UNIVERSITY OF CALIFORNIA,
IRVINE

Building Secure Systems Across All Layers

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Zhihao Yao

Dissertation Committee:
Professor Ardalan Amiri Sani, Chair
Professor Gene Tsudik
Professor Alfred Chen

2023

DEDICATION

I dedicate this thesis work to my loving wife and parents
who support and trust me unconditionally.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

In this dissertation I present,
my effort and my intent.
None would have been possible,
without those who made my journey wonderful.

I must thank my advisor,
for always showing me the right path;
You taught me how to debug,
with joy and patience instead of ugh [1] [2];

To my committee members, insightful and kind,
for their expertise and sharp mind;
You chisel my ideas with good care,
and help me reach beyond I can ever dare;

To my collaborators who shepherd me during this quest,
for your unwavering support, I am truly blessed;
You taught me how to work with *glee* [3],
and together, we achieve more than I can foresee.

To my dear dear wife,
for your trust and love which ease my strife;
You supported me with all your heart,
and do not even mind I become cranky working late;

To my parents, you gave me life,
which can hardly express by this little rhyme;
You gave me the best education you could,
and with unreserved love, you taught me life is good;

I must give credit where it's due,
to God whose blessings I drew;
With prayers, I *malloc* your infinite grace,
and never have to worry about the pesky *frees* [4].

I would like to acknowledge the funding support made by the National Science Foundation (NSF) Award #1617513, #1718923, #1846230, #1953932.

[1] In reference to Prof. Ardalan Amiri Sani's philosophy in addressing programming mistakes: patience, attention to detail, and adding prints. This philosophy has been instrumental in helping me overcome many challenges.

[2] Also in reference to our work that identifies bugs in various systems [228, 212]

[3] In reference to our work on discovering WebGL bugs, GLeeFuzz [212].

[4] In c programming, malloc and free are APIs for dynamic memory allocation. Programmers often mishandle them and cause memory errors and security hazards.

Portions of Chapters 1, 2, and 5 are a reprint of the material as it appears in "Sugar: Secure GPU Acceleration in Web Browsers" [264] In Proceedings of the ACM ASPLOS 2018, used with permission from ACM. The co-authors listed in this publication are Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, and Aparna Chandramowlishwaran.

Portions of Chapters 1, 3, and 5 are a reprint of the material as it appears in "Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks" [265] In Proceedings of the ACM CCS 2018, used with permission from ACM. The co-authors listed in this publication are Zhihao Yao, Saeed Mirzamohammadi, Ardalan Amiri Sani, and Mathias Payer.

Portions of Chapters 1, 4, and 5 are a reprint of the material as it appears in "Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware" [266] In Proceedings of the ACM MobiSys 2023, used with permission from ACM. The co-authors listed in this publication are Zhihao Yao, Seyed Mohammadjavad Seyed Talebi, Mingyi Chen, Ardalan Amiri Sani, and Thomas Anderson.

VITA

Zhihao Yao

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2020**
University of California, Irvine *Irvine, California*

Bachelor of Science *with honor* in Computer Science **2017**
University of California, Irvine *Irvine, California*

RESEARCH EXPERIENCE

Graduate Research Assistant **2017–2023**
University of California, Irvine *Irvine, California*

Research Intern **2018**
Microsoft Research *Redmond, Washington*

Undergraduate Research Assistant **2014–2017**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2017–2023**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

A Real-Time Electrical Load Forecasting and Unsupervised Anomaly Detection Framework **2023**
Applied Energy

REFEREED CONFERENCE PUBLICATIONS

Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware **Jun 2023**
ACM MobiSys

GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation **Aug 2023**
USENIX Security

Undo Workarounds for Kernel Bugs **Aug 2021**
USENIX Security

Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks **Oct 2018**
ACM CCS

Sugar: Secure GPU Acceleration in Web Browsers **Mar 2018**
ACM ASPLOS

SOFTWARE

OctopOS `https://github.com/trusslab/octopos`

GLeeFuzz `https://github.com/HexHive/GLeeFuzz`

Hecaton `https://github.com/trusslab/hecaton`

Milkomeda `https://github.com/trusslab/milkomeda`

Sugar `https://github.com/trusslab/sugar`

ABSTRACT OF THE DISSERTATION

Building Secure Systems Across All Layers

By

Zhihao Yao

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Ardalan Amiri Sani, Chair

In response to the growing number of cyber security attacks worldwide, layers of security remedies and patches have been introduced in recent hardware and software systems. Although these solutions have thwarted some attacks, the reactive approach of adding yet another layer of indirection is not a panacea for all security problems. The dissertation aims to create secure systems throughout the computing stack, mainly using two approaches: (1) mitigating security hazards in legacy systems, and (2) designing systems that provide isolation at the lowest possible layer.

First, it highlights the security hazard of vulnerabilities in the graphics stack. To address this, two solutions are introduced. For desktops, it leverages GPU virtualization to provide web apps with a dedicated virtual graphics plane. For mobile devices, it designs a system to automatically repurpose the browser's security checks to safeguard the mobile graphics interface.

Second, it presents a hardware design that is composed of statically-partitioned and physically-isolated trust domains, namely the Split-Trust hardware design. It introduces a few simple, formally-verified hardware components to enable a program to gain provably exclusive and simultaneous access to both computation and I/O on a temporary basis. To manage this hardware, it presents OctopOS, an OS composed of mutually distrustful subsystems.

Chapter 1

Introduction

Due to their ubiquity and capability, modern computers are often used to run security-sensitive applications alongside diverse, untrusted, and potentially malicious programs. For example, mobile users often run financial applications along with untrusted web apps; cloud providers launch programs from different tenants on the same physical machine. Commodity operating systems are not well suited for providing adequate isolation to the growing number of kernel subsystems, especially I/O device drivers. To address these issues, hardware and software vendors have introduced hardware virtualization, additional privilege modes, TEEs (e.g., Intel SGX, Arm TrustZone, Realm), microkernel-style I/O services, and software sandboxes. Although these solutions thwart some attacks, the remedies themselves are largely unverified and have become new attack surfaces, subject to side-channel, micro-architecture, and traditional memory bug attacks. This dissertation is motivated by the emerging challenges of establishing trust amidst the world of constantly evolving and increasingly sophisticated technology.

This dissertation is concerned with the increasing adoption of personal computers for security-sensitive applications, such as digital ID cards, medical devices, and legal documents. Indeed,

current mobile devices already support digital driver's licenses [87] and app-controlled insulin pumps [242]. Given the recent increase in technology adoption, it is especially important to advance system security. In particular, the dissertation takes the first steps toward my research agenda by (1) mitigating security hazards in legacy systems, and (2) designing systems that provide isolation at the lowest possible layer.

1.1 Mitigating Security Hazards in Legacy Systems

This aspect of the dissertation investigates the attack surface in commercial personal computers and attempts to mitigate security hazards by leveraging existing OS primitives and hardware features.

The graphics stack is the Achilles heel of system security because of the bloated Trusted Computing Base (TCB) of the graphics libraries and kernel-mode drivers. Even worse, the user-space access to the GPU driver is non-sandboxed for performance reasons, and this interface is further exposed to web apps through the WebGL APIs after security checks. We demonstrates that WebGL and local apps can exploit the bugs in the graphics stack to compromise the entire system, and it proposes two solutions. First, for desktops and laptops, we provide a dedicated virtual graphics plane to web apps by leveraging GPU virtualization supported by modern hardware. By refactoring the vGPU driver into the web app's address space, WebGL bypasses the OS kernel for driver code invocation, and the rendering is strongly isolated from the rest of the system graphics. Second, for mobile devices where GPU virtualization is not available, we developed a system to automatically repurpose the browser's WebGL security checks to safeguard the mobile graphics interface. Specifically, we create an in-process shield space implemented using page tables for securely deploying these checks within the same address space of an untrusted mobile app.

1.1.1 Secure GPU Acceleration in Web Browser

Modern personal computers have embraced increasingly powerful Graphics Processing Units (GPUs). Recently, GPU-based graphics acceleration in web apps (i.e., applications running inside a web browser) has become popular. WebGL is the main effort to provide OpenGL-like graphics for web apps and it is currently used in 53% of the top-100 websites. Unfortunately, WebGL has posed serious security concerns as several attack vectors have been demonstrated through WebGL. Web browsers' solutions to these attacks have been reactive: discovered vulnerabilities have been patched and new runtime security checks have been added. Unfortunately, this approach leaves the system vulnerable to zero-day vulnerability exploits, especially given the large size of the Trusted Computing Base of the graphics plane.

We present Sugar, a novel operating system solution that enhances the security of GPU acceleration for web apps by design. The key idea behind Sugar is using a dedicated virtual graphics plane for a web app by leveraging modern GPU virtualization solutions. A virtual graphics plane consists of a dedicated virtual GPU (or vGPU) as well as all the software graphics stack (including the device driver). Sugar enhances the system security since a virtual graphics plane is fully isolated from the rest of the system. Despite GPU virtualization overhead, we show that Sugar achieves high performance. Moreover, unlike current systems, Sugar is able to use two underlying physical GPUs, when available, to co-render the User Interface (UI): one GPU is used to provide virtual graphics planes for web apps and the other to provide the primary graphics plane for the rest of the system. Such a design not only provides strong security guarantees, it also provides enhanced performance isolation.

1.1.2 Safeguarding Mobile Graphics Stack

GPU-accelerated graphics is commonly used in mobile applications. Unfortunately, the graphics interface exposes a large amount of potentially vulnerable kernel code (i.e., the

GPU device driver) to untrusted applications. This broad attack surface has resulted in numerous reported vulnerabilities that are exploitable from unprivileged mobile apps. We observe that web browsers have faced and addressed the exact same problem in WebGL, a framework used by web apps for graphics acceleration. Web browser vendors have developed and deployed a plethora of security checks for the WebGL interface.

We introduce Milkomeda, a system solution for *automatically* repurposing WebGL security checks to safeguard the mobile graphics interface. We show that these checks can be used with minimal modifications (which we have automated using a tool called CheckGen), significantly reducing the engineering effort. Moreover, we demonstrate an in-process shield space for deploying these checks for mobile applications. Compared to the multi-process architecture used by web browsers to protect the integrity of the security checks, our solution improves the graphics performance by eliminating the need for Inter-Process Communication and shared memory data transfer, while providing integrity guarantees for the evaluation of security checks. Our evaluation shows that Milkomeda achieves close-to-native GPU performance at reasonably increased CPU utilization.

1.2 Isolation at the Lowest Possible Layer

The underlying principle of this dissertation is to minimize the number and complexity of hardware and software components that a computer owner needs to trust to withstand adversarial inputs. Since the invention of MIT CTSS in 1961, time-sharing systems have enabled various programs to run on the same underlying hardware and system software. Such sharing depends on the hardware and software to effectively sandbox and neutralize malicious programs, but unfortunately, efforts to add additional privilege modes and security mechanisms struggle to keep pace with the growth in exploits. Today, a simple application has to trust a labyrinth of system software (libraries, OS services, OS kernel including all

device drivers) and hardware (CPU, memory management, I/O devices, and even plug-and-play peripherals).

Specifically, smartphone owners often need to run security-critical programs on the same device as other untrusted and potentially malicious programs. This requires users to trust hardware and system software to correctly sandbox malicious programs. The trust has been proven unwarranted. This part of the dissertation aims at providing isolation at the lowest possible layer in the computing stack in order to support security-sensitive applications.

We present a hardware design that is composed of statically-partitioned and physically-isolated trust domains, namely the Split-Trust hardware design. It introduces a few simple, formally-verified hardware components to enable a program to gain provably exclusive and simultaneous access to both computation and I/O on a temporary basis, orchestrated by a minimal yet untrusted resource manager running in a separate domain. For example, security-sensitive applications can request a TEE domain and I/O resources such as storage and network for a certain amount of time. Once approved, the application gains verifiable, uninterrupted, and exclusive access to the resources, secluded from other trust domains at the hardware level. Compared to modern SoCs, our design prototyped on a CPU-FPGA board only incurs a small hardware cost. For security-critical programs, it significantly reduces the required trust compared to mainstream TEEs, and for normal programs, it achieves similar performance to a legacy machine.

Chapter 2

Sugar: Secure GPU Acceleration in Web Browsers

Web browsers have transformed the way we use computers in our daily lives. Starting as a program to navigate static content on the web, web browser is an undeniable pillar of user's experience on personal computers these days. Increasingly, applications running inside web browsers, or *web apps* for short, are capable of competing with their native counterparts in terms of functionality and performance. Many utility applications, such as word processing, presentation, and spreadsheet applications, which used to be available only as native apps, are now available as web apps as well. Indeed, Chromebooks by Google (which only provide a web browser environment for the user) demonstrate the vision that web apps are capable of replacing native apps altogether.

One area in which web apps have recently started to compete with native apps is GPU-accelerated graphics, e.g., for enhanced graphics in a web page, 3D games, and scientific visualization. Most notably, WebGL has recently emerged as a counterpart to OpenGL, promising a native-like graphics API for web apps. Indeed, WebGL has become popular

rapidly: 53% of top-100 websites now use WebGL (§2.1.1) and 96% of 48.8 million visitors to a series of websites used WebGL-enabled browsers [71].

Unfortunately, WebGL endangers the system security. This is because it exposes a large Trusted Computing Base (TCB) to web apps, which are untrusted. This TCB is the operating system's complex graphics plane, which includes the GPU and all the software graphics stack needed to operate it. Similar to OpenGL, WebGL exposes several APIs, the implementations of which span the browser, GPU libraries (including the OpenGL library), and the GPU device driver in the operating system kernel. Moreover, through WebGL, a web app can program different shaders (i.e., GPU kernel code) to run on the GPU, which can directly access the memory using Direct Memory Access (DMA). As a result, WebGL weakens the browser's ability to sandbox the web apps.

Indeed, browser vendors are aware of and concerned with the security implications of WebGL. For example, Microsoft did not initially support WebGL due to security concerns [14]. Browsers that do employ WebGL use various ad hoc solutions to protect against vulnerability exploits. First, they isolate the WebGL implementation in a separate process in the browser called the *GPU process* [30, 44]. Second, they perform runtime security checks on the WebGL API calls [70]. Whenever a new vulnerability is discovered in the graphics plane (which is not in the browser itself), a new security check is added to the GPU process, while vulnerabilities within the browser's WebGL implementation are directly patched. Third, browsers often "blacklist" [70] a system, not allowing the use of WebGL, if the system uses untested GPU device drivers and libraries. Unfortunately, these solutions have important shortcomings: first, while a separate GPU process can sandbox the WebGL implementation, it does not protect the operating system graphics plane from a malicious web app. As a result, a web app can mount various severe attacks on the browser or the system through WebGL, as we will show. Second, the API security checks and vulnerability patches are reactive and cannot protect against zero-day exploits. Finally, the blacklisting approach does

not provide any guarantees for white-listed systems.

To address the shortcomings of WebGL, we present Sugar (**S**ec**u**re **G**PU **A**ccele**r**ation)[1], a novel operating system solution that achieves secure GPU acceleration for web apps while providing high graphics performance. The key idea in Sugar is to leverage GPU virtualization to implement *virtual graphics planes* used by web apps. A virtual graphics plane consists of a dedicated virtual GPU (vGPU) and all the graphics stack needed to operate it, all sandboxed within the web app process. Currently, all the applications (native or web apps) and system services (such as the operating system window manager) use a single *physical graphics plane* in the system, which includes a physical GPU and its device driver in the kernel. However, as mentioned, this physical graphics plane exposed to untrusted web apps significantly increases the size of the TCB. In Sugar, a web app is given a dedicated virtual graphics plane, which is fully isolated from the rest of the system. The trusted components of the system, including the operating system window manager and the browser's core processes, use a separate graphics plane, hereafter called the *primary graphics plane*. The main property of the primary graphics plane is that it has exclusive access to the display and is used to (*i*) perform graphics acceleration for trusted components and (*ii*) display content rendered by various graphics plane on the screen providing a unified User Interface (UI).

We address the following challenges in Sugar. First, to enable a web app to use a dedicated vGPU in an isolated manner, we port the vGPU device driver as a user space library and link it with the web app process (§2.3). Second, we redesign the Chromium web browser's WebGL stack so that a web app is responsible for its own GPU rendering. To do this, a web app uses one of its own threads (called the GPU thread), rather than the browser's GPU process, for processing its WebGL commands, and only shares the final WebGL texture with the GPU process for compositing (§2.4).

We evaluate the security and performance of Sugar. We show that with Sugar, the TCB of

[1]Sugar is open sourced: https://trusslab.github.io/sugar/

the graphics stack exposed to web apps is 20 times smaller. We also demonstrate that Sugar effectively protects the system against 19 reported WebGL vulnerability exploits out of the 20 reports that we examined. Moreover, we show that it achieves high graphics performance: for benchmarks that normally achieve a framerate higher than the display refresh rate of 60 Hz, Sugar also provides a framerate higher than 60. For those that are normally around or below 60, Sugar achieves competitive framerate. As a result, Sugar achieves similar user experience for WebGL rendering.

We present two different designs of Sugar. We design single-GPU Sugar for machines with a single virtualizable GPU. Our main targets for this design are commodity desktops and laptops using Intel processors that incorporate a virtualizable integrated GPU (all Intel Core processors starting from the 4th generation, i.e., Haswell [243]). We design dual-GPU Sugar for machines with two physical GPUs, one of which is virtualizable. Our main targets for this design are high-end desktops and laptops that incorporate a second GPU in addition to the virtualizable integrated Intel GPU. In both designs, web apps use the virtual graphics planes created by the virtualizable GPU. The main difference between the two designs is the primary graphics plane. In single-GPU Sugar, the primary graphics plane uses the same underlying virtualizable GPU but has exclusive access to the display connected to it. In dual-GPU Sugar, the primary graphics plane uses the other GPU, which is connected to the display. As we will show, dual-GPU Sugar provides better security than single-GPU Sugar, especially against Denial-of-Service attacks. Moreover, dual-GPU Sugar achieves better graphics performance isolation. That is, web apps' usage of WebGL causes a smaller drop in the graphics performance of the rest of system and vice versa.

2.1 Current State of WebGL

In this section, we present an overview of WebGL. More specifically, we discuss the adoption of WebGL and its reported security vulnerabilities, which motivate our work.

2.1.1 Adoption

Adoption rate. To study the adoption of WebGL by top websites, we modify Chromium's HTMLCanvasElementModule::getContext() function to detect if a web page attempts to get a WebGL context (required to use WebGL). We then use this browser to analyze the top websites in the Alexa Top Sites list [45]. We analyze randomly-visited pages within each site for one minute. We sometimes manually visit some pages not covered by the random visit. Our analysis shows that *at least* 53% of the top-100 sites, 29.3% of the top-1000 sites, and 16.4% of the top-10,000 sites use WebGL.

As websites have increased their use of WebGL, browsers on personal computers are also increasingly WebGL-enabled. At the time of this writing, WebGL Stats reports that 96% of 48.8 million visitors to a series of contributing websites used WebGL-enabled browsers [71].

Uses of WebGL. Next, we investigate the reasons behind the use of WebGL in these websites. Driven by the demand for GPU-based graphics acceleration, many popular websites, including websites of Apple, Microsoft, Google, Facebook, and Baidu have adopted WebGL. For example, Apple utilizes WebGL to render the introduction pages for the macOS [46]; Microsoft creates numerous WebGL demonstrations, including one for the Assassin's Creed Pirates and one for the Dolby Audio Experience [55]; Google and Baidu use WebGL to enhance their map services [53, 47]; and Facebook and Unity work together to provide their users with WebGL-based online games [42].

In addition to being used for enhanced graphics in web pages and 3D games, WebGL is also widely adopted for scientific applications. Some examples include the simulation of the kinematic model of robots [138], the NASA Experience Curiosity website, which allows the public to learn about Mars and the Martian rover [56], the NGL Viewer, which visualizes molecules [225], the Thingiverse Customizer, which previews and edits 3D printing models [63], the LiverAnatomyExplorer, which facilitates medical education [117], and the NIST Digital Library of Mathematical Functions, which brings mathematical formulas to life [58].

2.1.2 Security

We study the WebGL vulnerability reports. We find that since its adoption, WebGL has seen several vulnerability exploits, most of which is solved by Sugar by design.

WebGL vulnerability statistics. We search for WebGL vulnerabilities in the National Vulnerability Database (NVD) [57] and Chrome bug report database [50]. We search these databases using the keyword "WebGL".

Figures 2.1 shows the number of WebGL vulnerabilities reported in these databases. They demonstrate that WebGL-related vulnerabilities have not decreased significantly over the years. These statistics confirm our hypothesis: WebGL introduces a large amount of trusted code, making it difficult to discover and patch all vulnerabilities in a timely manner. In Sugar, we eliminate most of these vulnerabilities (and many yet not discovered) by design, i.e., by sandboxing the entire graphic plane in the web app process.

WebGL vulnerability examples. We study 20 of the WebGL vulnerabilities in detail (including some reported in Figure 2.1 and others found through manual search, e.g., Firefox WebGL vulnerabilities). Our goal in this study is to understand the impact of these vulnerabilities and to determine whether Sugar can eliminate them.

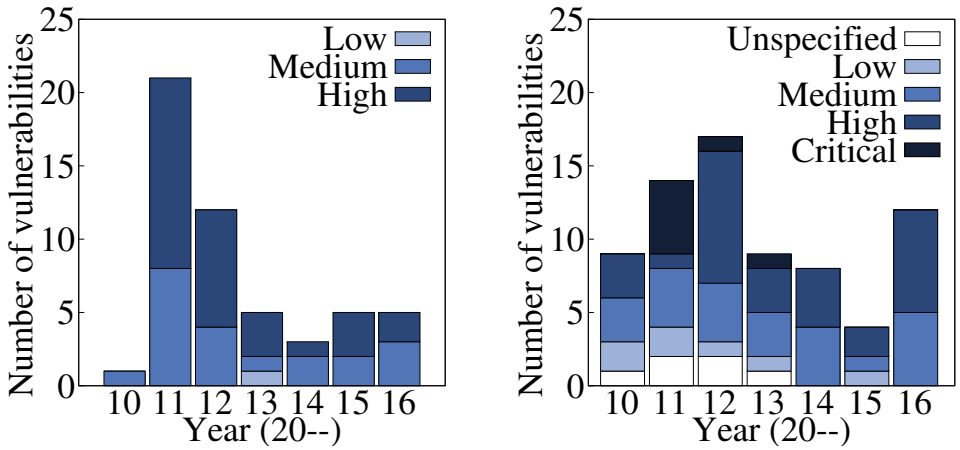


Figure 2.1: WebGL vulnerability statistics according to NVD (Left) and Chrome reports (Right). Note that the max severity level in NVD is "High".

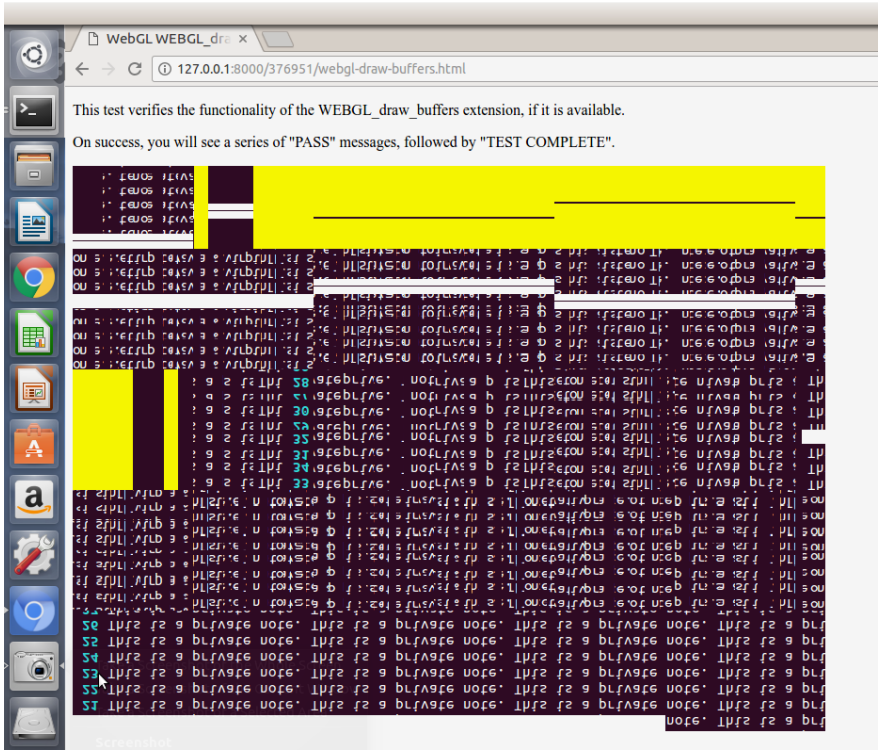


Figure 2.2: Exploiting vulnerability #10 in Table 2.1. The exploit manages to access to unauthorized parts of the GPU memory, which holds previously rendered UI content. In this snapshot, the exploit has accessed the content of a text previously edited in a native text editor.

Table 2.1 characterizes these vulnerabilities. For each of them, we take the following steps. First, we attempt to recreate the vulnerability exploit in the current version of the platform targeted by the exploit. The platform refers to the browser (e.g., Chrome or Firefox), the operating system (e.g., Linux, Windows, and macOS), and the GPU (e.g., Intel and NVIDIA). The 8th column in the table shows that we could recreate only 3 of the vulnerabilities in the current version of the platform since most have already been fixed.

Second, we attempt to recreate the exploits after removing the fix patch from the current version. This allows us to validate the vulnerability and potentially use it to evaluate Sugar. The 9th column in the table shows that we successfully recreated 3 more of the exploits this way (Figure 2.2 shows our successful recreation of vulnerability #10). For the rest, we could not take this approach since either the fix was in a closed source component, we did not have access to the patch, or the exploit targeted a platform we did not possess.

Third, we study the vulnerability reports in detail by analyzing the reports themselves along with the discussions and related reports. When possible, we also study the targeted vulnerable code and the fix. We describe the type of vulnerability and its potential impacts in the 3rd and 6th columns, respectively, according to our understanding. We have published our detailed study of these vulnerabilities on the Sugar's website[2].

Fourth, we investigate the severity of each vulnerability using the reports. We list the vendors report number and severity in the 4th column and the corresponding NVD report number and severity in the 5th column. Note that different vulnerability report databases have different scoring systems for capturing the severity. For example, NVD uses the Common Vulnerability Scoring System version 2 (CVSS v2), which has the following severity levels: Low, Medium, and High [62]. Chrome reports, on the other hand, uses the following levels: Low, Medium, High, and Critical. We use the levels used by the corresponding report.

[2]https://trusslab.github.io/sugar/webgl_bugs.html

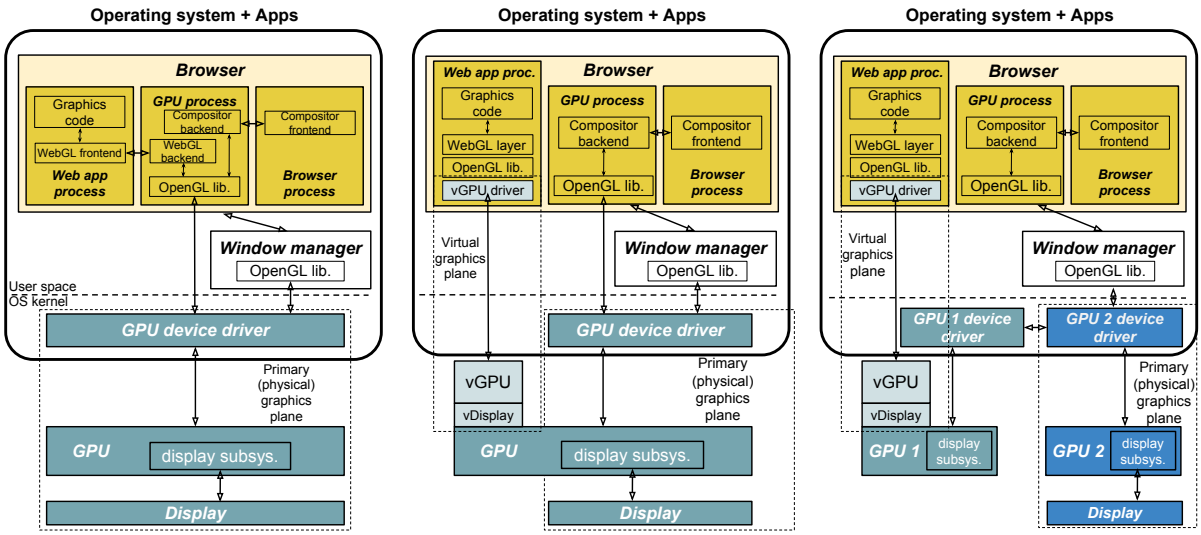


Figure 2.3: *(a) Existing WebGL architecture. (b) Single-GPU Sugar's architecture. (c) Dual-GPU Sugar's architecture. Note that the graphics planes bounding boxes in the figures only enclose the GPU and its driver, and not the graphics libraries, for simplicity.*

Finally, we investigate whether Sugar eliminates these vulnerabilities or not and show the results in the 10th and 11th columns in the table (for single-GPU Sugar and dual-GPU Sugar, respectively). We evaluate the effectiveness of Sugar for most of the vulnerabilities by analysis (shown in the table using "BA"). Also, for one vulnerability, we experimentally evaluate the effectiveness of Sugar. This is a vulnerability that (*i*) we have successfully recreated and (*ii*) target the platform used in Sugar's prototype (i.e., Chrome, Linux, and Intel GPU). We find that, out of the 20 vulnerabilities, single-GPU Sugar and dual-GPU Sugar can eliminate 17 and 19 of them, respectively. In §2.6.1, we provide more details on this evaluation.

2.2 Sugar's Design

Our preliminary study in §2.1 demonstrates various security problems in WebGL. In this section, we present the design of Sugar, an operating system solution that addresses many

of these problems by design.

Our key idea in Sugar is to use isolated graphics planes for web apps. We use the term *graphics plane* to refer to a GPU (or a vGPU) and the software stack required to use it. The key rationale behind our design is the observation that sharing a single physical graphics plane in the current operating system is the source of the security problems in WebGL. More specifically, in today's systems, all the applications, including the web apps, use the same physical graphics plane for hardware acceleration. Moreover, the operating system window manager and browser's core processes also use the same graphics plane. To make the matters worse, the GPU device driver (which is a key and large part of the graphics plane) runs in the operating system kernel making its vulnerabilities dangerous. Therefore, in Sugar, by using fully isolated graphics planes for web apps, we significantly reduce the size of the TCB.

Figure 2.3 (a) shows the existing architecture of the graphics plane and how it is used by web apps. In this architecture, web apps communicate to a GPU process in the browser for WebGL API calls. The GPU process in the browser performs security checks on the WebGL calls and then uses the OpenGL library to communicate with the GPU device driver to render the WebGL texture. The browser's compositor in the browser process determines the layout of the final browser's window and composites the entire UI using the GPU process. In doing so, it uses the WebGL texture previously rendered by the GPU process. Note that in Chrome, the compositing process is performed in two steps. First, a compositor thread in the web app process composites the web app's UI (by sending commands to the GPU process). The browser's compositor then composites the full browser window content. However, in our discussions and in the figures, we only show a single browser compositor in the browser process for simplicity.

The key idea in Sugar is to use virtualization support on modern GPUs, such as Intel integrated GPUs, to realize isolated graphics planes for web apps. A virtualizable GPU can create multiple virtual GPUs (vGPUs) all isolated from each other. vGPUs are normally

used by virtual machines. One of our contributions in Sugar is to enable an operating system process, e.g., a web app process in the browser, to program and use a vGPU. The process loads the entire graphics stack into its address space (including the device driver, which we transform into a library as discussed in §2.3). The vGPU along with its software stack is an isolated graphics plane used by the web app, referred to as a *virtual graphics plane*.

The operating system window manager, the browser core processes such as the GPU process, and the rest of the trusted applications use a different graphics plane for their operations. We refer to this graphics plane as the *primary graphics plane*. This graphics plane has a special requirement: exclusive access to the display. That is, it must have the unique ability to program the display controller in order to set the address of the framebuffer and to set the display mode (e.g., resolution). The operating system will then allow the window manager (but no other processes) to use the primary graphics plane's access to the display controller for UI management.

Figure 2.3 (b) and (c) show two variants of Sugar's architecture. Figure 2.3 (b) shows the single-GPU Sugar variant, in which we assume that the system has a virtualizable GPU. Our main targets for single-GPU Sugar are commodity desktops and laptops using Intel processors that incorporate an integrated virtualizable GPU (all Intel Core processors starting from the 4th generation, i.e., Haswell [243]). In this design, each web app uses a virtual graphics plane. Moreover, the primary graphics plane uses the same underlying GPU but is given exclusive access to the display subsystem by the GPU device driver.

Figure 2.3 (c) shows the dual-GPU Sugar variant, in which we assume that the system has two GPUs, one of which is virtualizable. This setup is common in high-end laptops and desktops that include one GPU in addition to the one provided by the Intel processor. It will also be available in the recently announced "Intel with Radeon Graphics", a multi-chip package that incorporates both an Intel integrated GPU and a Radeon GPU [82]. Similar to the single-GPU Sugar, this design uses a virtual graphics plane for a web app. The main

difference is the primary graphics plane. In this design, the primary graphics plane uses the other GPU in the system. Finally, note that in all the three architectures in Figure 2.3, the web app process is not allowed to directly communicate with the GPU device driver in the kernel.

WebGL texture retrieval in Sugar. In Sugar, a web app does not use the browser's GPU process for WebGL support. Instead, it makes calls into its own "GPU thread" (§2.4.1). The GPU thread executes the web app's WebGL commands in order to render the WebGL texture. The GPU process then retrieves this texture and uses it to composite the browser's UI (per compositing commands from the browser process).

Sharing a texture rendered by a web app with the GPU process requires transferring a graphics buffer (i.e., the buffer holding the WebGL texture) from the web app's virtual graphics plane to the primary graphics plane. While buffer sharing within a graphics plane is trivially enabled by the GPU (or vGPU) device driver, doing so across the planes is more challenging. We use two techniques to enable this. First, we use the virtual display (i.e., vDisplay) read-back capability of the Intel GPU virtualization. That is, for every frame, once the WebGL texture is ready, we use a simple GPU shader to *post the texture to the virtual display of the vGPU*. That is, we copy the texture to the framebuffer of this vGPU in a fullscreen mode. The Intel GPU device driver then encapsulates the virtual display framebuffer as a texture available to the browser's GPU process, which can use it for compositing.

While the previous technique is sufficient for single-GPU Sugar, it is not adequate for dual-GPU Sugar since it uses two different physical GPUs for the web app's virtual graphics plane and the primary graphics plane. Therefore, in our second technique, we use Linux `dma-buf` interface [54] to transfer buffers between the two GPU device drivers. With this interface, the Intel GPU exports the virtual display's framebuffer, which is then imported by the device driver of the other GPU.

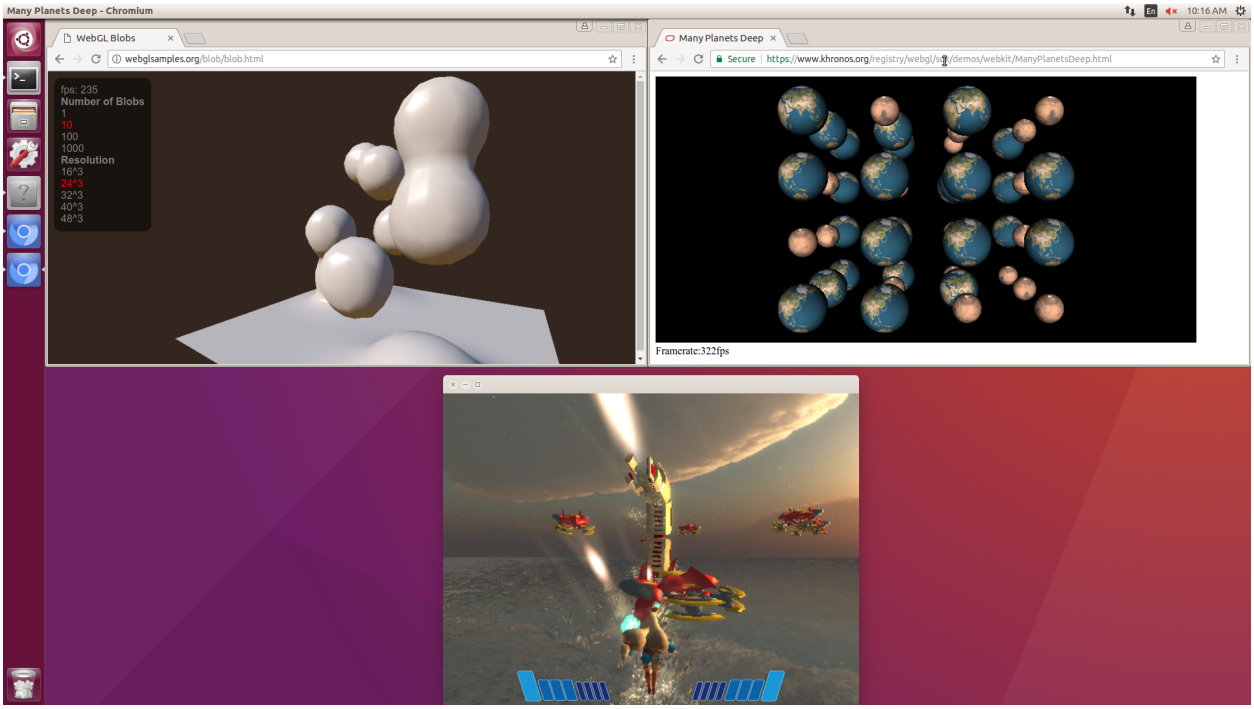


Figure 2.4: Screenshot of dual-GPU Sugar in action.

Co-rendering the UI. One of our key contributions is to use multiple graphics planes (potentially using different physical GPUs as in dual-GPU Sugar) to render the content of a single unified UI displayed to the user on the display. Figure 2.4 shows an example screenshot of UI in dual-GPU Sugar, which illustrates this point. In this screenshot, the blob texture in one browser session is rendered by one Intel vGPU, the planets texture in the second browser session is rendered by another Intel vGPU, and the native 3D game and the rest of the UI is rendered and composited by a Radeon GPU.

Single-GPU Sugar vs. dual-GPU Sugar. Single-GPU Sugar is deployable on any machine with a single virtualizable GPU, such as Intel integrated GPUs. However, when a second GPU is available, dual-GPU Sugar is preferred since it provides two advantages. First, it can protect against Denial-of-Service attacks caused by hanging the GPU (§2.6.1). If successful, these attacks cause the system UI to freeze, causing significant inconvenience to the user. Single-GPU Sugar in our current prototype cannot protect against these attacks because hanging a vGPU in the Intel GPU virtualization technology hangs the underlying

physical GPU as well. However, in dual-GPU Sugar, the primary graphics plane uses a separate physical GPU. Hence hanging the Intel GPU does not result in a UI freeze. Second, dual-GPU Sugar provides enhanced performance isolation, as we demonstrate in §2.6.2, since the web apps will not time-share the underlying GPU with the rest of the system. In other words, when using dual-GPU Sugar, web apps' usage of WebGL causes a smaller drop in the graphics performance of the rest of system and vice versa.

Indeed, many modern high-end desktops and laptops incorporate two GPUs. Therefore, one might wonder how existing systems leverage these two GPUs and how dual-GPU Sugar advances the state of the art. In most laptops and desktops, only one GPU is connected to the display and hence that is the only GPU used for graphics. Some systems can support different displays connected to different GPUs as well. In such cases, each display is fully controlled by a different GPU. In dual-GPU Sugar, only one GPU is connected to the display but the content on this display can be rendered by two GPUs (when a web app renders a WebGL texture). Such a seamless integration of two GPUs for graphics is one of our contributions.

Note that many existing systems use the second GPU for computation. Doing so is easier since the UI is supported by one GPU and the other GPU is simply treated as an accelerator.

Supporting multiple web apps. Sugar can support multiple web apps using WebGL simultaneously. It does so by assigning a separate vGPU to each of the web apps. However, Sugar is bound by the max number of vGPUs achieved by the virtualizable GPU. In our prototype, we find this number to be 3. Even though Intel GPU virtualization can theoretically support up to 8 vGPUs, some practical constraints in this technology limits this number (see §2.6.2). Given this limitation, one might wonder what Sugar can do if more web apps need to use WebGL. We see three possible options, which we plan to explore in the future. First, Sugar can simply prevent more web apps from using WebGL. Second, it can allow some user-selected white-listed web sites to use WebGL on top of the primary

graphics plane bypassing Sugar. Third, it can enable a group of web apps to share a single vGPU by assigning that vGPU to a separate GPU process, with which all these web apps communicate.

2.2.1 Threat Model

We assume that a web app, and hence the whole web app process, is untrusted. This is the common threat model for web apps as they are developed by potentially unknown developers and can contain malware. We assume that the rest of the browser, including the browser and GPU processes, and the operating system are trusted.

We attempt to protect the system against various attacks by a web app including integrity, confidentiality, and availability attacks (§2.1.2). We do not protect against side-channel attacks.

2.2.2 Trusted Computing Base

We define the TCB of WebGL architecture (either the existing one or Sugar) as the privileged parts of the software stack involved in performing hardware acceleration through the WebGL API. A privileged part refers to one residing outside the web app process (which is the sandbox for the web app code). The TCB of existing WebGL architecture includes the browser's GPU process, the OpenGL and GPU libraries, and the GPU device driver. The TCB of Sugar includes the GPU virtualization software (mainly an emulation layer), Sugar's code to attach a vGPU to a process (§2.3.1), and part of KVM for instruction decoding used in Sugar (§2.5).

It is important to note that we rely on the operating system kernel and root user to be protected from a web app. If a web app can gain root or kernel privileges, it can simply

bypass Sugar. We also trust the GPU hardware.

2.3 vGPU Driver as a Library

One of the key components of Sugar is enabling a web app to use a vGPU for WebGL rendering. In this section, we discuss how Sugar achieves this.

Before presenting our solution, we discuss a straw-man solution. This solution is to run a web app inside a virtual machine with access to a vGPU. Prior work has demonstrated a web browser design in which each web app runs inside a virtual machine, e.g., in Tahoma [135]. A similar approach is being used in Windows Defender Application Guard in Microsoft Edge [43]. However, one main drawback of this solution is that it requires significant re-vamping of the browser design. Moreover, even with hardware support for virtualization available in modern processors, CPU and memory virtualization still incurs some – although small – overhead, and hence this design does affect the overall performance of the browser, even for non-graphics tasks.

In Sugar, we take a different approach. That is, we enable the web app to directly access and use a vGPU without requiring a virtual machine. We achieve this by wrapping the vGPU's device driver inside a user space library, link the library to the web app process address space, and then attach the vGPU to the process. This reduces the required modifications to the browser to only the WebGL stack. Moreover, it avoids the overhead of CPU and memory virtualization.

2.3.1 Attaching a vGPU to an Operating System Process

As previously mentioned, our current focus is to use the virtualization support of Intel GPUs since they are commonly available on all modern desktops and laptops. Intel GPU virtualization is a mediated passthrough solution, which leverages hardware isolation features such as GPU page tables. In this solution, the vGPU device driver's attempts to access the vGPU's registers and page tables are trapped by the virtualization layer and emulated. However, the vGPU driver's access to performance critical resources, such as memory, is not trapped enabling high graphics performance.

To enable an operating system process to directly use a vGPU, we employ the following techniques. First, we map the registers of the vGPU into the process address space, but remove read and write permissions from these mappings. This allows the vGPU driver to access the registers, which is then trapped into the kernel, passed to the GPU virtualization layer, and emulated as needed. Note that this is possible since all the vGPU's registers are memory-mapped (i.e., Memory-Mapped I/O or MMIO).

Second, we deliver the interrupts for vGPU using operating system signals (SIGUSR1 in our prototype). When the vGPU driver disables the interrupts, we mask the signal. Similarly, when the driver re-enables the interrupts, we unmask the signals and deliver the pending ones.

Third, we add support for vGPU driver's programming of the GPU page tables. Intel GPUs include an MMU, which allows the device driver to control the GPU's access to memory using Direct Memory Access (DMA). Similarly, the vGPU device driver attempts to program the vGPU's MMU page tables. In this case, the GPU virtualization layer shadows the page tables. That is, it traps vGPU driver's attempt to update the tables and updates the actual tables. The page table shadowing is required for safety. It only allows the vGPU driver to map its own process memory pages into the page tables, which limits the vGPU's DMA

access to the web app process memory.

Shadowing the vGPU's page table in Sugar raises a challenge – determining the virtual and physical address spaces for the vGPU device driver. Normally, the vGPU device driver programs the page tables using the physical addresses of the virtual machine that it runs in. The GPU virtualization layer then translates these physical addresses to system physical addresses. However in Sugar, the vGPU driver runs in an operating system process, which only has a single virtual address space (and no notion of a physical address space). We solve this challenge by refactoring the vGPU device driver and using the process virtual address space as both the vGPU driver's virtual and physical address spaces (i.e., one-to-one mapping). In this case, the vGPU driver updates the page tables using its physical address space, which is identical to its virtual address space. This enables the virtualization layer to translate the vGPU's physical addresses by simply walking the process page tables.

Fourth, we pin in memory the process memory pages that can be accessed by the vGPU through DMA. This ensures that the physical pages will not be swapped out as long as they can be accessed by the vGPU. Pinning memory pages puts pressure on the operating system memory manager. In future work, we plan to explore techniques similar to that in [178] to minimize the number of pinned pages in Sugar.

Finally, graphics applications interact with the GPU driver through user space libraries. These libraries include the OpenGL library and some platform-specific GPU libraries such as the Direct Rendering Manager (DRM) libraries in Linux-based OSes. These libraries issue system calls to interact with the driver. We modify these libraries to instead issue a function call into the vGPU driver library for Sugar. Note that these modified libraries are only used by Sugar. The rest of the system can continue to use the unmodified versions of these libraries for their own access to the primary graphics plane.

2.3.2 Reusing the vGPU Driver Code

As mentioned, we run the vGPU driver as a library within a process. Existing vGPU driver from Intel is developed to run in an operating system kernel, and not in the user space. Therefore, one option for us was to rewrite the driver for user space. However, this approach would have required significant engineering effort since Intel's vGPU driver is almost the same as the Intel's actual GPU driver, which consists of about 123,000 LoC. Therefore, we decided to instead port the existing kernel driver to user space with a wrapper. We use User Mode Linux (UML) as our wrapper. UML ports Linux to run on top of the Linux syscall interface. We modify the build system of the UML so that it is built into a shared library, and not an executable.

We allocate memory for the library in two steps. First, at UML and driver's initialization time, we allocate a fixed chunk of memory, which is given to the SLAB page allocator of UML and used for small allocation calls in the driver (e.g., allocating memory for an object). Second, for larger memory allocations required for graphics buffers, we dynamically allocate more memory from the system. We believe that this design achieves a reasonable trade-off between performance and memory provisioning. Allocating all the memory at initialization time can result in over-provisioning of memory. On the other hand, allocating all the required memory dynamically can affect the performance especially due to small object allocations within the driver.

2.3.3 Surface Management for vGPU

A graphics plane typically requires a window manager to control and share the framebuffer between applications using that plane. The window manager allocates windows for applications. It also allocates renderable surfaces for these windows. Once the applications have filled these surfaces with their UI contents, the window manager composites all of these

surfaces on their corresponding window locations on the framebuffer. The operating system window manager shown in Figures 2.3 (b) and (c) operates on the primary graphics plane.

Similar to the primary graphics plane, a virtual graphics plane requires some form of window management to control the usage of its framebuffer. However, due to our design, a virtual graphics plane is used by a single web app, making a full-blown window manager unnecessary. Therefore, we use a baremetal surface manager in Sugar. The surface manager allocates a single fullscreen surface for the web app and posts the WebGL texture fullscreen to the vGPU's display. In §2.7, we explain how this design would cause challenges for web apps that use more than one WebGL texture and discuss a potential solution for it as well.

2.4 Browser's Support for Sugar

2.4.1 GPU Thread vs. GPU Process

As previously mentioned, modern web browsers, such as Chromium, use a dedicated process for GPU-related tasks, called the *GPU Process* [30, 44]. All other processes communicate with this process for using the GPU. In Sugar, web apps use dedicated vGPUs. Hence, the web app process handles all the GPU-related operations. To enable this, we create a thread in the web app process for GPU-related tasks, called the *GPU thread*. When needed, other threads in the web app process submit GPU-related tasks to this thread for execution.

The GPU thread executes mostly the same code that the GPU process does, with the following exceptions. First, the GPU process receives graphics operation requests through IPC from other processes whereas the GPU thread receives requests only from other threads in the same process. Second, the GPU process does not perform any display management operations. It only acquires a window and its surface from the operating system window

manager and renders the browser's final UI on that surface. In contrast, the GPU thread configures and manages its own virtual display.

It is important to note the rationale behind using a dedicated thread in the web app process for graphics operations. While it was possible for us to simply execute the graphics operations in the same thread that executes the web app's javascript code, we opted for a separate thread in order not to slow down the rest of operations in the web app since graphics operations can block for relatively long periods of time.

2.4.2 Rendering Synchronization

In the existing WebGL architecture, the GPU process orchestrates the submission of commands to the GPU based on their dependencies. Consider the following example – the browser process issues compositing commands to the GPU process. These commands rely on the web app's WebGL textures to be rendered first and used in compositing. To enable this, the browser process inserts a *sync point* in its commands declaring these dependencies. When encountering the sync point, the GPU process pauses the submission of the browser commands to the GPU. However, it continues to execute the web app's commands for WebGL. When these commands are fully executed and the WebGL texture is ready, the sync point is triggered and the GPU process resumes the execution of paused browser's compositing commands.

In Sugar, however, the commands for the web app's WebGL textures are executed within the web app itself, and hence the GPU process is not normally informed of their completion, causing the dependent commands to be paused indefinitely. We solve this problem by sending an IPC with the right sync point information to the GPU process from the web app when the WebGL texture is rendered and posted to the vGPU's display.

2.5 Implementation and Prototype

Our implementation has the following components: the Intel vGPU driver library, Intel GPU virtualization layer, Mesa (open source OpenGL Implementation) and DRM libraries, and the Chromium browser. We build both Intel vGPU driver and the GPU virtualization layer on top of the Intel driver with virtualization support, i.e., Intel GVT-g (2016-Q3 release of KVMGT [39]), which uses Linux kernel version 4.3.0. We build our libraries on top of Mesa version 12.0.6 and DRM version 2.4.70. Finally, we add support for Sugar to Chromium version 58.0.3023.0.

We added support to Intel GPU virtualization for attaching a vGPU to a process, as discussed in §2.3.1. This requires us to trap and emulate vGPU driver's accesses to vGPU registers and some protected memory regions. We use existing KVM's x86 instruction decoder to decode the trapped accesses.

We use an Intel Core i7-5775C processor in our prototype, which comes with an Iris Pro Graphics 6200 integrated GPU. While we have tested Sugar only on this GPU, we anticipate it to easily support other Intel GPUs with virtualization support as well since our vGPU driver library (§2.3) is derived from the existing vGPU driver, which support all Intel virtualizable GPUs.

Our prototype uses a desktop with the aforementioned CPU, 16 GB of memory, 500 GB of SSD, a 27" display, and an ASRock Z97 Extreme4 motherboard. For dual-GPU Sugar, we use a Radeon R9 290 discrete GPU as well. Based on our experience, it is important that the second discrete GPU is powerful enough to perform the compositing load needed for running WebGL at a high framerate. Even if the web app does not use this GPU for rendering, still the rest of the system uses it for the primary graphics plane. In an initial prototype, we used a weak Radeon HD 6450 GPU, which could not keep up with the compositing load, resulting in an overall slowdown. Indeed, in most dual-GPU systems, the second GPU is a

powerful one compared to the Intel integrated GPU.

EGL vs. GLX. On a Linux machine, Chromium by default uses the GLX framework for interfacing between OpenGL and the X window system. However, the Intel vGPU framebuffer read-back implementation, which we have used in the GPU processor, is based on the EGL framework. Therefore, we reconfigure Chromium to use EGL, which achieves similar performance to GLX. However, EGL sometimes causes some visual choppiness at high framerates. We plan to add GLX support to Sugar in the future to replace EGL.

GPU and display configurations. For single-GPU Sugar, we connect the display to the VGA port of the Intel GPU. For dual-GPU Sugar, we first update the BIOS settings to set the Radeon GPU as the primary GPU and enable the "iGPU Multi-Monitor" option to activate both GPUs. We then connect the display to the VGA port of the Radeon GPU.

Chromium build options. We follow Google's guidelines to build the "Release" version of Chromium, both for our baseline and Sugar experiments [61]. However, for enhanced WebGL performance for both, we turn off the "`dcheck_always_on`" option, which performs runtime assertions.

2.6 Evaluation

We evaluate the security and performance of Sugar.

2.6.1 Security

TCB analysis. In the current implementation of WebGL, the TCB exposed to the web app is large. It includes the code in the browser's GPU process, the graphics libraries (including OpenGL and DRM libraries in Linux), and the GPU device driver in the kernel. Table 2.2

presents the size of these components. It shows that the size of the TCB is about 738,000 LoC. In contrast, the TCB of Sugar is about 34,400 including 28,300 LoC for Intel GPU virtualization (mostly a GPU emulation layer), 1,500 LoC for Sugar for attaching a vGPU to a process as discussed in §2.3.1), and 4,600 LoC for the KVM x86 instruction decoder (this number includes the KVM code for instruction emulation too, which we do not use). Moreover, a full GPU virtualization has the potential to further reduce the size of TCB in Sugar by eliminating the GPU emulation layer.

Failure domain analysis. We analyze how effectively Sugar protects the system against the exploit of WebGL vulnerabilities. As shown in Table 2.1, we study 20 WebGL vulnerabilities and determine (either experimentally or by analysis) whether they are solved by Sugar or not. We determine that single-GPU Sugar manages to overcome 17 of these vulnerabilities and dual-GPU Sugar overcomes 19.

Sugar protects against most of these vulnerabilities because of the following reasons. First, it sandboxes all the vulnerable code in the web app process. Second, it isolates the GPU memory accessible to the web app as a result of GPU memory virtualization.

The additional vulnerabilities that dual-GPU Sugar overcomes (compared to single-GPU Sugar) is related to GPU hang problem (vulnerabilities #16 and #20). Single-GPU Sugar cannot protect against these vulnerabilities since, on Intel GPUs, a vGPU hang results in the same effect in the physical GPU. Moreover, neither single-GPU Sugar nor dual-GPU Sugar protects the system against vulnerability #8 in the table. This vulnerability leverages a timing side-channel, which is also successful in Sugar, as we mentioned in our threat model (§2.2.1).

Three of the vulnerabilities in the table require additional explanation. Vulnerability #12 (marked as Conditional (BA/C) in the table) leaks the system user-name and the browser's executable file system path due to a bug in the shader compiler in the GPU process. Sugar

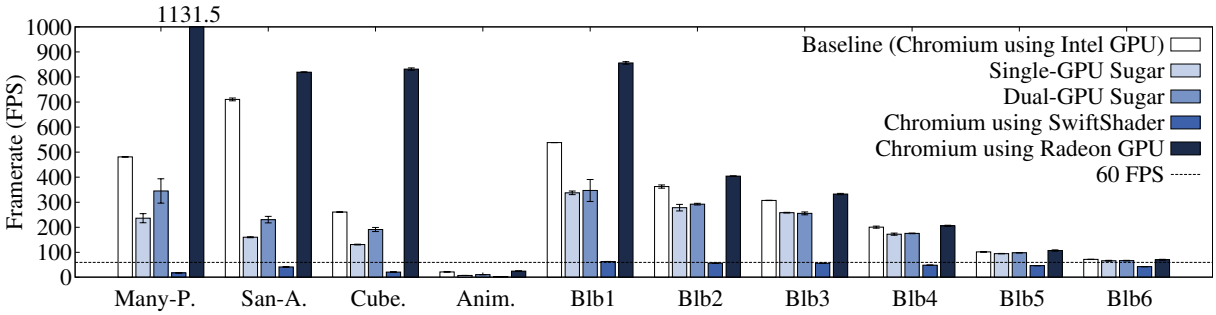
moves the shader compiler to the web app process since the compiler is part of the OpenGL library. This, on its own, does not solve the problem. However, it can be effective along with proper sandboxing of the web app process and preventing its access to such system info.

Vulnerabilities #16 and #20 that hang the GPU will result in the GPU device driver triggering a Timeout Detection and Recovery (TDR) operation, which resets the GPU hardware. Unfortunately, TDR has been shown to be often error-prone resulting in either a kernel panic or visual side effects [31]. While dual-GPU Sugar prevents these vulnerabilities from freezing the UI, it does trigger the TDR for the hung GPU, and hence can suffer from the bugs in TDR. We plan to address this problem in two ways in the future in dual-GPU Sugar. First, after a hang, the system can simply refuse to reset this GPU. It can continue to use the primary graphics plane but cannot use Sugar anymore until a full system reboot. Second, we are considering to move the TDR operation to the user space, which will at least eliminate the possible kernel panics.

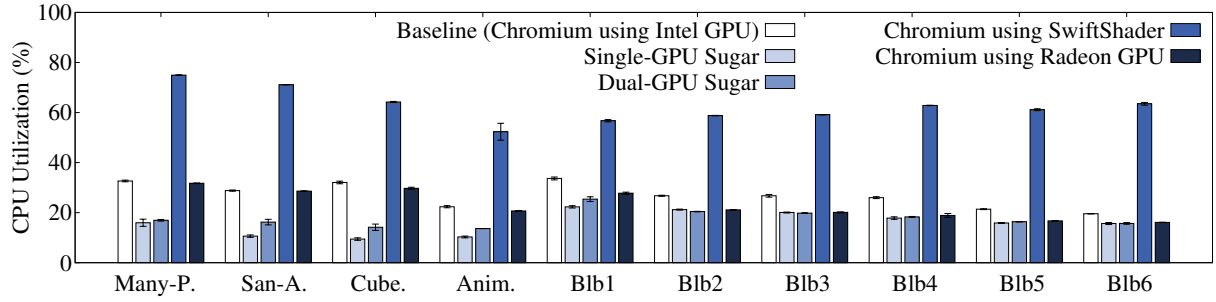
2.6.2 Performance

We evaluate the performance of Sugar with five benchmarks: Blob [66], Many-Planets [68], San-Angeles [69], Cubemap [67], and Animometer [65]. Note that the Blob benchmark can be configured with varying number of blobs and resolutions. Unless otherwise stated, we use the default settings.

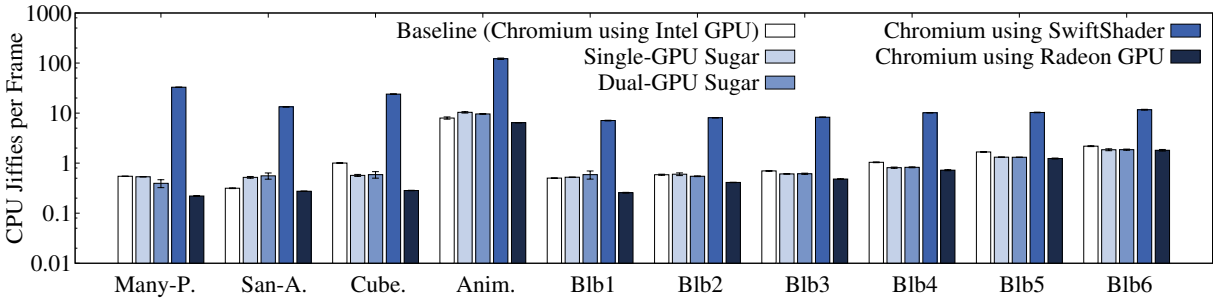
We configure the system as follows for the experiments. First, we use a default memory size for our vGPUs in all experiments. We determine the default memory size in Experiment 2 by comparing the performance of various configurations. Our default setting uses 64 MB of CPU-visible GPU memory (i.e., aperture) and 128 MB of CPU-non-visible GPU memory. Second, we use the EGL framework for Sugar (§2.5) but use GLX for the baseline experiments since GLX is used by default in Chromium. Third, as mentioned in §2.5, we use Mesa version



(a)



(b)



(c)

Figure 2.5: (a) Benchmarks' performance. (b) System's CPU utilization while executing the benchmarks. (c) Benchmarks' raw CPU usage.

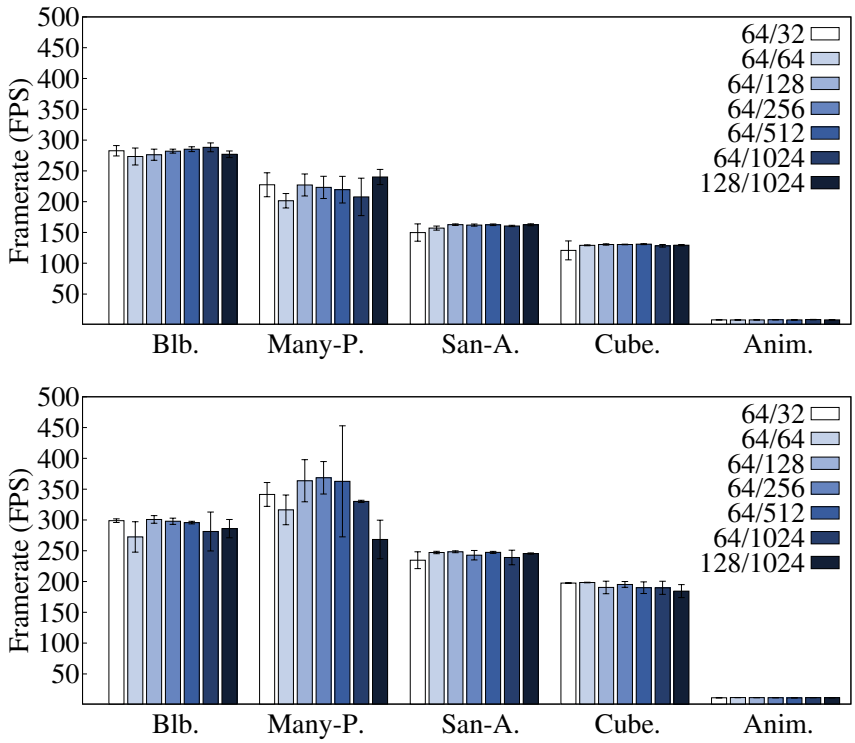


Figure 2.6: Effect of varying vGPU memory sizes for (Left) single-GPU Sugar and (Right) dual-GPU Sugar.

12.0.6 for Sugar. However, this version of Mesa does not properly support the high-end Radeon R9 290 GPU in our prototype [41]. Therefore, we use Mesa version 17.0.7 for the Radeon GPU in dual-GPU Sugar and in Radeon performance experiments since this version of Mesa has a fix for the aforementioned problem [36]. Moreover, for better comparison of single-GPU Sugar with dual-GPU Sugar, we use Mesa version 17.0.7 for the GPU process in Sugar (since in dual-GPU Sugar, while the web app process uses our Mesa library, the GPU process uses the Mesa library 17.0.7 needed for the Radeon-based primary graphics plane). And, for baseline experiments on the Intel GPU, we use Mesa version 12.0.6 for better comparison with Sugar (in which web apps use our Mesa library based on version 12.0.6).

Experiment 1: Sugar's performance. In the first set of experiments, we compare the performance of single-GPU Sugar and dual-GPU Sugar with the baseline Chromium

(running on the Intel GPU). Figure 2.5 (a) shows the results for our benchmarks. In this experiment, we use 5 other settings for the Blob benchmark other than the default ones. These settings include (1, 16ˆ3) for Blob1, (10, 24ˆ3) for Blob2 (the default), (10, 32ˆ3) for Blob3, (100, 32ˆ3) for Blob4, (1000, 40ˆ3) for Blob5, and (1000, 48ˆ3) for Blob6, where the two parameters determine the number of blobs and resolution, respectively.

Our results show that Sugar achieves high graphics performance. More specifically, we observe the following: First, for benchmarks that achieve a framerate higher than 60 (which is equal to the display refresh rate at 60 Hz), Sugar also achieves a performance higher than 60. Given that in practice, the browser caps the WebGL framerate to 60 (to synchronize with the display refresh rate), Sugar matches the baseline performance in this case (i.e., both baseline and Sugar will achieve 60 FPS in practice). Second, for benchmarks that have performance below or close to 60 FPS, Sugar achieves competitive performance as the baseline. As a result, our experiments show that Sugar will provide a similar user experience.

Our experiments also show that at max framerate, Sugar achieves noticeably lower performance than the baseline running on the Intel GPU. We believe that a large part of performance loss in Sugar is due to the overhead of GPU virtualization, as also reported in [243]. Therefore, a GPU virtualization solution with higher performance can further improve Sugar's performance. Other reasons behind Sugar's performance loss are (1) our use of operating system signals for interrupt delivery and (2) additional usage of GPU in the web app process to post the WebGL texture to its virtual display (§2.2). While the former can be eliminated by a faster interrupt delivery mechanism to the user space driver, the latter is fundamental to the design of Sugar. Hence, we measure this overhead for the default Blob benchmark and expect the same result for other benchmarks since the post operation requires filling up the same-sized virtual display in all benchmarks. Our experiments show that the time taken to complete the WebGL texture post operation is about 0.23 ms. To put this number in perspective, imagine a benchmark that achieves about 300 FPS on Sugar.

The frame time for this benchmark is about 3.3 ms. In this case, the WebGL texture post operation takes up 7% of the frame time. We believe that this is a small overhead.

Figure 2.5 (a) also shows the performance of running the same benchmarks on the Radeon GPU in our dual-GPU prototype as well. The Radeon GPU is a more powerful GPU than the Intel one and hence can achieve noticeably higher performance. Sugar is, however, bounded by the performance of the Intel GPU, which provides the vGPUs.

Figure 2.5 (b) shows the system's CPU utilization for the same set of benchmarks. We measure the CPU utilization by calculating the percentage of time in which the CPU cores are not idle. The results show that Sugar does not incur significant CPU utilization, which would affect other running processes in the system. To compare the CPU usage of different WebGL solutions, Figure 2.5 (c) shows the raw CPU usage per frame. We measure the CPU usage by calculating the total units of CPU time (in jiffies) needed to render a frame. The figure shows that Sugar does incur almost same CPU usage as the baseline. This is because while Sugar does use more CPU instructions for vGPU emulation, it eliminates IPC communication and shared-memory data transfer between the web app process and the GPU process.

The same figure also shows the results for an software renderer, Chrome SwiftShader. As the figure shows, single-GPU Sugar beats the SwiftShaders's performance by an average of 375% (as high as 1216% for one benchmark) while incurring 74% less CPU utilization and 92% less raw CPU usage.

Experiment 2: vGPU memory size. We attempt to understand the effect of vGPU memory size on the performance of Sugar. A vGPU memory consists of memory accessed by the CPU (also referred to as aperture) and memory not accessed directly by CPU. For Intel Iris Pro 6200 GPU used in our prototype, the overall size of these memory types are 250 MB and 3.75 GB, respectively. The system reserves 96 MB and 384 MB of these two

memory types, respectively, for the primary graphics plane. Moreover, the aperture size of a vGPU cannot be smaller than 64 MB on Linux according to Intel GPU virtualization guidelines [40]. Based on these constraints, we test the following configurations (represented as A/B where A and B refer to the aperture size and the size of CPU-non-visible GPU memory in MB): 64/32, 64/64, 64/128, 64/256, 64/512, 64/1024, and 128/1024.

Figure 2.6 shows the results. We observe the following. First, the small memory sizes of 32/64 and 64/64 result in a drop in performance. We believe that this is due to higher memory contention. Second, the large memory size of 128/1024 also results in a drop in performance. We believe that this is due to the overhead of memory pinning, which affects the overall performance of the system, including the browser. Based on these results, we choose 64/128 as the default memory size configuration for the rest of the experiments in this chapter. This is the configuration with the smallest amount of memory that shows no drop in performance. While we believe that this configuration is a good default one, we admit that different benchmarks might benefit from other configurations. It is, therefore, possible to extend Sugar's design to dynamically test and choose the right configuration for the current benchmark.

Experiment 3: supporting multiple web apps. We measure the scalability of Sugar when supporting multiple web apps. To do this, we run up to 3 web apps concurrently, each running the default Blob benchmark in a separate Chromium session and each occupying an equal portion of the screen. Given the vGPU memory size constraints mentioned earlier, we cannot run more than 3 vGPUs at a time. Moreover, when using three vGPUs, we reduce the GPU aperture size allocated for the system to 32 MB in order to free up enough aperture for the vGPUs.

Figure 2.7 shows the results. It shows that single-GPU Sugar sees a significant drop in performance when supporting more than one web app. Baseline and dual-GPU Sugar, on the other hand, see a more moderate drop. The more significant drop in single-GPU Sugar vs.

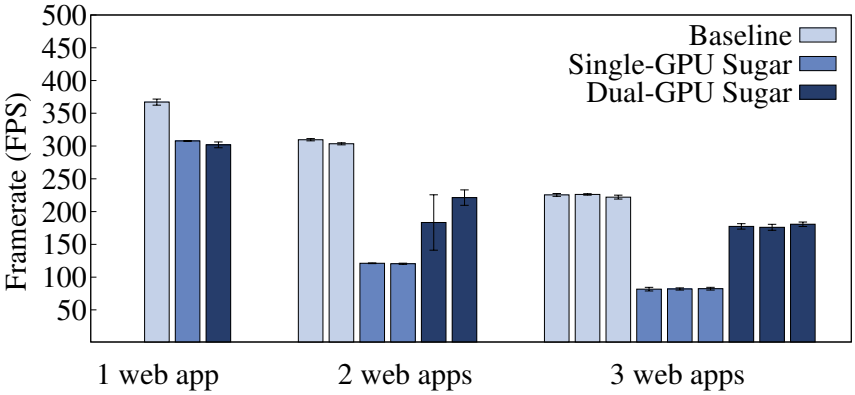


Figure 2.7: Supporting multiple web apps simultaneously.

dual-GPU Sugar is due to additional load on the primary graphics plane for compositing, which is sharing the same GPU with web apps. Moreover, the more significant drop in single-GPU Sugar vs. the baseline is due to the overhead of virtualization to the GPU device driver.

Experiment 4: performance isolation. We evaluate the effectiveness of dual-GPU Sugar in isolating the performance of a web app from the rest of the system. To do this, we run a native OpenGL benchmark (Unity WaveShooter [64]) at the same time as one of our WebGL benchmarks (Blob), each occupying almost half of the screen. Figure 2.8 (Left) shows the WebGL benchmark performance in this setup and Figure 2.8 (Right) shows the OpenGL benchmark performance. Each figure shows the results for baseline, single-GPU Sugar, and dual-GPU Sugar. Moreover, each figure shows the performance of the benchmark while running with or without the other benchmark. For the latter cases, we run the benchmark in half of the screen while the other half is empty. The figures show that the performance drop both in the WebGL and OpenGL benchmarks is smallest in dual-GPU Sugar. As previously mentioned, this is because the native app in Sugar runs on the secondary GPU while the web app uses an Intel vGPU. However, we see some drop even with dual-GPU Sugar. This is because the OpenGL benchmark competes with the browser's GPU process for access to the second GPU.

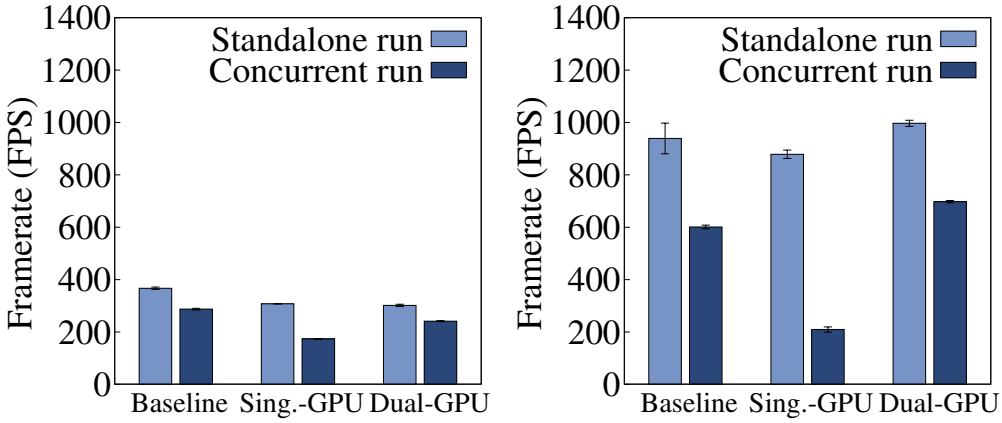


Figure 2.8: Performance isolation. (Left) A WebGL benchmark running standalone or concurrent with an OpenGL benchmark. (Right) An OpenGL benchmark running standalone or concurrent with a WebGL benchmark.

2.7 Limitations

Multiltiple WebGL textures in one web app. Sugar currently supports only web apps with a single WebGL texture simply because it uses the whole framebuffer of the virtual display for that texture. We plan to remove this limitation by having multiple WebGL textures share this framebuffer.

Tearing effect. Sugar suffers from some tearing effect at high framerate, where the displayed frame contains content from consecutively rendered frames. This is because the virtual display readback in the GPU process overlaps with consecutive posting of WebGL textures to the virtual display. We plan to solve this problem by using multiple framebuffers for the virtual display (similar to how multiple render buffers solves the tearing problem in existing graphics framework).

Other GPU Virtualization Solutions. Sugar can leverage any GPU virtualization solution and its performance and security trade-off will be determined by that of the solution. We chose Intel GPU virtualization due to its availability on almost all desktops and laptops. Virtualizable GPUs from NVIDIA [23] and AMD [37] provide better performance

and isolation (e.g., by using SR-IOV), but are mostly tailored for servers and hence much less commonly available on personal computers. Supporting these GPUs requires non-trivial engineering effort to port their drivers to user space.

Native apps. Sugar can be used to provide secure GPU acceleration for untrusted native apps too. Doing this requires modifying the operating system window manager so that it retrieves the rendered texture of the app from the vGPU's display framebuffer, similar to how the GPU process in the browser retrieves the web app's WebGL texture.

Vulnerability type	#	Vulnerability description	Official vulnerability report		Effect	Target platform (Browser:OS:GPU)	Reproduced by us		Solved by Sugar	
			Vendor (number, severity)	NVD (number, severity)			On current version	After removing patch	One-GPU	Dual-GPU
Integrity	1	Use-after-free [28, 26]	1028891, Crit.	CVE-2014-1556, High	Browser crash; execute arbitrary code	FF	✗	✗	✓(BA)	✓(BA)
	2	Write-after-free [17, 20]	149904, High	CVE-2012-5115, High	GPU process crash; unspecified impact	CHR:Mac	✗	N/A (CS)	✓(BA)	✓(BA)
	3	Memory allocation [34, 33]	1190526, Crit.	CVE-2015-7179, High	Browser crash; execute arbitrary code	FF:Win	✗	✗	✓(BA)	✓(BA)
	4	Integer overflow (for texture dimension) [16, 19]	145544, High	CVE-2012-2896, High	GPU process crash; unspecified impact	CHR:Lin	✗	✗	✓(BA)	✓(BA)
						CHR:Mac	✗	✗		
Confidentiality	5	Memory access control [11, 5]	656752, Crit.	CVE-2011-2367, Med.	Read of GPU memory	FF	✗	✓	✓(BA)	✓(BA)
	6	Uninitialized memory [12, 15]	659349, High	N/A	Read of GPU memory	FF	✗	✗	✓(BA)	✓(BA)
	7	Read unauthorized memory [13, 9]	684882, High	CVE-2011-3653, Med.	Read of GPU memory	FF:Mac: Intel	N/A (PNA)	N/A (PNA)	✓(BA)	✓(BA)
	8	Timing attack [10, 4, 6]	655987, High	CVE-2011-2366, Med.	Read of cross-domain image	FF	✗	✓	✗(BA)	✗(BA)
			N/A	CVE-2011-2599, Med.		CHR	✗	✗		
	9	Read-after-free [51, 52]	682020, Unsp.	CVE-2017-5031, Med.	Read of Browser GPU process memory	CHR:Win	✗	✗	✓(BA)	✓(BA)
	10	Uninitialized memory [24, 27]	376951, Med.	CVE-2014-3173, Med.	Potential read of other graphics buffers	CHR	✗	✓	✓(BA)	✓(BA)
	11	Memory access control [21, 22]	237611, Med.	CVE-2013-2874, Med.	Read of Browser's UI content	CHR:Win: NVIDIA	N/A (PNA)	N/A (PNA)	✓(BA)	✓(BA)
	12	Information leak [3, 8]	83841, Low	CVE-2011-2784, Med.	Reveal OS user-name and browser filesystem path	CHR:Win	✗	N/A (NAP)	✓ (BA/C)	✓ (BA/C)
	13	Unauthorized access [29, 25]	972622, Mod.	CVE-2014-1502, Med.	Using other WebGL contexts, e.g., reading their buffers	FF	✗	✗	✓(BA)	✓(BA)
	14	Uninitialized memory [32]	521588, Low	N/A	Reveal previous webpages UI	CHR	✗	✗	✓(BA)	✓(BA)
Availability	15	Out of GPU memory [18]	153469, High	N/A	Kernel Panic	CHR:Mac: NVIDIA	✗	N/A (CS)	✓(BA)	✓(BA)
	16	GPU hang [31]	483877, Unsp.	N/A	UI freeze; GPU TDR; kernel panic (platform dependent)	All	✓	N/A (NF)	✗(BA)	✓(BA)
	17	Compiler compute overflow [38]	593680, Unsp.	N/A	Browser hang	CHR:Lin	✓	N/A (NF)	✓	✓
	18	Invalid input (to shader compiler) [2]	70718, Med.	N/A	GPU process crash	CHR:Lin	✗	✗	✓(BA)	✓(BA)
	19	Invalid pointer deref. [1]	63617, Low	N/A	Window manager (X) crash	CHR:Lin	✗	N/A (old OS)	✓(BA)	✓(BA)
	20	GPU hang [7]	N/A	CVE-2011-2601, High	UI freeze; GPU TDR	Mac	✓	N/A (NF)	✗(BA)	✓(BA)

Table 2.1: *WebGL vulnerabilities. Abbreviations and short forms used in the table: Crit. = Critical, Med. = Medium, Mod. = Moderate, Unsp. = Unspecified, TDR = Timeout Detection and Recovery, CHR = Chrome, FF = Firefox, Lin = Linux, Mac = macOS, Win = Windows, CS = Closed Source, NF = Not Fixed yet, NAP = No Access to Patch, PNA = Platform Not Available to us, BA = By Analysis, BA/C = By Analysis/Conditional. In the 7th column, when a platform component is not specified, it means that the vulnerability applies to all types of that component. For example, CHR alone means the vulnerability applies to Chrome on all operating systems and GPUs.*

System	TCB (LoC)	Component	LoC
Baseline WebGL	738,000	- GPU device driver	123,000
		- Mesa library	441,000
		- DRM libraries	16,000
		- Chromium GPU process	158,000
Sugar	34,400	- Intel GPU virtualization	28,300
		- Sugar's code to attach a vGPU to a process	1,500
		- KVM x86 instruction decoder	4,600

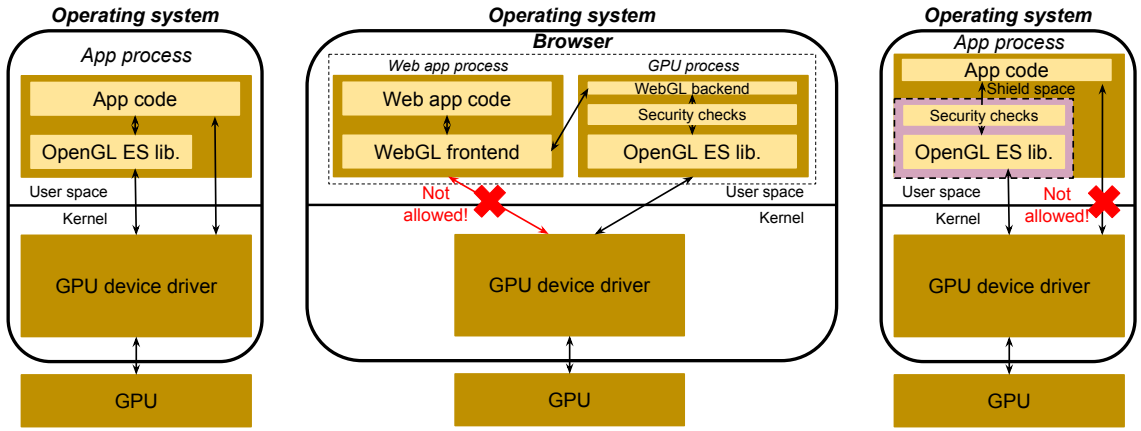
Table 2.2: WebGL TCB Analysis (assuming an Intel GPU). For Mesa and DRM libraries, before counting, we manually eliminate parts of the source trees specific to platforms and GPUs other than Linux and Intel GPU.

Chapter 3

Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks

Mobile GPUs have reached performance that rivals that of dedicated gaming machines. Many mobile applications (apps) such as games, 3D apps, Artificial Reality (AR) apps, and apps with high fidelity user interfaces (UI) leverage these high-performance GPUs. Mobile GPUs are typically accessed through the OpenGL ES API, which is a subset of the infamous OpenGL API and is designed for embedded systems.

Unfortunately, allowing untrusted apps to use the GPU has resulted in serious security issues. The GPU device driver in the operating system kernel is large (e.g., 32,000 lines of code for the Qualcomm Adreno device driver) and potentially vulnerable. Yet, to enable OpenGL ES, the operating system exposes the GPU device driver interface to unprivileged apps. This enables malicious mobile apps to issue requests directly to the device driver in the kernel, triggering deep vulnerabilities that can result in a full system compromise.



(a) (b) (c)

Figure 3.1: (a) Graphics stack in a mobile operating system. (b) WebGL stack in a web browser. (c) Graphics stack in Milkomeda.

Historically, apps that require GPU acceleration have been benign. On desktops, these apps include popular games, accelerated video decoders, parallel computational workloads, and crypto currency mining. Such apps are typically developed by well-known entities and are therefore trusted. On mobile devices, apps are untrusted and potentially malicious. Mobile apps run in a sandbox (i.e., the operating system process as well as the Java virtual machine) and are isolated from the rest of the system. Yet, direct access to the GPU device driver exposes a large unvetted attack surface to malicious apps. Unfortunately, this direct access seems unavoidable since it allows the app to get the best possible performance from the GPU. This has left the system designers with no choice but to sacrifice security for performance.

Another platform has faced a similar problem: web browser. WebGL exposes GPU acceleration to untrusted web apps written in JavaScript running in the browser. To mitigate the security threat, browsers perform various runtime security checks and keep state across WebGL calls. The WebGL API is mostly based on the OpenGL ES API and hence WebGL checks are designed based on the OpenGL ES specification [70] as well as newly reported vulnerabilities and exploits. Only calls with valid arguments (considering the current GPU state) are allowed, effectively whitelisting safe API interactions. Such an interposition layer greatly reduces the attack surface and restricts API calls to well-defined state transitions.

Browser vendors have invested significant resources into the development of security checks for WebGL. We introduce Milkomeda, a system that allows us to repurpose these security check for mobile apps. Milkomeda immediately safeguards the mobile graphics interface without reinventing the wheel.

We solve two important challenges in Milkomeda: minimizing porting effort and maintaining high graphics performance. First, trying to manually extract WebGL security checks from the browser's source code and package them for the mobile graphics stack is challenging and time-consuming, a lesson that we soon learned in the initial stages of this work. Milkomeda addresses this challenge with a tool, called CheckGen, that automatically extracts and packages WebGL security checks for the mobile graphics stack, making small interface modifications to the original code for resolving interface incompatibilities.

Second, maintaining high graphics performance for mobile apps is challenging. To protect the integrity of WebGL security checks, web browsers use a multi-process architecture. In this architecture, a web app process cannot directly invoke the GPU device driver needed for WebGL; it must instead communicate with a "GPU process" for WebGL calls. Hence, this architecture requires Inter-Process Communication (IPC) as well as shared memory data copying, which incur significant performance overhead. While such an overhead might be acceptable for web apps, it is intolerable for mobile apps, which demand high graphics performance. Milkomeda addresses this issue with a novel in-process shield space design, which enables the evaluation of the security checks in the app's process while protecting their integrity. The shield space allows to securely isolate the code and data of the graphics libraries as well as the security checks within an untrusted process. It provides three important properties: (*i*) it only allows threads within the shield space to issue system calls directed at the GPU device driver in the kernel; (*ii*) it allows the application's untrusted threads to enter the shield only through a designated call gate so that security checks cannot be circumvented; and (*iii*) it protects the code and data within the shield space from being

tampered with. These properties, collectively, allow Milkomeda to ensure that the security checks automatically ported from WebGL can efficiently vet graphics API calls within a mobile app.

We implement Milkomeda for Android and use the Chrome browser WebGL security checks in it. Our implementation is geared for ARMv8 processors, used in modern mobile devices. We evaluate Milkomeda on a Nexus 5X smartphone. We show that (*i*) for several benchmarks with a framerate of 60 Frames Per Second (FPS), which is the display refresh rate, Milkomeda achieves the same framerate, (*ii*) for a benchmark with lower FPS, Milkomeda achieves close-to-native performance, and (*iii*) Milkomeda incurs additional CPU utilization (from 15% for native execution to 26%, on average). Moreover, we show that the multi-process architecture increases the execution time of OpenGL ES calls by an average of 440% compared to Milkomeda, demonstrating the efficiency of Milkomeda in providing isolation.

We make the following contributions in this work.

- We demonstrate the feasibility of using a web browser's WebGL security checks to guard the mobile operating system graphics interface.
- We present a solution for extracting these checks from the browser and packaging them for mobile apps with minimal engineering effort.
- We provide a system solution for securely evaluating these checks in the app's own process in order to achieve high graphics performance.

Vulnerability Type	Examples
Privilege Escalation	CVE-2014-0972(Q), CVE-2016-2067(Q), CVE-2016-2468(Q), CVE-2016-2503(Q), CVE-2016-2504(Q), CVE-2016-3842(Q), CVE-2016-6730(N), CVE-2016-6731(N), CVE-2016-6732(N), CVE-2016-6733(N), CVE-2016-6734(N), CVE-2016-6735(N), CVE-2016-6736(N), CVE-2016-6775(N), CVE-2016-6776(N), CVE-2016-6777(N), CVE-2016-8424(N), CVE-2016-8425(N), CVE-2016-8426(N), CVE-2016-8427(N), CVE-2016-8428(N), CVE-2016-8429(N), CVE-2016-8430(N), CVE-2016-8431(N), CVE-2016-8432(N), CVE-2016-8434(Q), CVE-2016-8435(N), CVE-2016-8449(N), CVE-2016-8479(Q), CVE-2016-8482(N), CVE-2017-0306(N), CVE-2017-0307(N), CVE-2017-0333(N), CVE-2017-0335(N), CVE-2017-0337(N), CVE-2017-0338(N), CVE-2017-0428(N), CVE-2017-0429(N), CVE-2017-0500(M), CVE-2017-0501(M), CVE-2017-0502(M), CVE-2017-0503(M), CVE-2017-0504(M), CVE-2017-0505(M), CVE-2017-0506(M), CVE-2017-0741(M), CVE-2017-6264(N)
Unauthorized Memory Access	CVE-2016-3906(Q), CVE-2016-3907(Q), CVE-2016-6677(N), CVE-2016-6698(Q), CVE-2016-6746(N), CVE-2016-6748(Q), CVE-2016-6749(Q), CVE-2016-6750(Q), CVE-2016-6751(Q), CVE-2016-6752(Q), CVE-2017-0334(N), CVE-2017-0336(N), CVE-2017-14891(Q)
Memory Corruption	CVE-2016-2062(Q), CVE-2017-11092(Q), CVE-2017-15829(Q)
Denial of Service	CVE-2012-4222(Q)

Table 3.1: List of CVEs for Android GPU driver vulnerabilities in NVD. The letter in the parenthesis shows the GPU driver containing the vulnerability. Q, M, and N stand for Qualcomm, MediaTek, and NVIDIA GPU device drivers, respectively.

3.1 Background and Motivation

3.1.1 Current Graphics Stack in Mobile Devices

To leverage GPUs for graphics acceleration, mobile apps use the OpenGL for Embedded System (OpenGL ES) API, which is a subset of the OpenGL API targeted for embedded systems. The OpenGL ES library on a mobile device is provided by the GPU vendor and handles the standardized OpenGL ES API calls of the application. In doing so, it interacts with the GPU device driver in the operating system kernel by issuing system calls (syscalls for short). In Android, which is the focus of this chapter, this is done by issuing syscalls on a device file (e.g., `/dev/kgsl-3d0` for the Adreno GPU in a Nexus 5X smartphone). More specifically, this is done by opening the GPU device file and then issuing syscalls, e.g., `ioctl` and `mmap`, on the returned file descriptor. Figure 3.1a shows this architecture.

There are two reasons why this architecture is prone to attacks by malicious apps. First,

while well-behaved apps only use the OpenGL ES library to (indirectly) communicate with the GPU device driver, nothing stops the app from interacting with the GPU device driver in the kernel directly (as shown in Figure 3.1a). This is because the operating system gives the mobile app process permission to access the GPU device file to enable the OpenGL ES framework within the app process. Therefore, any code within the process can simply invoke the device driver in the kernel. This exposes a huge and easy-to-exploit attack surface to the app. For example, the `ioctl` syscall enables about 40 different functions for the Qualcomm Adreno GPU device driver, which is about 32,000 lines of kernel code in Nexus 5X's LineageOS Android source tree (v14.1) and has many vulnerabilities (Table 3.1).

Second, even indirect communication with the GPU driver through the OpenGL ES API is unsafe since this API is not designed with security in mind. Several attacks against a related interface, WebGL API (which is very similar to the OpenGL ES API – see §3.5.2), have been demonstrated [264]. Indeed, these attacks using the WebGL interface inspired many security checks in web browsers, which vet arguments of WebGL calls. These checks have, over time, hardened the WebGL interface. However, mobile apps lack such a checking framework for the OpenGL ES interface. Here we show that we can repurpose the security checks in WebGL for mobile apps.

3.1.2 Mobile Graphics Vulnerabilities

Reported vulnerabilities. We study Android GPU vulnerabilities by searching the National Vulnerability Database (NVD) [57] (note that we lack direct access to the bug trackers of Android and GPU vendors). We search for Android GPU driver vulnerabilities in NVD using the "Android" and "GPU" keywords. Table 3.1 shows the full list of CVEs we found. Overall, we found 64 CVEs, out of which 47 CVEs are privilege escalations, 13 are unauthorized memory accesses, 3 are memory corruptions, and one is a Denial of Service

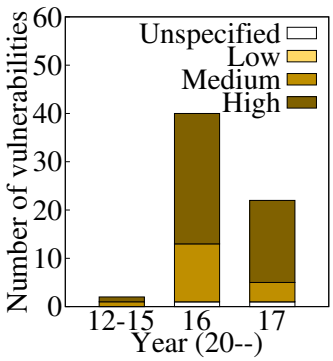


Figure 3.2: Severity and year of Android GPU vulnerabilities in NVD. The legend captures the severity according to CVSSv2.

(DoS).

Figure 3.2 shows the year and severity of these CVEs. There are two important observations. First, 73% of the reported vulnerabilities have the maximum severity level. The severity levels in the figure show NVD's score based on the Common Vulnerability Scoring System Version 2 (CVSSv2) [62]. The high severity of these vulnerabilities is because the GPU driver runs in kernel mode and is directly accessible by unprivileged apps. Second, the majority of these vulnerabilities are recent, i.e., reported in 2016 and 2017. This large number of mostly critical and new vulnerabilities show the pressing need to protect the interaction between unprivileged apps and the GPU driver.

Reproducing the vulnerabilities. We reproduce 3 of the aforementioned vulnerabilities by writing Proof-of-Concept (PoC) exploits to trigger them from an unprivileged Android app. We write the PoCs in C++ and integrate them in an Android application using the Android Native Development Kit (NDK) [72]. The three vulnerabilities are CVE-2016-2503, CVE-2016-2504, and CVE-2016-2468. Our PoCs trigger the reported vulnerabilities and force a kernel panic.

3.1.3 Graphics Stack in Web Browsers

To provide enhanced graphics functionality for web apps, web browsers introduced a framework called *WebGL*. WebGL provides an OpenGL ES-like API for web apps, enabling them to render high-performance 3D content using GPUs. Yet, browser vendors have been mindful of the security vulnerabilities in GPU device drivers. These vulnerabilities have been a great concern to them since web apps are completely untrusted and can be launched with a single click on a URL. As a result, supporting WebGL seemed like a significant security risk in the beginning, causing a large amount of discussions. For example, Microsoft first announced that WebGL is harmful and "is not a technology Microsoft can endorse from a security perspective" [14].

WebGL security solution. To mitigate these security concerns, the WebGL framework is equipped with a set of runtime security checks. Whenever a WebGL API is called by a web app, the parameters of the call are vetted before being passed to the underlying graphics library (which can be OpenGL ES, OpenGL, or Direct3D depending on the platform and operating system). These checks are mostly derived from the OpenGL ES specification [70], given that the WebGL API is similar to the OpenGL ES API (see §3.5.2 for incompatibilities). Moreover, when a new vulnerability or an exploit is discovered, a new check is added to prevent future exploits. For example, recently, a drive-by Rowhammer attack was demonstrated using the GPU through the WebGL API [148]. To mitigate it, Google and Mozilla both blocked a certain extension in WebGL [78]. While these security checks cannot protect against all unknown attacks (e.g., zero-day exploits), their accumulation over the past few years has greatly improved the state of WebGL security.

One might wonder whether existing checks in GPU device drivers are enough to guard them and whether WebGL security checks are redundant given the driver checks. Unfortunately, GPU device drivers do not include a comprehensive set of security checks and are vendor

specific. While some simple checks (such as checking for a null pointer) might exist, they are not systematically designed to properly vet the driver API calls. This is one important reason behind so many vulnerabilities in mobile GPU device drivers (Figure 3.2). On the other hand, WebGL checks have been comprehensively designed to protect against potentially malicious web apps.

However, deploying WebGL checks has an important cost for browsers: performance loss. This is mainly due to the architecture needed to protect integrity of check evaluation. More specifically, in order to control a web app's access to the GPU device driver, WebGL is deployed in a multi-process architecture [30, 44]. In this architecture, the web app cannot directly communicate with the GPU device driver as enforced by the operating system. Instead, it is only granted permission to communicate with the GPU driver through a proxy process, called the GPU process, which executes the WebGL API on behalf of the web app, albeit after security checking. The GPU process is a privileged process in the browser with access to the GPU device driver.

Figure 3.1b illustrates this architecture. The web app process uses a WebGL frontend framework, which uses Inter-Process Communication (IPC) and shared memory to serialize and pass the WebGL API calls of the web app to the WebGL backend in the GPU process. The backend performs the aforementioned security checks on these API calls, executes them if they pass the checks, and then returns the result to the web app process. This architecture degrades the performance of WebGL. This is because a WebGL call is now an IPC call rather than a function call and it requires serialization and deserialization of arguments. Moreover, the graphics data need to be copied to a shared memory segment by the web app.

3.1.4 WebGL Security Checks

In this subsection, we provide a high-level review of WebGL security checks based on available documents, e.g., [35], and our own study of Chromium browser source code. While our study focused on WebGL in Chromium, we believe that the provided review is valid for other browsers too. We group WebGL security checks into four categories.

Category I: checks on numeric values. WebGL validates numeric arguments passed as input to its APIs. For example, it checks for some arguments to be positive and rejects deprecated values. Some simple checks are hard-coded in the WebGL implementation using conditional statements. The rest are handled by *Validators*, which are automatically generated with a python script from a checklist manually derived from the OpenGL ES specification.

Category II: checks on correctness of API calls. WebGL (built on top of the OpenGL ES) is highly stateful. That is, some WebGL calls update the "rendering state". At any rendering state, only some WebGL API and arguments are valid according to the OpenGL ES specification. WebGL performs checks to enforce correct API usage. It records the API calls and uses them to infer the rendering state. It then uses this state to validate subsequent API calls. As an example, a call for a graphics operation on a graphics object is only valid if that object (identified by an integer handle) has already been created in a previous call. Therefore, upon handling such calls, WebGL first checks the existence of the corresponding graphics object.

Category III: checks on the shader code. Hardware acceleration using GPUs is primarily done through "shaders", which are submitted to the GPU for execution. The WebGL implementation translates the shader source code to the format used on the platform and validates it. For example, it does not allow non-ASCII characters in the shader source code as it has been reported that such characters can crash some shader compilers [2]. The

translation and validation is done through the Almost Native Graphics Layer Engine (ANGLE) compatibility layer. Also, WebGL disables the *glShaderBinary* API, which submits a compiled shader binary to the GPU, since it bypasses shader validation.

Category IV: platform workarounds. Chromium maintains a list of known graphics bugs and their respective workarounds. Then at runtime, depending on the platform (e.g., GPU model), it applies the necessary workarounds. For our experiment platform (i.e., Nexus 5X smartphone with a Qualcomm Adreno GPU), there are 15 workarounds at the time of this writing. For example, due to a bug in the Adreno OpenGL ES library, the initialization of shader variables in a loop causes the shader compiler to crash [49]. Chromium avoids this problem by disallowing the use of loops to initialize shader variables.

Preventing TOCTTOU attacks. Many parameters passed to the WebGL API are pointers. To prevent Time of Check to Time of Use (TOCTTOU) attacks, the WebGL implementation makes a "shadow copy" of the sensitive data pointed by these pointers, then validates and uses the shadow copies. Only security-sensitive data is shadowed. Others, such as a texture data passed to the `glTextImage2D` API, are not shadowed as they can only affect the rendered content. This selective shadowing helps with performance as it minimizes the required data copying.

Case Study: glTexImage2D in WebGL. The glTexImage2D API specifies a two-dimensional texture image [79]. Figure 3.3 shows a simplified version (for readability) of the IPC handler function for glTexImage2D in WebGL in Chrome (`HandleTexImage2D`). This function first retrieves non-pointer arguments from the IPC data structure. It then enforces simple checks on the width and height parameters and uses safe arithmetic functions to validate the image data size. It then calls `ValidateAndDoTexImage` for more security checks. This function uses validators to check whether the target texture type, the command type, and image data parameters are allowed according to the OpenGL ES specification [79]. Then, it checks the target texture's ability to work with the dimension and level of the

```
error::Error HandleTexImage2D(void* ipc_data) {
  TexImage2D_args& c = *static_cast<TexImage2D_args*>(ipc_data);
  GLenum target = static_cast<GLenum>(c.target);
  /* Get all other parameters from ipc_data */
  ...

  /* Get shared memory ID for image data */
  uint32_t pixels_shm_id = static_cast<uint32_t>(c.pixels_shm_id);
  uint32_t pixels_shm_offset = static_cast<uint32_t>(c.pixels_shm_offset);
  ...

  if (width < 0 || height < 0) {
    LOCAL_SET_GL_ERROR(GL_INVALID_VALUE, func_name, "dimensions < 0");
    return error::kNoError;
  }

  /* Validate image data size */
  if (!GLES2Util::ComputeImageDataSizesES3( ... ) {
    return error::kOutOfBounds;
  }

  /* Get image data pointer from shared memory */
  const void* pixels;
  if (pixels_shm_id) {
    pixels = GetSharedMemoryAs<const void*>(
    pixels_shm_id, pixels_shm_offset, pixels_size);
    if (!pixels)
      return error::kOutOfBounds;
  } else {
    pixels = reinterpret_cast<const void*>(pixels_shm_offset);
  }

  ValidateAndDoTexImage( ... );
  return error::kNoError;
}


void ValidateAndDoTexImage( ... ) {
  if (((args.command_type == DoTexImageArguments::kTexImage2D) &&
       !validators->texture_target.IsValid(args.target)) || ... ) {
     return false;
  }
  ValidateTextureParameters( ... );
  ValidForTarget( ... );

  TextureRef* local_texture_ref = GetTextureInfoForTarget(state, args.target);
  if (!local_texture_ref) {
    return false;
  }

  /* Apply necessary platform workarounds */
  ...

  /* DoTexImage updates the bookkeeping info for the affected objects and eventually call glTexImage2D */
  DoTexImage(texture_state, state, framebuffer_state, function_name, texture_ref, args);
}
```

Figure 3.3: WebGL's (simplified) handling of the glTexImage2D API including several security checks.

image data. It then attempts to retrieve the target texture information, which is collected when handling previous calls to create and operate on the texture. If the target texture information does not exist, it returns an error. After the arguments are validated, the function looks for and applies necessary platform workarounds. It then calls `DoTexImage` to update the bookkeeping state for the affected objects. Finally, it calls the actual OpenGL ES API function: `glTexImage2D`.

3.2 Threat Model

We assume that mobile apps are untrusted and potentially malicious, similar to web apps. This is because many mobile apps are developed by untrusted developers. Moreover, an "instant app" [80] can be launched with a single URL click and without installation.

We assume that the attacker uses one such mobile app to attack the system. This malicious app has full control over the user space process it runs in (excluding the shield space). It can run both Java and native code. It does the latter by loading arbitrary native libraries and calling them through the Java Native Interface (JNI). We assume that this malicious app tries to exploit vulnerabilities in the GPU device driver. To do so, the app uses the GPU device driver syscall interface (e.g., `ioctl` and `mmap` syscalls) or the OpenGL ES API (which indirectly invokes the GPU device driver syscalls). We do not trust any libraries used directly by the app in its process, including system libraries. We do trust the kernel, which we also leverage to set up a trusted shield space in the process address space. We set up the shield at application load time and before loading the application's code. Therefore, we assume that the shield is set up correctly and hence can be trusted.

3.3 Milkomeda's Design

Milkomeda protects the GPU kernel device driver from malicious apps by disallowing direct access to the driver and routing all OpenGL ES calls through a vetting layer. We repurpose the security checks developed for the WebGL framework for this layer. Note that this is fundamentally feasible since WebGL API is based on OpenGL ES (in §3.5, we describe how we automate porting and overcome incompatibilities). The question now becomes: what is the right architecture that satisfies security and performance constraints for deploying these checks for mobile apps? We first discuss two straw-main solutions before presenting ours.

Straw-man design I. One straight-forward design is the multi-process architecture used in the browser. That is, we can deploy a special process and force the app to communicate to this process for OpenGL ES support. This process then performs the security checks adopted from the browser and invokes the GPU device driver. This design provides isolation between the app code and the security checks since they execute in different processes. Therefore, the web app cannot easily circumvent the checks, unless it manages to compromise this specialized process or the operating system.

Unfortunately, there is one major drawback for this design: degraded performance. The graphics performance in this design is lower than that of the existing graphics stack for mobile apps due to the overhead of (*i*) IPC calls and shared memory data copy, needed for communication between the two processes, and (*ii*) serialization and deserialization of the API calls' parameters.

Straw-man design II. Another potential design is to deploy the checks in the app process itself. That is, we can deploy the checks as a shim layer on top of the existing OpenGL ES library. When the app calls the OpenGL ES API, the API call is first evaluated through the shim before being passed to the underlying API handlers. While this design achieves high graphics performance (only degraded by the minor performance overhead of evaluating

the security checks), it suffers from an important problem: the checks are circumventable. First, the app can directly call the GPU device driver itself, bypassing the library altogether. Second, the app can load and use a different OpenGL ES library, which does not incorporate the security checks. Third, the app can bypass the security checks in the existing library by jumping past the checks but before the API handlers.

Required guarantees. Based on these straw-man solutions, we come up with a set of principled guarantees that a solution must provide including three security guarantees and one performance guarantee.

- **Security guarantee I:** Untrusted app code cannot directly interact with the GPU device driver. All interactions between the app and the driver are vetted by security checks.
- **Security guarantee II:** the control-flow integrity of the security checks is preserved.
- **Security guarantee III:** the data integrity of the security checks and their intermediate states is preserved.
- **Performance guarantee:** the security check framework does not cause significant performance degradation for mobile graphics.

Milkomeda's design. In Milkomeda, we present a design that provides these guarantees. Milkomeda achieves **security guarantee I** by restricting the communications between the app and the GPU driver through a vetting layer, which can then perform security checks on the OpenGL ES API calls before passing them to the underlying GPU device driver. It does so using a novel shield space in the app's address space for executing the security checks. The operating system kernel only allows the threads in the shield space to interact with the GPU device driver. In Milkomeda, we reuse WebGL's security checks as the vetting layer for mobile graphics. Milkomeda achieves **security guarantee II** by enforcing the app's normal

threads to enter the shield at a single designated entry point in order to issue an OpenGL ES API call. The call is then vetted by the aforementioned security checks and, if safe, is passed to the OpenGL ES library in the shield space. Therefore, the app cannot jump to arbitrary code locations in the graphics libraries. Milkomeda achieves **security guarantee III** by protecting the memory pages of the shield space from the rest of the app, even though the shield space is within the app process address space. All the graphics libraries and their dependencies are loaded in the shield space and their code and data are protected from tampering by the app. Finally, Milkomeda achieves the **performance guarantee** since the graphics libraries execute in the same address space as the app, hence eliminating the need for IPC, shared memory data copy, and serialization/deserialization of API arguments. We will show in §3.7.2 that Milkomeda achieves high graphics performance for various mobile apps, although at the cost of moderately increased CPU utilization. Figure 3.1c illustrates Milkomeda's design.

3.4 Shield Space

Milkomeda's shield space regulates an app's access to the GPU device driver and enforces the app to interact with the OpenGL ES library at a designated entry point. Figure 3.4 shows a simplified view of shield's design. We create a shield space within the normal operating system process. A thread executing normally (i.e., outside the shield space) cannot access the memory addresses reserved for the shield space. It cannot execute syscalls targeted at the GPU device driver either. To execute an OpenGL ES API, a thread needs to issue a *shield-call*, which transfers the execution to *a single designated call gate* within the shield space (allocated in the shield memory). This thread is now trusted and can access the shield memory and interact with the GPU device driver. It executes the API call (after vetting it) and then returns from the shield-call. The shield space can be thought of as a more

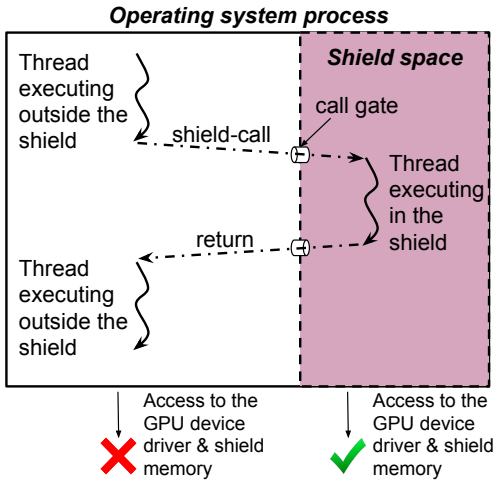


Figure 3.4: A simplified view of shield's design highlighting how a thread can use a shield-call to enter the shield space to interact with the GPU device driver.

privileged execution mode for the process, similar to existing privilege modes such as kernel or hypervisor.

Shield's design has two components: protected shield space memory and effective syscall filtering. The former enables the protection of the shield's code and data. The latter limits the GPU driver access permission to threads executing within the shield. We next elaborate on these two components. We then finish the section by providing details on the execution flow of an OpenGL ES API call in Milkomeda and by explaining how Milkomeda satisfies the guarantees of §3.3.

3.4.1 Protected Shield Space Memory

We isolate the shield space memory within the process address space. This space is a range of virtual addresses in the process address space that can only be accessed if the thread of execution has entered the shield space through a shield-call. Other threads within the process are not allowed to access the shield's memory.

We implement this protected memory space in the operating system kernel and by leveraging

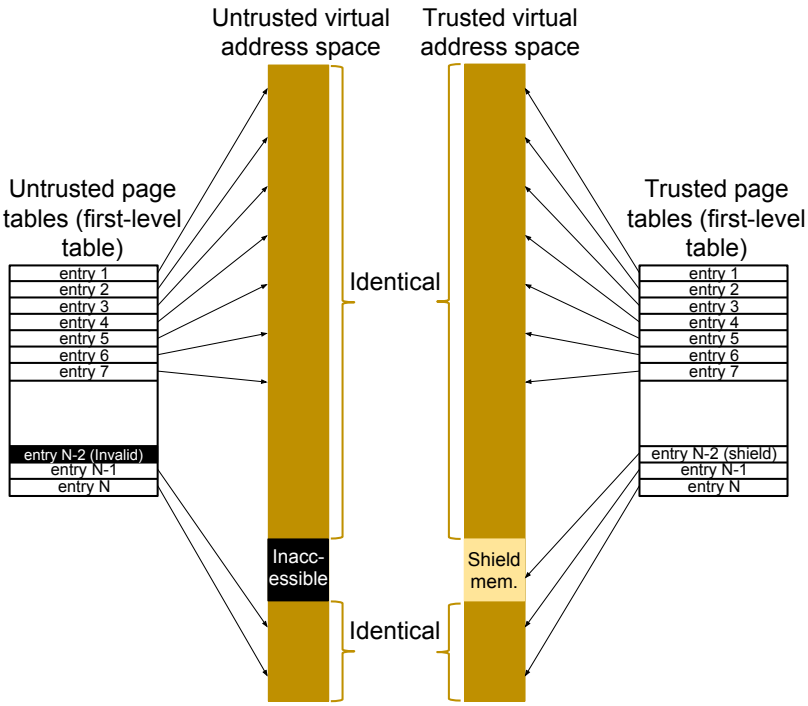


Figure 3.5: Implementation of shield space memory using page tables. The untrusted and trusted address spaces (mapped by the untrusted and trusted page tables and used, respectively, for the threads outside and within the shield space) are almost identical except for a contiguous range of addresses reserved for the shield space and accessible only through the trusted page tables.

page table translations. That is, we allocate two sets of page tables for the process, one to be used for threads executing outside the shield space (i.e., untrusted page tables) and one for threads executing within it (i.e., trusted page tables). The address space mapped by these two sets of page tables are mostly identical. They only differ in a fixed range of addresses, which is mapped by a single entry (or, if needed, a few entries) in the first-level page table. These addresses are marked as inaccessible in the untrusted page tables. They are however accessible in the trusted page tables. We choose to use the first-level page table entry to map the shield memory for performance: this design minimizes the operations needed to synchronize the trusted and untrusted page tables as synchronization is only needed when the first-level table is updated, which is rare. Figure 3.5 illustrates this concept.

All threads within the process use the untrusted page tables by default. They can, however,

request to enter the shield and use the trusted page tables. To do this, a thread needs to make a shield-call. We implement the shield-call with a syscall. Upon handling this syscall, the kernel programs the CPU core executing the thread to use the trusted page tables and resumes the execution at a designated call gate for the shield space. The code in the shield then handles the request and exits the shield space using another syscall. This exit syscall programs the CPU core to use the untrusted page tables, flushes the TLB, and returns. The thread can then resume its execution outside the shield. Note that the shield entry syscall does not need a TLB flush since the addresses used for the shield space are inaccessible in the untrusted page tables. Also, cache flush is not needed for the shield entry and exit syscalls for the same reason (i.e., the protected address range is inaccessible outside the shield and hence accesses to these addresses from outside the shield always fail).

While executing in the shield, a thread uses secure stack and heap memory. The secure stack is deployed by the kernel at shield entry syscall and removed upon exit. Heap allocation requests by threads within the shield are served from the reserved shield address range. This is guaranteed by the kernel, which simply checks the state of the requesting thread (i.e., whether it is executing in the shield or not) before allocating the virtual addresses. Our shield's design can support concurrent threads executing within the shield space. This is important as Android apps use multiple threads for graphics (e.g., one for hardware-accelerated UI compositing and one for 3D acceleration).

When in the shield, a thread can access all the process address space since the trusted page tables map all the address space. This allows the graphics libraries to access the memory allocated by the app directly, e.g., for data passed to the OpenGL ES API calls, avoiding the performance overhead of additional copies.

3.4.2 Effective Syscall Filtering

Milkomeda limits access to the GPU driver to only the shield space. More specifically, it allows only the threads in the shield space to interact with the GPU device driver. It achieves this using a set of checks at the entry points of the device driver in the kernel. These checks look at the state of the thread that issues the syscall for the GPU device driver. More specifically, in the kernel, Milkomeda marks the application's thread as either trusted (i.e., executing inside the shield) or untrusted (i.e., executing outside the shield) in the thread's Thread Control Block (e.g., Linux's `task_struct`). It only allows a syscall targeted at the GPU device driver if the thread issuing the syscall is marked as trusted. This requires adding only a handful of light-weight checks as the number of these syscall handlers in device drivers are limited (e.g., 6 handlers for the Qualcomm Adreno GPU driver including the handlers for `ioctl`, `mmap`, and `open` syscalls).

Note that we considered and even implemented another syscall filtering mechanism as our initial prototype. In this solution, we leveraged the Linux Seccomp syscall filtering mechanism, which allows us to configure the filter fully from user space [227]. We eventually settled for the aforementioned solution for two reasons: (*i*) our Seccomp filter required several comparisons to be evaluated for every syscall. While this overhead might not be noticeable for graphics operations, the filter needs to be evaluated for every syscall and hence can negatively affect the performance of apps that make many (even non-graphics) syscalls, such as apps stressing network or file I/O. (*ii*) Due to the limited functionality of the filter (e.g., inability to parse strings, access file systems, and dereference pointers), we had to implement a scheme that forwards all the `open` and `close` syscalls to the shield space for evaluation. While we managed to successfully build such a scheme, we noticed that it adds noticeable complexity to our system. Therefore, in light of better efficiency and lower complexity, we opted for the aforementioned solution, which only requires a few simple kernel checks that are executed only for GPU syscalls and hence do not affect other syscalls. Also, note that

while we add the checks in the driver entry points, they can also be added outside the driver right where the kernel calls into the driver entry points.

3.4.3 OpenGL ES API Call Execution Flow

In this subsection, we describe, the execution flow of an OpenGL ES API call in Milkomeda. Figure 3.6 shows this flow using pseudocode. First, the untrusted app code makes an OpenGL ES call. Second, this call is handled by a simple stub function in the untrusted part of the process. This stub function simply calls the syscall to enter the shield. Before doing so, it stores the arguments of the OpenGL ES call as well as the API number on the CPU registers. In our prototype based on ARMv8, up to 5 arguments are passed in CPU registers and the rest in memory. The OpenGL ES API numbers are known both in the stub function and in the shield space. In fact, existing OpenGL ES libraries already number the APIs. In case of an API number update by future OpenGL ES libraries, only the relevant libraries need to be updated.

Third, the shield entry syscall handler in the kernel securely transfers the execution to the designated call gate function in the shield space. To do so, the syscall handler saves the current state of CPU registers (to be restored on exit from the shield), sets up a secure stack for the thread, sets the program counter to the address of the call gate function, marks the thread as secure (§3.4.2), switches to use the secure page tables on the CPU core executing the thread, and finally exits, which then resumes the execution in user space in the designated call gate function.

Fourth, the call gate function identifies the called OpenGL ES API using the API number passed on a CPU register. It performs the security checks needed for the specific API call. If rejected, it returns an error. If passed, it calls the actual API handler in the OpenGL ES library. This handler then executes the API call, interacting with the GPU device driver

when needed, and gives back a return value. The call gate function then exits the shield using another syscall, passing the return value along.

Finally, the shield exit syscall handler in the kernel securely transfers the execution to the original caller of the shield entry syscall. To do so, it switches to use the untrusted page tables on the CPU core executing the thread, flushes the TLB (§3.4.1), marks the thread as untrusted, restores the previously saved CPU registers, gives the aforementioned return value to the caller by putting it on a CPU register, and exits. The app code then resumes its execution. To the app, it looks as if the shield entry syscall executed the graphics API, returning the result.

3.4.4 Satisfying the Required Guarantees

In this subsection, we discuss how Milkomeda achieves the four required guarantees discussed in §3.3.

Security guarantee I. The first guarantee states that only the threads within the shield be allowed to invoke the GPU device driver. We achieve this by using our syscall filtering mechanism (§3.4.2). The filter rejects syscalls targeted at the GPU device driver when issued by threads executing from outside the shield.

Security guarantee II. The second guarantee states that the control-flow integrity of the checks be preserved by forcing the app code to enter the shield space only at a designated call gate. We achieve this using our protected shield memory (§3.4.1). A thread cannot normally access the memory of the shield space as this region of memory is marked as inaccessible in the untrusted page tables. As a result, if it does attempt to jump to any location within the shield, it will result in a page translation fault. The only way to access the shield is to issue a shield-call, which resumes the execution at a predetermined call gate in the shield.

```c
/* Untrusted application code */
long foo(void)
{
...
/* Calls an OpenGL ES API */
return some_opengles_api(arg1, arg2, ...);
}
```

```c
/* Stub function for the OpenGL ES API in an untrusted user space library */
long some_opengles_api(long arg1, long arg2, ...)
{
  /* Store as many arguments on the CPU registers as possible.
   * If any, store the rest of the arguments in a memory buffer
   * Enter the shield with a syscall */
  return syscall(NR_SHIELD_ENTER, API_NUM, arg1, arg2, ...);
}
```

```c
/* Kernel implementation of shield entry syscall */
SYSCALL_DEFINE(shield_enter, long, api_num, long, arg1, long, arg2, ...)
{
  /* 1. Save current CPU registers
   * 2. Prepare secure stack for the thread
   * 3. Update the stack pointer and the program counter
   * 4. Mark the thread as secure
   * 5. Switch to the secure page tables
   * 6. Exit (which transfers the execution to the predefined userspace
   *    location for the call gate function) */
}
```

```c
/* The call gate function in the shield space */
void call_gate_func(long api_num, long arg1, long arg2, ...)
{
  /* 1. Determine the requested OpenGL ES API based on api_num
   * 2. Execute security checks for this API, return error if not safe */
  if (!is_opengles_call_safe(api_num, arg1, arg2, ...))
    return -1;

  /* 3. Call the actual OpenGL ES API */
  long rv = some_opengles_api_actual_function(arg1, arg2, ...)

  /* 4. Return from the shield, return the OpenGL ES call return value (rv) */
  syscall(NR_SHIELD_EXIT, rv);

  /* The execution never reaches here. */
}
```

```c
/* Kernel implementation of shield_exit syscall */
SYSCALL_DEFINE(shield_exit, long, rv)
{
  /* 1. Switch to the untrusted page tables
   * 2. Flush the TLB
   * 3. Mark the thread as untrusted
   * 4. Restore previously saved CPU registers
   * 5. Store the return value (rv) on a CPU register
   * 6. Exit (which returns to the untrusted app code outside the shield,
   *    to right after the shield entry syscall) */
}
```

Figure 3.6: Pseudocode demonstrating an OpenGL ES API call in Milkomeda.

Security guarantee III. The third guarantee states that the code and data within the shield are protected from tampering by untrusted code. This prevents untrusted code from compromising the integrity of the security checks in the shield since these checks rely not only on correct code for the checks but also on several global variables, e.g., to maintain state information about prior calls (§3.1.4). We achieve this by using our protected shield memory (§3.4.1). All the code and data of these security checks (including the stack and heap) are allocated within the shield and hence are protected.

Performance guarantee. The last guarantee states that performance loss should be minimized. Our solution eliminates the need for IPC, shared memory data copy, and serialization/deserialization of API calls. It does however add some overhead including two syscalls per OpenGL ES API call (one syscall to enter the shield space and one to exit it), saving and restoring the register state as well as changing the page tables at entry and exit syscalls, and TLB flushes in the exit syscall as well as in some context switches (§3.6.1).

3.5 Reusing WebGL Security Checks for Mobile Graphics

One of our key design principles is to aim for minimal engineering effort to port and reuse WebGL's security checks for mobile graphics. This is because these checks are still under active development. For instance, our study shows that 12 new patches have been added to these checks in just 2 months recently (March and April 2018). A solution that requires significant effort to port these checks to mobile graphics makes it challenging to keep the checks up-to-date. As a result, we developed a tool, called CheckGen, which automatically ports the WebGL security checks to be used for mobile graphics.

Figure 3.7 illustrates the role of the CheckGen tool. The left side of the figure depicts the

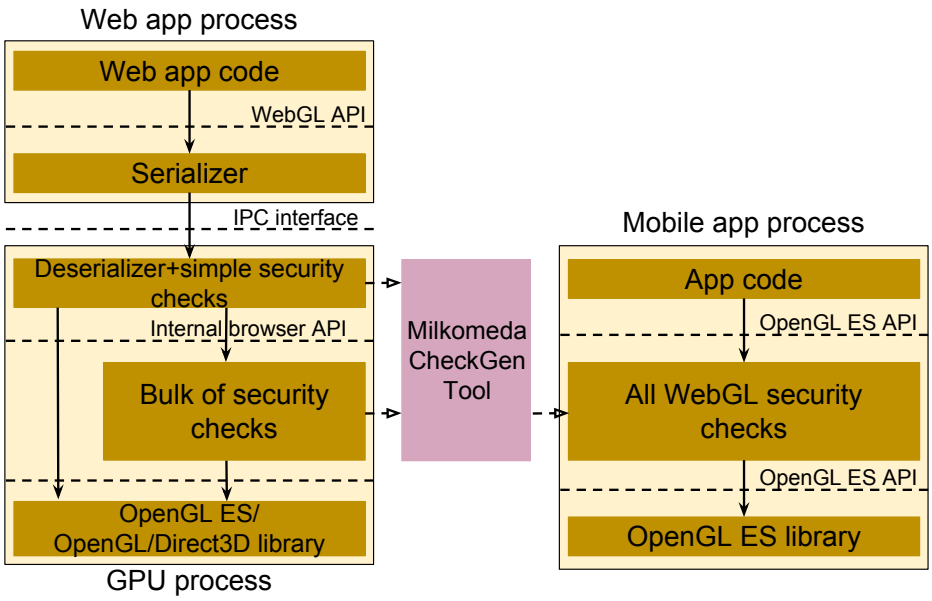


Figure 3.7: Milkomeda's CheckGen tool automatically transforms WebGL's security checks into a single layer to be used for mobile graphics.

WebGL stack, all the way from the web app to the underlying graphics library (OpenGL ES, OpenGL, or Direct3D depending on the platform and operating system). A WebGL API call is first serialized in the web app process and sent, using IPC and shared memory, to the GPU process. Inside the GPU process, the IPC is deserialized. Some simple security checks, such as validation of numeric values (§3.1.4) are performed in the same procedure that performs the deserialization. Some select API calls are then forwarded for more security checks and others are directly passed to the underlying graphics library. Therefore, as can be seen in the figure, the security checks are spread across two layers in the WebGL stack, a layer dedicated for checks and the deserializer. Our CheckGen tool receives the source code for these two layers and generates one single vetting layer with the OpenGL ES API as its input and output, which can then be used in the mobile graphics stack, shown on the right side of the same figure.

In the rest of this section, we discuss the challenges that we addressed in CheckGen.

3.5.1 Fixing the Interface for Security Checks

CheckGen transforms the input interface of WebGL's deserializer to the OpenGL ES interface, as expected by mobile apps (see Figure 3.7). The deserializer interface accepts a pointer to and the size of a shared memory segment as arguments for a WebGL call. It contains the code that extracts OpenGL ES API arguments from this shared memory segment, and then performs simple security checks. To transform this interface to the OpenGL ES interface, CheckGen uses the OpenGL ES interface definition. Moreover, it removes all the deserialization code and only keeps the simple security checks of this layer using pattern matching. The bulk of the security checks provided in the next layer (Figure 3.7) are then used without any modifications.

3.5.2 WebGL and OpenGL ES Incompatibilities

The WebGL and OpenGL ES API have a few differences. More specifically, the Chromium project documents two incompatibilities between WebGL and OpenGL ES 2.0 [81]. First, WebGL does not support client-side vertex arrays [77], which store vertices and their attributes in the system memory instead of the GPU memory. This is not due to security and mainly because this API is slow (indeed, it is being deprecated in OpenGL ES 3.0). Therefore, WebGL fails calls to this API. However, this feature is required by the OpenGL ES 2.0 specification, and indeed used by many mobile apps, e.g., by two of the mobile app benchmarks used in our evaluation. Therefore, we enable this feature in Milkomeda and remove a Chrome WebGL check due to this incompatibility. An alternative option is to emulate this feature on top of other OpenGL ES APIs.

Second, WebGL does not support the GL_FIXED attribute type. It suggests using GL_FLOAT instead since GL_FIXED "requires the same amount of memory as GL_FLOAT, but provides a smaller range of values" [48]. Chromium converts this type [81]. Because Milkomeda is

built on top of OpenGL ES, which requires support for GL_FIXED, we modify the checks to accept GL_FIXED. Our understanding is that this does not cause a security problem. Alternatively, we can also convert this type.

3.6 Implementation

We implement Milkomeda for Android operating system on 64-bit ARMv8 processors, which are commonly used in all recent mobile devices (see §3.8 for a discussion on support for ARMv7 processors). We use Google Chromium's WebGL security checks in our implementation. Milkomeda's implementation consists of two parts: the shield and the CheckGen tool. Below, we provide implementation details on these two components.

3.6.1 Shield Integration

The core of the shield's functionality is implemented in the Linux kernel. This includes the implementation of the protected memory space and syscall filtering. Our implementation consists of about 500 LoC, making the solution easy to reason about and easy to port.

The shield space needs to be set up by the process at its initialization time. This is done through one syscall that activates the shield for a range of addresses in the address space. The activation syscall creates the secondary set of page tables and marks the designated address range as inaccessible in the default page tables. Moreover, the same syscall sets the shield's call gate address and prepares secure stacks for threads to execute in the shield. Note that once the shield is activated, it cannot be deactivated by the process anymore.

In our current prototype, we fix the shield address space size to be 1 GB. This is because (*i*) 1 GB of address space is mapped by a single entry in the first-level page table (when using the 4

kB translation granulate with three levels of address translation in ARMv8 [106]), simplifying the implementation and (*ii*) 1 GB is large enough for all the trusted code (including the graphics libraries, security checks, and the libraries they depend on). Note that we do not allocate memory for the shield space unless needed. That is, we only reserve 1 GB of the address space, but the actual backing memory is only allocated and mapped when needed (e.g., when a library is loaded or when trusted code performs dynamic memory allocation). Increasing the shield address space size, if needed, is trivial by using more of the first-level page table entries. Also, note that reserving 1 GB of the address space does not put pressure on the operating system memory management for finding unallocated memory addresses for the app. This is because the virtual address space in ARMv8 is large (256 GB of address space when using the aforementioned paging mode, which uses 38-bit virtual addresses effectively [106]). Finally, when setting up the shield, we choose one entry in the first-level page table that is yet unused. The chosen entry then determines the start and end addresses of the shield space.

To protect the integrity of the security checks, it is important that all code and data used by these checks are isolated from the rest of the app. To do this, we load the security checks, the graphics libraries, as well as all the libraries they rely on in the shield space. This means that we have duplicate copies of several libraries in the process address space, one for use by the untrusted code in the app and one to be used by the protected code in the shield. One noteworthy example is LibC. We initialize two instances of LibC, one for the untrusted code and one for the graphics-related code in the shield. This ensures that all the global variables and dynamic allocations of LibC and other libraries used by the trusted code are in the shield space as well and hence protected. This design increases the memory usage of the app (since it needs to load more libraries). Moreover, it puts more pressure on the code cache. However, these libraries are shared between all apps hence amortizing the overhead. Moreover, as part of our future work, we plan to investigate sharing the library code (but not data) between the trusted and untrusted space in the process address space to eliminate

this additional overhead.

These libraries need to be loaded and the shield needs to be activated before untrusted app code is loaded. We implement this for Android in the app's launch sequence. We bypass the Zygote process (which forks a pre-configured process) and execute the launch sequence from scratch. In the future, to accelerate the launch time of Milkomeda apps, we can create a secondary Zygote process with Milkomeda's shield preconfigured. Our implementation allows us to select the apps that need to be protected by Milkomeda by specifying the app's package name in Android system properties. This capability can be used by the operating system admin or the user in various ways: first, it is possible to enable Milkomeda on all apps. Second, it is possible to enable Milkomeda by default but whitelist some trusted apps. Finally, it is possible use Milkomeda for only a set of blacklisted apps.

Milkomeda does not require any modifications to the app. Indeed, it can support binary code, i.e., `.apk` executable packages in Android. To achieve this, Milkomeda employs a shim graphics library outside the shield space that implements the OpenGL ES API. When called by the app, it issues a shield-call and passes the API number and its arguments (see the OpenGL ES stub function in Figure 3.6).

Milkomeda does not allow any OpenGL ES API call to register a callback. Otherwise, such a callback can be exploited by malware to execute arbitrary code within the shield space. Fortunately, there is only one OpenGL ES API with a callback: glDebugMessageCallback. We disable this debug API in Milkomeda.

Milkomeda's shield implementation is thread-safe. Each thread entering the shield has its own secure stack. Indeed, our benchmarks in §3.7.2 use multiple threads for graphics. These threads enter the shield separately and potentially concurrently. Thread scheduling is also safely done in Milkomeda. We have modified the kernel context switch procedure so that the right page tables (secure vs. untrusted) are used for a thread, and the TLB is flushed,

when needed, to prevent an untrusted thread from accessing the TLB entries for the shield space.

Milkomeda does not allow delivering a signal to a thread within the shield space. This is important to ensure the integrity of execution within the shield.

3.6.2 CheckGen's Implementation

We implement CheckGen in Python. It compiles the security checks as a set of shared libraries by reusing part of the Chromium source code. In addition to the regular build process, which produces the unified browser executable, Chromium also supports a component build. We leverage the component build to generate the aforementioned shared libraries.

OpenGL ES represents the graphics state with a *context* object. In order to properly vet the graphics API calls, we create a separate instance of the security checks for each graphics context (similar to WebGL).

Chromium implements GPU driver and library bug workarounds for specific vendors and operating systems (§3.1.4). Similarly, we apply the workarounds for the GPU used in the target mobile device, e.g., the Adreno GPU in our prototype.

We solve one challenge with respect to the IDs of graphics objects in OpenGL ES. OpenGL ES assigns integer IDs to graphics resource objects, such as texture objects. In WebGL, in order to minimize the round trip delay for management of IDs, the web app process itself generates the ID immediately upon creating an object and uses these locally generated IDs in future operations [81]. The GPU process uses the real IDs returned by the OpenGL ES library, and maintains a mapping between the web app process-generated IDs and the real IDs. As this is a performance optimization needed in the multi-process architecture [81], we disable it in CheckGen. Note that this does not affect the security of Milkomeda because

the real IDs are not considered secrets.

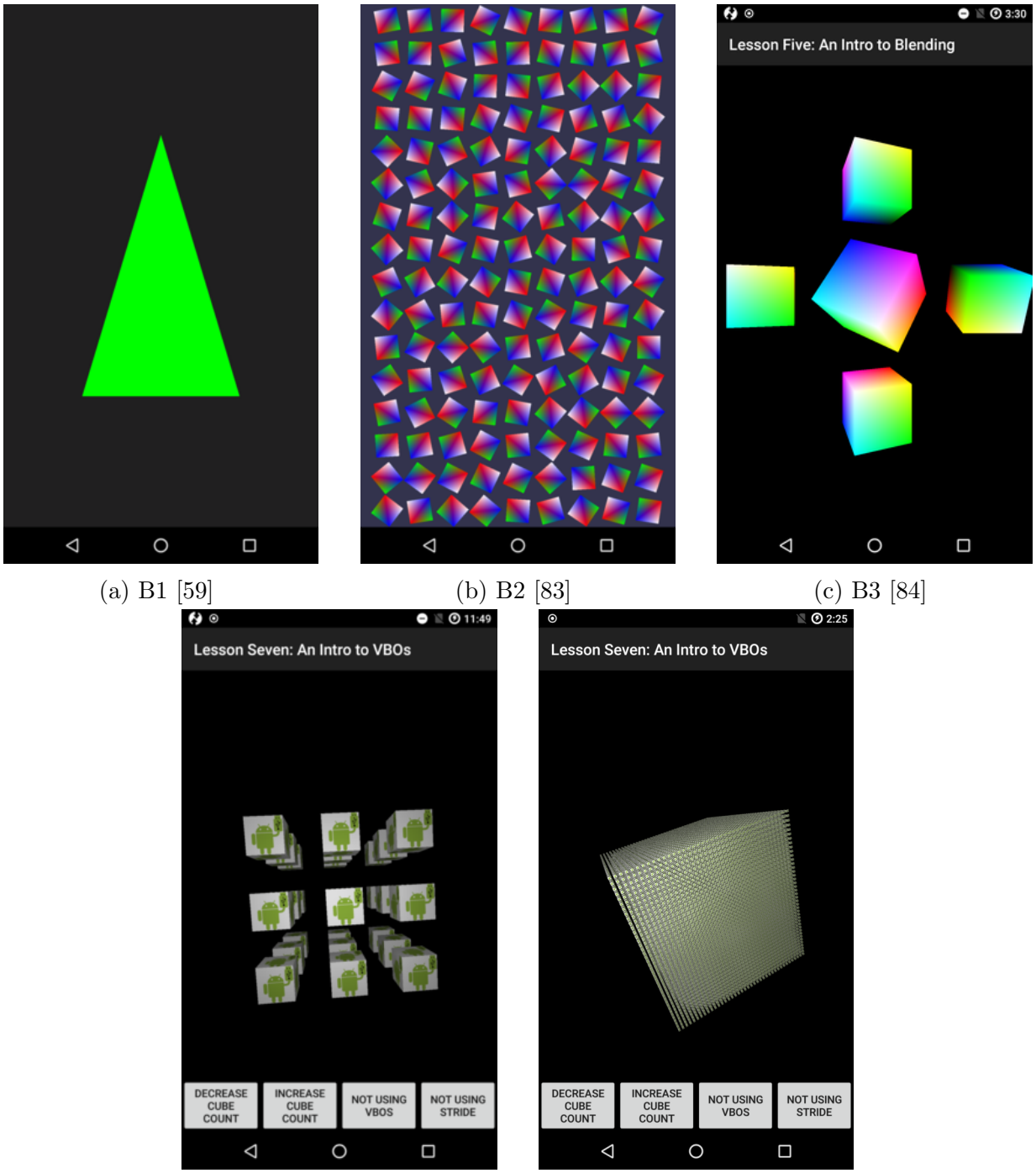
3.7 Evaluation

We evaluate Milkomeda on a Nexus 5X smartphone. This smartphone has 2 GB of memory, four ARM Cortex-A53 cores as well as two ARM Cortex-A57 cores (ARM big.LITTLE), and an Adreno 418 GPU. We use Android 7.1.2 (LineageOS 14.1).

3.7.1 Security Analysis

In this section, we discuss the attacks that Milkomeda can and cannot protect against and compare with the multi-processor architecture deployed in web browsers.

First, an attacker may try to directly invoke the GPU device driver syscalls. Milkomeda prevents this attack as only the shield space is allowed to interact with the GPU device driver. The multi-process architecture prevents this attack too as the web app process is not given permission to interact with the GPU driver. Second, the attacker may try to jump past the security checks and directly execute the unvetted OpenGL EL API. Milkomeda prevents this attack since a thread cannot enter the shield space at arbitrary entry points. Similarly, the multi-process architecture does not allow this attack since a thread in one process cannot jump to and execute code in a different process. Third, an attacker may try to trigger the driver vulnerabilities through the OpenGL ES API calls. Milkomeda leverages WebGL's security checks to stop these attacks. Any such attack that is successful against Milkomeda is also successful against the multi-process architecture. Fourth, an attacker may try to leverage a vulnerability in the Trusted Computing Base (TCB) of Milkomeda in order to bypass the security checks. The TCB of Milkomeda is the operating system kernel as well as all the code inside the shield space. This is almost a subset of the TCB in the multi-process



(a) B1 [59] (b) B2 [83] (c) B3 [84]

(d) B4 [85] (e) B5

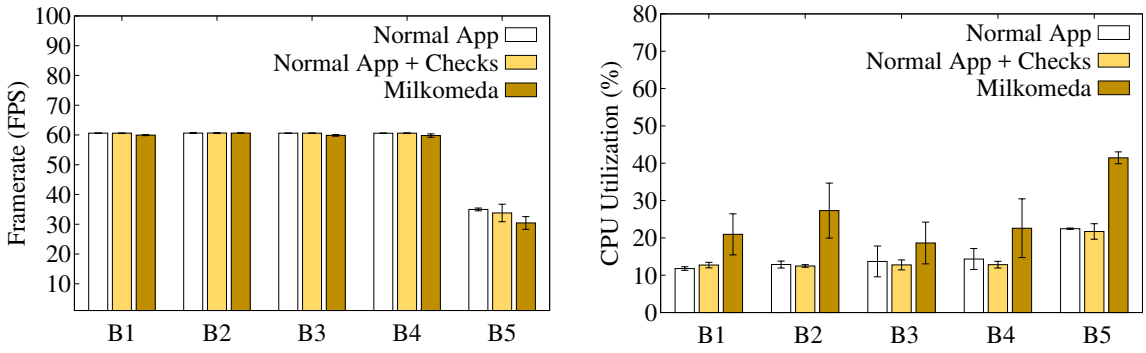
Figure 3.8: Graphics benchmarks used in evaluation. We derive B5 from B4 by increasing the number of cubes significantly.

architecture, which does not need the small amount of kernel code needed to implement the shield space but requires more code in the GPU process to support composing of the browser's UI as well as IPC and shared memory code used for communication. Therefore, most such attacks are also effective against the multi-process architecture.

We evaluate the effectiveness of Milkomeda in preventing vulnerability exploits. We have investigated all 64 CVEs in Table 3.5. We managed to find enough information on 45 of them for analysis (including patches, source code, and PoC). For these 45, we have confirmed that Milkomeda prevents all of them. This is because all of these CVEs directly invoke the GPU device driver APIs, which are prevented in Milkomeda.

With these CVEs neutralized, an attacker can try to use the OpenGL ES API to mount attacks. Similar attacks have been attempted through the WebGL APIs (which is quite similar to the OpenGL ES API) [264]. Since WebGL checks are designed to protect against such attacks in the browser, they protect against similar attacks on mobile devices.

We note that the WebGL security checks may miss some zero-day attacks [264]. However, these checks provide two benefits. First, they prohibit attacks using known vulnerabilities in the GPU driver. Second, they limit unknown attacks due to the additional state verification. The WebGL security checks limit the arguments of the graphics APIs (e.g., they return early if an argument is not valid per OpenGL ES specification). Some vulnerabilities are caused by invalid arguments that violate the OpenGL ES specification. Therefore, constraining API calls prevents invalid OpenGL ES API inputs and thereby stops some, but not all, unknown attacks. Milkomeda is therefore a mitigation, comparable to ASLR or stack canaries, that stops some attack vectors and makes other attack vectors harder.



(a) (b)

Figure 3.9: (a) Graphics performance. (b) CPU utilization. In both of the figures, B1 to B5 represent the five benchmarks shown in Figure 3.8. Each bar in the figure shows the average over six runs and the error bar shows the standard deviation.

3.7.2 Graphics Performance and CPU Usage

We measure the mobile graphics performance using the achieved framerate, which determines the number of frames rendered in one second. We use 5 mobile app benchmarks in our evaluation. We choose these apps as they focus on GPU-based graphics and they span a range of apps with simple to complex graphics operations. Figure 3.8 shows snapshots of these benchmarks (B1-B5). We derive the fifth benchmark (B5) by modifying B4 to render 64,000 (40^3) cubes rather than 27 (3^3). We run each benchmark six times. We discard the first 100 frames in each run to eliminate the effect of initialization in the measurements.

Figure 3.9a shows the framerate in our benchmarks. It shows the measurement for three different configurations: normal app, normal app + checks, and Milkomeda. The first configuration is the performance of the benchmarks using an unmodified graphics stack, i.e., the state of the art. The second configuration represents the performance of the security checks without the shield's space to protect their integrity. This configuration is not secure. Yet, it allows us to measure the overhead needed for evaluating the security checks on OpenGLES APIs. The third configuration is Milkomeda, in which not only the security checks are evaluated, but also the shield space is used to protect the integrity of the checks.

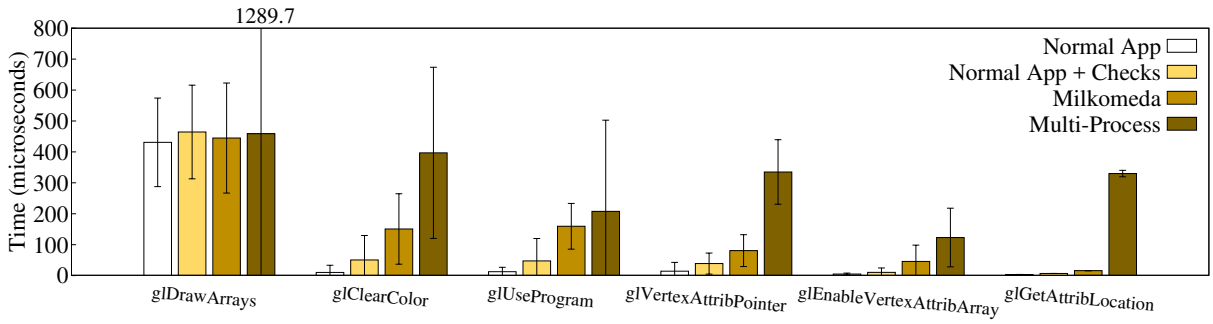


Figure 3.10: Execution time of several OpenGL ES API calls. Each bar in the figure shows the average over all invocations of the API in three runs and the error bar shows the standard deviation.

The results show the following. First, for benchmarks with 60 FPS framerate, Milkomeda manages to maintain the 60 FPS graphics performance. Note that in Android, framerate is capped at a maximum of 60 FPS, which is the display refresh rate. Therefore, for these benchmarks, Milkomeda achieves the maximum graphics performance. Second, for a benchmark with lower FPS, Milkomeda achieves a close-to-native performance. Overall, the results show that Milkomeda does not impact the user experience.

However, the extra security in Milkomeda comes at a cost: higher resource usage. Figure 3.9b shows the CPU utilization of the system when executing the same benchmarks. It shows that Milkomeda increases the CPU utilization from 15% (for normal execution) to 26%, on average. We note that this additional CPU utilization is not prohibitively high. However, if the system is highly utilized, e.g., by various background tasks, then the graphics performance gets affected more significantly in Milkomeda compared to normal apps.

3.7.3 Comparison with the Multi-Process Design

As mentioned in §3.1.1, browsers deploy the WebGL security checks in a separate process from the web app process to protect the integrity of checks. To compare the overhead of this approach with Milkomeda, we implement such a multi-process architecture for mobile apps.

That is, in the mobile app process, we forward the OpenGL ES API calls over IPC (using sockets) to another process for execution. We also use shared memory to pass the data.

Our multi-process prototype does not support all OpenGL ES API calls (it supports around 30 of them) since supporting each API call requires us to understand the semantics of the parameters and write the proper serialization and deserialization code for it. Therefore, we report the execution time of a few OpenGL ES API calls that we do support (average of three runs of the experiment). Figure 3.10 shows the results. As can be seen, the multi-process architecture increases the execution time of these API calls significantly (an average increase of 440% compared to Milkomeda).

3.8 Limitations

Other GPU frameworks. While OpenGL ES is the main framework using the GPU in mobile devices, it is not the only one. Notably, OpenCL and CUDA leverage the GPU for computation. Milkomeda disallows any code outside the shield space to interact with the GPU device driver. Therefore, our current prototype blocks the usage of such frameworks. We plan to address this problem in two steps. First, we will load these frameworks in the shield space and allow the app to use them by making proper shield-calls. Note that this step immediately improves the state of the art, which needs to give unrestricted access to the app for communication with the GPU device driver. In our solution, the app's access will be regulated and limited to a higher-level API (i.e., the GPU framework API). Second, we will evaluate the security of the interface of these frameworks and, if needed, investigate adding proper vetting for them as well.

Use the shield space to improve WebGL performance. As mentioned, web browsers deploy a multi-process architecture to protect the integrity of the security checks (see Fig-

ure 3.1b). We plan to use the shield space to employ the WebGL backend (including the security checks) in the web app process and improve the WebGL performance.

Supporting ANGLE. As mentioned in §3.1.4, WebGL uses ANGLE's shader validator. ANGLE, in addition to the shader verifier, is being orthogonally equipped with a set of security checks. While it does not yet provide a comprehensive set of checks as current WebGL checks (e.g., no support for OpenGL ES version 3.0), it is under active development and will likely add the missing checks, as evident from a discussion by Google on the potential integration of all security checks [74]. We plan to update our CheckGen tool to also automatically reuse ANGLE's security checks for the mobile graphics interface.

Supporting ARMv7 processors. Our shield implementation in Milkomeda targets ARMv8 processors, used in modern mobile devices. We plan to support older ARMv7 processors as well. For that, we will use a smaller part of the process address space for the shield space since the address space is limited for these 32-bit processors. We will also consider implementing the shield space memory using ARM memory domains available in ARMv7 processors [105], which will not require changes to the kernel. Note that, unfortunately, ARM memory domains are not available on ARMv8 processors. We believe that if such hardware support existed on these processors, the shield's overhead could be reduced.

Chapter 4

Minimizing a Smartphone's TCB with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware

Because of their ubiquity and portability, modern smartphones are often used to run security-critical programs along with diverse, untrusted, and potentially malicious programs. For example, most of us perform financial tasks, such as banking and payments [214] on our smartphones. Many of us also run health-related programs, e.g., to receive test results and diagnoses from our health providers. There is also interest in using these devices to perform life-critical tasks such as controlling an insulin pump [242] or monitoring breathing [203], although security concerns currently pose a roadblock [242].

Realizing this computing paradigm should be straightforward. The job of an operating system (OS) is to isolate security-critical programs from other programs running on the same hardware. Yet, this has proven to be challenging in practice due to vulnerabilities in system software (e.g., OS, hypervisor, and device drivers) [247, 96, 76, 92, 93, 271, 209,

112, 132] and hardware (e.g., processor, memory, interconnects, and I/O devices including their firmware) [172, 190, 175, 244, 208, 254, 136]. Malicious programs can exploit these vulnerabilities to take control of the machine and any program running on it. We must trust that hardware and system software can effectively sandbox and neutralize malicious programs, but this trust often proves to be misplaced.

To address this challenge, a new approach has emerged. It uses *Trusted Execution Environments (TEEs)* to host security-critical programs without requiring trust in the OS. Unfortunately, today's TEEs still require us to trust the hardware and the security monitor implementing the TEE guarantees. This trust has also proven unjustified. Existing TEEs have fallen victim to various attacks, e.g., hardware-based side-channel attacks [122, 244, 127, 200, 198, 156, 226, 189, 272, 183], attacks exploiting software vulnerabilities [126, 91, 220, 137], and attacks based on design flaws [163, 182, 257, 181].

In this work, we present a solution to enable smartphones to be used for both security-critical and non-critical programs. Our goal is to minimize the Trusted Computing Base (TCB). More specifically, our goal is to minimize the number and complexity of hardware and software components that need to be trusted by the smartphone owner, when executing a security-critical program, to fend off adversarial inputs.

Our key principle is *provably exclusive access* to hardware and software components. That is, we design a solution to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use.* The exclusive use of a component makes it unreachable to attackers.

More concretely, we present a hardware design for a smartphone. Called a *split-trust hardware*, it comprises multiple trust domains, one or multiple for TEEs, one for each I/O device, one for a resource manager, and one for hosting a commodity OS, e.g., Android, and its programs. The trust domains are *statically-partitioned* and *physically-isolated*: they each have

their own processor and memory (and one I/O device in the case of an I/O domain) and do not share any underlying hardware components; they can only communicate by message passing over a hardware mailbox. Moreover, we introduce a few simple, *formally-verified* hardware components that enable a program to gain provably exclusive access to one or multiple domains.

We then present OctopOS, an OS to manage this hardware. Unlike existing OSes, which have a single, trusted-by-all nucleus, i.e., the kernel, OctopOS comprises mutually distrustful subsystems: a TEE runtime for security-critical programs, I/O services, a resource manager, and a compatibility layer for a commodity, untrusted OS. The fundamental aspect of OctopOS is that components *do not trust, but verify* messages received from other components.

We rigorously evaluate the TCB of our machine. We show that it significantly reduces the TCB compared to mainstream TEEs and achieves one close to the lower bound.

We present a complete prototype of our machine (hardware and OS) on top of a CPU-FPGA board (Xilinx Zynq UltraScale+ MPSoC ZCU102). We use the powerful ARM Cortex A53 CPU to host the commodity, untrusted OS (PetaLinux) and its programs with high performance. We use the FPGA to build the other trust domains: two TEEs, a resource manager, and four I/O domains (an input domain, an output domain, a storage domain, and a network domain). We use (weak) microcontrollers for these other domains.

Using our prototype, we build two important security-critical programs for our machine:[1] (*i*) a banking program that can securely interact with the user, and (*ii*) an insulin pump program that can securely execute its algorithm and communicate with an (emulated) glucose monitor and pump.

Using our prototype, we show that the added hardware cost is small (i.e., 1-2%) compared

[1]We open source our hardware design and formal verification proofs at `https://github.com/trusslab/octopos_hardware`, and OctopOS and security-critical programs at `https://github.com/trusslab/octopos`.

to modern SoCs used in smartphones. Moreover, we show that *security-critical program can achieve usable performance despite the use of weak microcontrollers for all TEE and I/O domains*. We also show that *normal programs can achieve the same compute and I/O performance as on a legacy machine*, which is defined as a machine using the same powerful CPU as our untrusted domain but with that CPU being in full control of all I/O devices and main memory.

Secure hardware trend. Our vision of using physical isolation and exclusive use for security is in line with recent hardware trends from the smartphone industry. Apple has integrated the Secure Enclave Processor (SEP) into its products [88] and used it to secure user's secret data and to control biometric sensors (i.e., Touch ID and Face ID) [89]. Similarly, Pixel 6 uses the tensor security core to host security-critical tasks such as key management and secure boot [173]. Our work takes this vision further by allowing user-provided, third-party security-critical programs (including those that rely on I/O devices) to use dedicated hardware by developing a model for how that can be safely done.

4.1 Trust in Existing Systems

The TCB in a system comprises the hardware and software components that need to be trusted. Historically, the OS has been a trusted part of the system and hence part of the TCB (Figure 4.1 (a)). As commodity OSes have become more complex over the years, more and more vulnerabilities have been found in them, allowing malware to exploit them and compromise the OS [96, 247, 92, 76, 271, 128, 209, 112, 132]. As an example, there have been about 1700 security vulnerabilities reported in the Linux kernel just since 2016 [96]. Therefore, trust in commodity OSes is not warranted.

There have been several attempts to build trustworthy OSes. These include microkernels [99,

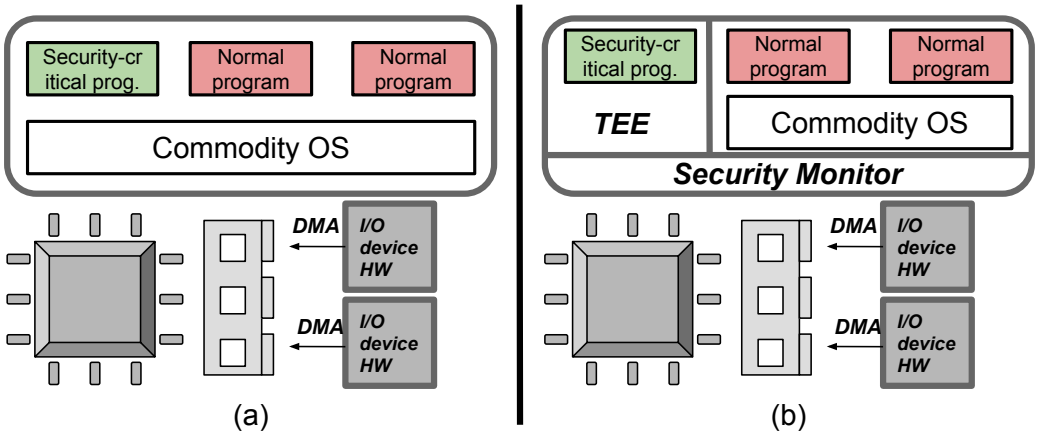


Figure 4.1: *(a) Traditional design where the OS isolates security-critical programs from normal programs. (b) Use of a TEE to isolate a security-critical program.*

188, 153, 174, 141], exokernels and library OSes [142, 170, 215, 113], formally verified OSes (and hypervisors) [174, 159, 160, 248, 205, 233, 184, 185, 240], and OSes written in safe languages [143, 168, 180, 204]. While effective, these solutions require replacing commodity OSes with a new OS. This is a challenging task due to the abundance of existing programs, device drivers, and developers for commodity OSes. More importantly, using these OSes still requires trust in hardware, which is not warranted either, as we will discuss.

About two decades ago, a new approach started to gain popularity. The idea is to create an isolated environment, called a TEE, to host a security-critical program. This allows the use of a commodity OS, but relegates it to be only in charge of untrusted, normal programs such as games, utility apps, and entertainment platforms. The TEE enables a security-critical program to ensure its own integrity and confidentiality, but leaves the OS in charge of resource management (and hence the availability guarantee). Therefore, one does not need to trust the OS when running a security-critical program, reducing the TCB. Figure 4.1 (b) illustrates this design. It shows a *security monitor* is used to isolate a TEE from the OS. The security monitor can be implemented purely in software (i.e., a hypervisor) [131, 165] or using a combination of hardware and software. ARM TrustZone and Intel SGX are examples of the latter. Others include AMD Secure Encrypted Virtualization (SEV), Intel Trusted

Domain Extensions (TDX), ARMv9's Realms [157], and Keystone for RISC-V [176].

Despite their success, existing TEE solutions still have a large TCB including the security monitor and several hardware components such as the very complex processor, memory, I/O devices in some cases, and dynamically-programmable protection hardware such as address space controllers and MMUs. Unfortunately, all of these components can be compromised by an adversary. For examples, hypervisors contain many vulnerabilities [93, 109]. TEE OSes in TrustZone have also contained vulnerabilities and have been exploited in the past [126, 91, 220, 137]. AMD SEV has also been shown to contain several vulnerabilities due to design flaws [163, 182, 257]. AMD's response to these vulnerabilities have been enhanced versions of SEV, called SEV-ES and SEV-SNP. Unfortunately, these versions have also fallen to attacks exploiting side channels [183] or additional design flaws [181].

Hardware components have been exploited as well. Hardware-based side-channel attacks have recently emerged as a serious threat to computing systems. For example, SGX enclaves and TrustZone have been compromised using several such attacks [122, 244, 127, 200, 198, 156, 226, 189, 272]. The core reason behind this is that existing machines run the untrusted OS and TEEs on the same hardware, sharing underlying microarchitectural features such as cache [122, 189, 272, 198, 156, 226] and speculative execution engine [190, 244, 175, 127], as well as architectural ones such as virtual memory [200]. The memory subsystem has also proved vulnerable to Rowhammer attacks [172, 221, 246, 260, 158, 192]. The complexity of these hardware components ensures that more vulnerabilities are likely to be discovered and exploited. For example, researchers have recently demonstrated a suite of new side channels using the CPU interconnect [208], the x87 floating-point unit, and Advanced Vector extensions (AVX) instructions (among others) [254].

4.2 Key Goal and Principle

4.2.1 Trust Definitions

We define two types of trust in the TCB: *strong trust* and *weak trust*. We say a component is strongly trusted if it needs to guard against *adversarial inputs*. An example is an OS that is trusted to isolate a program from other malicious programs, which can issue adversarial syscalls to the OS concurrently to the protected program. This component must be trusted to prevent these other programs from exploiting any vulnerabilities in it. This is challenging as demonstrated by the plethora of reported exploits.

We say that a component is weakly trusted if it just needs to operate correctly in the absence of adversarial inputs. An example is an OS that only serves a single program (and assuming application-level networking). This component must only be trusted to not exert buggy behavior under normal usage. This can be (more) easily achieved in practice.

Due to their obvious criticality, in this work, we focus on the strongly-trusted components in the TCB. For brevity, when talking about TCB, we mainly refer to these components.

Finally, we note that all components of the TCB need to be trusted not to have any backdoors implanted by an adversary.

4.2.2 Key Goal

Our goal in this work is to minimize *both the number and complexity* of (strongly-trusted) components in the TCB. Our rationale for the former is obvious: the fewer trusted components, the better. Our rationale for the latter is that it is difficult for complex hardware or software components to adequately protect themselves against attacks; by contrast, sim-

pler components can fend off attacks through comprehensive testing, analysis, and formal verification.

4.2.3 Key Principle

Our key principle to achieve this goal is *provably exclusive access* to hardware and software components. That is, we design our machine to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use*. More specifically, our goal is to have most components, especially complex ones such as the processor and system software, (1) be reset to a clean state before use, (2) then used exclusively by a security-critical program in a verifiable fashion through remote and/or local attestation, and (3) then again reset to a clean state right after use. In this case, such a component does not need to be (strongly) trusted anymore as it cannot be reached by an attacker while serving the security-critical program, nor does it need to worry about residual state from the security-critical program while serving other, potentially malicious, programs.

To realize this principle, we introduce a novel *split-trust hardware design* (§4.3). We then introduce an OS for this hardware, called OctopOS (§4.4).

4.3 Split-Trust Hardware

Modern machines leverage hardware with a *hierarchical privilege model*. That is, hardware provides multiple privilege levels, each with more privilege than previous ones, with one all-powerful level to "rule them all."[2] This model results inevitably in several complex components in the TCB such as the processor, protection hardware, and system software.

[2]A reference to Tolkien's The Lord's of the Rings.

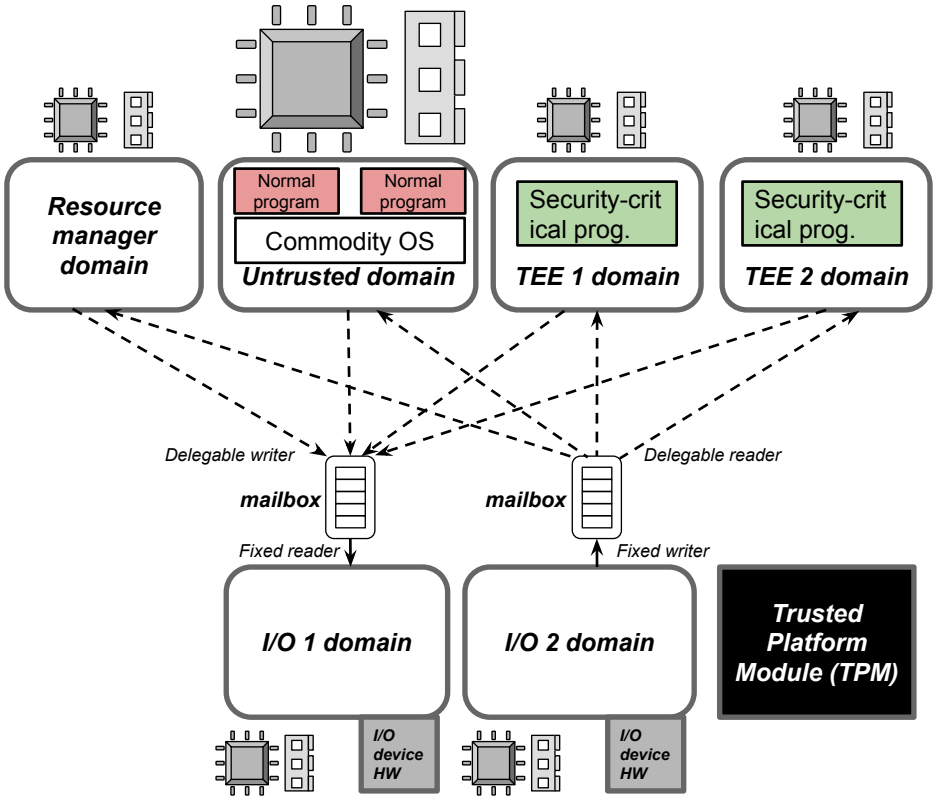


Figure 4.2: *Simplified overview of the split-trust hardware. The figure does not show all mailboxes for clarity.*

In this chapter, we demonstrate a novel hardware design, the *split-trust* hardware, in which the hardware is split into multiple isolated trust domains. Each domain is intended for one aspect of the machine: one or multiple for TEEs, one for each I/O device (i.e., an I/O domain), one for a commodity OS and its untrusted programs (i.e., the untrusted domain), and one for a resource manager, which is in charge of *constrained* resource scheduling and access control. The benefit of the split-trust hardware is that a security-critical program can *exclusively* take control of and use its own domain and *exclusively* communicate with other domains (§4.3.2), e.g., for I/O and IPC, hence significantly reducing the TCB. Figure 4.2 shows a simplified view of this hardware design. Next, we discuss its key aspects.

4.3.1 Physical Isolation and Static Partitioning

We follow two important principles in our hardware design. (1) Domains must be *physically isolated* (i.e., share no hardware components). (2) The isolation boundary between them cannot be programmatically and dynamically modified as *there is no trusted-by-all hardware or software component* to be tasked with that. This implies that we cannot rely on programmable protection hardware, such as an MMU, IOMMU, or address space controller, to enforce isolation. As a result, our design *statically partitions* the hardware resources between domains.

More specifically, each trust domain has its own processor. We use a powerful CPU for the untrusted domain, which accommodates a commodity OS and its (untrusted) programs, to achieve high performance. This CPU is similar to the powerful CPU used in modern smartphone SoCs. We use weaker microcontrollers for other domains in order to keep the hardware cost small. Each domain has its own memory as well and domains do not (and cannot) share memory.

Each I/O domain also has exclusive control of an I/O device, which is wired to and only programmable by the processor of that domain and which directly interrupts that processor. (We will discuss how DMA is handled in §4.3.5.)

4.3.2 Exclusive Inter-Domain Communication

To be able to act as one machine, the domains need to be able to communicate. We introduce a simple, yet powerful, hardware primitive for this purpose: *verifiably delegable hardware mailbox*. At its core, a mailbox is a hardware queue, allowing two domains (i.e., the writer and reader) to communicate through message passing.

The key novelty of our mailbox is how it enables exclusive communication using its *delegation*

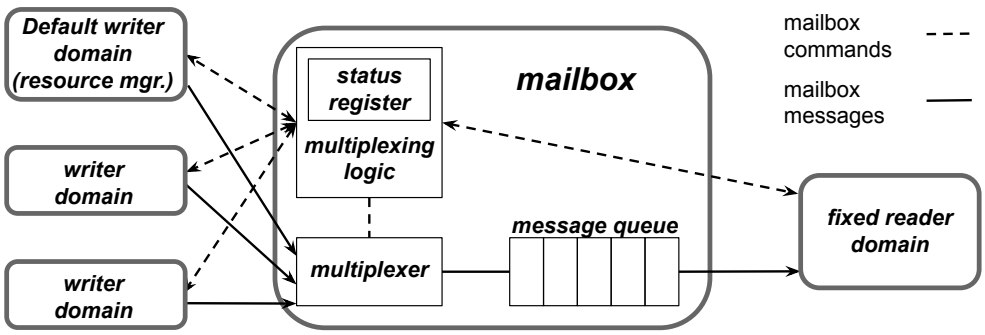


Figure 4.3: *Mailbox design.*

model. A mailbox has a fixed end (reader or writer) and a delegable one. The fixed end is hard-wired to a specific domain. The delegable one is wired to multiple domains, but only one can use it at a time, enforced by a hardware multiplexer within the mailbox. This end is by default (i.e., after a mailbox reset) under the control of the resource manager domain. But the resource manager can delegate it to another domain, which is then able to *exclusively* communicate with the domain on the fixed end of the mailbox.

Figure 4.3 shows the design of the mailbox with a fixed reader. For example, consider the serial output domain in our prototype. It is the fixed reader of a mailbox. Any domain with write access to the mailbox can (exclusively) send content to the output domain to be displayed in the terminal.

The delegation model of our mailbox has another important property: *limited yet irrevocable* delegation. When the resource manager delegates the mailbox to a domain, it sets a *quota* for the delegation in terms of both the maximum number of allowed messages and maximum delegation time. As long as the quota has not expired (i.e., *a session*), the domain can use the mailbox and the resource manager cannot revoke its access to the mailbox. The session expires when either the message limit or the time limit expires. (The message limit can be set to infinite, but not the time limit.)

This delegation model enables a limited form of availability, which we refer to as *session*

availability. That is, a domain with exclusive communication access to another domain can be sure to retain its access for a known period of time or number of messages. This is critical for some security guarantees on smartphones. For example, a security-critical program can ensure that the User Interface (UI) will not be hijacked or covered with overlays when the program is interacting with the user [130, 263]. Or a security-critical program that has authenticated to and hence unlocked a sensitive actuator domain (e.g., insulin pump) can ensure that no other program can hijack the session and manipulate the actuator. We leverage session availability in our own apps (§4.6).

As the resource manager is not trusted by other domains, the delegation must be *verifiable*. The mailbox hardware provides a facility for this verification. As Figure 4.3 shows, all domains connected to the mailbox can read a status register from the mailbox hardware. The status register specifies the domain that can read/write to the mailbox and the remaining quota. The domain with delegated access can therefore verify its access and quota. (Other domains will receive a dummy value when reading the status register for confidentiality.)

Domains transmit both commands and data to each other through mailboxes. Because commands are typically short but data messages are typically long, we use two types of mailboxes to optimize the hardware design, namely *control-plane mailboxes* and *data-plane mailboxes*. These two types of mailboxes share the same hardware properties, but have different sizes (i.e., message size and queue size).

4.3.3 Power Management

Our mailbox primitive cannot, on its own, guarantee session availability. This is because we need to ensure that during a session, the domains used by a security-critical program remain powered up (given adequate energy in the battery).

The Power Management Unit (PMU) normally takes commands from the resource manager. The resource manager uses this capability to reset other domains when needed, e.g., reset a TEE domain before running a new program, or apply Dynamic Voltage Frequency Scaling (DVFS) to manage the system's power consumption. (We do not support DVFS for the domains in our prototype. Hence, in the rest of the chapter, we mainly focus on the reset interface, although similar principles can be applied to DVFS.)

However, the resource manager is not a trusted component; hence it may try to reset a domain during a session. Therefore, we add a simple hardware component, called the *reset guard*, for controlling all the reset signals that are local to a domain, which ensures that as long as the quota on a mailbox has not expired, the domains on both sides of the mailbox (including the domain's mailboxes) cannot be reset, hence ensuring session availability. The resource manager simply fails to reset a domain if the domain has an ongoing delegation. Once the quota expires (or if the access to the delegable end of the mailbox is yielded), the mailbox is returned back to the resource manager, and the resource manager is allowed to reset and reuse the domains (assuming no other ongoing delegations).

4.3.4 Hardware Root of Trust

A hardware root of trust is needed during remote attestation to convince the party in charge of a security-critical program of the authenticity of the hardware and the correctness of the loaded program. We use a Trusted Platform Module (TPM) to realize the root of trust for the split-trust hardware.

Why TPM? TPM, as specified by the Trusted Computing Group (TCG), is a tamper-resistant security co-processor connected to the main processor over a bus [241]. Traditionally, it provides security features for the machine as a whole, such as the measurements of the loaded software. This makes TPM unsuitable for more fine-grained security features, such

as remote attestation of a specific program. As a result, in-processor TEE solutions, such as SGX, integrate the root of trust in the processor itself, tightly coupling it with various features of the processor (such as virtual memory and cache), further bloating the trusted processor.

Our key insight is that TPM can provide fine-grained security features for a split-trust machine since different components of this OS run on separate processors. This allows the machine to enjoy the security benefits of TPM without suffering from its main limitation.

To integrate TPM into a split-trust machine, we need a different set of parameters (i.e., the number of Platform Configuration Registers (PCRs) and their access permissions, i.e., localities) from the ones found in existing TPM chips, in order to provide one PCR per domain and securely extend it with the measurement of software loaded in the domain. The bootloader of a domain measures the boot image and extends the corresponding PCR with the measurement, and the PCR values are then used to provide a cryptographic proof of the software loaded into the domain (§4.4.1).

4.3.5 High Performance I/O

By default, the data plane of I/O domains are implemented over mailboxes. However, this raises a performance concern due to additional data copies (to and from mailbox). While the performance overhead is acceptable for TEE domains, it is not so for the untrusted domain. An important hardware primitive that enables a legacy machine to achieve high I/O performance is DMA. To safely use DMA in our machine, we introduce *domain-bound DMA*, defined with the following two restrictions. (1) The DMA engine is hard-wired to only read/write to the memory of the untrusted domain. (2) The DMA engine can stream data in/out of the I/O device only when the I/O domain is used by the untrusted domain.

We achieve this with a simple hardware component called the *arbiter*, which is a switch that decides if the data streams of the I/O device is connected to a DMA engine or to a simple FIFO queue accessible to the I/O domain.

4.3.6 Domain and Mailbox Reset

Domains and their mailboxes need to be reset before and after use (§4.2). We reset the mailboxes directly in hardware upon delegation, yield, and session expiration. We leave the resetting of the domains to the resource manager, albeit under the limitations enforced by the reset guard (§4.3.3). Even though the resource manager is untrusted, this does not pose a problem since a program can verify, using local and remote attestation through TPM as well as some measures provided by the domain runtime that (1) a domain has been reset, (2) it has not been used since last reset, (3) it will be reset after use and before use by other domains. We provide more details on the verification process with an example in §4.4.1.

4.4 OctopOS

We introduce OctopOS, an OS to manage the split-trust hardware. Unlike existing OSes, which have an all-powerful trusted-by-all kernel, OctopOS is composed of *mutually-distrustful components*. These components include I/O services for I/O domains, a runtime for TEE domains, a resource manager, and a compatibility-layer for the untrusted domain.

4.4.1 Fundamental Aspect

The fundamental aspect of OctopOS is that components *do not trust, but verify* any messages received from other untrusted components. We illustrate this aspect with one example.

Imagine a security-critical program that needs access to the input and output domains in order to interact with the user (e.g., to ask for username and password). The program, running in a TEE domain, sends a message to the resource manager and asks for the two domains to be delegated to it for a certain amount of time, e.g., one minute. More specifically, the program asks the resource manager to delegate the mailboxes of the input and output domain to the TEE domain. The resource manager waits for these domains to become available (if not at first), resets them, and then performs the delegation if it deems the request reasonable (e.g., if it is not for a very long period of time). It then responds to the TEE domain, confirming the successful delegation.

At this point, the security-critical program performs a series of verifications before it uses these domains. First, it uses the status register of the delegated mailboxes to verify that (1) its own domain is given exclusive access to the mailbox and (2) the delegation quota is correct (since otherwise the session might end abruptly, allowing the resource manager to hijack the program's interaction with the user). Second, the program needs to ensure that the right software has been loaded into the input and output domains and that the domains have been reset (otherwise the resource manager could install a keylogger/eavesdropper in these domains or simply inject code into them by exploiting their vulnerabilities). It performs the verification by checking the PCR value of each of the domains from the TPM. The PCR value provides a cryptographic proof of all the software loaded into a domain. Moreover, our I/O services further extend the PCR of their domain upon handling their first request. This way, the PCR value proves freshness (or lack thereof), i.e., that the domain has been reset prior to delegation.

Performing all these verifications on every interaction with other domains would be a daunting task, if it were to be done by the developer of a security-critical program. Therefore, OctopOS provides all of these for the developers in its components in the form of high-level API. We next discuss each of these components in more detail.

4.4.2 Components

I/O Services

Each I/O domain runs a service to manage it. The I/O service incorporates the software stack needed to program and use an I/O device, e.g., device driver. In addition, it provides an API that can be called (through messages) by any domain that has exclusive access to the mailboxes of the corresponding I/O domain.

There are two types of I/O devices. The first is *non-restricted I/O devices*. These are devices that can be used by a security-critical program without any restrictions during a session, such as the serial output and network devices in our prototype. For these devices, we ensure that the I/O domain is reset before and after use by another domain.

The second type is *restricted I/O devices*. These are devices that cannot be used freely by a security-critical program during a session and require the resource manager to enforce restrictions (i.e., fine-grained access control). In our prototype, storage is of this type since it contains data of other programs. Even if the data are encrypted, they need to be protected if a general availability guarantee is needed (§4.7.4). For these devices, we still ensure exclusive access to the domain during the session. We also ensure reset after use. However, we cannot ensure the domain is reset to a clean state before use. This is because after reset, the resource manager needs to communicate with the I/O service to restrict its usage, e.g., limit the storage domain to using only one partition allocated for a security-critical program, before delegating the domain's mailboxes to a TEE domain.

We have carefully designed an API for such I/O services. The core of the API revolves around the notion of an *I/O resource*. For example, in the case of the storage service, each partition is a resource. The API allows the manager to allocate resources and bind them to specific security-critical programs. It also allows the program to authenticate itself in order

to use the resource and to verify the status of the service. We omit the details of the API due to space limitation.

Finally, we note that this design adds the storage service to the TCB when it is used by a security-critical program (§4.7.4). In contrast, other I/O services are not directly reachable by the adversary when used exclusively by a TEE domain and hence are not part of the TCB.

TEE Runtime

In order to facilitate the development of security-critical programs, we have developed a runtime for TEEs, which provides a high-level API. A program may choose to utilize this runtime (which is part of the TCB), or its own.

We provide several categories of functions in this API: (1) Requesting and verifying access to other domains; this category also helps the program manage the remaining quota of mailboxes by calling a callback function upon quota updates, so that the program can decide whether to continue using the mailbox or not. It depends on the program's security goals to notify the user that the quota is about to expire. (2) High-level abstractions for using I/O services such as socket-based networking and terminal prints. (3) Assistance with the TPM, e.g., to request a remote attestation report. (4) Support for secure IPC between TEE domains. (5) Security-critical routines such as cryptographic primitives.

Resource Manager

At a high level, the resource manager is in charge of resource scheduling, access control, and system-wide, untrusted I/O functionalities. More specifically, it performs the following three tasks. First, it makes constrained scheduling decisions. When a new security-critical

Property	Proved theorems
Mailbox exclusive access	Domains without exclusive access to mailbox cannot change which domain has exclusive access, nor the remaining quota.
	If a domain does not yield its exclusive access, its exclusive access is guaranteed as long as the quota has not expired.
	The domain with exclusive access to the mailbox can correctly read or write from/to the queue.
	The domains without exclusive access to the mailbox cannot read/write to the queue.
Mailbox limited delegation	When given exclusive access, a domain cannot use the mailbox more than its delegated quota.
	When the quota delegated to a domain expires, the domain loses exclusive access.
Mailbox verifiable excl. access	The domain with exclusive access can correctly verify its exclusive access and remaining quota.
	The domain on fixed end of mailbox can correctly verify domain with exclusive access on the other end and remaining quota.
Mailbox default excl. access	After reset, the resource manager domain has exclusive access by default.
	The resource manager domain does not lose its exclusive access unless it delegates it.
	When a domain loses exclusive access (yield/expiration), the exclusive access will be given to the resource manager domain.
Mailbox confidentiality	Domains without exclusive access cannot use mailbox's verification interface to learn which domain has exclusive access and remaining quota.
	Upon delegation/yield/expiration, the data in the queue is wiped.
Reset Guard	The reset signal does not get forwarded if any other domain is using one of the domain's mailboxes.
	The reset signal does not get forwarded if the domain is using any of the other domain's mailboxes.
Arbiter control	The control interface can change its state between trusted and untrusted.
	Nothing other than the control interface can change the arbiter's state.
Arbiter excl. access	When an arbiter is connected to a trusted domain, a mailbox can correctly read or write data.
	When an arbiter is connected to an untrusted domain, a DMA engine can correctly read or write data.
ROM	A memory can be transformed into read-only access, a change that is irreversible.

Table 4.1: *Theorems we prove for our hardware components. Proving some of these require proving lemmas not listed here.*

program needs to execute, or when an existing one requests exclusive communication with another domain (for I/O or IPC), the manager checks the availability of resources, grants the request, or blocks it until the resource is available. Compared to schedulers in commodity OSes, scheduling in OctopOS is more restricted. This is because the resource manager cannot preempt a domain as long as mailbox quotas have not expired (§4.3.2). Second, the resource manager restricts the usage of some I/O domains to enforce fine-grained access control, as discussed in §4.4.2. Finally, the manager implements system-wide, untrusted I/O functionalities. For example, as the manager is the initial client of the input and output domains, it implements the shell (i.e., the UI). The UI, however, can be delegated to security-critical programs upon request.

Untrusted Domain's Compatibility Layer

In OctopOS, a commodity OS runs in the untrusted domain, and hence by definition manages its own processor and memory. (In contrast, OctopOS is in charge of managing all the domains and their interactions with each other.) Yet, the commodity OS is not given direct control of I/O devices as they are managed by separate I/O domains.

We address this issue by developing a compatibility layer for the untrusted OS. In our prototype, which uses PetaLinux, the compatibility layer consists of several kernel modules, each pretending to be a device driver. Transparent to Linux and its program, they communicate to the resource manager to get access to the I/O services' mailboxes (or to set up DMA) and then communicate to them. These Linux drivers can be used to run Android in the untrusted domain as well.

4.5 Prototype

We have built a prototype of the split-trust hardware and OctopOS on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. We use the Cortex A53 ARM processor on the SoC for the untrusted domain in order to achieve high performance for the commodity OS (PetaLinux) and its programs. We use the FPGA to synthesize 7 simple Microblaze microcontrollers (i.e., no MMU and no cache): two TEE domains, the resource manager domain, and four I/O domains (serial input, serial output, storage, and Gigabit Ethernet). (Note that we are limited to I/O devices in the development board and hence could not use more smartphone-specific I/O devices such as WiFi. However, our principles and approaches apply equally to these other I/O devices as well.) We leverage the (single-threaded) Standalone library [261] to program the microcontrollers. We use the entirety of the main memory for the untrusted domain. For other domains, we use a total of 3.2 MB of on-chip memory

including some ROM for bootloaders and some RAM. We run the TPM (emulator) [169] on a separate Raspberry Pi 4 board connected to the main board through serial ports. We use another Microblaze microcontroller to mediate the communications of the domains with the TPM.

In addition, we use the FPGA to synthesize the mailboxes (12 in total), the arbiter for DMA for the network domain (other domains do not support DMA), the reset guard, as well as 11 hardware queues for permanent domain connections (such as for all domains to communicate with TPM or for TEE domains to communicate with the resource manager). The control-plane mailboxes have the capacity of 4 messages of 64 B each, and the data-plane mailboxes have the capacity of 4 messages of 512 B each. As a concrete example, our storage domain has 4 mailboxes: two for its control plane (send/receive) and two for its data plane (send/receive).

As mentioned in §4.3.1, an I/O device is only programmable by its domain. This includes access to registers and receiving interrupts from the I/O device. In our prototype, we use I/O interrupts only for the network device and use polling for the rest. The interrupts to the network domain's microcontroller is from the FIFO queue that holds the packets and are only used when the domain serves a TEE domain (§4.3.5). When serving the untrusted domain, the domain-bound DMA engine directly interrupts the A53 processor on DMA completion.

We faced two noteworthy limitations in our prototype. First, while we have strived for our domains to share no hardware, currently, all our domains share the same clock source and our FPGA-based domains share the same power domain. Second, the on-board SD card reader and flash memory are directly programmable by the A53 processor and hence could not be used for the storage domain. Our solution was to connect a MicroSD card reader directly to FPGA through Pmod [146]. This provides physical isolation for the storage domain, but significantly degrades its performance due to Pmod's limited throughput. Therefore, for performance evaluation, we instead use DRAM as our storage (we partition out a chunk

of DRAM and use it exclusively for the storage domain). This allows us to stress the performance of the mailboxes of the storage domain and get an upper bound for our storage performance, which we cannot do with the Pmod prototype.

We note that requiring an FPGA board to experiment with our machine may pose a road block for many researchers. Therefore, we also develop an emulator for our hardware design. The emulator runs on a Linux-based host OS such as Ubuntu and is able to fully boot and run OctopOS.

Overall, we have implemented OctopOS and our hardware emulator in about 39k lines of C code (including 5k of modified drivers from Xilinx and crypto libraries). We report the LoC for our hardware below.

4.5.1 Verified Hardware Design

The split-trust hardware has only four simple hardware components that are part of the TCB (§4.7.4): mailbox, DMA arbiter, reset guard, and ROM (for bootloaders). We have implemented these components in 1630 lines of Verilog code as well as 800 lines of Python code.

The simplicity of our trusted hardware components enables us to formally verify them. We use SymbiYosys to perform formal verification [154]. SymbiYosys is a front-end for Yosys-based formal hardware verification flows. We took a pragmatic approach to infer 20 theorems (some comprising multiple lemmas) from our guarantees. Formal verification ensures that our hardware design satisfies these theorems and hence our guarantees. Indeed, we have discovered and fixed a delegation logic error during verification.

We use the SMTBMC engine, which uses k-induction to formally verify our hardware design against these theorems. Table 4.1 shows the list of theorems we prove for our hardware

```
reg init = 1;
always @(posedge clk) begin
  if (init) assume (!aresetn);
  if (aresetn) begin
    q_expired <=
      (remain_quota == 0) && (owner != `ID_RM);
    t_expired <=
      (remain_time == 0) && (owner != `ID_RM);
    if (t_expired || q_expired)
      assert (owner == `ID_RM);
  end
  init <= 0;
end
```

Figure 4.4: Simplified formal verification code for the theorem, "when the quota delegated to a domain expires, the domain loses exclusive access" (Table 4.1 Row 6).

components. Overall, we developed 3000 lines of SystemVerilog code for our hardware verification. We describe all the theorems in a separate document, which can be found in our hardware repository[3]. Below, we present one example.

Theorem example. As shown in Figure 4.4, we demonstrate the Verilog code (adjusted for readability) that we develop for verifying the theorem that "when the quota delegated to a domain expires, the domain loses exclusive access" (Table 4.1 Row 6). As specified by the pseudo-code below, the SMTBMC engine proves that on the rising edge of a clock cycle, when either the time limit or quota limit becomes zero, the new owner is determined to be the resource manager.

In lines 5-6, the "q_expired" register compares the remaining quota limit with zero, and in lines 7-8, the "t_expired" register compares the remaining time limit with zero. In both cases, the expired registers are not triggered if the current owner is the resource manager. Line 9 checks if the time limit or quota limit has expired, and if so, the new owner must be the resource manager.

[3] https://github.com/trusslab/octopos_hardware/raw/main/docs/OctopOS-TRM.pdf

4.6 Security-Critical Programs

We discuss two security-critical programs that we have built for our machine. These programs are simplified yet representative of real-world applications.

I. Secure banking. Our secure banking program allows a user to securely log in to their account and view their account balance. The program leverages several features of our machine. First, it uses exclusive access to the UI (i.e., shell) as well as our session availability guarantee to make sure all inputs come from the user (and not malware) and that outputs are only displayed to the user. On legacy machines that do not support session availability, it has been shown that user's interaction with a banking app can be hijacked or covered with overlays [130, 263, 97]. Upon getting exclusive access to the UI, the program needs to convince the user that they are interacting securely with the program. It does so by displaying a secret established *a priori* between the user and the bank. Moreover, the program utilizes the runtime APIs to monitor the quota left for the UI session, and prompts the user to stop interacting with the program if the quota is low.

Second, the program uses exclusive access to the network domain to transfer confidential information. One might wonder why it is not adequate to use a secure networking protocol, such as TLS, for this purpose. Such protocols leave open some side-channel attack vectors [259], which our exclusive network access closes against on-device attackers; external network side-channel attacks are still possible. Note that a secure networking protocol is still needed for protecting the data against adversaries outside our machine (although we have not incorporated such a protocol in our prototype yet).

Finally, the program uses remote attestation to enable the bank server to verify the integrity of the program running on the user's device before any sensitive account information is released or any commands are accepted. Specifically, (1) the server provides the program with a challenge (i.e., a nonce), and the program passes the challenge to the TPM, which

generates an attestation report. (2) The program sends the report to the server, which verifies it and then sends the expected PCR values of the I/O services to the program, (3) which then uses them for local attestation of I/O domains (including that of the network service).

II. Secure insulin pump. Diabetic patients need to administer insulin to control the glucose level in their blood. New glucose monitor and insulin pumps have recently emerged that can be programmed through a smartphone, although security concerns currently requires using a dedicated smartphone [242]. (We note that some patients use an open source, unofficial Android app [94] to control the pump, albeit at their own risk.) Our machine can enable the use of user's own smartphone to securely execute these life-critical tasks.

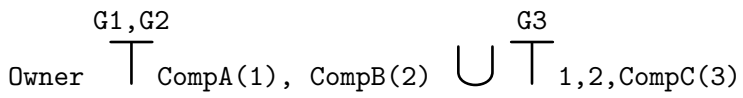
We build two versions of this security-critical program in our OS. The first version allows the user to directly program the insulin pump (in which case a glucose monitor is not used). The second version automatically reads the user's glucose level and uses that (and previous historical readings) to decide how much insulin to pump.

These programs leverage our session availability and exclusive access to the insulin pump (and the glucose monitor in the second version of the app), e.g., via Bluetooth or through the headphone jack. This way, the program can securely authenticate itself to these devices and not worry that the session may be hijacked. The program also uses exclusive access to the network domain to securely communicate with the health provider's server, which uses remote attestation to enable the provider's server to trust the program, similar to our secure banking program. Finally, the second version of this program needs to be executed in fixed intervals and store its sensor readings across sessions. This requires a stronger availability guarantee, called *general availability* (as opposed to the more limited session availability). For this, it trusts the resource manager and the storage domain, as discussed in §4.7.4. The first version does not need the additional trust since it only requires session availability.

4.7 TCB and Security Analysis

4.7.1 TCB Notation

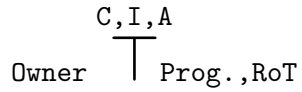
We introduce and use a simple, compact notation for TCB, discussed here with an abstract example:



The key operator is the $\top$ sign, which resembles a T (as in Trust). It helps denote *the set of (strongly-trusted) components in the TCB*. The elements on top of the $\top$ sign, e.g., `G1`, are the security guarantees, e.g., confidentiality and integrity. This allows for differentiating trust assumptions for different guarantees and combining them using the $\cup$ sign. The elements in front of the $\top$ sign are the trusted components. For succinctness, we tag a repeating component with a number in parenthesis on its first appearance and use the number in other locations.

4.7.2 Lower Bound of TCB

Assuming that the program communicates with the outside world, the lower bound can be achieved if the machine is dedicated to executing a security-critical program:



where C, I, A stand for Confidentiality, Integrity, and Availability. This shows that the owner at the very least needs to trust the (security-critical) program and the Root of Trust (RoT). The trust in the program is fundamental: the program needs to protect itself against adversarial inputs, e.g., malicious network packets. (This could imply trust in the network interface card. However, we assume that the network interface card is isolated by the program, e.g., using an IOMMU). The program in the TCB includes the runtime used by the program to interact with the hardware.

The trust in the RoT is also fundamental and stems from the fact that an adversary controlling the machine may try to fool the verifier of remote attestation by attempting to attack and compromise the RoT. The trust in the RoT includes trust in the bootloader, the ROM used to store the bootloader, the hardware/firmware used for remote attestation, e.g., TPM, as well as the hardware vendor that certifies attestation reports.

Finally, note that we do not consider the processor to be part of the TCB because the program can sanitize the adversarial inputs and prevent them from reaching the processor in a meaningful way.

4.7.3 TCB of Existing Systems

First, we consider a traditional system that uses an OS to provide isolation:

$$\texttt{Owner} \quad \overset{\texttt{C,I,A}}{\top} \quad \texttt{Prog.,OS,Proc.,Mem.,I/O,interconn.,P.HW,RoT}$$

This shows that the owner needs to trust the hardware including the processor, memory, I/O devices, protection hardware (`P.HW`) such as MMU and IOMMU, and interconnects. Moreover, the OS is also trusted, including device drivers. In this case, the program includes the libraries used by the program to interact with the OS and hardware.

Next, we write the TCB for a popular TEE solution for smartphones, TrustZone, in Formula 4.1. `SM` is the security monitor (i.e., the secure world OS and monitor code). We note that TrustZone allows the secure world to take full control of an I/O device, i.e., secure I/O (`Sec-I/O`). Yet, this device and its driver are exposed to multiple programs in the secure world and hence are trusted. Another noteworthy issue is that, in general, the OS is trusted when availability is needed as it is in charge of resource scheduling. However, in TrustZone, the secure world OS (part of the `SM` in the formula) can be configured to handle some of the interrupts and hence can control the availability of the corresponding resources [86].

4.7.4 Our TCB

Formula 4.2 shows the TCB of our machine. `As`, `Ag`, `SD`, and `RM` stand for session availability, general availability, storage domain, and resource manager, respectively. Our system requires trust in a few cases that were not part of the lower bound. First, for confidentiality, integrity, and session availability, the owner needs to trust the mailboxes used by the program, the arbiter (if domain-bound DMA is used), and the domain reset guard as these components interact with untrusted components. As discussed in §4.5.1, the simple design of these components allowed us to formally verify them, making this trust acceptable.

$$\texttt{Owner} \overset{\texttt{C,I}}{\top} \texttt{Prog.(1),SM(2),Processor(3),Mem.(4),Sec-I/O(5),interconn.(6),} \\ \texttt{P.HW(7),RoT(8)} \bigcup \overset{\texttt{A}}{\top}_{\texttt{1,2,3,4,5,6,7,8,OS}} \tag{4.1}$$

$$\texttt{Owner} \overset{\texttt{C,I,As}}{\top} \texttt{Prog.(1),mailbox(2),reset-guard(3),arbiter(4),RoT(5)} \bigcup \\ \overset{\texttt{Ag}}{\top}_{\texttt{1,2,3,4,5,RM,SD}} \tag{4.2}$$

Second, if a program needs general availability guarantees (e.g., it needs to be executed in fixed intervals) and needs to store data across sessions, it needs to trust the resource manager domain and the storage domain. The only way to eliminate the trust in the storage domain for general availability is to have separate storage devices for each security-critical program. Unfortunately, this is prohibitively expensive. Note that we assume that the program protects the confidentiality and integrity of its stored data using proper cryptographic primitives, although we have not implemented that in our prototype.

It is noteworthy that our machine eliminates the need to trust several complex hardware and software components such as the processor, memory, I/O devices, the interconnects (since our machine does not share any buses between trust domains) and system software (security monitor, OS, and device drivers), compared to existing TEEs. Overall, the TCB of our machine is significantly smaller than modern, popular TEEs. Moreover, our TCB is rather close to the lower bound. Achieving a smaller TCB for a machine that can host security-critical and untrusted programs concurrently would be challenging.

4.7.5 Security Analysis

Threat model. We assume an attacker can run malicious programs in the machine and tries to exploit any software or hardware vulnerabilities. We also assume that adversary can send malicious packets over the network to the machine. Below, we discuss various such attacks and their implications. Physical attacks are out of scope.

Software vulnerability-based exploits. Vulnerabilities in trusted software components would lead to attacks. An attacker that compromises the program can obviously change its behavior. An attacker that compromises the bootloader (including the code that cleans up the state in a domain upon reset) can falsify the remote attestation report or access/impact data from other sessions. An attacker that can compromise the storage service can delete the program's data. An attacker that can compromise the resource manager can starve the program of resources (but cannot impact the availability of a session once it is granted). An attacker that manages to compromise other software components, e.g., I/O services, other security-critical programs, and the untrusted OS, cannot mount an attack on the program.

Hardware vulnerability-based exploits. In a split-trust machine, unlike existing TEEs, vulnerabilities in many complex hardware components such as the processor cannot be exploited since the adversary never shares the underlying hardware with the security-critical program. Therefore, the attacker cannot leverage various hardware-based attacks such as cache side-channel attacks, interconnect side-channel attacks, speculative execution attacks, and Rowhammer attacks. Only vulnerabilities in the trusted hardware components (i.e., mailbox, arbiter, reset guard, ROM, and TPM) would lead to attacks. The first four are formally verified (§4.5.1) and TPM is a mature and secure technology.

Timing side-channel attacks. All trusted software and hardware components are vulnerable to timing side-channel attacks. In our machine, the only components that may expose useful timing channels are the TPM and the program runtime. Such attacks (and

others) have been demonstrated on TPMs before [171, 234, 125, 162, 199]. As TPM is a mature technology, vulnerabilities get fixed. Indeed, there have been several works that formally verify various aspects of the TPM standard [129, 230, 256]. We have not analyzed the timing channel of the runtime we have developed for security-critical programs.

Power management attacks. These types of attacks can induce faults in the victim program's execution by manipulating the frequency or voltage of the processor and have been demonstrated against TEEs [238, 219, 201]. As mentioned in §4.3.3, our machine does not allow power management of a domain in a session, and hence mitigates such attacks.

Power management data can also be used as a side channel. More specifically, an attacker may try to monitor the voltage and frequency of a domain (which changes according to DVFS) and use that as a side channel to extract secrets from a domain. We note that our current prototype is not vulnerable to this side channel since our TEE domains do not support DVFS. However, our hardware can support the use of DVFS-capable processors for TEE domains. In such a case, we will need to close this channel. To do so, we will need to ensure that the PMU does not leak any information about a domain to another domain. This can be done rather trivially within the PMU firmware, which should be formally verified and hardened.

Hertzbleed [252] turns a power side channel into a timing attack. We leave it to the program and its runtime to migitate such an attack.

Remote network attacks. Similar to a legacy machine, a security-critical program must protect itself against malicious network messages in our machine. However, our machine provides some protection against network attacks that target the network stack. This is because it sandboxes the network device and its device driver in its own domain. As a result, programs that do not use the network at the time of a exploit are protected from these attacks. This is in contrast to a legacy machine in which a single successful exploit of

the kernel-based network stack may result in a full takeover.

Out of scope: physical attacks. We assume that the adversary does not have physical access to the device. Therefore, we do not protect against physical attacks. However, if the program does not use any I/O devices, it can use on-chip computation and memory encryption to protect its secrets against physical attacks [133, 161, 269]. These are orthogonal to our design and hence can simply be added to our machine. However, we note that if a program uses I/O devices, no general solution can be used to prevent physical attacks. While storage and network devices can use encryption (i.e., full-disk encryption), other devices such as output devices, cameras, sensors, and actuators cannot be universally protected.

4.8 Evaluation

Our FPGA-based hardware implementation serves two purposes. First, we use it to estimate the hardware cost of our solution in terms of chip area. Second, it provides a bound on the performance impact of the solution. A deployed solution would likely replace the FPGA components with higher-performance non-reprogrammable ASIC elements, such as an integrated SoC or specialized chiplets [202].

However, despite the use of FPGA and weak microcontrollers for TEE and I/O domains, we show that security-critical programs can achieve decent performance, while normal programs can achieve the same compute and I/O performance as on a legacy machine.

4.8.1 Hardware Cost

We calculate an estimate for the number of transistors needed for our additional hardware components (all the components synthesized on the FPGA in our prototype). We calculate

FPGA resource	Count	Equivalent transistor count
Look-up table	69,999	2,519,964
Flip flop	63,188	1,516,512
Block RAM	27,061,649 (bits)	162,369,894

Table 4.2: *Extra hardware cost in our machine.*

this estimate by measuring the number of look-up tables, flip flops, and block RAMs used by our hardware and converting them to transistor count using the following estimates: 6 NAND gates per look-up table [216], 6 transistors per NAND gate [236], 24 transistors for each flip flop [232], and 6 transistor for each bit of on-chip memory (assuming a conventional 6-transistor SRAM cell [108]). Our calculation shows that our machine requires about 166.4 M additional transistors (162 M of which are used for on-chip memory). Table 4.2 shows the breakdown. This compares favorably with the number of transistors used in modern SoCs in smartphones. For example, Apple A15 Bionic and HiSilicon Kirin 9000 use 15 B transistors [229, 149]. This means that, if our solution is added to an SoC or implemented as a chiplet [202], the additional hardware cost would likely be 1-2%.

4.8.2 Performance

We measure various performance aspects of our machine. Note that all domains except the untrusted one use an FPGA with a 100 MHz clock. The Ethernet controller IP uses an external 50 MHz clock. Therefore, our results represent a lower bound on our machine's performance; we expect superior performance on ASIC. We repeat each experiment 5 times and report the average and standard deviation.

Mailbox performance. We measure the throughput and latency of communication over our mailbox. For throughput, we measure the time to send 10,000 messages of 512 B over a data-plane mailbox. For latency, we measure the round trip time to send a 64 B message and receive an acknowledgment over a control-plane mailbox. We perform these

Configuration	Throughput (MB/s)	Latency (μs)
A53-Microblaze	7.07±0	18.2±0
Microblaze-Microblaze	9.64±0.01	15.26±0.05

Table 4.3: *Mailbox performance.*

experiments in two configurations: one for communication between the hard-wired ARM Cortex A53 (the untrusted domain) and an FPGA-based Microblaze microcontroller, and one for communication between two FGPA-based Microblaze microcontrollers. Table 4.3 shows the results. One might wonder why the A53-Microblaze configuration achieves lower performance. We believe this is because this configuration requires the data to pass the FPGA boundary, hence passing through voltage level shifters and isolation blocks [262]. Moreover, the FPGA is in a different clock domain than A53.

Storage performance. We measure the performance of our storage domain, which uses the mailbox for its data plane (i.e., no DMA). To do so, we perform 2000 reads/writes of 512 B each. We evaluate three configurations: a best-case configuration where the storage domain directly performs reads/writes (hence giving us an upper bound on the DRAM-based storage performance), and two configurations where the untrusted domain or a TEE domain uses the storage service over the domain's mailboxes. Table 4.4 shows the results. They show that our mailbox-based storage domain can achieve decent performance (as can also be seen from our boot-time measurements reported below). It also shows that the additional copies caused by the mailbox add noticeable overhead compared to the best-case scenario. To further improve this performance for the untrusted domain, one can use domain-bound DMA for the storage domain.

Network performance. We measure the performance of our network domain, which uses domain-bound DMA for high performance for the untrusted domain (§4.3.5). We evaluate three configurations, similar to those used for storage experiments. For measuring the throughput for the baseline and the untrusted configurations, we use iPerf; for round-trip time (RTT) measurements, we use Ping. For the TEE configuration, we develop custom

Configuration	Read throughput (MB/s)	Write throughput (MB/s)
Best-case	8.13±0.00	6.10±0.00
Untrusted dom.	4.17±0.09	4.06±0.00
TEE domain	4.39±0.00	3.93±0.00

Table 4.4: *Storage performance.*

Configuration	Throughput (Mbit/s)	RTT (ms)
Baseline	943±0	0.17±0.01
Untrusted domain	943±0	0.17±0.02
TEE domain	0.567±0.001	23.92±0.02

Table 4.5: *Network performance.*

programs for measurements. For all experiments, we connect the board to a PC, which acts as a server. Table 4.5 shows the results. They show that our domain-bound DMA is capable of matching the performance of a legacy machine. Moreover, the network performance for a TEE is usable.

We believe, based on some tests that we have conducted, that it is possible to further improve the TEE network performance by about 10 X. This is because, currently in the network domain, we add an artificial delay between accessing the mailbox and the network IP, which limits performance. We do so to prevent data corruption, which according to our extensive investigation, is caused by a bug in the Ethernet AXI IP from Xilinx (potentially the bug discussed in [90]). Since the IP is closed source, we are not able to fix the bug.

Boot time and breakdown. We measure the boot time of our machine. All the boot images are transferred from the storage domain to their corresponding domains over mailboxes. Due to presence of multiple domains, booting OctopOS from a partition in the storage service is a carefully choreographed dance, requiring steps taken by bootloaders in each domain and the resource manager. Due to space limitations, we do not provide the details of the boot process, but measure and report it. Our measurements show that it takes 4.03±0.00 s to boot all domains excluding the untrusted domain, which takes an additional 8.65±0.01 s to boot.

Untrusted program performance. We use the network file system to evaluate the performance of an untrusted program. Our benchmark reads 100 files each containing 10,000 random numbers from a network file system, sorts them, and writes them back to the same file system. We choose this benchmark since it stresses CPU, memory, and network (for which we have domain-bound DMA). Our evaluation shows the benchmark takes the same amount of time (3.86±0.03 s) on our machine as on a legacy machine with the same A53 processor, RAM, and Gigabit Ethernet (3.84±0.04 s).

Security-critical program performance. We measure the execution time of two security-critical programs. In our experiments, we assume that no other domain needs and hence competes for the I/O domains. This allows for simple optimizations, e.g., proactively resetting the network domain.

The first program is secure banking (§4.6). We measure the time it takes to launch, including time needed to acquire keyboard, serial, and network, to perform attestation, and finally, to display prompts for user credentials. Our measurements show the overall execution time is 2.38±0.42 s.

We also develop and evaluate a more performance-intensive security-critical program. It reads a 1 MB file from the storage domain, computes its hash, and sends the hash over the network to a server. The overall execution time is 1.75±0.00 s. Looking at the breakdown, it takes 0.30±0.00 s to launch (including time needed to acquire exclusive access to storage and network, excluding local attestation through TPM), 0.22±0.00 s to read the file from storage, 1.21±0.00 s to compute the hash, and 0.01±0.00 s to send the hash over the network. To better assess this execution time, we write a normal program to perform similar tasks on a legacy machine with the A53 processor, RAM-FS, and Gigabit Ethernet. This program takes 0.23±0.00 s to execute.

If an I/O domain is in use when we run the security-critical program, there will be two types

of additional delay. First, our security-critical program needs to wait for the I/O domain to become available. Second, in the case of the network, the program needs to wait for the network domain to perform ICMP route discovery and other network protocols, which can take around 4.12±0.96 s in our current prototype (without any optimizations). But note the app can mitigate part of these delays by overlapping them with other parts of its execution.

Programming effort. We evaluate the programming effort for both types of developers. We report the programming effort required to develop a security-critical program on top of OctopOS. Currently, the runtime provides 49 APIs for the application developers to use. The secure banking program presented in §4.6 has 482 lines of code, which includes 58 lines for the main logic, 107 lines for the user interface, 207 lines for network communication (including attestation), and 93 lines for managing delegated resources. The secure insulin pump program (second version) has 563 lines of code, which includes 217 lines for the main logic, 200 lines for network communication (including attestation), and 128 lines for managing delegated resources.

The network domain has 7217 lines of code (including modified drivers from Xilinx). The storage, keyboard, and serial domain have 1091, 154, and 165 lines of code, respectively. These numbers exclude the domains' bootloaders and lower-level OctopOS code for hardware support, such as our mailbox driver, which an I/O service developer can reuse.

Impact of exclusive I/O use. We evaluate the impact of executing a security-critical program that uses storage on the storage performance of the untrusted domain. More specifically, we launch a security-critical program in a TEE that exclusively reads 1 MB from and writes 1 MB to storage, while the untrusted domain is reading a 100 MB file data (which normally takes 24.26±0.31 s to finish). Our measurements show that the security-critical program causes a 2.58±0.03 s gap where the untrusted domain cannot access the storage.

4.8.3 Energy Consumption

We report the estimated energy consumption of running security-critical programs on our hardware. We measure the actual execution time of each domain, and multiply the time by the per-domain power estimation. The estimation is obtained by running the power report program on our hardware design using the Xilinx Vivado software.

Our measurements show the energy consumption of all the domains involved in launching the banking program (including booting, initialization, requesting resources, and performing attestation) is 3.21±0.64 Joules.

We also measure the energy consumption of the other security-critical program that reads a 1 MB file, hashes it, and sends the hash over network. The energy consumption of all the domains that are involved is 2.03±0.42 Joules. In comparison, we measure the estimated energy consumption of the 1 MB file hashing experiment on the legacy machine. Xilinx Vivado software estimate the runtime energy of the A53 processor on the SoC of our FPGA board to be to be 2.74 watts (with no DVFS). We calculate the energy consumption of the program running on the legacy machine to be 0.63±0.00 Joules.

To provide a frame of reference, note that the overall amount of energy in a fully charged battery in a modern smartphone (i.e., Google Pixel 7) is 60517 Joules.

4.9 Thoughts on Scalability, Performance, and Usability

Scalability and Performance. The exclusive use of TEE domains limits the number of concurrent security-critical programs. Moreover, our choice of using weak microcontrollers, small amounts of memory, and I/O without DMA for TEEs limit the performance of security-

critical programs. We believe that the former is not a serious issue since we do not expect a large number of security-critical programs executing simultaneously in a smartphone.

The latter is mostly a non-issue either since security-critical programs are more concerned with security guarantees than performance. However, there are exceptions, for example, authentication of the user by applying machine learning algorithms to photos taken by the camera or privacy-preserving federated learning [197]. We believe that these programs can leverage accelerators (which will be available in the machine in the form of additional I/O domains). Indeed, Nider et al. propose a machine with no CPU and several self-managed devices [206], showing the diminished role of CPU for performance. We also note that our design allows for using more powerful processors for the TEE domains, albeit at the cost of additional hardware budget.

One might wonder whether we can use a single DMA engine to improve performance of an I/O device for all TEE domains. This is not feasible since domains' memories are physically separated. Instead, we can potentially use multiple domain-bound DMA engines, one for each TEE domain.

Usability. We argue that the exclusive use of hardware resources by security-critical programs in our machine does not cause usability problems for normal programs, for three reasons. First, security-critical programs in smartphones already use some I/O devices exclusively. For example, the UI (display and touchscreen) is used exclusively (e.g., when using TrustZone-based Protected Confirmation [73]) due to its small form factor.

Second, the performance impact on other I/O types, such as networking and storage, can be minimal when security-critical programs use short sessions, e.g., a few seconds. In §4.8.2, we experimentally demonstrate this impact for storage. Moreover, TCP network connection keepalives persist for tens of seconds. Further, since smartphone network connections are frequently dropped during handoffs, most widely used applications transparently re-establish

lost connections without user visible changes. Security-critical programs can be designed to initiate, use, and close their connections in a single session (a practice that we use in our own security-critical programs).

It is also possible to mitigate these issues using multiple I/O domains of the same type. For example, all smartphones have both WiFi and cellular network interfaces. One can imagine allowing normal programs to share and use one of these while security-critical programs use the other (through two separate I/O domains in our hardware).

Third, most security-critical programs rely on only a subset of the I/O domains. For example, our insulin pump program (second version) mainly requires access to its sensor and pump as well as a brief access to storage. While this program is running, all other I/O domains, e.g., network, UI, and even storage, can be used by normal programs.

We finally note that any attempt to allow simultaneous sharing of hardware resources will undoubtedly increase the TCB. For example, enabling multiple domains to render to the display simultaneously will require trusting the display domain in our machine.

Chapter 5

Related Work

5.1 Graphics and I/O Device Security

5.1.1 Graphics Security

Sugar enhances WebGL's security. It uses virtual GPUs available on modern Intel GPUs to fully sandbox the WebGL graphics stack all the way down to the GPU device driver. A similar approach can be used to safeguard the graphics stack used by apps. Unfortunately, mobile GPUs do not support virtualization. Therefore, in Milkomeda, we attempt to improve the mobile graphics security by leveraging existing software-based security checks in web browsers.

SchrodinText [100], VButton [186], and Truz-Droid [268] protect integrity or confidentiality of content shown on the mobile display. SchrodinText achieves this by modifying the OS graphics stack to perform most of the text rendering stages without access to the text to be displayed. It uses the hypervisor and ARM TrustZone secure world to display the text. VButton and Truz-Droid use the ARM TrustZone secure world to control the display and

touchscreen and use them to show content to the user securely, collect inputs, and verify them. In all of these systems, the OS is assumed to be untrusted whereas the user and the app are trusted. Unlike these systems, Milkomeda does not modify the existing OS graphics stack. It assumes that the OS is trusted but the app is not. It then safeguards the graphics stack against malicious apps.

AdSplit [231], AdDroid [211], and LayerCake [224] isolate the code used to render an embedded UI component, e.g., ads. Their goal is to protect the app from untrusted embeddings. These solutions, at a high level, are similar to Sugar and Milkomeda that renders various parts of the UI in isolation. However, Sugar focuses on GPU accelerated graphics and leverages GPU virtualization in its design, none of which is addressed in these systems. In Milkomeda, we protect the system from untrusted apps, which try to exploit the vulnerabilities in the GPU device driver.

GPU virtualization. Cells supports OS-level virtualization of mobile devices and supports secure sharing of the GPU between multiple virtual phones through virtualization [104]. Paradice paravirtualizes I/O devices, including GPU, using the UNIX device file boundary [102]. Sugar uses an existing GPU virtualization solution for secure GPU access by web apps.

5.1.2 Device Driver Vulnerabilities and Mitigations

The core of most vulnerabilities in the graphics stack is the GPU device driver. Device drivers are known to have many vulnerabilities, more than the rest of the kernel [132, 209, 271]. Other related work tries to mitigate vulnerabilities in device drivers. Microkernels execute the device drivers in user space daemons [141]. Microdriver [151] and Glider [103] move parts of the device drivers to user space. Nooks safeguard against faults in device drivers using lightweight protection domains in the kernel [237]. SafeDrive does so using language

techniques [273].

In Milkomeda, we target existing systems that unfortunately do not leverage the aforementioned mitigation techniques. Instead, our observation is that WebGL security checks have been successfully deployed. Therefore, we try to leverage these solutions that can mitigate the GPU device driver vulnerabilities without requiring any modifications to the device drivers themselves and hence are easily applicable to various platforms.

User space I/O. Sugar allows the user space web app process to directly use a vGPU and hence is related to all user space I/O solutions. For example, Arrakis [213] and IX [115] decouple the control and data planes of the networking and storage stacks in the operating system and run the data plane in the user space by leveraging virtualized I/O devices. However, unlike existing solutions, Sugar focuses on GPU and integrates with web browsers.

Application's direct access to hardware. The nonkernel gives applications direct access to devices [116]. Combined with GPU virtualization, the nonkernel can be used to assign vGPUs to different applications. However, the nonkernel will not be able to effectively assign vGPUs to web applications without support in the browser, as in Sugar. Dune [114] gives applications direct access to virtualization hardware extensions. Similarly, Sugar gives an application direct access to a vGPU.

Alternative device driver designs. A main source of security concern with GPU access in the browser is the GPU kernel device driver's vulnerabilities. There are solutions that improve the device driver's risk on the system security and hence are related to Sugar. For example, microkernels move the device driver to the user space [141, 177, 145, 155, 223]. SUD [120], Microdriver [151], and Glider [103] move either part or all of the driver to the user space. Indeed, SUD and Glider use UML to achieves this, similar to Sugar (§2.3.2). LeVasseur et al. [179] move the device driver to a virtual machine for better isolation. Moreover, Nooks [237] and SafeDrive [273] keep the driver in the kernel but protect against its vulner-

abilities using runtime and language solutions, respectively. In contrast, Sugar is the first to run a full vGPU device driver as a library and show that it can be effectively integrated with web apps within the web browser.

5.2 Access Control and Isolation

5.2.1 OS-level Access Control

Milkomeda employs a light-weight syscall filtering mechanism to limit the process's access to the GPU device driver to only the code within the shield space. This is a form of access control enforced by the OS. Initial related work started with system call vetting based on `ptrace` but quickly moved towards a kernel-level caching mechanism [217]. AppArmor [75] enforces a configurable system call policy on a per-process basis. SELinux [60] hardens kernel and user-space and restricts interactions between processes and the kernel without enforcing an explicit system call policy. Capsicum [253] enforces capabilities on a per-process basis for Unix systems. Seccomp is an efficient, kernel-based vetting mechanism that evolved out of all these proposed systems and enables per-process system call vetting [227]. These systems are restricted to per-process checks with some context of the application. In contrast, our access control mechanism enforces a policy for a subset of code in the process address space.

CASE enforces isolation between modules of a mobile app [274]. CASE's approach can be used to isolate some libraries within the process. However, on its own, CASE is not able to restrict access to the GPU device driver to only a subset of the code. Moreover, CASE leverages information hiding to conceal the handlers of these modules and hence prevent jump to arbitrary locations within the modules. In contrast, Milkomeda leverages a hardware-protected shield space to achieve this.

5.2.2 Process-Level and Thread-Level Partitioning

Several related work evaluates process-level partitioning at different levels of granularity. Related work primarily focuses on separation policies and inference of a separation policy, not the separation enforcement mechanism. Provos et al. [218] provide a case study on how to break the OpenSSH server into smaller protected components (similar to how QMail compares to sendmail). Privtrans [123] automates the privilege separation process through an inference process. Wedge [118] extends Privtrans with capabilities while Salus [235] provides dynamically adjustable enforcement policies. Dune [114] leverages VT-x extensions to reduce separation overhead on per-page basis, improving performance of separation mechanisms. All these mechanisms share the limitation that they cannot handle multiple threads in a single compartment.

Recently, process-level partitioning has been extended with thread-awareness. Arbiter [251] provides fine-grained synchronization of memory spaces between threads but incurs prohibitive overhead. SMV [167] leverages a page-based separation scheme to enable fast compartment switching on a per-thread basis and provides a fine-grained API.

Light-weight Contexts [191] create independent protection units within a process. SandTrap uses two sets of page tables for a process to provide different address spaces for its threads [222]. In contrast, Milkomeda's shield space provides a protected space for graphics code to execute and limits the process' access to the GPU device driver to only this space. While the shield space share some underlying techniques with these systems (e.g., using a syscall to change the address space and using separate page tables for a process), shield is specialized and designed for enforcing graphics security check integrity. Specifically, using two first-level page tables to efficiently implement an in-process shield space and enabling it to securely control and vet the accesses of threads to the GPU driver is the novelty of the shield's design. IMIX provides hardware support for in-process memory isolation [147]. In

contrast, Milkomeda's shield space is designed for existing hardware.

5.2.3 Browser Security and Web App Isolation.

Other solutions have attempted to protect the browser and the system against an untrusted web app, e.g., by sandboxing it inside a picoprocess [166], in an exokernel browser [196], inside a virtual machine [135, 43], or by reducing the TCB of the browser [239]. Moreover, Xax [140] and Native Client [267] enable secure execution of native code in the browser using hardware protection mechanisms and software fault isolation, respectively. These solutions are orthogonal to our work, which focuses on secure GPU acceleration for web apps.

5.2.4 Other Mitigations

Control-Flow Hijacking Mitigation. In Milkomeda, we protect the control flow of the execution of the security checks by running them in an isolated shield space. An orthogonal approach to protect the control flow inside a process is control-flow integrity (CFI) [98, 124]. CFI restricts control flow through indirect control flow transfers to well known and valid targets, prohibiting calls to unaligned instructions or indirect function calls to invalid targets. The set of allowed targets depends on the underlying analysis but is at least the set of valid functions. Even the most basic CFI policy protects against an attacker hijacking the control flow past the check at the beginning of a function. While most existing CFI mechanisms are static and the set of valid targets is tied solely to the code location, some recent CFI mechanisms embrace context sensitivity. PathArmor [245] and PittyPat [139] track path constraints, increasing precision of CFI mechanisms to path awareness. Protecting applications against control-flow hijacking is orthogonal to separating two execution contexts. CFI ensures that bugs inside a context cannot compromise control flow, while Milkomeda protects a privileged kernel component by leveraging existing security checks from a different

domain.

Fault Isolation. Fault isolation restricts interactions between (at least) two compartments in a single address space. Software fault isolation [250] and Native Client [267] leverage binary rewriting and restrictions on binary code to separate compartments and control interactions. MemTrace [210] executes x86 programs and additional security checks in an x86_64 process, protecting checks and metadata by moving them past the 32-bit address space of the original program. Limitations of these existing solutions are performance overhead and the need of *a priori* rewriting and verification to ensure the encapsulation along with restrictions on the address space. Milkomeda is oblivious to the unprotected compartment and shield simply places a secure compartment inside the untrusted process and controls interactions between the untrusted part of the process and the trusted component.

Instead of using a software-based mechanism, hardware-based fault isolation enables separation at low performance overhead. The early work on flicker [194] leverages a Trusted Platform Module (TPM) chip to enforce strong isolation. TrustVisor [193] increases the TCB by moving from the TPM chip to the hypervisor and leveraging a software TPM to minimize overhead. Several architectures such as Loki [270], CODOM [249], or CHERI [258] leverage some form of tagged memory to enforce strong separation and isolation at low overhead by overhauling the underlying memory architecture. All these systems share that they require heavy hardware changes. Milkomeda is geared towards existing hardware and does not need any new CPU or memory features.

Milkomeda is also related to solutions that sandbox untrusted code. For example, Boxify [110] and PREC [164] sandbox Android apps and Native Client sandboxes native code in the Chrome web browser [267]. In contrast, Milkomeda protects a vetting layer from an untrusted app within its own process.

Library operating systems and other sandboxes. Library operating systems, such

as Exokernel [142] and Drawbridge [215], improve the system security by executing the operating system management components as a library in the application's process address space. Indeed, Sugar can be thought of as an exokernel design for GPU acceleration and hence is complementary to this line of work. Haven uses Intel Software Guard Extension (SGX) to protect an application from the untrusted cloud, and uses a library OS in the enclave. While Sugar uses a library OS-like architecture, it cannot protect the web app from an untrusted system.

5.3 Trusted Computing

5.3.1 Physical Isolation

Physical isolation and static partitioning. Notary [107] safeguards approval transactions by running its agent on a separate SoC from the ones running the kernel and the communication stack. Split-trust hardware shares the idea of using physically-isolated trust domains and also resets the domains before and after use by other programs. In contrast, our approach show how to safely mediate access to shared I/O devices for a workload of concurrent security-critical and untrusted programs.

Likewise, I/O-Devices-as-a-Service (IDaaS) suggests that I/O devices should have their own separate microcontrollers (and observes that they often do) and advocates for hardening their interfaces against potentially malicious kernel behavior [101]. Split-trust hardware also uses separate I/O microcontrollers but does not require trust in the microcontroller software, by resetting the I/O domain between uses.

Physical isolation and dynamic partitioning. IRONHIDE introduces dynamic spatial partitioning of processor cores and their communication channels to form isolated en-

claves [207]. Non-virtualized composable microprocessor [95] proposes a new server architecture that dynamically partitions CPU cores, memory, and accelerators. In contrast, we statically partition the hardware resources, resulting in a simpler design and a smaller TCB (i.e. no security monitor).

5.3.2 Exclusive Use

Flicker [195] uses the late launch feature of Intel Trusted Execution Technology (TXT) [150], to exclusively run a program on the processor. The exclusive use of the hardware results in minimizing the trusted components. However, Flicker's design requires stopping all other programs (including untrusted ones) when running a security-critical program. Our approach can run untrusted programs and security-critical programs concurrently (albeit with the limitation that I/O domains cannot be shared). Consider our secure insulin pump program (§4.6), which might need to be run frequently while the user is actively doing other, less security-critical, tasks on the main processor. Realizing this in Flicker can result in significant disruptions to other programs and to the user as a result.

5.3.3 Time Protection

Ge et al. add time protection to seL4, which closes many of the available side channels in commodity processors [152]. As Chapter 4 mentions, some processors do not provide mechanisms needed to close channels. Moreover, channels using busses could not be closed, and they have recently been shown to be effectively exploitable [208]. Our approach in the split-trust hardware of using completely separate hardware for security-critical programs addresses these concerns for these programs. We do, however, note that our approach (as it stands) does not scale to support all (normal) programs, which may have their own security needs. Therefore, we believe that time protection remains an important abstraction to be

explored for when the same processor is asked to host multiple programs.

5.3.4 Trusted Execution Environment

Secure I/O for TEEs. SGXIO uses a hypervisor and a TPM to create a trusted path for an SGX enclave to access an I/O device [255]. The solution requires the enclave program not only to trust SGX's firmware and hardware, but also the hypervisor. CURE [111] adds a few hardware primitives in order to allow the security monitor to assign a peripheral (i.e., access to MMIO registers and DMA target addresses) to an enclave. These primitives are designed to be programmed by a trusted-by-all security monitor (unlike the split-trust hardware).

Other TEE solutions. Komodo is a verified security monitor that can create enclaves for security-critical programs [144]. Li et al. formally verify the firmware in Realms, part of ARM confidential computing [187]. Use of formal verification warrants the strong trust in the security monitor/firmware, but not the ARM processor that hosts both security-critical and untrusted programs. For example, Li et al. mention that "[p]rotection against known software error injection attacks and side-channel attacks require appropriate usage of architectural mitigations and are beyond the scope of this paper."

Sanctum uses hardware modifications to RISC-V alongside a software security monitor to create isolated enclaves [134]. Compared to SGX, Sanctum enclaves are protected against both cache and page fault side-channel attacks. MI6 time-partitions hardware resources and implements a rigorous "purge" operation that erases microarchitectural and memory states associated with a security-sensitive program [119]. None, however, addresses other potential hardware vulnerabilities such as interconnect side channels.

SANCTUARY leverages the Address Space Controller hardware to enable strong isolation in TrustZone's normal world [121]. SANCTUARY still requires a security monitor to program the controller.

Chapter 6

Conclusions

This dissertation presents three system solutions that enhance system security. Sugar and Milkomeda mitigate security hazards in the graphics stack. Split-trust hardware/OctopOS is a new hardware and software design that provides isolation at the lowest possible layer. We showed that our systems effectively enhance system security by utilizing existing hardware features and designing new hardware and software. Our systems achieve good performance and strong security guarantees with reasonable performance overhead.

First, we presented *Sugar*, a system solution for enhancing the security of GPU acceleration for web apps. Sugar leverages modern GPU virtualization to provide web apps with a dedicated and isolated virtual graphics plane. We showed that Sugar reduces the TCB exposed to web apps and that it eliminates various vulnerabilities already reported in the WebGL framework. Furthermore, our extensive evaluation showed that Sugar's performance is high, providing similar user-visible performance with existing less secure systems.

Second, we presented *Milkomeda*, a system solution to protect the mobile graphics interface against exploits. We showed, through a study, that the mobile graphics interface exposes a large amount of vulnerable kernel code to potentially malicious mobile apps. Yet, mobile

apps' access to the OpenGL ES interface is not vetted. Browser vendors have invested significant effort to develop a comprehensive set of security checks to vet calls for the WebGL API. Milkomeda repurposes the existing WebGL security checks to harden the security of the mobile graphics interface. Moreover, it does so with almost no engineering effort by implementing a tool, CheckGen, which automates the porting of these checks to be used for mobile graphics. We also introduced a novel shield space design that allows us to securely deploy these checks in the app's process address space for better performance. Our evaluation showed that Milkomeda achieves high graphics performance for various mobile apps, although at the cost of moderately increased CPU utilization.

Finally, we presented *Split-trust hardware/OctopOS*, which minimizes the TCB when executing security-critical programs. Smartphone owners expect to use their devices for a mixture of security-critical and ordinary tasks, yet this requires trust that the hardware and system software is able to isolate those tasks from each other, trust that is often misplaced. We presented a hardware design with multiple statically-partitioned, physically-isolated trust domains, coordinated using a few simple, formally-verified hardware components, along with OctopOS, an OS to manage this hardware. We described a complete prototype implemented on a CPU-FPGA board and showed that it incurs a small hardware cost. For security-critical programs, our machine significantly reduces the TCB compared to existing solutions, and achieves usable performance. For normal programs, it achieves similar performance to a legacy machine.

Bibliography

[1] Chromium issue 63617: Closing multiple WebGL tabs at the same time causes segfault in Xorg. https://bugs.chromium.org/p/chromium/issues/detail?id=63617, 2010.

[2] Chromium Issue 70718: Crashes when opening a page with webgl. https://bugs.chromium.org/p/chromium/issues/detail?id=70718, 2011.

[3] Chromium issue 83841: User information leakage esp local paths, username in webgl getProgramInfoLog. https://bugs.chromium.org/p/chromium/issues/detail?id=83841, 2011.

[4] CVE-2011-2366: Timing attack steals cross-domain images (Firefox). https://nvd.nist.gov/vuln/detail/CVE-2011-2366, 2011.

[5] CVE-2011-2367: Read of GPU memory through Firefox WebGL. https://nvd.nist.gov/vuln/detail/CVE-2011-2367, 2011.

[6] CVE-2011-2599: Timing attack steals cross-domain images (Chrome). https://nvd.nist.gov/vuln/detail/CVE-2011-2599, 2011.

[7] CVE-2011-2601: The GPU support functionality in Mac OS X does not properly restrict rendering time, which allows remote attackers to cause a denial of service. https://nvd.nist.gov/vuln/detail/CVE-2011-2601, 2011.

[8] CVE-2011-2784: Chrome WebGL reveals local path in logs. https://nvd.nist.gov/vuln/detail/CVE-2011-2784, 2011.

[9] CVE-2011-3653: Read of cross-origin image through Firefox WebGL. https://nvd.nist.gov/vuln/detail/CVE-2011-3653, 2011.

[10] Firefox bug 655987 - Respond to the WebGL cross-domain image theft vulnerability. https://bugzilla.mozilla.org/show_bug.cgi?id=655987, 2011.

[11] Firefox bug 656752: WebGL crash in gleRunVertexSubmitImmediate. https://bugzilla.mozilla.org/show_bug.cgi?id=656752, 2011.

[12] Firefox bug 659349: WebGL allows access to uninitialized graphics memory. https://bugzilla.mozilla.org/show_bug.cgi?id=659349, 2011.

[13] Firefox bug 684882 - Random video memory grabbed into WebGL cube map textures on Mac OS, including on 10.7.1, on Intel GPUs. https://bugzilla.mozilla.org/show_bug.cgi?id=684882, 2011.

[14] Microsoft considers WebGL harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx, 2011.

[15] WebGL - More WebGL Security Flaws. http://www.contextis.com/resources/blog/webgl-more-webgl-security-flaws/, 2011.

[16] Chromium issue 145544: Security: integer overflow in gpu process with webgl. https://bugs.chromium.org/p/chromium/issues/detail?id=145544, 2012.

[17] Chromium issue 149904: Security: webgl - after running out of memory, buffer can still be written. https://bugs.chromium.org/p/chromium/issues/detail?id=149904, 2012.

[18] Chromium issue 153469: Security: Nvidia Kernel Panic. https://bugs.chromium.org/p/chromium/issues/detail?id=153469, 2012.

[19] CVE-2012-2896: Integer overflow in Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2012-2896, 2012.

[20] CVE-2012-5115: Bug in graphics drivers allows for "wild writes" in Chrome. https://nvd.nist.gov/vuln/detail/CVE-2012-5115, 2012.

[21] Chromium Issue 237611: Security: Screen capture via WebGL texture. https://bugs.chromium.org/p/chromium/issues/detail?id=237611, 2013.

[22] CVE-2013-2874: Read of screen data through Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2013-2874, 2013.

[23] NVIDIA GRID K1 and K2 Graphics-Accelerated Virtual Desktops and Applications. NVIDIA White Paper, 2013.

[24] Chromium Issue 376951: Security: webgl draw buffers extension can expose uninitialized video memory to webpage. https://bugs.chromium.org/p/chromium/issues/detail?id=376951, 2014.

[25] CVE-2014-1502: Bug in Firefox WebGL allows for rendering cross-domain content. https://nvd.nist.gov/vuln/detail/CVE-2014-1502, 2014.

[26] CVE-2014-1556: Crafted WebGL content constructed with Cesium JavaScript library allows for arbitrary code execution. https://nvd.nist.gov/vuln/detail/CVE-2014-1556, 2014.

[27] CVE-2014-3173: Read of uninitialized memory in Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2014-3173, 2014.

[28] Firefox bug 1028891: WebGL app crashes Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1028891, 2014.

[29] Firefox bug 972622 - WebGL.compressedTex(Sub)Image2D doesn't call MakeCurrent. https://bugzilla.mozilla.org/show_bug.cgi?id=972622, 2014.

[30] GPU Accelerated Compositing in Chrome. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome, 2014.

[31] Chromium issue 483877: Bad shader can cause kernel crash. https://bugs.chromium.org/p/chromium/issues/detail?id=483877, 2015.

[32] Chromium Issue 521588: Security: leaking previous webpage through webGL canvas preserveDrawingbuffer and scissor. https://bugs.chromium.org/p/chromium/issues/detail?id=521588, 2015.

[33] CVE-2015-7179: Incorrect allocation of memory allows attackers to execute arbitrary code or cause a denial of service. https://nvd.nist.gov/vuln/detail/CVE-2015-7179, 2015.

[34] Firefox bug 1190526 - Overflow in VertexBufferInterface::reserveVertexSpace causes memory-safety bug. https://bugzilla.mozilla.org/show_bug.cgi?id=1190526, 2015.

[35] WebGL* in Chromium*: Behind the scenes. https://software.intel.com/en-us/articles/webgl-in-chromium-behind-the-scenes, 2015.

[36] A Mesa fix lands for the Radeon R9 290 issue. https://www.phoronix.com/scan.php?page=news_item&px=DRI3-Mesa-Fix-Gears-290, 2016.

[37] AMD Multiuser GPU: Hardware-Enabled GPU Virtualization for a True Workstation Experience. AMD White Paper, 2016.

[38] Chromium Issue 593680: WebGL test "temp expressions should not crash" freezes browser. https://bugs.chromium.org/p/chromium/issues/detail?id=593680, 2016.

[39] [iGVT-g] [ANNOUNCE] 2016-Q3 release of KVMGT. https://lists.01.org/pipermail/igvt-g/2016-November/000976.html, 2016.

[40] iGVT-g Setup Guide. https://github.com/01org/Igvtg-kernel/blob/2016q3-4.3.0/iGVT-g_Setup_Guide.txt, 2016.

[41] Radeon R9 290 performing poorly with Mesa 12.1-dev and Linux 4.7. https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.7-R9-290-Regression, 2016.

[42] Unity and Facebook Collaborate on WebGL Gaming. https://developers.facebook.com/blog/post/2016/08/18/FB-Unity-Alpha, 2016.

[43] Windows Defender Application Guard for Microsoft Edge. https://blogs.windows.com/msedgedev/2016/09/27/application-guard-microsoft-edge/#mBwrd1ATV1aluMyd.97, 2016.

[44] A new multi-process model for Firefox. https://hacks.mozilla.org/2017/06/firefox-54-e10s-webextension-apis-css-clip-path/, 2017.

[45] Alexa Top 500 Global Sites. http://www.alexa.com/topsites, 2017.

[46] Apple macOS Sierra. https://www.apple.com/macos/sierra, 2017.

[47] Baidu Map. http://map.baidu.com, 2017.

[48] Best Practices for Working with Vertex Data. https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html, 2017.

[49] Certain types of loops in WebGL shaders cause GLSL compiler crashes on Adreno. https://bugs.chromium.org/p/chromium/issues/detail?id=784817, 2017.

[50] Chrome Issues. https://bugs.chromium.org/p/chromium/issues/list, 2017.

[51] Chromium Issue 682020: Security: WebGL - Use After Free in Buffer11::updateBufferStorage(). https://bugs.chromium.org/p/chromium/issues/detail?id=682020, 2017.

[52] CVE-2017-5031: Use after free in Chrome ANGLE. https://nvd.nist.gov/vuln/detail/CVE-2017-5031, 2017.

[53] Google Maps. https://www.google.com/maps, 2017.

[54] Linux dma-buf. https://www.kernel.org/doc/html/v4.10/driver-api/dma-buf.html, 2017.

[55] Microsoft Edge TestDrive demos. https://developer.microsoft.com/en-us/microsoft-edge/testdrive/tags/webgl, 2017.

[56] NASA Experience Curiosity. https://eyes.nasa.gov/curiosity, 2017.

[57] National Vulnerability Database. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd, 2017.

[58] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov, 2017.

[59] OpenGL ES Benchmark 1. https://github.com/googlesamples/android-ndk/tree/master/hello-gl2, 2017.

[60] SELinux. https://wiki.centos.org/HowTos/SELinux, 2017.

[61] The Chromium Projects: GN build configuration. https://www.chromium.org/developers/gn-build-configuration, 2017.

[62] The Common Vulnerability Scoring System version 2. https://www.first.org/cvss/v2/, 2017.

[63] Thingiverse Customizer. https://www.thingiverse.com/customizer, 2017.

[64] Unity WaveShooter OpenGL benchmark. https://github.com/unity3d-jp/WaveShooter, 2017.

[65] WebGL Animometer benchmark. http://kenrussell.github.io/webgl-animometer/Animometer/tests/3d/webgl.html, 2017.

[66] WebGL Blob benchmark. http://webglsamples.org/blob/blob.html, 2017.

[67] WebGL Cubemap benchmark. http://webglsamples.org/dynamic-cubemap/dynamic-cubemap.html, 2017.

[68] WebGL Many-Planets benchmark. http://www.khronos.org/registry/webgl/sdk/demos/webkit/ManyPlanetsDeep.html, 2017.

[69] WebGL San-Angeles benchmark. http://www.khronos.org/registry/webgl/sdk/demos/google/san-angeles/index.html, 2017.

[70] WebGL Security. http://www.khronos.org/webgl/security/, 2017.

[71] WebGL Statistics. http://webglstats.com, 2017.

[72] Android NDK. https://developer.android.com/ndk/index.html, 2018.

[73] Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html, 2018.

[74] ANGLE: From OpenGL to Direct3D and back again. https://docs.google.com/presentation/d/1CucIsdGVDmdTWRUbg68IxLE5jXwCb2y1E9YVhQo0thg/pub?slide=id.g26efd2cf6_0178, 2018.

[75] AppArmor. https://wiki.ubuntu.com/AppArmor, 2018.

[76] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[77] Client-Side Vertex Arrays. https://www.khronos.org/opengl/wiki/Client-Side_Vertex_Arrays, 2018.

[78] Drive-by Rowhammer attack using on Android. https://arstechnica.com/information-technology/2018/05/drive-by-rowhammer-attack-uses-gpu-to-compromise-an-android-phone/, 2018.

[79] glTexImage2D specification – OpenGL ES 3.0. https://www.khronos.org/registry/OpenGL-Refpages/es3.0/html/glTexImage2D.xhtml, 2018.

[80] Google Play Instant. https://developer.android.com/topic/google-play-instant/, 2018.

[81] GPU Command Buffer - The Chromium Projects. https://www.chromium.org/developers/design-documents/gpu-command-buffer, 2018.

[82] Intel with Radeon Graphics. https://www.anandtech.com/show/12220/how-to-make-8th-gen-more-complex-intel-core-with-radeon-rx-vega-m-graphics-launched, 2018.

[83] OpenGL ES Benchmark 2. https://github.com/googlesamples/android-ndk/tree/master/gles3jni, 2018.

[84] OpenGL ES Benchmark 3. https://github.com/learnopengles/Learn-OpenGLES-Tutorials (Lesson 5), 2018.

[85] OpenGL ES Benchmark 4. https://github.com/learnopengles/Learn-OpenGLES-Tutorials (Lesson 7), 2018.

[86] Arm CoreLink GIC-600 Generic Interrupt Controller Technical Reference Manual. https://developer.arm.com/documentation/100336/0106/operation/security, 2019.

[87] Apple announces first states signed up to adopt driver's licenses and state IDs in Apple Wallet. https://apple.co/3RT9ETU, 2021.

[88] Apple Platform Security - Secure Enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web, 2021.

[89] Apple Platform Security - Touch ID and Face ID security. https://support.apple.com/guide/security/touch-id-and-face-id-security-sec067eb0c9e/web, 2021.

[90] Common AXI Themes on Xilinx's Forum (see Section "Out-of-protocol designs" for the discussion on a bug in Xilinx's Ethernet-lite controller). https://zipcpu.com/blog/2021/03/20/xilinx-forums.html, 2021.

[91] CVE Details. Op-tee: Vulnerability Statistics. https://www.cvedetails.com/product/56969/Linaro-Op-tee.html, https://www.cvedetails.com/product/42749/Linaro-Op-tee.html, https://www.cvedetails.com/product/36161/Op-tee-Op-tee-Os.html, 2021.

[92] CVE Details. Windows 10: Vulnerability Statistics. https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html, 2021.

[93] CVE Details. XEN: Vulnerability Statistics. https://www.cvedetails.com/product/23463/XEN-XEN.html, 2021.

[94] AndroidAPS app documentation. http://wiki.aaps.app/en/latest/, 2022.

[95] Calling for the Return of Non-Virtualized Microprocessor Systems. https://www.sigarch.org/calling-for-the-return-of-non-virtualized-microprocessor-systems/, 2022.

[96] CVE Details. Linux Kernel: Vulnerability Statistics. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, 2022.

[97] GoatRAT Attacks Automated Payment Systems. https://labs.k7computing.com/index.php/goatrat-attacks-automated-payment-systems, 2023.

[98] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. ACM CCS*, 2005.

[99] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proc. Summer 1986 USENIX Conference*, 1986.

[100] A. Amiri Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*, 2017.

[101] A. Amiri Sani and T. Anderson. The Case for I/O-Device-as-a-Service. In *Proc. ACM HotOS*, 2019.

[102] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong. I/O Paravirtualization at the Device File Boundary. In *Proc. ACM ASPLOS*, 2014.

[103] A. Amiri Sani, L. Zhong, and D. S. Wallach. Glider: A GPU Library Driver for Improved System Security. *Technical Report 2014-11-14, Rice University*, 2014.

[104] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a Virtual Mobile Smartphone Architecture. In *Proc. ACM SOSP*, 2011.

[105] ARM. Architecture Reference Manual, ARMv7-A and ARMv7-R edition. *ARM DDI*, 0406A, 2007.

[106] ARM. Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. *ARM DDI*, 0487A.a (ID090413), 2013.

[107] A. Athalye, A. Belay, M. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proc. ACM SOSP*, 2019.

[108] P. Athe and S. Dasgupta. A Comparative Study of 6T, 8T and 9T Decanano SRAM cell. In *Proc. IEEE Symposium on Industrial Electronics & Applications*, 2009.

[109] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*, 2016.

[110] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. USENIX Security Symposium*, 2015.

[111] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proc. USENIX Security Symposium*, 2021.

[112] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys*, 2006.

[113] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *Proc. USENIX OSDI*, 2014.

[114] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proc. USENIX OSDI*, 2012.

[115] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. USENIX OSDI*, 2014.

[116] M. Ben-Yehuda, O. Peleg, O. Agmon Ben-Yehuda, I. Smolyar, and D. Tsafrir. The nonkernel: A Kernel Designed for the Cloud. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2013.

[117] S. Birr, J. Mönch, D. Sommerfeld, U. Preim, and B. Preim. The LiverAnatomy-Explorer: A WebGL-Based Surgical Teaching Tool. *IEEE Computer Graphics and Applications*, 2013.

[118] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proc. USENIX NSDI*, 2008.

[119] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2019.

[120] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proc. USENIX ATC*, 2010.

[121] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. SANCTUARY: Arming trustzone with user-space enclaves. In *Proc. Internet Society NDSS*, 2019.

[122] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[123] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium*, 2004.

[124] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 2017.

[125] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *Proc. ACM CCS*, 2013.

[126] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in Trustzone-assisted TEE Systems. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[127] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[128] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2011.

[129] L. Chen and J. Li. Flexible and Scalable Digital Signatures in TPM 2.0. In *Proc. ACM CCS*, 2013.

[130] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. USENIX Security*, 2014.

[131] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[132] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[133] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. De Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proc. ACM ASPLOS*, 2015.

[134] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. USENIX Security Symposium*, 2016.

[135] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2006.

[136] N. V. Database. CVE-2021-0200: Out-of-bounds write in the firmware for Intel(R) Ethernet 700 Series Controllers before version 8.2 may allow a privileged user to potentially enable an escalation of privilege via local access. `https://nvd.nist.gov/vuln/detail/CVE-2021-0200`.

[137] N. V. Database. Vulnerability summary for cve-2015-6639.

[138] U. Dey, P. K. Jana, and C. S. Kumar. Modeling and Kinematic Analysis of Industrial Robots in WebGL Interface. In *IEEE International Conference on Technology for Education*, 2016.

[139] Ding, R. and Qian, C. and Song, C. and Harris, B. and Kim, T. and Lee, W. Efficient Protection of Path-Sensitive Control Security. In *Proc. USENIX Security Symposium*, 2017.

[140] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proc. USENIX OSDI*, 2008.

[141] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proc. ACM SOSP*, 2013.

[142] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*, 1995.

[143] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proc. ACM EuroSys*, 2006.

[144] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proc. ACM SOSP*, 2017.

[145] A. Forin, D. Golub, and B. N. Bershad. An I/O System for Mach 3.0. In *Proc. USENIX Mach Symposium*, 1991.

[146] K. Franz. Add a microSD Slot with the Pmod MicroSD. `https://digilent.com/blog/add-a-microsd-slot-with-the-pmod-microsd/`, 2021.

[147] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *Proc. USENIX Security Symposium*, 2018.

[148] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *Proc. IEEE Security and Privacy (S&P)*, 2018. bibtex: frigo2018.

[149] A. Frumusanu. Huawei Announces Mate 40 Series: Powered by 15.3bn Transistors 5nm Kirin 9000. `https://www.anandtech.com/show/16156/huawei-announces-mate-40-series`, 2020.

[150] W. Futral and J. Greene. *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*. Apress Media LLC, Springer Nature, 2013.

[151] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proc. ACM ASPLOS*, 2008.

[152] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser. Time Protection: The Missing OS Abstraction. In *Proc. ACM EuroSys*, 2019.

[153] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, 2000.

[154] Y. GmbH. SymbiYosys (sby) Documentation. `https://symbiyosys.readthedocs.io/en/latest/index.html`, 2021.

[155] D. B. Golub, G. G. Sotomayor, and F. L. Rawson, III. An Architecture for Device Drivers Executing As User-Level Tasks. In *Proc. USENIX MACH III Symposium*, 1993.

[156] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proc. ACM European Workshop on Systems Security (EuroSec)*, 2017.

[157] R. Grisenthwaite. Arm CCA will put confidential compute in the hands of every developer. `https://www.arm.com/company/news/2021/06/arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer`, 2021.

[158] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2018.

[159] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. Deep Specifications and Certified Abstraction Layers. In *Proc. ACM POPL*, 2015.

[160] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. USENIX OSDI*, 2016.

[161] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.*, 2016.

[162] S. Han, W. Shin, J. Park, and H. Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In *Proc. USENIX Security Symposium*, 2018.

[163] F. Hetzelt and R. Buhren. Security Analysis of Encrypted Virtual Machines. In *Proc. ACM VEE*, 2017.

[164] T. Ho, D. Dean, X. Gu, and W. Enck. PREC: Practical Root Exploit Containment for Android Devices. In *Proc. ACM CODASPY*, 2014.

[165] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*, 2013.

[166] J. Howell, B. Parno, and J. Douceur. Embassies: Radically Refactoring the Web. In *Proc. USENIX NSDI*, 2013.

[167] T. C. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proc. ACM CCS*, 2016.

[168] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 2007.

[169] IBM. Software TPM Introduction. `http://ibmswtpm.sourceforge.net/ibmswtpm2.html`, 2021.

[170] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. ACM SOSP*, 1997.

[171] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proc. USENIX Security Symposium*, 2007.

[172] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proc. ACM ISCA*, 2014.

[173] D. Kleidermacher, J. Seed, B. Barbello, S. Somogyi, and P. . T. s. t. Android. Pixel 6: Setting a new standard for mobile security. `https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html`, 2021.

[174] 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 SOSP*, 2009.

[175] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[176] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. ACM EuroSys*, 2020.

[177] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5), 2005.

[178] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir. Page Fault Support for Network Controllers. In *Proc. ACM ASPLOS*, 2017.

[179] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. USENIX OSDI*, 2004.

[180] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proc. ACM SOSP*, 2017.

[181] M. Li, Y. Zhang, and Z. Lin. CrossLine: Breaking "security-by-crash" based Memory Isolation in AMD SEV. In *Proc. ACM CCS*, 2021.

[182] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proc. USENIX Security Symposium*, 2019.

[183] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. CipherLeaks: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *Proc. USENIX Security Symposium*, 2021.

[184] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A Secure and Formally Verified Linux KVM Hypervisor. 2021.

[185] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proc. USENIX Security Symposium*, 2021.

[186] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proc. ACM MobiSys*, 2018.

[187] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proc. USENIX OSDI*, 2022.

[188] J. Liedtke. Improving IPC by Kernel Design. *ACM SIGOPS Operating Systems Review*, 1993.

[189] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*, 2016.

[190] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proc. USENIX Security Symposium*, 2018.

[191] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proc. USENIX OSDI*, 2016.

[192] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *Proc. ACM HotOS*, 2021.

[193] 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 (S&P)*, 2010.

[194] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM EuroSys*, 2008.

[195] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM EuroSys*, 2008.

[196] J. Mickens and M. Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proc. ACM SOSP*, 2011.

[197] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis. PPFL: Privacy-Preserving Federated Learning with Trusted Execution Environments. In *Proc. ACM MobiSys*, 2021.

[198] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *Proc. Springer International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[199] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. TPM-Fail: TPM meets Timing and Lattice Attacks. In *Proc. USENIX Security Symposium*, 2020.

[200] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *Proc. USENIX Security Symposium*, 2020.

[201] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[202] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. Loh, M. Subramony, and S. White. Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product. In *Proc. ACM/IEEE ISCA*, 2021.

[203] R. Nandakumar, S. Gollakota, and N. Watson. Contactless Sleep Apnea Detection on Smartphones. In *Proc. ACM MobiSys*, 2015.

[204] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proc. USENIX OSDI*, 2020.

[205] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proc. ACM SOSP*, 2017.

[206] J. Nider and A. Fedorova. The Last CPU. In *Proc. ACM HotOS*, 2021.

[207] H. Omar and O. Khan. IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications. In *Proc. IEEE HPCA*, 2020.

[208] R. Paccagnella, L. Luo, and C. W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *Proc. USENIX Security Symposium*, 2021.

[209] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS*, 2011.

[210] M. Payer, E. Kravina, and T. R. Gross. Lightweight Memory Tracing. In *Proc. USENIX ATC*, 2013.

[211] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proc. ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012.

[212] H. Peng, Z. Yao, A. Amiri Sani, D. Tian, and M. Payer. GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation. In *Proc. USENIX Security Symposium*, 2023.

[213] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*, 2014.

[214] A. Phaneuf. State of mobile banking in 2020: top apps, features, statistics and market trends. `https://www.businessinsider.com/mobile-banking-market-trends`, 2019.

[215] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM ASPLOS*, 2011.

[216] M. Posner. How many ASIC Gates does it take to fill an FPGA? `https://blogs.synopsys.com/breakingthethreelaws/2015/02/how-many-asic-gates-does-it-take-to-fill-an-fpga/`, 2015.

[217] N. Provos. Improving Host Security with System Call Policies. In *Proc. USENIX Security Symposium*, 2003.

[218] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proc. USENIX Security Symposium*, 2003.

[219] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Proc. ACM CCS*, 2019.

[220] Quarklab. BREAKING SAMSUNG'S ARM TRUSTZONE. `https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf`, 2019.

[221] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *Proc. USENIX Security Symposium*, 2016.

[222] A. Razeen, A. R. Lebeck, D. H. Liu, A. Meijer, V. Pistol, and L. P. Cox. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *Proc. ACM MobiSys*, 2018.

[223] D. S. Ritchie and G. W. Neufeld. User Level IPC and Device Management in the Raven Kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, 1993.

[224] F. Roesner and T. Kohno. Securing Embedded User Interfaces: Android and Beyond. In *Proc. USENIX Security Symposium*, 2013.

[225] A. S. Rose and P. W. Hildebrand. NGL Viewer: a web application for molecular visualization. *Nucleic Acids Res*, 2015.

[226] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proc. Springer International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[227] SECure COMPuting with filters. `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`, 2018.

[228] S. M. Seyed Talebi, Z. Yao, A. Amiri Sani, Z. Qian, and D. Austin. Undo Workarounds for Kernel Bugs. In *Proc. USENIX Security Symposium*, 2021.

[229] S. Shankland. Apple's A15 Bionic chip powers iPhone 13 with 15 billion transistors, new graphics and AI. `https://www.cnet.com/tech/mobile/apples-a15-bionic-chip-powers-iphone-13-with-15-billion-transistors-new-graphics-and-ai/`, 2021.

[230] J. Shao, Y. Qin, D. Feng, and W. Wang. Formal Analysis of Enhanced Authorization in the TPM 2.0. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2015.

[231] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. USENIX Security Symposium*, 2012.

[232] Y. Shizuku, T. Hirose, N. Kuroki, M. Numa, and M. Okada. A 24-transistor static flip-flop consisting of nors and inverters for low-power digital vlsis. In *Proc. IEEE International New Circuits and Systems Conference (NEWCAS)*, 2014.

[233] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for Design and Verification of Information Flow Control Systems. In *Proc. USENIX OSDI*, 2018.

[234] E. R. Sparks. A Security Assessment of Trusted Platform Modules. *Dartmouth College Undergraduate Theses. 53*, 2007.

[235] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel Support for Secure Process Compartments. *EAI Endorsed Transactions on Security and Safety*, 2015.

[236] V. Strumpen. Introduction to Digital Circuits: Basic Digital Circuits. `http://bibl.ica.jku.at/dc/build/html/basiccircuits/basiccircuits.html`, 2015.

[237] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*, 2003.

[238] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKscrew: Exposing the Perils of Security-Oblivious Energy Management. In *Proc. USENIX Security Symposium*, 2017.

[239] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proc. USENIX OSDI*, 2010.

[240] R. Tao, J. Yao, X. Li, S. Li, J. Nieh, and R. Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proc. ACM SOSP*, 2021.

[241] T. C. G. (TCG). TPM 2.0 Library. `https://trustedcomputinggroup.org/resource/tpm-library-specification/`, 2019.

[242] D. Team. NEWS: OmniPod Tubeless Insulin Pump to Offer Smartphone Control Soon. `https://www.healthline.com/diabetesmine/omnipod-smartphone-control-diabetes`, 2019.

[243] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proc. USENIX ATC*, 2014.

[244] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. USENIX Security Symposium*, 2018.

[245] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *Proc. ACM CCS*, 2015.

[246] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proc. ACM CCS*, 2016.

[247] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[248] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proc. USENIX Security Symposium*, 2016.

[249] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proc. ACM/IEEE ISCA*, 2014.

[250] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proc. ACM SOSP*, 1993.

[251] J. Wang, X. Xiong, and P. Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *Proc. USENIX ATC*, 2015.

[252] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *Proc. USENIX Security Symposium*, 2022.

[253] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proc. USENIX Security Symposium*, 2010.

[254] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *Proc. USENIX Security Symposium*, 2021.

[255] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proc. ACM CODASPY*, 2017.

[256] S. Wesemeyer, C. J. Newton, H. Treharne, L. Chen, R. Sasse, and J. Whitefield. Formal Analysis and Implementation of a TPM 2.0-based Direct Anonymous Attestation Scheme. In *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2020.

[257] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[258] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. ACM/IEEE ISCA*, 2014.

[259] Y. Xiao, M. Li, S. Chen, and Y. Zhang. Stacco: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proc. ACM CCS*, 2017.

[260] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proc. USENIX Security Symposium*, 2016.

[261] Xilinx. Xilinx Standalone Library Documentation. OS and Libraries Document Collection. UG643 (v2021.1) June 16, 2021.

[262] Xilinx. Zynq UltraScale + Device. Technical Reference Manual. UG1085 (v2.2) December 4, 2020.

[263] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. ACM MobiSys*, 2019.

[264] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran. Sugar: Secure GPU Acceleration in Web Browsers. In *Proc. ACM ASPLOS*, 2018.

[265] Z. Yao, S. Mirzamohammadi, A. Amiri Sani, and M. Payer. Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks. In *Proc. ACM CCS*, 2018.

[266] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson. Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware. In *Proc. ACM MobiSys*, 2023.

[267] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2009.

[268] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proc. ACM MobiSys*, 2018.

[269] M. H. Yun and L. Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proc. Internet Society NDSS*, 2019.

[270] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proc. USENIX OSDI*, 2008.

[271] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS*, 2015.

[272] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[273] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proc. USENIX OSDI*, 2006.

[274] S. Zhu, L. Lu, and K. Singh. Case: Comprehensive Application Security Enforcement on COTS Mobile Devices. In *Proc. ACM MobiSys*, 2016.