Trusted Display on Untrusted Commodity Platforms

Miao Yu
ECE Department and CyLab
Carnegie Mellon University
miaoy1@andrew.cmu.edu

Virgil D. Gligor
ECE Department and CyLab
Carnegie Mellon University
virgil@andrew.cmu.edu

Zongwei Zhou*
ECE Department and CyLab
Carnegie Mellon University
zongwei@alumni.cmu.edu

ABSTRACT

A trusted display service assures the confidentiality and authenticity of content output by a security-sensitive application and thus prevents a compromised commodity operating system or application from surreptitiously reading or modifying the displayed output. Past approaches have failed to provide trusted display on commodity platforms that use modern graphics processing units (GPUs). For example, full GPU virtualization encourages the sharing of GPU address space with multiple virtual machines *without* providing adequate hardware protection mechanisms; e.g., address-space separation and instruction execution control. This paper proposes a new trusted display service that has a minimal trusted code base and maintains full compatibility with commodity computing platforms. The service relies on a GPU separation kernel that (1) defines different types of GPU objects, (2) mediates access to security-sensitive objects, and (3) emulates object whenever required by commodity-platform compatibility. The separation kernel employs a new address-space separation mechanism that avoids the challenging problem of GPU instruction verification without adequate hardware support. The implementation of the trusted-display service has a code base that is two orders of magnitude smaller than other similar services, such as those based on full GPU virtualization. Performance measurements show that the trusted-display overhead added over and above that of the underlying trusted system is fairly modest.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Security kernels*

[*]Current affiliation: VMware Inc, 3401 Hillview Ave, Palo Alto, CA 94304.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813719.

Keywords

GPU separation kernel; Trusted display

1. INTRODUCTION

A trusted display service provides a protected channel that assures the confidentiality and authenticity of content output on selected screen areas. With it users can rely on the information output by a security-sensitive application (SecApp) without worrying about undetectable screen "scrapping" or "painting" by malicious software on commodity systems; i.e., the display output is surreptitiously read or modified by a compromised commodity operating systems (OSes) or applications (Apps).

Security architectures that isolate entire SecApps from untrusted commodity OSes and Applications (Apps) [35] implement trusted display functions via trusted path [55, 56]. That is, a user's explicit activation of the trusted-path effectively removes all untrusted OS and Apps access to the display device and assigns the device to a SecApp for the entire duration of a session. Unfortunately, the exclusive use of display devices via trusted path does not allow *both* untrusted OS/Apps and SecApps to output content concurrently on a user's screen; i.e., untrusted output cannot be displayed until after the trusted path releases the screen at the end of the SecApp session. As a consequence, it would not be possible to maintain the typical multi-window user experience for applications that comprise both trusted and untrusted components and use the same display screen.

Problem. Some past approaches that allow trusted display of output with different sensitivity on the same screen concurrently have been based on encapsulating and protecting graphics cards within high-assurance security kernels [14, 42, 17]. In addition to requiring changes of commodity OSes, adopting such an approach for the entire graphics processing unit (GPU) would not work since the complexity of modern GPU functionality[1] would rule out maintaining a small and simple code base for the security kernel, which is a prerequisite for high assurance. For example, the size of Intel's GPU driver for Linux 3.2.0 - 36.57 has over 57K SLoC, which is more than twice the size of a typical security kernel [54]. Furthermore, GPU functions operate asynchronously from the CPUs [46, 53] to improve graphics performance and introduce concurrency control for multi-threading in the trusted

[1]Modern GPUs include graphics/computation accelerators [39, 16]. They are equipped with hundreds of processors [32] to provide complex functions of 2D/3D hardware rendering, general-purpose computing on GPU (GPGPU), and hardware video encoding/decoding.

code base. This would invalidate all correctness proofs that assume single-thread operation [27, 47].

Full GPU virtualization [46, 45] can be used to enable concurrent display of both trusted and untrusted output on a user's screen without requiring commodity OSes/Apps modification. However, full GPU virtualization, which is largely motivated by improved performance, relies on address-space sharing between different virtual machines (VMs) and the GPU *without* providing adequate hardware mechanisms for protecting different VMs' code and data within the GPU; e.g., address-space separation and instruction execution control. As a concrete example, we illustrate a class of new attacks that exploit the inadequacy of address-space separation on fully virtualized GPUs; viz., Section 2.2. Moreover, full GPU virtualization intrinsically requires a large trusted code base; e.g. supporting native GPU drivers/Apps requires emulating *all* accesses to *all* GPU configuration registers for the VMs scheduled to access the GPU. Thus, adopting full GPU virtualization for high-assurance trusted display would be impractical.

Solution. The trusted display design presented in this paper satisfies the following four requirements.

- it allows the confidentiality and authenticity of display contents to be assured to whatever degree of rigor deemed necessary by minimizing and simplifying the trusted-display code base.
- it avoids redesign and modification of underlying trusted-system components, and preserves their correctness properties; e.g., proofs of high-assurance micro-kernels and micro-hypervisors [27, 47].
- it preserves full compatibility with commodity platforms; i.e., it does not require any modification of commodity OS/Apps code and GPU hardware or reduce their functionality.
- it maintains a typical user's perception and use of application output and relies on easily identifiable screen geometry; e.g., it uses different windows for trusted and untrusted screen areas.

The central component of our trusted display design is a GPU Separation Kernel (GSK) that (1) distinguishes different types of GPU objects, (2) mediates access to security-sensitive objects, and (3) emulates object access whenever required by commodity-platform compatibility. The GSK employs a new address-space separation mechanism that avoids the challenging problem of GPU instructions verification without adequate hardware support. The implementation of the trusted display service has a code base that is two orders of magnitude smaller than other similar services, such as those based on full GPU virtualization.

Outline. In Section 2, we provide a brief overview of GPU functions to enable the reader understand the vulnerabilities of GPU virtualization to adversary attacks, and challenges of trusted display on commodity platforms. In Section 3, we define the adversary threats, security security properties that counter them, and an informal security model that satisfies these properties. In Section 4, we describe the detailed design and implementation of the trusted display system, and in Section 5, we evaluate our implementation. The related work in this area is presented in Section 6, common use of trusted display is briefly discussed in Section 7, and conclusions are provided in Section 8.

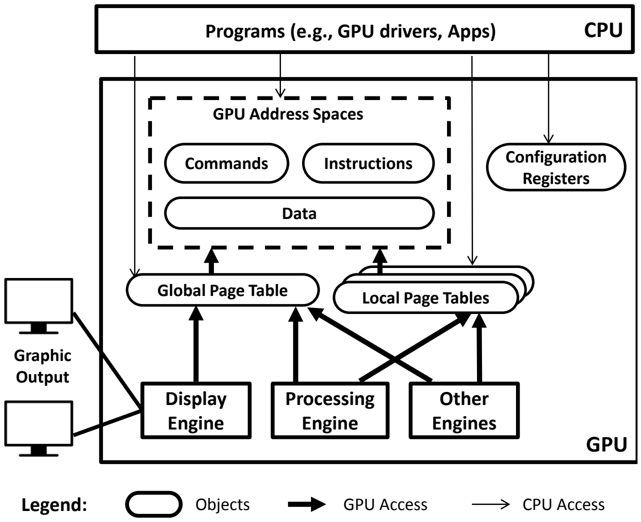


Figure 1: Overview of a modern GPU architecture.

2. COMMODITY GPU ARCHITECTURE AND SECURITY VULNERABILITIES

In this section, we present an overview of common architecture features of modern GPUs to enable an unfamiliar reader understand their vulnerability to attacks. The GPU architecture described herein is common to widely available commodity devices from vendors such as Intel [23, 2], AMD [6], Nvidia [4], and ARM [3, 10].

2.1 GPU Architecture Overview

CPU programs (e.g. GPU drivers and Apps) control GPU execution via four types of *objects*, namely data, page tables, commands, and instructions that are stored in GPU memory, and GPU configuration registers; viz., Figure 1.

CPU programs produce the instructions and commands that are executed by GPU hardware. For example, instructions are executed on GPU processor cores, process input data, and produce results that are used by display engines. In contrast, commands are executed by dedicated command processors and are used to configure the GPU with correct parameters; e.g., specify stack base address used by instructions. Groups of commands are submitted for processing in dedicated command buffers; e.g., they are received in input (*ring*) buffers from drivers and (*batch*) buffers from both applications and drivers.

As shown in Figure 1, a GPU also contains several *engines* [46, 23, 7], such as the *processing engine* and *display engine*. The processing engine executes instructions on multiple GPU cores for computation acceleration. It references memory regions known as the *GPU local address space* via the *GPU local page tables*. The display engine parses screen pixel data stored in *frame buffers* according to the engine's configurations, and outputs images for display. Other engines perform a variety of functions such as device-wide performance monitoring and power management.

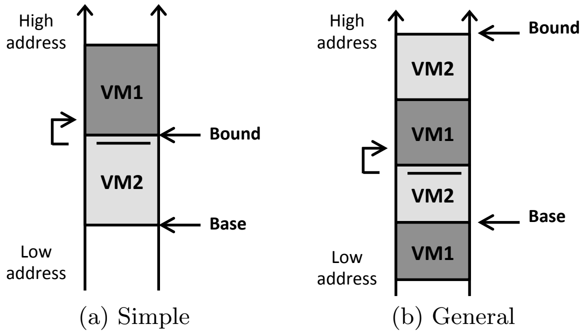
The display engine defines several basic configurations for *frame buffer* presentation; e.g. geometry and pixel formats. Furthermore, it provides the data paths from *frame buffers* to external monitors. For example, the screen output may comprise a combination of multiple screen layers, each of which contains a separate *frame buffer*. In this case, GPUs support a *hardware cursor* as the front layer of the screen and display it over the primary image. Since a single GPU may be connected to multiple screen monitors, a monitor may consume the *same* frame buffers as another monitor, which implies that GPU memory protection requires a controlled-sharing mechanism. Furthermore, an image presented on a screen may be torn as the result of frame-buffer updates by CPU programs during screen refreshing. To address this synchronization problem, display engines of modern GPUs also provides a *V-Sync interrupt* to notify CPU programs of the time when it is safe to update a frame buffer [49].

Although the GPU architecture illustrated in Figure 1 is common to many commodity GPUs, some of these GPUs differ in how memory is accessed and managed. For example, Intel's GPUs use a *global page table* (GGTT) for memory access in addition to local page tables. The GGTT maps the memory region referred as the *GPU global address space*, which includes *frame buffers*, command buffers, and GPU *memory aperture*, which is shared between CPU and GPU. In contrast, AMD and Nvidia GPUs do not have a GGTT and allow direct access to GPU physical memory address space[2]. This implies that GPU memory access may also differ in different GPUs; e.g., the processing engine of Nvidia's GPU can access only the local address space [45, 26], whereas the Intel and AMD's[3] can also access the global address space [23, 6, 2].

2.2 Address Space Separation Attacks

A fully virtualized GPU shares its global address space with multiple virtual machines (VMs) to support *concurrent* accesses to its memory [46]. For example, while the GPU's display engine fetches a VM's *frame buffer* to display its content, the GPU's processing engine generates content for other VMs' *frame buffers*. Furthermore, the hardware design of the GPU's processing engines (e.g. Intel, AMD) allows instructions to access the global address space. Because full GPU virtualization supports native drivers, any malicious VMs can submit GPU instructions that access another VM's GPU data for screen output.

Figure 2(a) illustrates this simple attack. Here, a malicious VM2 submits valid GPU instructions that ostensibly address GPU memory inside VM2's address space but in fact access victim VM1's GPU memory. For example, VM2 can submit malicious instructions that contain large address offsets which fall into VM1's GPU address space[4]. Unless an additional "base-and-bound" mechanism for address space protection is supported by GPU address translation, the GPU's processing engine would allow the malicious VM2 to access victim VM1's GPU output data thereby violating confidentiality and authenticity.



(a) Simple (b) General

Figure 2: GPU address-space separation attacks.

We note that some fully virtualized GPUs support a single "base-and-bound" pair of registers for address space protection; e.g., Intel GPUs limit memory access range of GPU instructions by correct setting of the "base-and-bound" register pair for GPU command execution [23]. These GPUs can mediate memory accesses and deny address-space violations by GPU instructions and commands issued by malicious VMs [46].

Unfortunately, a *single pair* of base and bound registers is insufficient to counter all address-space separation attacks mounted by malicious VMs. These attacks are enabled by another important performance optimization of full GPU virtualization. That is, address space "ballooning" [46] allows the GPU to directly access virtual memory at addresses provided by guest VMs. This optimization improves GPU memory-access performance and greatly reduces complexity of GPU programming. Without it, trusted code would have to translate the referenced GPU virtual addresses for every object, and even recompile GPU instructions on the fly. For example, AMD's GPU instructions perform register-indirect memory accesses, and hence would require such recompilation for address translation.

However, address space ballooning allows the GPU memory of a guest VM to be mapped into two or more non-contiguous blocks in GPU global address space; e.g., one in GPU memory aperture for exchanging data between CPU and GPU, and the other in non-aperture space for holding GPU data. As a consequence, the separated memory blocks cannot be protected by the setting of the single pair of "base-and-bound" registers in the GPU commands; e.g., viz., Intel GPU. As illustrated in Figure 2(b), malicious VM2 uses the simple attack of Figure 2(a) but this time it can access victim VM1's GPU memory despite base-and-bound protection, because one of VM1's GPU memory blocks falls between two of VM2's non-contiguous memory blocks. It should be noted that the simple attack succeeds for other GPUs, not just Intel's; e.g. some instructions in AMD GPUs can perform register-indirect memory accesses, without specifying added address-space protection [7].

2.3 Challenges of Commodity Platforms

Implementing a trusted display service on untrusted commodity OS and hardware platforms that support SecApp isolation faces three basic challenges.

Incompatibility with commodity platforms. The goal of maintaining object-code compatibility with untrusted OSes that directly access GPU objects in an unrestricted manner poses a dilemma. If one re-designs and re-implements GPU functions on commodity OSes to block memory accesses that breach address space separation, one introduces object-code

[2] To simplify presentation, we consider that these GPUs use a GGTTs with flat mappings (e.g. virtual addresses are identical with physical addresses) even though the GGTT does not exist in these GPUs.

[3] Although this feature is undocumented by AMD's GPUs, it is supported in the open source GPU driver provided by AMD [6].

[4] Other full GPU virtualization approaches [45] are also subject to such attacks.

incompatibility. If one does not, one forgoes trusted display. To retain compatibility, access to GPU objects by untrusted commodity OS/Apps code must be emulated by the trusted-system, which increases the trusted code base and makes high-assurance design impractical.

Inadequate GPU hardware protection. The inadequacy of the hardware for memory protection has already been noted in the literature for Intel GPUs [46]. The address-space separation attack by malicious GPU instructions of Section 2.2 illustrates another instance of this problem and suggests that simplistic software solutions will not work. For example, verifying address offsets of GPU instructions before execution does not work because operand addressing cannot always be unambiguously determined due to indirect branches [23] and register-indirect memory accesses [23, 7].

Unverifiable code base. Even if, hypothetically, all the OS/Apps functions that access GPU objects could be isolated and made tamper-proof, their code base would be neither small (i.e., tens of thousands of SLoC) nor simple, and hence the formal verification of their security properties would be impractical. A large number of diverse GPU instructions and commands spread throughout different drivers and application code provide access to a large number of GPU objects; e.g., a GPU can have 625 configuration registers and 335 GPU commands, as shown in Section 5. Furthermore, since the underlying trusted base (e.g., micro-kernel or micro-hypervisor) must protect different SecApps on a commodity platform, the functions that access GPU objects directly must be implemented within the trusted base. Hence, these functions' code would have to preserve all existing assurance of the underlying trusted base; i.e., their security properties and proofs must compose with those of the trusted base. These challenges have not been met to date.

3. SECURITY MODEL

In this section, we define the threats posed by an adversary to trusted display and present security properties that counter these threats. Furthermore, we present an *informal GPU security model* that satisfies those properties in commodity systems.

3.1 Threats

An adversary can leak a SecApp's security-sensitive output via *screen scraping* attacks whereby the content of display output in a GPU's memory is read by a malicious program of a compromised commodity OS/App or SecApp. The adversary can also modify the SecApp's output content, configuration (e.g., geometry, pixel format, *frame buffer*'s base address) via *screen painting* attacks whereby a malicious program modifies GPU memory and configuration registers. For example, to launch both attacks the adversary can breach the separation of GPU's address spaces. These breaches can be implemented by unauthorized access to GPU objects, either directly by CPU programs (e.g., drivers, applications), or indirectly by GPU commands and instructions that cause the GPU to access other GPU objects in an unauthorized manner. Furthermore, the adversary can manipulate the display engine's data paths and overlay a new *frame buffer* over a SecApp's display thereby breaking the integrity of SecApps' display output without touching its contents.

In this paper, we do not consider hardware, firmware, side-channels, device peer-to-peer communication and shoulder-surfing attacks [20]. We ignore I/O channel isolation attacks, which have already been addressed in the literature [56, 55]. We also omit denial of service (DoS) attacks, such as manipulation of GPU configurations to disable screen output; e.g., disable-then-resume GPU, color shifts. For a well designed SecApp, it would be difficult for an adversary to launch a DoS attack that would remain unnoticed by an observant user.

3.2 Security Properties

A security model for trusted display on commodity systems must satisfy three abstract properties (defined below) that are intended to counter an adversary's threats. To express these properties, we partition the GPU objects into two groups: security *sensitive* and *insensitive* objects. Intuitively, the security-sensitive GPU objects are those that can be programmed by untrusted software (e.g., malicious drivers, applications) to break the confidentiality or authenticity of trusted display output, and those which can be tainted by access to other sensitive GPU objects. For example, sensitive GPU objects include directly accessible objects, such as *frame buffers*, page tables, configuration registers, and objects that can affect the security of other objects, such as GPU commands, and instructions, which can modify GPU page table structures. Furthermore, because GPU objects are mapped into GPU address spaces, the corresponding virtual and physical GPU memory regions are regarded as *sensitive*. In contrast, the security-insensitive GPU objects cannot affect the confidentiality and authenticity of trusted display even if they are manipulated by malicious software.

The three security properties that must be satisfied by trusted display designs and implementations are expressed in terms of the separation of sensitive-insensitive objects and their accesses, complete mediation of accesses to sensitive objects, and minimization of the trusted code base that implements the separation and mediation properties.

P1. Complete separation of GPU objects and their accesses. The trusted display model must partition *all GPU objects* into security-sensitive and security-insensitive objects and must define all *access modes* (e.g., content read, write, configuration modification) for the security sensitive objects and their memory representations.

P2. Complete mediation of GPU sensitive-object access. The trusted display model must include a *GPU separation kernel* that must satisfy the following three properties. The kernel must:
(1) mediate *all* accesses to the security-sensitive objects according to a defined GPU access mediation policy;
(2) provide a GPU access-mediation policy that defines the access invariants for security-sensitive objects; and
(3) be protected from tampering by untrusted OS/Apps and SecApps[5];

P3. Trusted code base minimization. The GPU separation kernel must: (1) have a small code base to facilitate formal verification, and (2) preserve the existing assurance

[5] The isolation of the GPU separation kernel can be easily achieved using the services of existing micro-kernel or micro-hypervisor security architectures [27, 56]

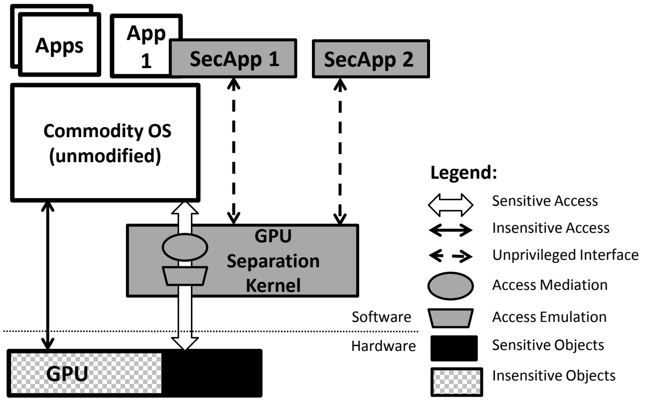


Figure 3: GPU Separation Kernel Architecture.

of the underlying Trusted Computing Base (TCB) necessary for its protection.

3.3 Separation of GPU Objects and Accesses

All security properties of trusted display ultimately rely on the complete separation of GPU objects and their accesses by GPU design and implementation; e.g., GPU separation kernel (GSK) discussed below. This requires the analysis of all interfaces between the software components (untrusted commodity OS/Apps, and SecApps) and GPU objects. The results of this analysis, which are abstracted in Figure 3 and Table 1, enable the design of the access mediation mechanism, policy, and access emulation. For example, this analysis shows that SecApps may provide only display content and geometry configurations to the GPU without sending any commands or instructions, and hence do *not* need direct access to GPU objects. Hence, all their direct accesses to GPU objects are blocked by the GPU separation kernel.

3.4 GPU Separation Kernel

3.4.1 Access Mediation Mechanism

The mediation mechanism of the GSK distinguishes between two types access outcomes, namely direct access to security-insensitive GPU objects, and verified (mediated) access to security-sensitive GPU objects. Every CPU access to sensitive objects must be mediated even though GPU hardware lacks mechanisms for intercepting all GPU commands and instructions individually at run time. Since most GPU commands and instruction references cannot be mediated at run time, they must be verified *before* submitting them to the GPU. Although previous work [46] illustrates how to verify GPU commands, it does not address the verification of GPU instructions, which is more challenging, as argued in Section 2.3.

GPU instructions are limited to three types of operations: arithmetic/logic operations, control flow operations, and memory access operations [23, 7, 50, 39]. Arithmetic/logic operations run on GPU cores only and do not affect GPU memory or other GPU objects. However, an adversary may exploit control flow and/or memory access operations to break the confidentiality and authenticity of trusted display contents. Mediating each of these operations individually without hardware support would be prohibitive since it would significantly enlarge and add complexity to the GSK code base and hence diminish its security assurance. This would also add significant overhead to the OS's graphics performance.

Table 1: GPU object and access separation.

GPU Objects	Untrusted OS/Apps	SecApps
Data	Mediated sensitive access and direct insensitive access	Submitted via GPU separation kernel
Configuration Registers		No access
Page Tables		
Commands		
Instructions	Confined by address space separation	

To resolve the above access-mediation problem, we use an efficient *address-space separation* mechanism. Instead of verifying individual instruction access, this mechanism confines the memory access of GPU instructions; i.e., it limits memory accesses only to those allowed by *local* GPU page tables. As a consequence, GPU address-space separation attacks no longer succeed since GPU instructions can no longer reference GPU memory via the shared GGTT. As the result, the mediation mechanism does not require any GPU instruction modification.

The mediation mechanism must also protect command buffers from modification by malicious GPU instructions and prevent TOCTTOU attacks. For example, some command buffers must be mapped into the local address space of untrusted OS/Apps in GPU memory. However, malicious GPU instructions can modify the GPU commands *after* command verification and invalidate the verification results for GPU commands at run time. Nevertheless, GPU address-space separation hardware can still protect the integrity of GPU command buffers via *write* protection. The confidentiality of command-buffer content does not need extra protection[6] and hence the accesses to GPU instructions that read command buffers need not be mediated.

3.4.2 Access Mediation Policy

GPU access mediation policy comprises a set of "access invariants" that are enforced by the GPU separation kernel. These invariants are designed to ensure the security of the SecApps' display output and must hold at all intermediate points during trusted-display operation. They yield both secure-state invariants and transition constraints in a state-transition model of security policy [18].

Access invariants. As argued in Section 3.1, an adversary's attacks may either breach the confidentiality and authenticity of trusted display content (i.e., *content security*), or destroy the integrity of its configurations (i.e., *configuration integrity*). For example, the adversary can modify the configurations of both SecApps' display and sensitive GPU memory content. Hence, our access mediation policy is defined in terms of *invariants* for GPU object accesses that must be maintained for both content security and configuration integrity.

- *GPU data.* Content security requires the following invariants: (1) no untrusted *read* of the trusted display's

[6] Our adversary model omits side channels and inference analyses that may deduce sensitive output from command content.

frame buffer; and (2) no untrusted *write* to sensitive GPU data.

- *GPU page tables.* The following invariants must hold for GPU address space separation: (1) no untrusted OS/Apps can map sensitive GPU memory to be writable in any GPU local page tables; (2) no untrusted OS/Apps can map the trusted display's *frame buffer* to be readable in any GPU local page tables; (3) untrusted OS/ Apps must have a *single mapping* to sensitive GPU memory in GPU global address space; and (4) GPU instructions uploaded by untrusted OS/Apps cannot reference the GPU's global address space.

- *GPU configuration registers.* Configuration integrity requires the following invariants: (1) no untrusted re-configuration of SecApps' display; and (2) no untrusted re-configuration of sensitive GPU memory. Content security requires the following invariant: no untrusted *read* of the trusted display's *frame buffer*, and no untrusted *write* to sensitive GPU memory.

  In addition, the invariant that untrusted access to configuration cannot violate the access invariants of GPU page tables must also be enforced.

- *GPU commands.* Content security requires the following invariants: (1) no GPU command can *read* trusted display's *frame buffers*; and (2) no GPU command can *write* sensitive GPU memory.

  In addition, the invariant that GPU commands cannot violate (1) any GPU configuration register invariants, and (2) GPU page table invariants must also be enforced.

Soundness. In order to show these invariants are sound, we need to show that if they hold, malicious access to and configuration of the above sensitive GPU objects is prevented. To show this, we note that an adversary can perform only four types of attacks that represent combinations of two basic methods (i.e., unauthorized access to trusted-display content and alteration of its configurations) exercised either directly or indirectly. To prevent direct manipulation of trusted-display content, GPU mediation policy ensures no untrusted access to the trusted display's *frame buffers* is possible. And to prevent indirect manipulation of trusted-display content, GPU mediation policy ensures that no untrusted write to other sensitive GPU memory is possible.

The protection of the trusted-display configuration protection is similar. To prevent direct tampering of output, untrusted re-configuration of trusted display is disallowed. Indirect re-configuration is prevented by requiring the same access invariants as those for indirect access to the display content. Finally, additional invariants are required to ensure that GPU address space separation is maintained in order to avoid complex GPU instructions verification.

3.4.3 Access Emulation

The GSK maintains full object-code compatibility with commodity platforms by retaining the common access methods (e.g., memory-mapped and port I/O) of commodity GPUs and by emulating the expected returns when untrusted commodity OS/Apps perform security-sensitive object operations. We use the security model to identify the minimal set of object accesses and functions that require emulation; viz., Section 4.5. This enables us to minimize the GSK code base; viz., Table 2 in Section 4.4, which shows that only a small number of mediated GPU objects requires function emulation. For example, direct access to security-insensitive objects and security-sensitive access to objects used only by commodity OS/Apps (i.e., outside trusted display) do not need emulation. In contrast, full GPU virtualization has to emulate accesses to *all* GPU objects of the VMs scheduled to access the GPU. In particular, it has to emulate a wide variety of accesses to *all* GPU configuration registers[7] and thus requires a large trusted code base.

When object-access emulation is required by the security model, the GSK returns the expected access results as defined in the GPU specifications without actually accessing the real GPU objects. It does this by "shadowing" the real objects; viz., Section 4.5. For example, to locate OS's frame buffer, GSK emulates accesses to frame buffer base by accessing this register's shadow. Furthermore, for sensitive-object accesses that violate security invariants, the GSK simply drops *write* accesses and returns dummy values for *read* accesses[8].

3.5 Verifiable code base

The GSK code base is both small and simple, and hence verifiable, for the following three reasons. First, as shown in Section 5.1, the number of security-sensitive GPU objects is very small. Most of the GPU objects are security-insensitive, and can be direct accessed without kernel mediation.

Second, the GSK outsources most GPU functions (including all GPU functions used by commodity software and GPU objects provisioning for trusted display) to untrusted OS/Apps because it can verify all untrusted-code results very efficiently. The verification is driven by the policy invariants. Furthermore, only a small number of sensitive GPU objects require function emulation and this takes only a small amount of code, as shown in Section 5.1. Thus, implementing the GPU functions themselves (e.g., the large and complex native GPU drivers) within the GSK becomes unnecessary. The GSK also exports GPU driver code to SecApps using standard techniques [56]; i.e., the traditional GPU software stack already deprivileges *frame buffer rendering* functions and management logic and exports them to user libraries. The GSK uses a similar approach, except that it requires SecApps to provide their own display contents. (Recall that SecApps cannot directly access any GPU objects.)

Third, GSK perserves existing assurance of the underlying trusted code bases. This is because GSK relies on existing security primitives and services already provided by the underlying trusted code bases; e.g., CPU physical memory access control [25, 9], and Direct Memory Access control [8, 24].

[7] Failure to emulate all accesses causes incompatibility with commodity OS/Apps; e.g., Tian *et al.* [46] virtualize GEN6_PCODE_MAILBOX register without emulating its functions, which causes GPU initialization errors in VMs.

[8] To date, we have not found any malware-free commodity OS/Apps code that would be denied accesses to security-sensitive objects.

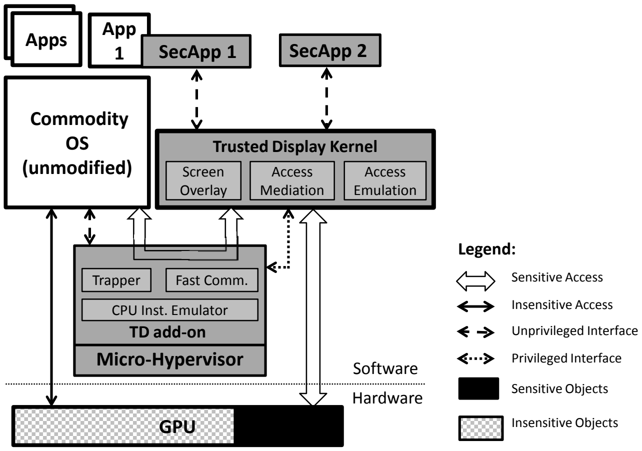


Figure 4: Architecture of the trusted display service. The grey area denotes the trusted base of SecApps.

4. DESIGN AND IMPLEMENTATION

We design the GSK as an *add-on security* architecture [19] based on two components: a *Trusted Display Kernel (TDK)* and a *trusted display (TD) add-on* to the underlying micro-hypervisor (*mHV*). This section first describes the system architecture, and then presents its detailed design for the trusted display.

4.1 Architecture Overview

As illustrated in Figure 4, mHV runs underneath all the other software components, protects itself, and hosts a TD add-on component. The TD add-on extends mHV and takes advantage of the mHV primitives to isolate its execution in a similar manner as that used in past work [35, 56]. The TD add-on notifies Trusted Display Kernel (TDK) about untrusted OS/Apps' requests to access sensitive GPU objects, since the TDK is the only software component in the trusted-display service that is allowed to access these objects directly. The TDK runs at the OS privilege level and provides trusted-display services to user-level SecApps that generate sensitive display content via CPU rendering. The TDK also mediates accesses to sensitive GPU objects by native drivers of untrusted OS/Apps and emulates these accesses whenever necessary.

TDK. The TDK includes three components. The first is the screen overlay component, which displays SecApps output over that of untrusted OS/Apps (Section 4.2).

The second component mediates access to all GPU sensitive objects that reveal or modify SecApps's overlayed display (Sections 4.3 and 4.4). The access mediation mechanism uses the CPU's protection rings to prevent direct SecApps access to GPU objects and uses the privileged mHV interfaces to program TD add-on to intercept sensitive accesses to GPU objects by untrusted OS/Apps.

The third component emulates access to security-sensitive objects to assure object-code compatibility with untrusted OS/Apps. shadowing sensitive GPU objects (Section 4.5). To emulate untrusted accesses, this component either configures CPU/GPU to operate on the shadow GPU objects or simulates object accesses.

TD add-on. The trusted display (TD) add-on supports the TDK in the mediation of security-sensitive accesses to GPU objects by untrusted OS/Apps. It implements traditional hypercalls to receive TDK-initiated communications, and fast communication channels to notify the TDK of access requests received from untrusted OS/Apps. The hypercalls enable the TDK to define the security-sensitive GPU objects so that the TD add-on knows what access requests from untrusted OS/Apps to trap. Once an access is tapped, the TD add-on uses its CPU instruction emulator to identify the object-access requested, access mode, and access parameters; e.g. the new value to be written. This information is sent to the TDK for the mediation and emulation of the requested access to the object, via a fast communication channel.

The TD add-on is implemented using the TrustVisor [35] extension to XMHF, which isolates SecApps and enables them to request on-demand isolated I/O operations [56].

4.2 Screen Overlay

The screen overlay component of TDK displays SecApps' output over that of untrusted commodity software. The screen overlay provides interfaces to SecApps and performs *frame buffer merging* in response to SecApps' requests. Frame buffer merging can be done either purely by software (i.e., by *"software overlay"*) or by hardware acceleration (i.e., by *"hardware overlay"*). In software overlays, TDK shadows the screen frame buffer when TDK is initialized. During the trusted-display session, the TDK merges the SecApps' display contents with the untrusted *frame buffer* using their geometry information, and outputs the resulting image to the shadow frame buffer. Then, the TDK programs the display engine to present the shadow *frame buffer* on the display. In comparison, hardware overlays are supported by some modern GPUs that layer one *frame buffer* over others. This improves the SecApps' CPU performance by eliminating frame-buffer merging by the CPU.

Frame buffer merging does not necessarily ensure that the SecApps display content is layered over all untrusted OS/Apps content. The hardware cursor can still be shown over the merged image, and hence TDK must ensure that the hardware cursor operation is trustworthy. To do this, the TDK provides and protects its cursor image and the corresponding GPU configurations[9]. Similarly, the TDK emulates all hardware overlay not used by SecApps. Thus, SecApps can display over all untrusted contents.

SecApps also provide their display geometries to the TDK to prevent SecApp-output overlaps. Furthermore, the TDK provides a V-sync interrupt to SecApps to prevent image tearing. To enhance human perception of trusted screen areas, a SecApp always starts its display at the center of the screen. The current trusted-display implementation supports hardware overlays only, and allows a single SecApp to execute at a time though a multi-window version implementation does not pose any additional security problems.

4.3 Access Mediation Mechanism

In general, the access mediation mechanism of TDK interposes between commodity software and GPU by intercepting Memory-Mapped I/O (MMIO) and Port I/O (PIO), in a similar manner to that of previous systems [46, 45]. The access mediation mechanism also performs two tasks, namely

[9] The cursor location needs protected only if the mouse device is a trusted-path device. This issue is orthogonal to the design of the trusted-display service.

GPU address space separation, and GPU command protection, as described in Section 3.4.1.

GPU Address Space Separation. GPU address space separation mediates instructions accesses by limiting their scope to GPU local page tables. Nevertheless, GGTT may also contain mappings to security-insensitive objects to be used by GPU instructions. To satisfy these requirements, the TDK shadows GGTT to separate GPU address space as follows:

(1) TDK shadows GGTT in a GPU local page table (GGTT'), and updates GGTT' whenever GGTT is modified.

(2) TDK verifies the access invariants for GGTT'.

(3) TDK forces GPU instructions execution to use GGTT' for all GPU engines except the display engine, which uses GGTT.

Note that in Step 3, the TDK forces GPU instructions to use GGTT' instead of GGTT in two steps. First, the TDK wraps related GPU commands into a batch buffer. Second, it sets the batch buffer to use GGTT'. As a result, GPU instructions preserve both their original functionality and security of the trusted display.

Forcing GPU instructions to use a new GPU local page table poses some implementation challenges. For example, it is possible that no spare slot exists to load GGTT'. Our solution is to randomly kick out a GPU local page table, switch to GGTT' to execute the GPU instructions, and then switch back the original GPU local page table after the GPU instruction execution finishes. In principle, it is also possible that a single GPU command group uses all GPU page tables. Although we have never encountered this situation in normal GPU driver operation [2, 6, 4], our TDK splits the command group into smaller pieces and reuses the solution to the first challenge described above. The TDK also pauses the verification of new GPU command submission in this case and resumes when this command group is executed.

GPU Command Protection. The TDK also protects GPU command buffers from malicious GPU instructions. As GPU page tables support *read-write* access control in many modern GPUs (e.g. Nvidia GPUs [4], AMD GPUs [6] and recent Intel GPUs [23, 2]), the TDK can protect GPU command buffers by mapping their *read-only* accesses to GPU local page tables. However, some Intel GPUs provide different hardware protection mechanisms. For example, GPU privilege protection disallows execution of security sensitive GPU commands from the batch buffers provided by applications. Using this feature, the TDK can enforce the security invariants for GPU commands by de-privileging these commands from the batch-buffer accesses mapped in GPU local page tables.

Table 2: Trusted Display Kernel Minimization. Legend: Bold letters denote object categories that contribute a significant number of GPU objects. (*) denotes categories where objects need mediation. The underline denotes mediated objects that do not require function emulation.

Category		Examples	Mediation		Func.
			Virt.	TDK	Emu.
GPU Data	Display Engine Data	Shadow Framebuffer	Yes	Yes	Yes
	Processing Engine Data	**Other VM's GPU Data**	**Yes**	**No**	**No**
	Data of Other Engines	Performance Report	Yes	No	No
GPU Configuration	Config Registers Used By TD	*Presentation*: SecApps' Geometry, V-Sync Enable	Yes	Yes	Yes
		Data Path: Framebuffer Bases, Target Display	Yes	Yes	Yes
		Others: Ring Buffer Base	Yes	Yes	Yes
	Other Access Sensitive GPU Memory	Performance Report Buffer Base	Yes	Yes	No
	Others		**Yes**	**No**	**No**
GPU Page Tables	Global Page Table	GGTT	Yes	Yes	Yes
	Local Page Tables		**Yes**	**Yes**	**No**
GPU Commands	Access Config Regs./ Page Tables	Update GPU Page Table	Yes	Yes	Yes
	Access Other Sensitive GPU Objects	Batch Buffer Base	Yes	Yes	No
	GPU Processing	**3D Commands**	**Yes***	**No**	**No**
	Others		No	No	No
GPU Instructions			**Yes***	**No**	**No**

4.4 Access Mediation Policy

The access mediation policy of the TDK enforces the security invariants for sensitive GPU objects references; viz., Section 3.4.2. The TDK identifies a GPU object by its address and verifies the validity of the object access by enforcing the corresponding invariants. The TDK enhances the performance of GPU command mediation without sacrificing trusted-display security [46]. In particular, the TDK monitors specific GPU configuration registers and performs the batch verification of an entire group of submitted GPU commands instead of trapping and mediating single GPU command individually. Furthermore, the TDK protects the verified GPU commands from malicious modification in TOCTTOU attacks, by shadowing the ring buffers and *write*- protecting batch buffers as illustrated by Tian *et al.* [46].

The access mediation policy has a small code base, since sensitive GPU objects comprise only a small group among all the GPU objects. As illustrated in Table 2, the TDK needs to mediate accesses to only a few GPU objects of each type and all accesses for GPU global page tables. However, the mediation of page-table access is simple due to their limited formats and sizes. In addition, special GPU protections built in hardware, such as GPU privilege protection, further reduces the mediation effort. It should also be noted that the TDK forces GPU commands to use GGTT' instead of GGTT whenever possible to reduce the mediation overhead of GPU command accesses. Section 5.1 presents the TDK code base minimization results.

Table 2 also shows that TDK has to mediate access to far fewer GPU objects than full GPU virtualization [46, 45]. This is largely due to fine-grained object separation provided by our security model. For GPU data, full GPU virtualization must mediate all the accesses to other VM's GPU data, whereas TDK needs to mediates the accesses only to a small set of trusted-display sensitive GPU data and GPU configuration registers. In contrast, full virtualization approaches need to mediate all accesses to GPU configuration registers to isolate the GPU configurations among different VMs. Full GPU virtualization also requires mediation of accesses to GPU instructions by design, whereas the TDK uses address space separation to avoid verifying individual GPU instructions. In addition, full GPU virtualization needs to mediate accesses to more GPU commands than the TDK, due to the virtualization of the GPU processing engines. In Section 5.1, we will provide a detailed evaluation on the mediation-policy trusted base.

4.5 Access Emulation

The TDK emulates accesses to four categories of GPU objects (viz., Table 2) operating on the shadow objects instead of the originals, as follows.

- For GPU data accesses, the TDK allocates dummy memory with equal size of the sensitive GPU data, and invokes TD add-on to remap the sensitive GPU data to the dummy memory for untrusted OS/Apps.
- For GPU configuration registers, the TDK maintains shadow registers, and updates their values on untrusted accesses and chronology events (e.g. V-Sync) according to their function definitions. TDK also keeps all the pending updates if the corresponding register requires *stateful* restoring during trusted-display finalization; e.g., register update relies on previous updates.
- For GPU page tables, the TDK updates the shadow GGTT' whenever the original GGTT is updated.
- For GPU commands, the TDK modifies their parameters or results to access shadow objects.

Table 2 shows that only a small number of mediated GPU objects requires function emulation. The result is unsurprising since many sensitive GPU objects are not used by the trusted display. Section 5.1 shows the results of access emulation minimization.

4.6 TD add-on

Trapper. The trapper intercepts sensitive GPU object accesses by untrusted OS/Apps, as specified by the TDK during the trusted display session. It use the CPU instruction emulator to obtain the trapped instruction's access information; e.g., object being accessed, access mode, and access parameters such as a new value to be written. Then it notifies the TDK with the access information and busy waits for TDK's access decision. When TDK returns the access decision, the trapper resumes the execution of the untrusted OS/Apps.

CPU Instruction Emulator. CPU instruction emulation is needed only for instructions that access security-sensitive GPU objects, and hence its complexity is lower than that of general purpose instruction emulators and disassemblers. For example, only a small subset of CPU instructions can be used to access GPU objects; e.g., instructions for MMIO and PIO. Also, the only instruction functions that need emulation are those operating in protected mode.

Fast Communication Channel. The fast communication channels facilitate communications initiated by the TD add-on with the TDK on multi-core platforms. They employ shared memory to hold the communication data and use Inter-Processor Interrupts (IPI) for cross-core notification [9, 25, 56]. However, these channels differ from previous approaches in two ways. First, the communication initiator, namely the TD add-on, busy waits for TDK's response after sending a request. Second, the communication receiver notifies *ready* responses via shared memory, instead of issuing IPI back to the TD add-on. Thus, these channels avoid expensive context switches between mHV and TDK, and improve the system performance. More importantly, fast communication channels preserve the mHV's sequential execution and verified security properties [47], since the TD add-on neither receives or handles inter-processor interrupts.

4.7 Life-Cycle

As is the case with previous secure I/O kernel designs [56], the TDK boots on-demand. This requires the mHV to isolate the execution of TDK from commodity OS/Apps and the TDK to isolate its memory from SecApp access.

Initialization. The TDK configures the trusted display services when invoked by a SecApp. The untrusted OS/App provisions GPU objects (e.g. shadow frame buffer, V-Sync interrupt) and pins the related GPU memory in GPU global address space. Then the OS/App registers the configuration via an OS-hypervisor interface. After configuration, the TD add-on switches execution to the TDK. The TDK disables interrupts, pauses GPU command execution, and calls the TD add-on to specify and enable the interception of sensitive GPU objects accesses from untrusted OS/Apps. Next, the TDK initializes GPU access emulation and verifies all GPU objects accesses according to the security invariants. Lastly, the TDK configures shadow memories (e.g. shadow ring buffer, shadow frame buffer) to start the trusted display service, resumes GPU command execution, enables interrupts, and and returns to the SecApp. Trusted display initialization is not needed for a SecApp unless all previous SecApps that used the trusted display terminated. Finalization reverses these steps and zeros shadow memories without verifying GPU objects again.

Untrusted Code Accesses to Sensitive GPU Objects. The TDK mediates the sensitive GPU object access using the access information received from the TD add-on, allows the access if the relevant security invariants are satisfied, and emulates the access if necessary, as described in Section 4.5. Then it returns the access decision to the TD add-on.

5. EVALUATION

We implemented and evaluated our system prototype on an off-the-shelf HP2540P laptop, which is equipped with a dual-core Intel Core i5 M540 CPU running at 2.53 GHz, 4 GB memory and an integrated Intel 5^{th} generation GPU (IronLake) with screen resolution of 1200 * 800. The laptop runs 32-bit Ubuntu 12.04 as the commodity OS with Linux kernel 3.2.0-36.57. We implemented a test SecApp that outputs a still image in all experiments.

Table 3: Number of GPU Objects Requiring Access Mediation.

GPU Object	Mediation in		Total
	TDK	Full GPU Virtualization [46]	
GPU Data	˜6 MB	All other VM's data	2 GB
GPU Configuration Registers[10]	39	711	625
GPU Page Tables	All		
GPU Commands	21	43	269
GPU Instructions	0	14	66

5.1 Trusted Code Base

Access Mediation Comparison. The number of GPU objects that require access mediation by TDK is much smaller than the number of GPU objects mediated in full GPU virtualization approaches [46]; viz., Table 3. This comparison is based on the Intel 7^{th} generation GPUs (Haswell), which has an open-source driver (released by Intel) and detailed documentation. For GPU data, Haswell GPU maps a 2 GB GPU memory into the GGTT. Full GPU virtualization hosts the bulk of other VM's GPU data in the global address space, whereas in our system the sensitive GPU memory is mapped in only about 6 MB. The memory used for sensitive GPU objects includes the shadow *framebuffers* (3750 KB for screens with 1200 * 800 resolution and 4 bytes per pixel), GGTT' (2052 KB), and other very small sensitive GPU memory areas; e.g., shadow ring buffers (128 KB). Note that the ratio of sensitive GPU objects to all GPU objects may vary, since the protection of multiple local GPU page tables requires more mediation of GPU data accesses and also increases the mapped memory space.

The TDK has to mediate access to far fewer GPU configuration registers than full GPU virtualization. That is, access to 39 out of 625 GPU configuration registers require mediation, 13 of which are needed for hardware overlays. In contrast, full GPU virtualization must mediate accesses to all GPU configuration registers to share all GPU functions securely among the different VMs that access the GPU. It also mediates access to more GPU commands than the TDK since it needs to perform additional tasks such as the virtualization of the GPU 2D/3D processing engine. In addition, the TDK does not need to mediate accesses of individual GPU instructions due to its use of the address-space separation mechanism.

Access Emulation Minimization. The TDK has to emulates the functions of only 20 configuration registers and 12 GPU commands since the trusted display only uses a subset of the sensitive GPU objects. As is the case with access mediation, full GPU virtualization needs to emulate the functions of *all* configuration registers to support all VMs that access the GPU, and more GPU commands than the TDK to virtualize the GPU 2D/3D processing engine.

Code Base Minimization. We use the SLOCCount tool to measure the code base of our trusted-display service.

Table 4: Code base size of trusted display service.

(a) Micro-hypervisor

Modules	SLoC
XMHF[11] + TrustVisor	28943
CPU Instruction Emulator	1090
Fast Communication Channel	144
Trapper	66
Total	**30243**

(b) GPU code in TDK

Modules	SLoC
Screen Overlay	177
Access Mediation	2865
Access Emulation	1571
Utility Code	973
Total	**5586**

Table 5: Access mediation overhead of TDK.

GPU Configuration Registers	GPU Page Tables	GPU Commands
2.61 μs	2.69 μs	8.86 μs

As shown in Table 4(a), the TD add-on modules (i.e., trapper, CPU instruction emulator, fast communication channel) add a total of 1300 SLOC, which is minuscule number compared to existing code bases of similar functionality. For example, CPU instruction emulator contributes most of this code, and yet it is an order of magnitude smaller than the code base of general purpose CPU instruction emulation tools; e.g., diStorm3.3[12] has 11141 SLOC. To implement this emulator we stripped the CPU instruction emulator of the Xen hypervisor (i.e., 4159 SLoC of Xen-4.5.0) of unnecessary code and shrank its size by 73.80%; i.e., from 4159 to 1090 SLOC. Were we to use diStorm3.3 instead, we would have bloated our micro-hypervisor's code size by over 38% (11141/(30,243-1,090)) and undoubtedly invalidated the existing formal assurances of XMHF.

Table 4(b) shows the code size of the TDK. The access mediation code for the GPU uses most of the code base since it contains both the mediation mechanism (1741 SLoC) and policy (1124 SLoC). A large portion of the code in these modules (i.e., 2227 SLoC) can be reused across different GPUs, including all the utility code and helper functions of other modules in the TDK. In particular, supporting different GPUs of the same vendor only requires minimal code modification because the GPU object changes are incremental; e.g., IronLake specific code in TDK takes only 178 SLoC.

In contrast, the code size of full GPU virtualization approaches [46, 45] is much larger. It contains a Xen hypervisor of 263K SLOC [56] and a privileged root domain that has over 10M SLOC.

5.2 Performance on micro-benchmarks

In this section, we measure the performance overhead of commodity OS/Apps and SecApps during a trusted-display session. Specifically, we measure the overhead of the access mediation and access emulation components of the TDK to evaluate their impact on untrusted commodity OS/Apps. We also evaluate the overhead added by screen overlays to SecApps. Finally, we illustrate the overhead incurred by the TD add-on component of the trusted-display service.

[10] We count registers using the same functional clustering as in Intel's documentation. This differs slightly from Tian *et al.*'s count [46], which lists registers individually.

[11] XMHF with fine-grained DMA protection takes 24551 Source Lines of Code (SLoC) [56]

[12] https://code.google.com/p/distorm/

Table 6: Overhead of GPU address space separation.

Initialization	Run-time		
GGTT Shadowing	Modify GGTT'	Apply Invariants	Page Table Switch[13]
40.25 ms	0.04 μs	0.10 μs	11.60 μs

Table 7: TD add-on overhead.

Trapper	CPU Instruction Emulator	Fast Communication Channel
11.79 μs	5.43 μs	3.60 μs

Access mediation. The run-time overhead of access mediation is small, though its magnitude varies for different GPU objects. The mediation of access to GPU configuration registers, GPU page tables, and GPU data adds a modest performance penalty to commodity OS/Apps during SecApp run-time. As shown in Table 5, the TDK spends 2.61 μs on mediating access to a GPU configuration register, and 2.69 μs on mediating access to a new GPU page table mapping, on average. However, the TDK does not spend any time on GPU data mediation on the fly, because the sensitive GPU data has been remapped by TD add-on. On average, access mediation for GPU commands takes 8.86 μs. We note that GPU commands mediation overhead may vary; e.g. mediation of the GPU batch-buffer *start* command may require access verification for the entire GPU command batch buffer. However, in most cases, batch-buffer mediation by TDK code is unnecessary since the GPU hardware protection mechanisms can be used instead.

We also measured the performance of GPU address-space separation and GPU command protection since they are important components of the access-mediation mechanism. Table 6 shows the overhead of GPU address space separation added to untrusted OS/Apps. GGTT shadowing incurs the overwhelming portion of the overhead since it needs to parse every mapping in GGTT and construct the mapping in GGTT'. Fortunately, this overhead is incurred only at TDK initialization, and subsequent SecApps display operations do not require GGTT shadowing. In contrast, the run-time overhead incurred by untrusted OS/Apps is small, as shown in Table 6. For GPU command protection, the TDK implementation uses the GPU privilege protection mechanism and takes 0.07 μs on the average to de-privilege a single GPU batch buffer.

Access emulation. Similar to access mediation, TDK's access emulation has a small runtime overhead, which varies for different GPU objects. For example, TDK's overhead for emulating access to GPU page tables is 0.24 μs, on average. We do not measure access emulation costs of GPU data, GPU commands, nor GPU configuration registers. The reason for this is that GPU data accesses are not intercepted by TDK or TD add-on and their access emulation is done by memory remapping. Although the overhead of GPU command-access and configuration-register emulation varies widely with the types of objects and functions used, accesses to these objects are either infrequent or cause a small overhead.

Screen overlay. As anticipated, our experiments confirm that hardware overlays have much better performance than software overlays. Our test SecApp uses a 100 * 100 display area size with a screen resolution of 1200 * 800. Software overlays take 4.10 ms to process a display request of this SecApp. In contrast, hardware overlays take only 0.03 ms, which decreases the overhead of handling a display request in software by 99.27%. Software overlays decrease CPU performance for SecApps as the screen resolution increases. For example, software overlays takes 42.59 ms on displays with 4096 * 2160 resolution, which are becoming increasingly popular. Such an overhead would cause excessive frame rate drops at 60Hz or higher display refresh cycles, which would result in visually choppy display images. We note that the overhead of software overlays increases when performed in GPU memory due to the reading and writing of *frame buffers* by the CPU.

TD Add-on. The TD add-on component takes 20.82 μs to trap and operate on a single sensitive GPU object access by untrusted OS/Apps. Table 7 illustrates the overhead breakdown.

First, the trapper takes 11.79 μs, on average, to intercept an MMIO access to GPU object and resume untrusted OS/Apps execution. (Note that this measurement does not include the PIO access interception.) This overhead is large due to the round-trip context switch between the mHV and untrusted OS/Apps. However, new hardware architectures make the trapper's overhead negligible.

Second, the CPU instruction emulator takes 5.43 μs, on the average, to parse the trapped CPU instruction due to accessing sensitive GPU objects from untrusted OS/Apps. However, this emulation overhead is well amortized since native GPU drivers [2, 6, 4] tend to use a single unified function for configuration updates of each type of GPU objects. Thus, the CPU instruction emulator can cache the emulation result for future use.

Third, our evaluation results show that the fast communication channels are much more efficient than switching between untrusted OS and TDK. In our implementation, the fast channel takes 3.60 μs for the communication round-trip between mHV and TDK when they run on different cores. In contrast, when the mHV needs to switch to TDK and back on the same core, the overhead is 722.42 μs. As a result, fast communication channels on multi-core systems can reduce the communication overhead by 99.50%.

5.3 Performance on macro-benchmarks

We use Linux GPU benchmarks to evaluate the performance impact of the trusted-display service on commodity OS/Apps software. Specifically, we use 3D benchmarks from Phoronix Test Suite [5], including OpenArena, UrbanTerror, and Nexuiz. We also use Cairo-perf-trace [1] 2D benchmarks, including firefox-scrolling ("Firefox"), gnome-system-monitor ("Gnome"), and midori-zoomed ("Midori"). However, we did not run LightsMark in 3D benchmark or firefox-asteroids in 2D benchmark as in previous studies [46], because these workloads have not been available to us.

Our evaluation uses three settings that are designed to distinguish the overhead caused by the underlying mHV from that caused by the trusted display service. These settings are: Ubuntu 12.04 with no security component added ("native"), mHV running *without* the trusted-display service ("TD_off"), and a SecApp using the trusted-display service

[13] We evaluate Haswell GPUs instead, because Intel's open source GPU drivers support local page tables on GPUs newer than IronLake [2]

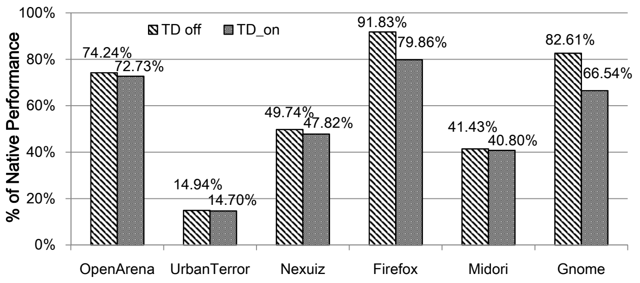


Figure 5: Performance of the trusted display service on 2D and 3D benchmarks.

running on the top of mHV (”TD_on”). The native setting does not load any of out trusted code; i.e., neither the mHV nor trusted-display service code. In the TD_off setting, both the mHV and TD add-on code are loaded, but the TD add-on code is never invoked because the TDK is not loaded. Thus, whatever performance overhead arises in the TD_off setting it is overwhelmingly caused by the security services of the *unoptimized* mHV. The TD_on setting measures the overhead of the trusted-display service over and above that of mHV, and hence is of primary interest.

Figure 5 shows that the TD_on setting achieves an average 53.74% of native performance, while TD_off achieves 59.13%. Thus, the trusted-display service is responsible for only 10% of the overhead whereas the *unoptimized* mHV is largely responsible for the rest. We believe that the performance of the mHV (i.e., XMHF) can be improved significantly with a modest engineering effort; e.g., mapping large pages instead of small pages in the nested page table for CPU physical memory access control, and hence decreasing the page-walking overhead and frequency of nested page table use. We also believe that new hardware architectures, such as Intel's SGX, will make the hypervisor overhead negligible.

Furthermore, the data of Figure 6 show that most frame jitter is caused by the unoptimized mHV, and the trusted-display service does not increase the amount of frame jitter. We obtained these data by measuring the frame latencies of the OpenArena workload using the tools provided by the Phoronix Test Suite. These data show that the frame latencies of TD_on and TD_off settings are similar, whereas those of the native and TD_off settings are different. Specifically, the standard deviations are 6.62, 14.69, 14.49 for Figure 6(a), Figure 6(b), Figure 6(c), respectively.

6. RELATED WORK

6.1 Trusted Display

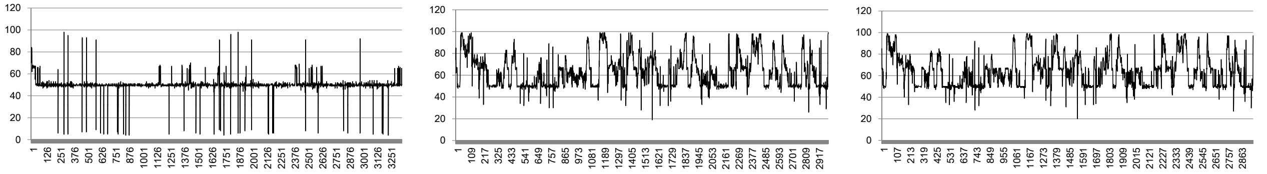
GPU isolation. Several previous approaches provide trusted display services using security kernels. For example, Nitpicker [17], EWS [42] and Trusted X [15] support a trusted windowing system. Glider [41] could also be used to provide a trusted display service since it isolates GPU objects in the security kernel. However, these approaches are unsuitable for *unmodified* commodity OSes, because security kernels are object-code incompatible with native commodity OSes. Past research efforts to restructure commodity OSes to support high-assurance security kernels have failed to meet stringent marketplace requirements of timely availability and maintenance [30, 19]. In contrast, our system does not require any modification of widely available commodity OSes.

Other approaches provide trusted display by exclusively assigning GPU to SecApp. Recent work [48, 51] uses the device pass-through feature of modern chipsets [8, 24] for this assignment. Other work [33, 11] isolates the GPU with a system's TCB. Recent implementations of trusted path [55, 56] also isolate communication channels from SecApps to GPU hardware. However, once assigned to a SecApp, the GPU cannot be accessed by untrusted commodity OS/App code until the device is re-assigned to that code. Thus, a commodity OS/App cannot display its content during SecApp's exclusive use of the trusted display, unless the OS/App trusts SecApps unconditionally. Our system solves this problem by allowing untrusted OS/Apps and SecApps to use GPU display function *at the same time*. As a result, commodity OS/Apps do not need to rely on external SecApps for display services.

GPU virtualization. GPU virtualization can provide trusted-display services by running SecApps in a privileged domain and untrusted OS/Apps in an unprivileged domain. The privileged domain can emulate the GPU display function in software [44, 28] for the untrusted OS/Apps. However, other GPU functions, such as image-processing emulation, are extremely difficult to implement in software and take advantage of this setup due to their inherent complexity [46, 13]. As a result, GPU emulation cannot provide all GPU functions to the untrusted OS/Apps, and hence this approach is incompatible with commodity software. Smowton [43] paravirtualizes the user-level graphics software stack to provide added GPU functions to untrusted OS/Apps. Unfortunately, this type of approach requires graphics software stack modification inside untrusted OS/Apps, and hence is incompatible with commodity OS software.

Full GPU virtualization approaches [46, 45] expose all GPU objects to unprivileged VMs access, and hence allow untrusted OS/Apps to use unmodified GPU drivers. However, these approaches share all GPU functions between privileged domain and unprivileged domain, and hence require complex mediation and provide only low assurance. Existing full GPU virtualization approaches are subject to GPU address space isolation attacks, and hence are inadequate for trusted-display services. Furthermore, full GPU virtualization requires extensive emulation of accesses to a large number of GPU objects, in order to retain compatibility with the VMs that share the GPU. Our system solves the problem by sharing only the GPU display function between the untrusted OS/Apps and the SecApps. Thus, it needs to mediate only accesses to GPU objects that affect trusted display's security. Hence it needs to emulate accesses to a much smaller set of GPU objects, which helps minimize the trusted code base for high assurance system development.

Special devices. High-bandwidth digital content protection (HDCP) [31, 40] employs cryptographic methods to protect display content transmitted from GPU to physical monitor. HDCP requires encryption/decryption circuits, and hence hardware modification of both GPUs and physical monitors. Similarly, Hoekstra *et al.* [21] also require crypto support in GPU to provide trusted display. Intel Identity Protection Technology with Protection Transaction Display is also reported to rely on special CPU and GPU features [22]. In contrast, our system does not require any



(a) Native (b) TD_off (c) TD_on

Figure 6: Latency evaluation of OpenArena. The vertical axis represents latency in milliseconds and the horizonal axis represents frame index.

modification of existing commodity hardware. It could also use HDCP to defend against certain hardware attacks; e.g., malicious physical monitors.

Other cryptography-based approaches [52, 36] decode concealed display images via optical methods; e.g., by placing a transparency, which serves as the secret key, over concealed images to decode them. These approaches are similar in spirit to the use of *one time pads*, and hence need physical monitor modification for efficient, frequent re-keying. Other systems [38] add decryption circuitry to displays, and hence also require commodity hardware modification, which fails to satisfy our design goals.

6.2 GPU Data Leakage

Recent research [12, 29] shows that GPU data leakage may occur because security-sensitive applications may not zero their freed GPU data memory in a timely manner. Thus, malicious applications can probe GPU data of other applications by using native GPU drivers (including user-level libraries) on commodity GPU. Other recent work [34] also focuses on GPU data leakage in virtualized environments and shows that the untrusted OS can manipulate GPU configuration registers to leak GPU data. An early instance of the data leakage problem [46] shows that full GPU virtualization approaches could solve the problem in principle by mediating access to GPU data, configuration registers, commands and page tables. Nevertheless the GPU address space separation attack of Section 2 shows that malicious GPU instructions can still break GPU data confidentiality and authenticity of full GPU virtualization. Our system prevents these attacks since the GSK mediates accesses to sensitive GPU objects and emulates unsafe accesses.

7. DISCUSSION

Direct GPU Access by SecApps. This paper proposes a trusted display design that can be used by the vast majority of SecApps, and has few basic GPU requirements. Most SecApps can render their display content on the CPU and then submit it to the trusted display kernel for output. To support SecApps that require GPU commands and instructions, we suggest the use of either GPU pass-through mechanisms [48, 51], or GPUs that have added hardware isolation features [37]. Full GPU virtualization [46, 45] does not provide sufficient security for SecApps that require direct GPU access.

Recommendations for GPU Hardware Enhancement. Separating sensitive and insensitive GPU registers and memory into different aligned pages would help reduce the trap-and-mediate overhead in commodity OSes and improve OS's run-time performance. GPU hardware overlays [23] provide dedicated memory buffers inside the GPU for programs to enable independent rendering of images and videos on top of the main display screen. Our system – and other trusted display solutions – could leverage these features to further reduce the size of the trusted code base. The address-space separation attacks described in Section 2.2, which are enabled by inadvertent side-effects of a GPU optimization, should serve as a warning to GPU designers that serious security analyses are required when introducing new features that are intended to enhance performance in GPUs.

Uniprocessor Support. On uniprocessor platforms, our trusted display service pauses the untrusted OS/Apps when executing the SecApp. This simplifies the design of GPU isolation since SecApps cannot run concurrently with untrusted commodity software.

8. CONCLUSION

Modern commodity GPUs have increasingly rich functions and higher performance, and yet lack adequate hardware isolation mechanisms for trusted display. Worse yet, full GPU virtualization techniques intended to optimize performance violate the existing rudimentary isolation mechanisms and expose user output to significant vulnerabilities. We design and implement a trusted display service that is compatible with commodity hardware, applications, OSes and GPU drivers, has a trusted code base that is orders of magnitude smaller than previous systems, and preserves relatively high OS graphics performance. Our design also helps identify key areas where added hardware protection mechanisms would enhance GPU object and resource isolation.

Acknowledgment

We are grateful to Yueqiang Cheng, Kun Tian, Yusuke Suzuki, the members of intel-gfx, radeon, nouveau IRC channels, and the CCS reviewers for their useful comments and suggestions. This research was supported in part by CMU CyLab under the National Science Foundation grant CCF-0424422 to the Berkeley TRUST STC. The views and conclusions contained in this paper are solely those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

9. REFERENCES

[1] Cairo-perf-trace. `http://www.cairographics.org/`.

[2] Intel graphics driver. `https://01.org/linuxgraphics/`.

[3] Lima graphics driver for ARM Mali GPUs. `http://limadriver.org/`.

[4] Nouveau graphics driver. http://nouveau.freedesktop.org/.

[5] Phoronix Test Suite. http://www.phoronix-test-suite.com/.

[6] Radeon graphics driver. http://www.x.org/wiki/radeon/.

[7] AMD. AMD Radeon documentation. http://www.x.org/wiki/RadeonFeature/#index10h2.

[8] AMD. AMD I/O virtualization technology (IOMMU) specification. AMD Pub. no. 34434 rev. 1.26, 2009.

[9] AMD. AMD 64 Architecture Programmer's Manual: Volume 2: System Programming. Pub. no. 24593 rev. 3.23, 2013.

[10] ARM. Open source Mali-200/300/400/450 GPU kernel device drivers. http://malideveloper.arm.com/develop-for-mali/drivers/open-source-mali-gpus-linux-kernel-device-drivers.

[11] Y. Cheng and X. Ding. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 201–218, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] R. Di Pietro, F. Lombardi, and A. Villani. CUDA leaks: information leakage in GPU architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[13] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009.

[14] J. Epstein, C. Inc, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Danner, C. R. Martin, M. Branstad, G. Benson, and D. Rothnie. A high assurance window system prototype. *Journal of Computer Security*, 2(2):159–190, 1993.

[15] J. Epstein, J. McHugh, R. Pascale, H. Orman, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype b3 trusted x window system. In *Computer Security Applications Conference, 1991. Proceedings., Seventh Annual*, pages 44–55, Dec 1991.

[16] K. Fatahalian and M. Houston. A closer look at GPUs. *Commun. ACM*, 51(10):50–57, Oct. 2008.

[17] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *Proc. Annual Computer Security Applications Conference*, 2005.

[18] M. Gasser. Building a secure computer system. In *Van Nostrand Reinhold, New York*, 1988.

[19] V. D. Gligor. Security limitations of virtualization and how to overcome them. In *Proc. International Workshop on Security Protocols, Cambridge University*, 2010.

[20] B. Hoanca and K. J. Mock. Screen oriented technique for reducing the incidence of shoulder surfing. In *Security and Management*, pages 334–340, 2005.

[21] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[22] Intel. Deeper levels of security with intel(r) identity protection technology. http://ipt.intel.com/Libraries/Documents/Deeper_Levels_of_Security_with_Intel%C2%AE_Identity_Protection_Technology.pdf.

[23] Intel. Intel processor graphics programmer's reference manual. https://01.org/linuxgraphics/documentation/driver-documentation-prms.

[24] Intel. Intel virtualization technology for directed I/O architecture specification. Intel Pub. no. D51397-006 rev. 2.2, 2013.

[25] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual: Volume 3: System programming guide. Pub. no. 253668-048US, 2013.

[26] S. Kato. Implementing open-source CUDA runtime. In *Proc. of the 54the Programming Symposium*, 2013.

[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. ACM Symposium on Operating Systems Principles*, 2009.

[28] I. T. Lab. Qubes OS. https://qubes-os.org/.

[29] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 19–33, 2014.

[30] S. Lipner, T. Jaeger, and M. E. Zurko. Lessons from VAX/SVS for high assurance VM systems. *IEEE Security and Privacy*, 10(6):26–35, 2012.

[31] D. C. P. LLC. High-bandwidth digital content protection system. www.digital-cp.com/files/static_page_files/8006F925-129D-4C12-C87899B5A76EF5C3/HDCP_Specification%20Rev1_3.pdf.

[32] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, May 2008.

[33] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

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

[35] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proc. IEEE Symposium on Security and Privacy*, 2010.

[36] M. Naor and A. Shamir. Visual cryptography. In *Advances in Cryptology–EUROCRYPT'94*, pages 1–12. Springer, 1995.

[37] Nvidia. Virtual GPU technology. http://www.nvidia.com/object/virtual-gpus.html.

[38] P. Oikonomakos, J. Fournier, and S. Moore. Implementing cryptography on TFT technology for secure display applications. In *Smart Card Research and Advanced Applications*, pages 32–47. Springer, 2006.

[39] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[40] X. Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkely, CA, USA, 1st edition, 2014.

[41] A. A. Sani, L. Zhong, and D. S. Wallach. Glider: A GPU library driver for improved system security. *CoRR*, abs/1411.3777, 2014.

[42] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

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

[44] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. European Conference on Computer Systems*, 2010.

[45] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why not virtualizing GPUs at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[46] K. Tian, Y. Dong, and D. Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[47] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. IEEE Symposium on Security and Privacy*, 2013.

[48] VMWare. Graphics acceleration in view virtual desktops. http://www.vmware.com/files/pdf/techpaper/vmware-horizon-view-graphics-acceleration-deployment.pdf.

[49] Wikipedia. Screen tearing. http://en.wikipedia.org/wiki/Screen_tearing.

[50] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[51] Xen. Xen VGA passthrough fetched on 2015-01-19). http://wiki.xen.org/wiki/Xen_VGA_Passthrough.

[52] H. Yamamoto, Y. Hayasaki, and N. Nishida. Secure information display with limited viewing zone by use of multi-color visual cryptography. *Optics express*, 12(7):1258–1270, 2004.

[53] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan. VGRIS: Virtualized gpu resource isolation and scheduling in cloud gaming. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 203–214, New York, NY, USA, 2013. ACM.

[54] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[55] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[56] Z. Zhou, M. Yu, and V. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 308–323, May 2014.