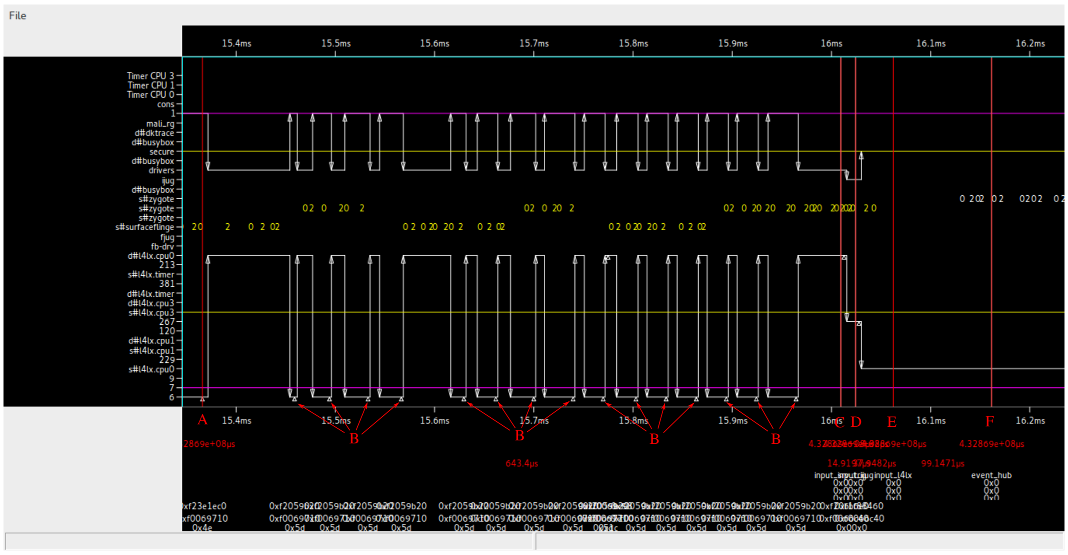


Figure 6.4.11: Visualized trace of an input event traveling through the system. The interrupt of the touch screen registers in the kernel and kicks off the driver (A). The driver queries the device, issuing bus transaction on an I^{2}C bus (Bs mark the I^{2}C bus controller interrupts). Eventually, the event is complete (C) and is delivered to `ijug` (D), L^{4}Linux (E), and Android's event hub (F).

shows the raw data smoothed with a five-second (50-samples) wide sliding window. They show the power consumption of the device during three subsequent runs of the high-level 3D benchmark Cube. The referencing experiment represents the final stage of the prototypical implementation. The copying experiment simulates the compositing approach of `mag`. The native experiment is the non-virtualized Cyanogenmod installation that was used for comparison throughout this chapter. One can see clearly that with the copying experiment, the benchmark takes almost three times as long to complete as with the other two, which is consistent with the lower frame rate measured earlier. And due to the higher CPU utilization, the instantaneous power consumption is higher by roughly 300 milliwatts. This is a great improvement over the initial implementation of the secure virtualized smartphone. Comparing the power consumption of the final stage with the native experiment indicates that the former is almost on par with the latter. The fact that the ∼200 milliwatts gap between the plots also exists in the idle phase (starting around second 60), leads to the hypothesis that there exists a consumer that is not handled by the virtualized smartphone's power management, a consumer unrelated to the secure GUI and GPU and, therefore, out of the scope of this work. The only power management that is, in

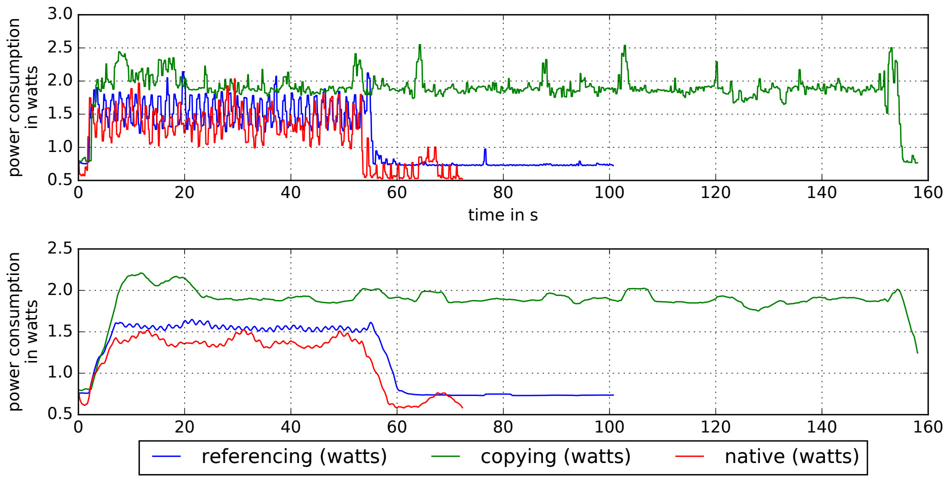


Figure 6.4.12: These graphs show the instantaneous power consumption of the device while running the high-level 3D benchmark Cube. Three experiments are compared: The "referencing" experiment denotes the final stage of the prototype. The "copying" experiment simulates the compositing approach. The "native" experiment is the non-virtualized experiment. The top graph shows the raw data, whereas the bottom graph shows the raw data filtered by a five-second wide sliding window.

fact, done by the GPU server is the power control of the GPU, which saves roughly 5 milliwatts when idle. This can be seen in Figure 6.4.13, which depicts the power consumption of the device while the clock and power supply of the idle GPU are switched on an off in 5 second intervals.

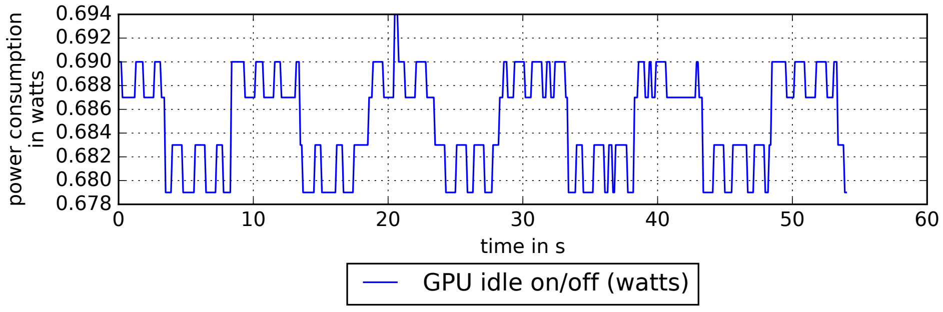


Figure 6.4.13: This graph shows the power consumption of the device while the clock and power supply of the idle GPU is switched on and of in five-second intervals.

TCB complexity

Module	SLOC
Fiasco.OC	28,943
moe	4,044
ned[8]	16,078
sigma0	1,023
io	12,864
total	62,952

(a) Runtime environment

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

(b) Secure GUI modules

Table 6.5.1: The left table shows the sizes of the modules of the underlying runtime environment, including the kernel. They form the common TCB. The right table shows the sizes of the modules that constitute the prototypical implementation. The sizes are given in *source lines of code* (SLOC), and were measured using David A. Wheeler's "SLOCCount". Both tables were previously published at MOST2015 [24].

One of the key design principles of the secure smartphone was to keep the trusted computing base (TCB) small. This principle was driving the decision for choosing the μ-kernel Fiasco.OC and it was driving the design of the compartmentalized architecture described here. Table 6.5.1b shows the size of the modules making up the prototypical implementation. With less than 7,000 lines of code, this prototype of a secure GUI architecture, including graphics acceleration, is outright minuscule, especially when compared with front-end GPU virtualization schemes, such as Xen3D[9] [60], which adds around 80,000 lines of code to the TCB, and is larger than the combined complexities of the runtime environment (see Table 6.5.1a) and secure GUI of the secure smartphone.

While code lines added to, e.g., a monolithic kernel all add to the "same" TCB, in a decomposed system, such as the prototype under discussion, each subsystem, when compromised, has its individual set of implications on the security attributes of the system. These implications naturally depend on the privileges that a subsystem is pooling, that is, the memory that it can access, the services of other subsystems that it is allowed to use, the communication channels that it maintains with other, possibly dependent, parties,

[8]With 14,124 SLOC, the bulk of the code that makes up the bootstrapper `ned` is contributed by a lua interpreter.

[9]Xen3D is chosen here for comparison, because it is one GPU virtualization scheme that aims at TCB reduction.

and not to forget, the IO resources that it can access. The latter, as can be seen in Chapter 3, is particularly interesting. Access to IO resources means control over one or more of the systems peripheral devices, which have their own implications on the security properties of the system. In a way, these devices can be modeled as subsystems offering services to the subsystem under consideration; the distinction between hardware and software subsystems diminishes. The individuality of the security implications of the subsystems have the effect that higher-level subsystems can choose whether a subsystem belongs to their TCB or not, depending on the security requirements that they have. Therefore, a discussion of these implications for the secure GUI modules is warranted.

GPU Server and GPU

The privileges of the GPU Server are few. It maintains communication channels with its clients, with the IO server, with dataspace providers, with a service allowing controlling the clock and power of the GPU, and through MMIO, it controls the GPU and its MMU. The latter ones, that is, all but the clients, can be considered trusted and belong to the GPU server's TCB; the clients do not, and they must be considered untrusted. In Chapter 3, it was established that with the assumed hardware model, control over the GPU and its MMU is equivalent to full physical memory access. This means that a compromised GPU server voids all isolation guarantees of the system; therefore, it must unconditionally be considered part of the TCB of all subsystems.

Display driver, display controller, and **fjug**

The privileges of the display driver are very similar to those of the GPU server. It maintains communication channels to its client, to the IO server, to a dataspace provider, to power and clock service, and it controls the display controller through MMIO. Here the client is the output switcher, which need not be considered inherently hostile for now. For the sake of discussion, it can be assumed that the display controller, being a DMA capable device, is only able to read from a configurable memory location and send it to a screen. This means, the corresponding driver cannot "learn" what the device read and therefore cannot exfiltrate it. It is conceivable that exfiltration can be done by an attacker from observing the screen, which implies physical access to the device, or at least proximity. The implications of a compromised display controller driver are therefore:

- **Availability of graphical output**
  The display driver can inhibit graphical output.

- **Integrity of graphical output**
  The display driver can replace or modify the graphical output of its clients.

- **Confidentiality**
  The display driver can, in a limited way, exfiltrate arbitrary data from physical memory through the screen.

It can be argued that the display driver belongs to the TCB only conditionally because the security implications, if it is subverted, are constrained. In other words, the display driver is only part of the TCB of those subsystems that rely on the attributes enumerated above. For example, data integrity can be upheld even if the display driver is compromised.

The switcher module `fjug`, as it is currently implemented, has similar capabilities as the driver. It can influence the **availability and integrity of graphical output**; but because it cannot arbitrarily program the display controller, **impairment of confidentiality is limited to the graphics output** of its clients. A tweak, which was briefly hinted at in Section 4.6, can remove even this security implication. If the framebuffers were allocated by a trusted third and `fjug` was only allowed to refer to the buffers using symbolic names, there would be no way by which `fjug`, if compromised, could learn about the content of its client's framebuffers.

Input driver and `ijug`

As are the drivers discussed before, the input driver is embedded into the system by a web of communication channels. It provides a client with input events and receives IO resources from the IO server. The input devices may be accessed in various ways as discussed in Section 4.5. But regardless of the method of device access, it may be assumed, for the sake of argument,[10] that unlike with the GPU and display controller, the input devices may not be used as agents for accessing out-of-bounds memory resources. Therefore, subverting the input driver has implications on the system, analogous to the driver's facilities. It can block input events, thus influencing the availability of the service; it can modify or inject events, thus challenging the integrity of the service. The driver could record input events, thus threatening their confidentiality. Exfiltration of these records, however, is all but straightforward. It cannot access mass storage or open a network connection, and therefore, it would need the cooperation of a subsystem with which it maintains a communication channel, in order to make for a viable key logger. Moreover, the driver is ignorant of the client to which it currently sends events, which is problematic for an attacker in two ways. First, it makes this channel unreliable for exfiltration, e.g., if one of the clients would be cooperative. Second, it increases the state space that the the attacker needs to track, for example, if the attacker whants to make inferences on the state of the screen, which is needed for interpreting touch events. The switcher `ijug` does not have this last limitation. Anyway, the components of the input path can impair the **availability and integrity of input events**. In addition, the **confidentiality of input events** could be compromised if a reliable exfiltration method is found. All other guarantees that the system provides remain intact.

[10]For clarification: As discussed in Section 4.5 there were two implementations of the input path. One was decomposed and had an L^{4}Linux instance driving the input devices. This one was used in the performance evaluation. But the assumptions made here do not hold for this implementation. The other implementation had a small input driver that was combined with the switcher, and it thus lacked decomposition. The argumentation in this section is conducted as though the input path comprised the switcher and the driver as small, decomposed components, as it was intended by the author, although it was never implemented in this way.

7 Conclusion

This work was performed in context of the SiMKo3 project, which had the goal of designing and implementing a secure smartphone catering to the security interests of multiple stakeholders; this goal was to be achieved using virtualization. It addresses the intricacies that arise when providing multiple strongly isolated VMs on a mobile handset with a means to present a secure and snappy graphical user interface to the user. The architecture presented here provides a trusted and identifiable path between the VMs and the user and allows for the use of graphics acceleration while preserving a high standard of isolation.

For the sake of reduced complexity in both runtime and implementation, the hardware abstraction, often expected from device virtualization, was dropped. As a result, the hardware specifics reached high up into the VMs and even into their application layers. But the price is small, considering that this is typical for a smartphone firmware, which is tightly coupled to each device. The resulting small and customizable trusted computing base in conjunction with the low runtime overhead of the architecture presented here, is predestined for the application in a wide variety of fields where embedded computing, high assurance, and the need for visualization meet.

Besides the field of mobile hand-held computing, automotive infotainment and visual assistance systems come into mind. The prototype presented here used rehosting, a very intrusive form of paravirtualization, for reusing legacy operating system components. Moreover, the secure GUI and GPU virtualization architecture presented here can be combined with a whole range of system types, from specially tailored, compartmentalized multi-server systems to virtualizing separation kernel-based systems.

Outlook

A controversial subject to which this work might give a new twist, is enabling graphics acceleration in ARM TrustZone. Secure applications deployed in such a secure runtime environment are used to store and apply—without leakage—key material for certification. But without a reliable means to track the user's intent as to when and how this material shall be used, these applications are useless in many fields, such as payment and cryptographic signatures. A user interface is required, and the industry demands it be snappy. But graphics acceleration is generally put on a level with large API frameworks such as OpenGL and is dismissed for bloating the TCB. By acknowledging the sensibleness of dropping the hardware abstraction in a tightly integrated firmware, it is conceivable that a secure runtime environment's GUI could be enhanced by using a small GPU server, such as the one presented here, in conjunction with a small set of precompiled rendering programs, thereby cutting down on the TCB bloat of a full OpenGL API stack.

Closing Words

In his inspiring Ph.D. thesis on securing graphical user interfaces, Norman Feske postulated, "that an achievable low complexity of the trusted [GPU] driver portion lies in the order of 5,000 to 10,000 SLOC"—Feske [32]. As it appears, he was close.

Acknowledgements

I would like to thank my Professor, Jean-Pierre Seifert, for his guidance. I would like to thank my committee members Felix Freiling, Konrad Rieck, and Sebastian Möller. I would like to thank the people who contributed to the SiMKo3 project, without whom this work would have been hardly possible. Among them were: Dirk Jansen, Adam Lackorzynski, Matthias Lange, Steffen Liebergeld, Christian Ludwig, Jan Nordholz, Matthias Petschick, John Tapsell, Julian Vetter, Michael Voigt, Alexander Warg, Jean Wolter, and many more. I thank Felicitas Hetzelt and Marta Piekarska, who worked with me on exploring GPU vulnerabilities. I thank Tobias Fiebig and Bhargava Shastry, who worked with me on the Nemesys project, for many inspiring discussions. A special thanks goes to Michael Peter who was also a member of the SiMKo3 team, but more than that, I call him my mentor and a friend. Thank you, Michael, for your constant motivation, support, criticism, and inspiration. I would like to thank my mother, Marianne Danisevskis, who taught me the satisfaction that comes with accomplishing something through hard work; the thought of her instilled strenght in me when I felt exhausted. And I would like to thank my father, Peteris Danisevskis, who taught me that all the hard work is meaningless, if one does not live once in a while. You are the best parents one could wish for. I would like to thank my sister, Friederike Danisevskis, who, to me, is a paragon of confidence and independence. Finally, I thank my wife, Sandra Vastag, and my son for their love, support, and most importantly, patience. Time is lost forever, so I will never be able to repay you for all the weekends I spent working – nevertheless, I will try.

References

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

[2] Arm architecture reference manual (armv7-a and armv7-r edition). Available at http://infocenter.arm.com/help/index.jsp as of November, 29$^{\text{th}}$, 2015.

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

[4] Gallium3d technical overview. Available at https://www.freedesktop.org/wiki/Software/gallium as of February, 25$^{\text{th}}$, 2016.

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

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

[7] Opengl es the standard for embedded accelerated 3d graphics. Available at https://www.khronos.org/opengles/ as of November, 30$^{\text{th}}$, 2015.

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

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

[10] Arm mali gpu opengl es application optimization guide, 2011. Available at http://malideveloper.arm.com/downloads/Mali_Optimization_Guide_3.0.pdf as of March, 2$^{\text{nd}}$, 2016.

[11] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In John Paul Shen and Margaret 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] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *Operating Systems Review*, 44(4):3–18, 2010.

[13] Chaitrali Amrutkar, Kapil Singh, Arunabh Verma, and Patrick Traynor. On the disparity of display security in mobile and traditional web browsers. 2011.

[14] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: a virtual mobile smartphone architecture. In Wobber and Druschel [64], pages 173–187.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[16] Kenneth C. Barr, Prashanth P. Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *Operating Systems Review*, 44(4):124–135, 2010.

[17] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 931–948. IEEE Computer Society, 2015.

[18] Prashanth Bungale. Arm virtualization: Cpu & mmu issues, 2010. Available at `https://labs.vmware.com/download/68/` as of August 10th, 2015.

[19] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating*

Systems, May 20-23, 2001, Elmau/Oberbayern, Germany, pages 133–138. IEEE Computer Society, 2001.

[20] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 1037–1052. USENIX Association, 2014.

[21] Rob Clark. Kilroy. `https://github.com/robclark/kilroy`.

[22] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In Wobber and Druschel [64], pages 189–202.

[23] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. The design, implementation, and evaluation of cells: A virtual smartphone architecture. *ACM Trans. Comput. Syst.*, 30(3):9, 2012.

[24] Janis Danisevskis, Michael Peter, Jan Nordholz, Matthias Petschick, and Julian Vetter. Graphical user interface for virtualized mobile handsets. 2015. Paper available at `http://ieee-security.org/TC/SPW2015/MoST/papers/s1p1.pdf` and slides available at `http://ieee-security.org/TC/SPW2015/MoST/slides/s1p1.pdf` as of Sebtember, 24th 2015.

[25] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. Dark side of the shader: Mobile gpu-aided malware delivery. In Hyang-Sook Lee and Dong-Guk 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.

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

[27] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[28] Wim Van Eck and Neher Laborato. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4:269–286, 1985.

[29] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[30] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. Tivos: Trusted visual i/o paths for android. *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[31] 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.

[32] Norman Feske. *Securing graphical user interfaces.* PhD thesis, Technische Universität Dresden, 2009.

[33] Tobias Fiebig, Janis Danisevskis, and Marta Piekarska. A metric for the evaluation and comparison of keylogger performance. In Chris Kanich and Patrick Lardieri, editors, *7th Workshop on Cyber Security Experimentation and Test, CSET '14, San Diego, CA, USA, August 18, 2014.* USENIX Association, 2014.

[34] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[35] Jacob Gorm 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.

[36] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 66–77. ACM, 1997.

[37] Yuichi Hayashi, Naofumi Homma, Mamoru Miura, Takafumi Aoki, and Hideaki Sone. A threat for tablet pcs in public space: Remote visualization of screen images using em emanation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 954–965, New York, NY, USA, 2014. ACM.

[38] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: A scalable graphics system for clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 129–140, New York, NY, USA, 2001. ACM.

[39] ARM Inc. Corelink level 2 cache controller l2c-310. Website. Available at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246h/DDI0246H_l2c310_r3p3_trm.pdf` as of July 16th, 2015.

[40] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1055–1062. ACM, 2013.

[41] Adam Lackorzynski, Alexander Warg, and Michael Peter. Virtual processors as kernel interface. In *Twelfth Real-Time Linux Workshop*, 2010.

[42] H. Andrés Lagar-cavilla, M. Satyanarayanan, Niraj Tolia, and Eyal de Lara. Vmm-independent graphics acceleration. In *Proceedings of VEE 2007*. ACM Press, 2007.

[43] Matthias Lange and Steffen Liebergeld. Crossover: secure and usable user interface for mobile devices with multiple isolated OS personalities.

In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 249–257. ACM, 2013.

[44] Donald C. Latham. *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, Dec 1985.

[45] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 17–30. USENIX Association, 2004.

[46] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Soft layering for virtual machines. In *13th Asia-Pacific Computer Systems Architecture Conference, ACSAC 2008, Hsinchu, China, August 4-6, 2008*, pages 1–9. IEEE, 2008.

[47] Steffen Liebergeld, Michael Peter, and Adam Lackorzynski. Towards modular security-conscious virtual machines. In *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[48] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[49] Tongbo Lua, Xing Jin, Ajai Anathanarayanan, and Wenliang Du. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security*, pages 227–243. Springer Berlin Heidelberg, 1012.

[50] Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[51] M. Peter, M. Petschick, J. Vetter, J. Nordholz, J. Danisevskis, and J.-P. Seifert. Undermining isolation through covert channels in the fiasco.oc microkernel. In Omer H. Abdelrahman, Erol Gelenbe, Gokce Gorbil, and Ricardo Lent, editors, *Information Sciences and Systems 2015*,

volume 363 of *Lecture Notes in Electrical Engineering*, pages 147–156. Springer International Publishing, 2016.

[52] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. Virtual machines jailed. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, 2009.

[53] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[54] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[55] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In Steven M. Bellovin and Greg Rose, editors, *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000.* USENIX Association, 2000.

[56] John M. Rushby. Design and verification of secure systems. In *SOSP*, pages 12–21, 1981.

[57] Sumit Semwal. Dma buffer sharing api guide. Available at `https://www.kernel.org/doc/Documentation/dma-buf-sharing.txt` as of November, 9th, 2015.

[58] Jonathan Shapiro. Debunking linus's latest, 2006. Available at `http://www.coyotos.org/docs/misc/linus-rebuttal.html` as of August 11th, 2015.

[59] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the eros trusted window system. In *USENIX Security Symposium*, pages 165–178, 2004.

[60] Christopher 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.

[61] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[62] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 121–132. USENIX Association, 2014.

[63] Alexander Warg and Adam Lackorzynski. Rounding pointers: Type safe capabilities with c++ meta programming. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 3:1–3:5, New York, NY, USA, 2011. ACM.

[64] Ted Wobber and Peter Druschel, editors. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. ACM, 2011.

[65] Ka-Ping Yee. User interaction design for secure systems. In Robert H. Deng, Sihan Qing, Feng Bao, and Jianying 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.

[66] Thomas M. Zeng. The android ion memory allocator, February 2012. Available at `https://lwn.net/Articles/480055/` as of November, 9th 2015.

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