Sugar: Secure GPU Acceleration in Web Browsers

Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, Aparna Chandramowlishwaran
University of California, Irvine
(z.yao,zonghenm,yingtong,ardalan,amowli)@uci.edu

Abstract

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.

CCS Concepts • **Security and privacy → Trusted computing**; **Virtualization and security**; **Browser security**;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173186

Keywords GPU acceleration, Web browser, Virtualization, Systems security

ACM Reference Format:
Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, Aparna Chandramowlishwaran. 2018. Sugar: Secure GPU Acceleration in Web Browsers. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3173162.3173186

1 Introduction

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) and 96% of 48.8 million visitors to a series of websites used WebGL-enabled browsers [66].

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, 43]. Second, they perform runtime security checks on the WebGL API calls [65]. 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" [65] 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 these shortcomings, 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 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 [99]). 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.

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 (§4). 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 (§5).

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

2 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 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

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

Sites list [44]. 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 [66].

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 [45]; Microsoft creates numerous WebGL demonstrations, including one for the Assassin's Creed Pirates and one for the Dolby Audio Experience [52]; Google and Baidu use WebGL to enhance their map services [46, 50]; and Facebook and Unity work together to provide their users with WebGL-based online games [41].

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 [78], the NASA Experience Curiosity website, which allows the public to learn about Mars and the Martian rover [53], the NGL Viewer, which visualizes molecules [95], the Thingiverse Customizer, which previews and edits 3D printing models [58], the LiverAnatomyExplorer, which facilitates medical education [75], and the NIST Digital Library of Mathematical Functions, which brings mathematical formulas to life [55].

2.2 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) [54] and Chrome bug report database [47]. We search these databases using the keyword "WebGL".

Figures 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 1 and others found through manual search, e.g., Firefox

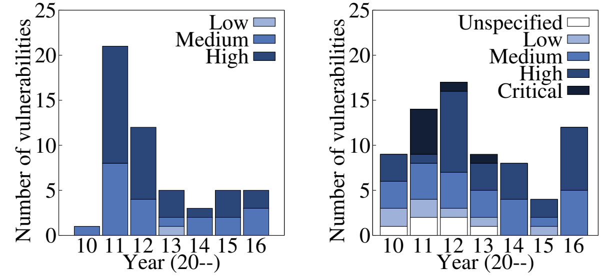

Figure 1. WebGL vulnerability statistics according to NVD (Left) and Chrome reports (Right). Note that the max severity level in NVD is "High".

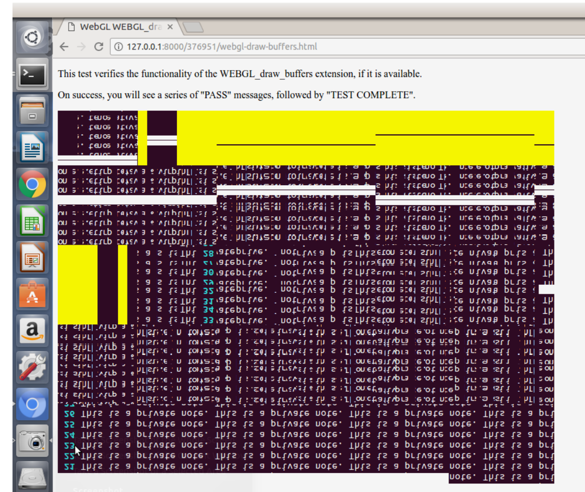

Figure 2. Exploiting vulnerability #10 in Table 1. The exploit manages to access to unauthorized parts of the GPU memory, which holds previously rendered UI content. In this snapshot, the exploit has accessed the content of a text previously edited in a native text editor.

WebGL vulnerabilities). Our goal in this study is to understand the impact of these vulnerabilities and to determine whether Sugar can eliminate them.

Table 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 shows

Vulnerability type	#	Vulnerability description	Official vulnerability report: Vendor (number, severity)	Official vulnerability report: NVD (number, severity)	Effect	Target platform (Browser: OS:GPU)	Reproduced by us: On current version	Reproduced by us: After removing patch	Solved by Sugar: One-GPU	Solved by Sugar: Dual-GPU
Integrity	1	Use-after-free [26, 28]	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 [33, 34]	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 [5, 11]	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 [9, 13]	684882, High	CVE-2011-3653, Med.	Read of GPU memory	FF:Mac: Intel	N/A (PNA)	N/A (PNA)	✓(BA)	✓(BA)
	8	Timing attack [4, 6, 10]	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 [48, 49]	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 [25, 29]	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 [37]	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 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.*

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 [57]. Chrome reports, on the other hand, uses the following levels: Low, Medium, High, and Critical. We use the levels used by the corresponding report.

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

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

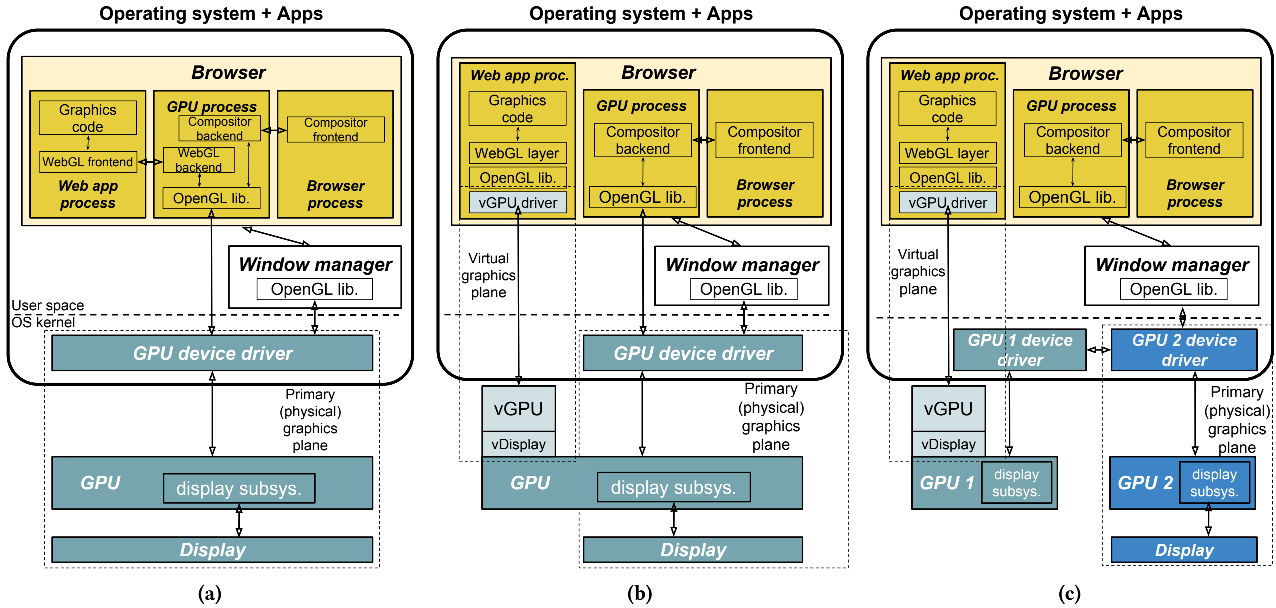

Figure 3. *(a) Existing WebGL architecture. (b) Single-GPU Sugar's architecture. (c) Dual-GPU Sugar's architecture. Note that the graphics planes bounding boxes in the figures only enclose the GPU and its driver, and not the graphics libraries, for simplicity.*

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 §7.1, we provide more details on this evaluation.

3 Sugar's Design

Our preliminary study in §2 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 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 §4). 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 3 (b) and (c) show two variants of Sugar's architecture. Figure 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 [99]). 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 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 [67]. 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 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" (§5.1). 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

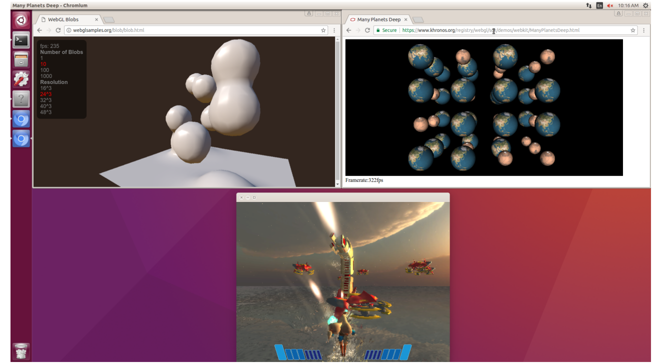

Figure 4. Screenshot of dual-GPU Sugar in action.

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 [51] 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.

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 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 (§7.1). 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 §7.2, 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 §7.2). 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.

3.1 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 (§2.2). We do not protect against side-channel attacks.

3.2 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 (§4.1), and part of KVM for instruction decoding used in Sugar (§6).

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.

4 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 [77]. A similar approach is being used in Windows Defender Application Guard in Microsoft Edge [42]. However, one main drawback of this solution is that it requires significant revamping 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.

4.1 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 [87] 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.

4.2 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.

4.3 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 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 §9, 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.

5 Browser's Support for Sugar

5.1 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, 43]. 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.

5.2 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.

6 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 [38]), 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 §4.1. 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 (§4) 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

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

Table 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.

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 [56]. However, for enhanced WebGL performance for both, we turn off the "`dcheck_always_on`" option, which performs runtime assertions.

7 Evaluation

We evaluate the security and performance of Sugar.

7.1 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 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 §4.1), 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 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 (§3.1).

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.

7.2 Performance

We evaluate the performance of Sugar with five benchmarks: Blob [61], Many-Planets [63], San-Angeles [64], Cubemap [62], and Animometer [60]. 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

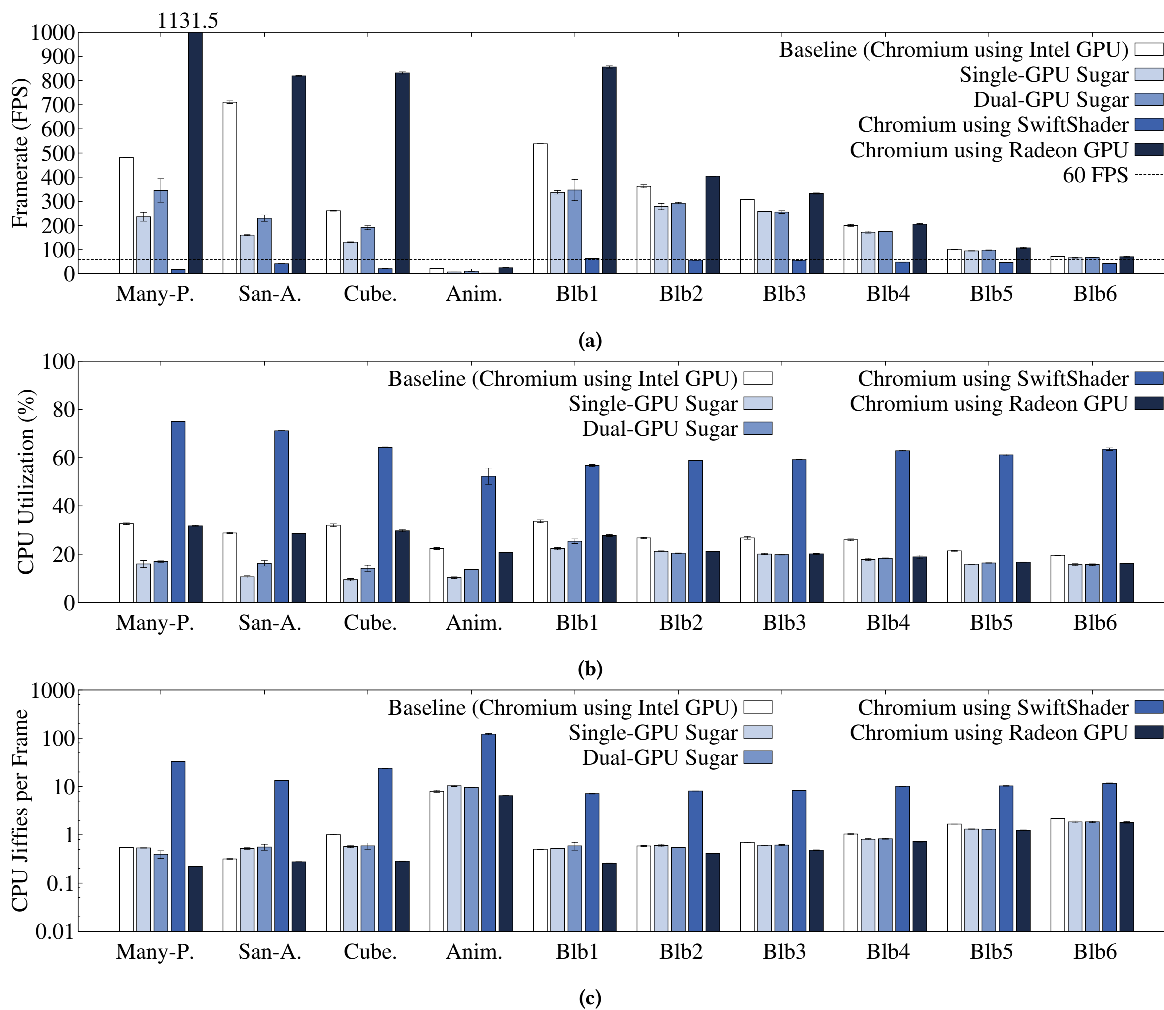

Figure 5. (a) Benchmarks' performance. (b) System's CPU utilization while executing the benchmarks. (c) Benchmarks' raw CPU usage.

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 (§6) but use GLX for the baseline experiments since GLX is used by default in Chromium. Third, as mentioned in §6, we use Mesa version 12.0.6 for Sugar. However, this version of Mesa does not properly support the high-end Radeon R9 290 GPU in our prototype [40]. 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 [35]. 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 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,

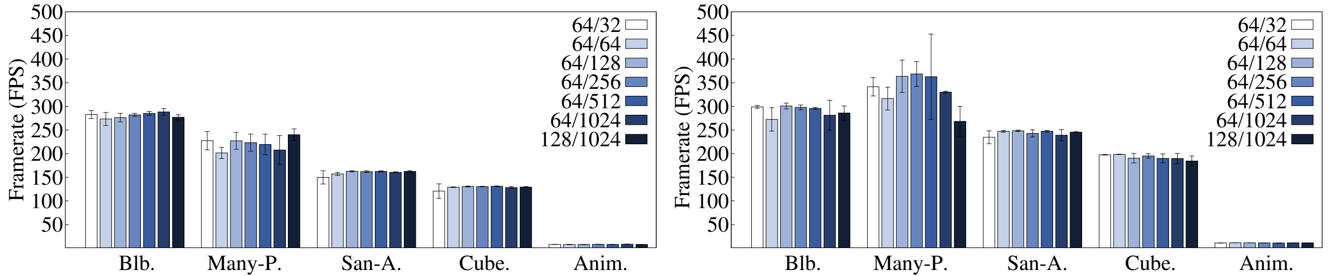

Figure 6. Effect of varying vGPU memory sizes for (Left) single-GPU Sugar and (Right) dual-GPU Sugar.

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 [99]. 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 (§3). 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 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 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 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 [39]. 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 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 the paper. 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 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. 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 [59]) at the same time as one of our WebGL benchmarks (Blob), each occupying almost half of the screen. Figure 8 (Left) shows the WebGL benchmark performance in this setup and Figure 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.

8 Related Work

Browser security & web app isolation. Other solutions have attempted to protect the browser and the system against

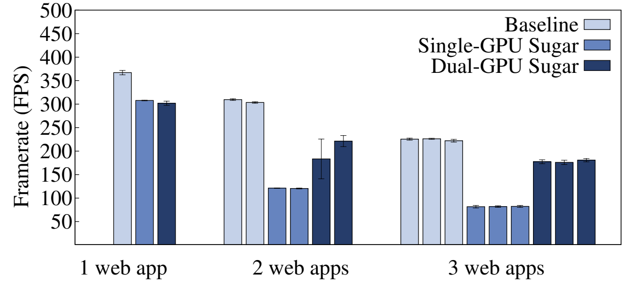

Figure 7. Supporting multiple web apps simultaneously.

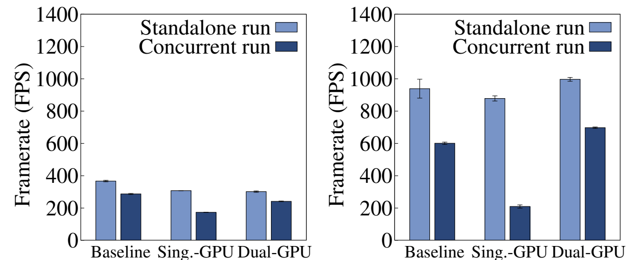

Figure 8. Performance isolation. (Left) A WebGL benchmark running standalone or concurrent with an OpenGL benchmark. (Right) An OpenGL benchmark running standalone or concurrent with a WebGL benchmark.

an untrusted web app, e.g., by sandboxing it inside a pico-process [85], in an exokernel browser [89], inside a virtual machine [42, 77], or by reducing the TCB of the browser [98]. Moreover, Xax [79] and Native Client [100] 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.

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 [91] and IX [73] 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.

Secure UI embeddings. AdSplit [96], AdDroid [90], LayerCake [94], and SchrodinText [68] demonstrate secure embedding of UI interfaces. These solutions, at a high level, are similar to Sugar 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.

Library operating systems and other sandboxes. Library operating systems, such as Exokernel [81] and Drawbridge [92],

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 operating system in the enclave. While Sugar uses a library operating system-like architecture, it cannot protect the web app from an untrusted system.

GPU virtualization. Cells supports operating system-level virtualization of mobile devices and supports secure sharing of the GPU between multiple virtual phones through virtualization [71]. Paradice paravirtualizes I/O devices, including GPU, using the UNIX device file boundary [69]. In contrast, Sugar uses existing GPU virtualization solutions for secure GPU access by web apps.

Application's direct access to hardware. The nonkernel gives applications direct access to devices [74]. 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 [72] 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 [80, 82, 84, 86, 93]. SUD [76], Microdriver [83], and Glider [70] move either part or all of the driver to the user space. Indeed, SUD and Glider use UML to achieves this, similar to Sugar (§4.2). LeVasseur et al. [88] move the device driver to a virtual machine for better isolation. Moreover, Nooks [97] and SafeDrive [101] keep the driver in the kernel but protect against its vulnerabilities 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.

9 Limitations and Future Work

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 [36] 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 nontrivial 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.

10 Conclusions

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

Acknowledgments

The work was supported by NSF Award #1617513. The authors thank Zhen Wang and the anonymous reviewers for their insightful comments.

References

[1] 2010. 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] 2011. Chromium Issue 70718: Crashes when opening a page with webgl. https://bugs.chromium.org/p/chromium/issues/detail?id=70718. (2011).

[3] 2011. 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] 2011. CVE-2011-2366: Timing attack steals cross-domain images (Firefox). https://nvd.nist.gov/vuln/detail/CVE-2011-2366. (2011).

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

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

[7] 2011. 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] 2011. CVE-2011-2784: Chrome WebGL reveals local path in logs. https://nvd.nist.gov/vuln/detail/CVE-2011-2784. (2011).
[9] 2011. CVE-2011-3653: Read of cross-origin image through Firefox WebGL. https://nvd.nist.gov/vuln/detail/CVE-2011-3653. (2011).
[10] 2011. Firefox bug 655987 - Respond to the WebGL cross-domain image theft vulnerability. https://bugzilla.mozilla.org/show_bug.cgi?id=655987. (2011).
[11] 2011. Firefox bug 656752: WebGL crash in gleRunVertexSubmitImmediate. https://bugzilla.mozilla.org/show_bug.cgi?id=656752. (2011).
[12] 2011. Firefox bug 659349: WebGL allows access to uninitialized graphics memory. https://bugzilla.mozilla.org/show_bug.cgi?id=659349. (2011).
[13] 2011. 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] 2011. Microsoft considers WebGL harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx. (2011).
[15] 2011. WebGL - More WebGL Security Flaws. http://www.contextis.com/resources/blog/webgl-more-webgl-security-flaws/. (2011).
[16] 2012. Chromium issue 145544: Security: integer overflow in gpu process with webgl. https://bugs.chromium.org/p/chromium/issues/detail?id=145544. (2012).
[17] 2012. 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] 2012. Chromium issue 153469: Security: Nvidia Kernel Panic. https://bugs.chromium.org/p/chromium/issues/detail?id=153469. (2012).
[19] 2012. CVE-2012-2896: Integer overflow in Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2012-2896. (2012).
[20] 2012. CVE-2012-5115: Bug in graphics drivers allows for “wild writes” in Chrome. https://nvd.nist.gov/vuln/detail/CVE-2012-5115. (2012).
[21] 2013. Chromium Issue 237611: Security: Screen capture via WebGL texture. https://bugs.chromium.org/p/chromium/issues/detail?id=237611. (2013).
[22] 2013. CVE-2013-2874: Read of screen data through Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2013-2874. (2013).
[23] 2013. NVIDIA GRID K1 and K2 Graphics-Accelerated Virtual Desktops and Applications. NVIDIA White Paper. (2013).
[24] 2014. 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] 2014. CVE-2014-1502: Bug in Firefox WebGL allows for rendering cross-domain content. https://nvd.nist.gov/vuln/detail/CVE-2014-1502. (2014).
[26] 2014. 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] 2014. CVE-2014-3173: Read of uninitialized memory in Chrome WebGL. https://nvd.nist.gov/vuln/detail/CVE-2014-3173. (2014).
[28] 2014. Firefox bug 1028891: WebGL app crashes Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1028891. (2014).
[29] 2014. Firefox bug 972622 - WebGL.compressedTex(Sub)Image2D doesn’t call MakeCurrent. https://bugzilla.mozilla.org/show_bug.cgi?id=972622. (2014).
[30] 2014. GPU Accelerated Compositing in Chrome. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome. (2014).
[31] 2015. Chromium issue 483877: Bad shader can cause kernel crash. https://bugs.chromium.org/p/chromium/issues/detail?id=483877. (2015).
[32] 2015. 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] 2015. 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] 2015. Firefox bug 1190526 - Overflow in VertexBufferInterface::reserveVertexSpace causes memory-safety bug. https://bugzilla.mozilla.org/show_bug.cgi?id=1190526. (2015).
[35] 2016. 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).
[36] 2016. AMD Multiuser GPU: Hardware-Enabled GPU Virtualization for a True Workstation Experience. AMD White Paper. (2016).
[37] 2016. Chromium Issue 593680: WebGL test "temp expressions should not crash" freezes browser. https://bugs.chromium.org/p/chromium/issues/detail?id=593680. (2016).
[38] 2016. [iGVT-g] [ANNOUNCE] 2016-Q3 release of KVMGT. https://lists.01.org/pipermail/igvt-g/2016-November/000976.html. (2016).
[39] 2016. iGVT-g Setup Guide. https://github.com/01org/Igvtg-kernel/blob/2016q3-4.3.0/iGVT-g_Setup_Guide.txt. (2016).
[40] 2016. 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).
[41] 2016. Unity and Facebook Collaborate on WebGL Gaming. https://developers.facebook.com/blog/post/2016/08/18/FB-Unity-Alpha. (2016).
[42] 2016. Windows Defender Application Guard for Microsoft Edge. https://blogs.windows.com/msedgedev/2016/09/27/application-guard-microsoft-edge/#mBwrd1ATV1aluMyd.97. (2016).
[43] 2017. A new multi-process model for Firefox. https://hacks.mozilla.org/2017/06/firefox-54-e10s-webextension-apis-css-clip-path/. (2017).
[44] 2017. Alexa Top 500 Global Sites. http://www.alexa.com/topsites. (2017).
[45] 2017. Apple macOS Sierra. https://www.apple.com/macos/sierra. (2017).
[46] 2017. Baidu Map. http://map.baidu.com. (2017).
[47] 2017. Chrome Issues. https://bugs.chromium.org/p/chromium/issues/list. (2017).
[48] 2017. Chromium Issue 682020: Security: WebGL - Use After Free in Buffer11::updateBufferStorage(). https://bugs.chromium.org/p/chromium/issues/detail?id=682020. (2017).
[49] 2017. CVE-2017-5031: Use after free in Chrome ANGLE. https://nvd.nist.gov/vuln/detail/CVE-2017-5031. (2017).
[50] 2017. Google Maps. https://www.google.com/maps. (2017).
[51] 2017. Linux dma-buf. https://www.kernel.org/doc/html/v4.10/driver-api/dma-buf.html. (2017).
[52] 2017. Microsoft Edge TestDrive demos. https://developer.microsoft.com/en-us/microsoft-edge/testdrive/tags/webgl. (2017).
[53] 2017. NASA Experience Curiosity. https://eyes.nasa.gov/curiosity. (2017).
[54] 2017. National Vulnerability Database. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd. (2017).
[55] 2017. NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov. (2017).
[56] 2017. The Chromium Projects: GN build configuration. https://www.chromium.org/developers/gn-build-configuration. (2017).
[57] 2017. The Common Vulnerability Scoring System version 2. https://www.first.org/cvss/v2/. (2017).
[58] 2017. Thingiverse Customizer. https://www.thingiverse.com/customizer. (2017).
[59] 2017. Unity WaveShooter OpenGL benchmark. https://github.com/unity3d-jp/WaveShooter. (2017).
[60] 2017. WebGL Animometer benchmark. http://kenrussell.github.io/webgl-animometer/Animometer/tests/3d/webgl.html. (2017).

[61] 2017. WebGL Blob benchmark. http://webglsamples.org/blob/blob.html. (2017).

[62] 2017. WebGL Cubemap benchmark. http://webglsamples.org/dynamic-cubemap/dynamic-cubemap.html. (2017).

[63] 2017. WebGL Many-Planets benchmark. http://www.khronos.org/registry/webgl/sdk/demos/webkit/ManyPlanetsDeep.html. (2017).

[64] 2017. WebGL San-Angeles benchmark. http://www.khronos.org/registry/webgl/sdk/demos/google/san-angeles/index.html. (2017).

[65] 2017. WebGL Security. http://www.khronos.org/webgl/security/. (2017).

[66] 2017. WebGL Statistics. http://webglstats.com. (2017).

[67] 2018. 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).

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

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

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

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

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

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

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

[75] S. Birr, J. MÃűnch, D. Sommerfeld, U. Preim, and B. Preim. 2013. The LiverAnatomyExplorer: A WebGL-Based Surgical Teaching Tool. *IEEE Computer Graphics and Applications* (2013).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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