Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks

Zhihao Yao[★], Saeed Mirzamohammadi[★], Ardalan Amiri Sani[★], Mathias Payer[†]

[★]UC Irvine, [†]EPFL and Purdue University

ABSTRACT

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.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Operating systems security**; **Mobile platform security**; **Browser security**;

KEYWORDS

Mobile Graphics Security; WebGL Security

1 INTRODUCTION

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.

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 [12] 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

CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5693-0/18/10...$15.00
https://doi.org/10.1145/3243734.3243772

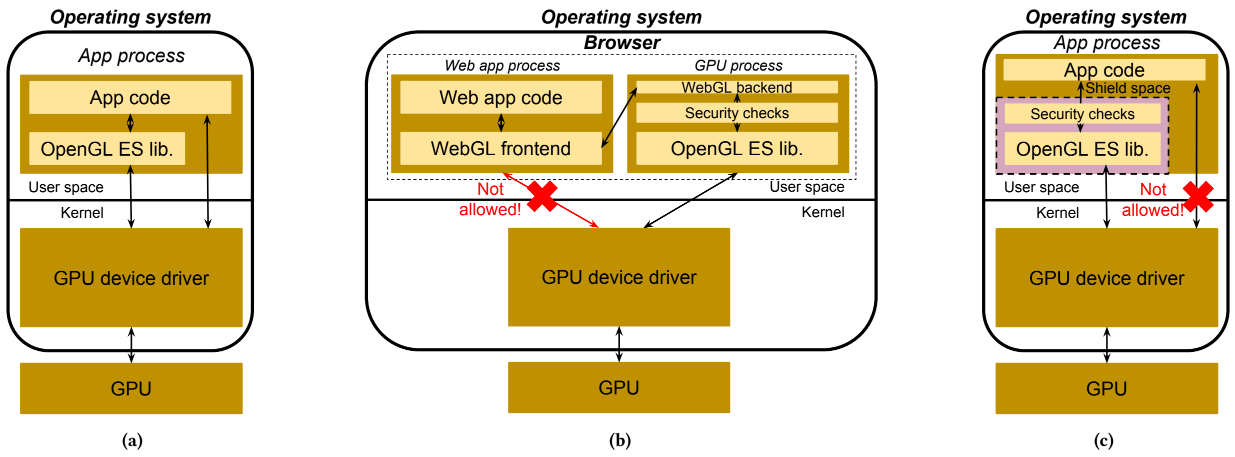


Figure 1: (a) Graphics stack in a mobile operating system. (b) WebGL stack in a web browser. (c) Graphics stack in Milkomeda.

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 paper.

- 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.

2 BACKGROUND & MOTIVATION

2.1 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 our paper, 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 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 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

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 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.

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 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 §6.2), have been demonstrated [63]. 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.

2.2 Mobile Graphics Vulnerabilities

Reported vulnerabilities. We study Android GPU vulnerabilities by searching the National Vulnerability Database (NVD) [8] (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 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 (DoS).

Figure 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) [11]. 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

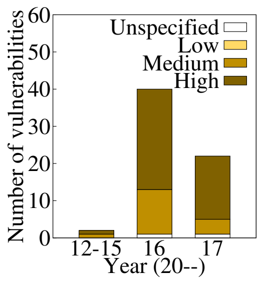


Figure 2: Severity and year of Android GPU vulnerabilities in NVD. The legend captures the severity according to CVSSv2.

the Android Native Development Kit (NDK) [13]. 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.

2.3 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" [2].

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 [12], given that the WebGL API is similar to the OpenGL ES API (see §6.2 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 [38]. To mitigate it, Google and Mozilla both blocked a certain extension in WebGL [17]. 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 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 [3, 5]. 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 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.

2.4 WebGL Security Checks

In this subsection, we provide a high-level review of WebGL security checks based on available documents, e.g., [4], 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 [1]. 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 [7]. 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 [18]. Figure 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 [18]. Then, it checks the target texture's ability to work with the dimension and level of the 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

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

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 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" [19] 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.

4 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 §6, 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 §8.2 that Milkomeda achieves high graphics performance for various mobile apps, although at the cost of moderately increased CPU utilization. Figure 1c illustrates Milkomeda's design.

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

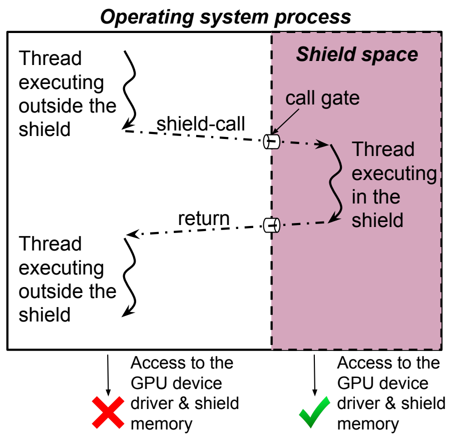

Operating system process
Shield space
Thread executing outside the shield
shield-call
call gate
Thread executing in the shield
return
Thread executing outside the shield
Access to the GPU device driver & shield memory
Access to the GPU device driver & shield memory

Figure 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.

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 §4.

5.1 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 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 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

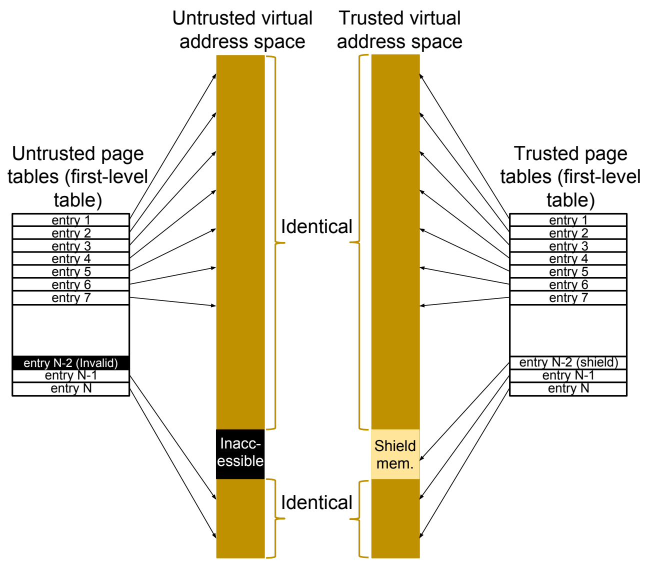


Figure 5: Implementation of shield space memory using page tables. The untrusted and trusted address spaces (mapped by the untrusted and trusted page tables and used, respectively, for the threads outside and within the shield space) are almost identical except for a contiguous range of addresses reserved for the shield space and accessible only through the trusted page tables.

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.

5.2 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 [53]. 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.

5.3 OpenGL ES API Call Execution Flow

In this subsection, we describe, the execution flow of an OpenGL ES API call in Milkomeda. Figure 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 (§5.2), 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 (§5.1), 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.

5.4 Satisfying the Required Guarantees

In this subsection, we discuss how Milkomeda achieves the four required guarantees discussed in §4.

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 (§5.2). 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 (§5.1). 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.

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 (§2.4). We achieve this by using our protected shield memory (§5.1). 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

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

```
/* 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, ...);
}
```

```
/* 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) */
}
```

```
/* 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. */
}
```

```
/* 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 6: Pseudocode demonstrating an OpenGL ES API call in Milkomeda.

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 (§7.1).

6 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

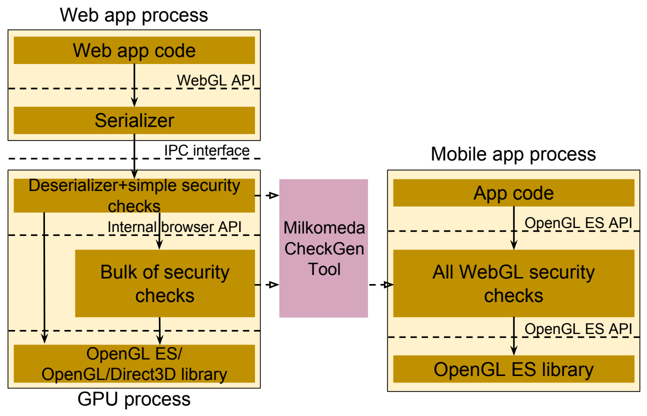


Figure 7: Milkomeda's CheckGen tool automatically transforms WebGL's security checks into a single layer to be used for mobile graphics.

a result, we developed a tool, called CheckGen, which automatically ports the WebGL security checks to be used for mobile graphics.

Figure 7 illustrates the role of the CheckGen tool. The left side of the figure depicts the 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 (§2.4) 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.

6.1 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 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 7) are then used without any modifications.

6.2 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 [20]. First, WebGL does not support client-side vertex arrays [16], 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" [6]. Chromium converts this type [20]. 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.

7 IMPLEMENTATION

We implement Milkomeda for Android operating system on 64-bit ARMv8 processors, which are commonly used in all recent mobile devices (see §10 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.

7.1 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 [28]), 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 [28]). 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 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 §8.2 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.

7.2 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 (§2.4). 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 [20]. 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 [20], we disable it in CheckGen. Note that this does not affect the security of Milkomeda because the real IDs are not considered secrets.

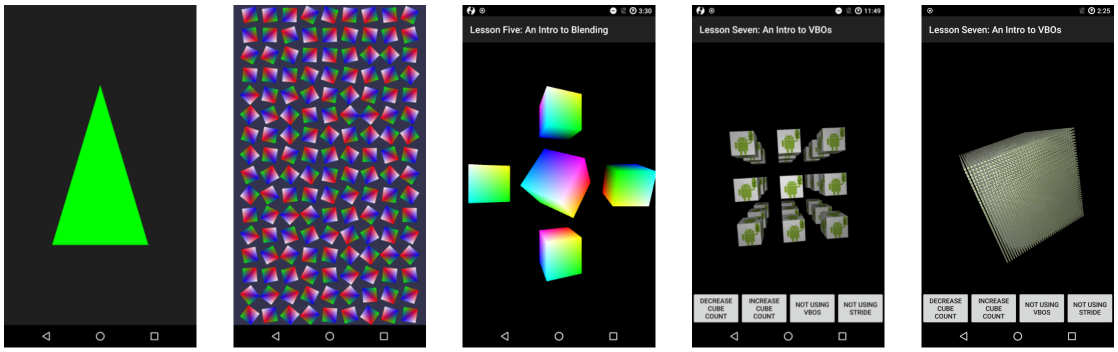
8 EVALUATION

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).

8.1 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,



(a) B1 [9] **(b) B2 [21]** **(c) B3 [22]** **(d) B4 [23]** **(e) B5**

Figure 8: Graphics benchmarks used in evaluation. We derive B5 from B4 by increasing the number of cubes significantly.

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 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 6. 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) [63]. 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 [63]. 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.

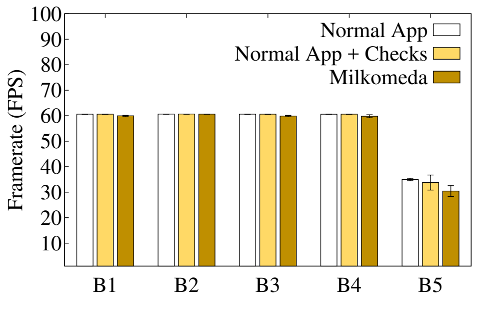
8.2 Graphics Performance & CPU Usage

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 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.

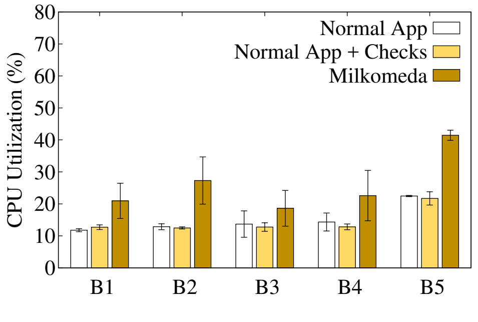
Figure 9a shows the framerate in our benchmarks. It shows the measurement for three different configurations: normal app, normal app + checks, and Milkomeda. The first configuration is the performance of the benchmarks using an unmodified graphics stack, i.e., the state of the art. The second configuration 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.

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 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



(a)



(b)

Figure 9: (a) Graphics performance. (b) CPU utilization. In both figures, B1 to B5 represent the five benchmarks shown in Figure 8. Each bar in the figure shows the average over six runs and the error bar shows the standard deviation.

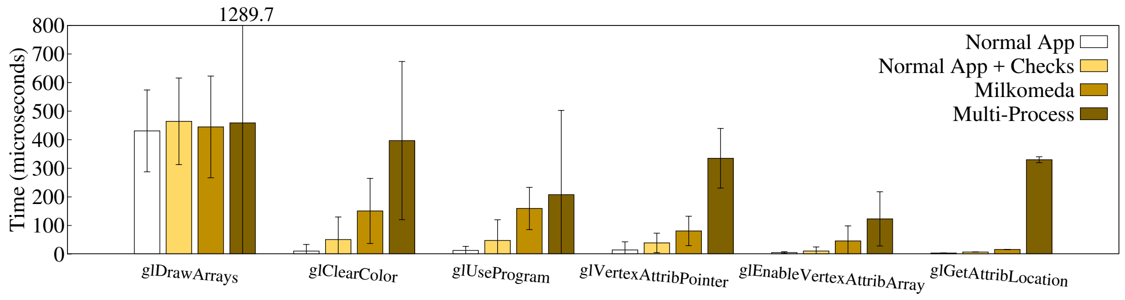


Figure 10: Execution time of several OpenGL ES API calls. Each bar in the figure shows the average over all invocations of the API in three runs and the error bar shows the standard deviation.

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.

8.3 Comparison with the Multi-Process Design

As mentioned in §2.1, 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 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).

9 RELATED WORK

9.1 Graphics Security

Sugar [63] 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 [25], VButton [42], and Truz-Droid [65] protect integrity or confidentiality of content shown on the mobile display. SchrodinText achieves this by modifying the operating system 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 operating system is assumed to be untrusted whereas the user and the app are trusted. Unlike these systems, Milkomeda does not modify the existing operating system graphics stack. It assumes that the operating system is trusted but the app is not. It then safeguards the graphics stack against malicious apps.

AdSplit [54], AdDroid [48], and LayerCake [52] isolate the code used to render an embedded UI component, e.g., ads. Their goal is to protect the app from untrusted embeddings. In contrast, in Milkomeda, we protect the system from untrusted apps, which try to exploit the vulnerabilities in the GPU device driver.

9.2 Device Driver Vulnerabilities & 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 [34, 46, 67]. Other related work tries to mitigate vulnerabilities in device drivers. Microkernels execute the device drivers in user space daemons [36]. Microdriver [39] and Glider [26] move parts of the device drivers to user space. Nooks safeguard against faults in device drivers using lightweight protection domains in the kernel [56]. SafeDrive does so using language techniques [68].

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.

9.3 Operating System-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 operating system. Initial related work started with system call vetting based on `ptrace` but quickly moved towards a kernel-level caching mechanism [49]. AppArmor [15] enforces a configurable system call policy on a per-process basis. SELinux [10] hardens kernel and user-space and restricts interactions between processes and the kernel without enforcing an explicit system call policy. Capsicum [61] 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 [53]. 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 [69]. 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.

9.4 Process-Level & 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. [50] provide a case study on how to break the OpenSSH server into smaller protected components (similar to how QMail compares to sendmail). Privtrans [32] automates the privilege separation process through an inference process. Wedge [31] extends Privtrans with capabilities while Salus [55] provides dynamically adjustable enforcement policies. Dune [30] 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 [60] provides fine-grained synchronization of memory spaces between threads but incurs prohibitive overhead. SMV [41] 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 [43] 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 [51]. 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 [37]. In contrast, Milkomeda's shield space is designed for existing hardware.

9.5 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) [24, 33]. 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 [57] and PittyPat [35] 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.

9.6 Fault Isolation

Fault isolation restricts interactions between (at least) two compartments in a single address space. Software fault isolation [59] and

Native Client [64] leverage binary rewriting and restrictions on binary code to separate compartments and control interactions. MemTrace [47] 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 [45] leverages a Trusted Platform Module (TPM) chip to enforce strong isolation. TrustVisor [44] 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 [66], CODOM [58], or CHERI [62] 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 [29] and PREC [40] sandbox Android apps and Native Client sandboxes native code in the Chrome web browser [64]. In contrast, Milkomeda protects a vetting layer from an untrusted app within its own process.

10 LIMITATIONS AND FUTURE WORK

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 Figure 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 §2.4, 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 [14]. 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 [27], 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.

11 CONCLUSIONS

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, a framework for GPU-based graphics acceleration for web apps. 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 using 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 shows that Milkomeda achieves high graphics performance for various mobile apps, although at the cost of moderately increased CPU utilization.

ACKNOWLEDGMENTS

This work was supported by NSF awards #1617513, #1513783, and ONR award N00014-17-1-2513. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. We thank the paper shepherd, Adwait Nadkarni, and the anonymous CCS reviewers for their insightful comments. The pseudocode used in Figure 6 was adopted and modified from a similar one suggested by one of the reviewers.

REFERENCES

[1] 2011. Chromium Issue 70718: Crashes when opening a page with webgl. https://bugs.chromium.org/p/chromium/issues/detail?id=70718.
[2] 2011. Microsoft considers WebGL harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx.
[3] 2014. GPU Accelerated Compositing in Chrome. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.
[4] 2015. WebGL* in Chromium*: Behind the scenes. https://software.intel.com/en-us/articles/webgl-in-chromium-behind-the-scenes.
[5] 2017. A new multi-process model for Firefox. https://hacks.mozilla.org/2017/06/firefox-54-e10s-webextension-apis-css-clip-path/.
[6] 2017. Best Practices for Working with Vertex Data. https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/

OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html.
[7] 2017. Certain types of loops in WebGL shaders cause GLSL compiler crashes on Adreno. https://bugs.chromium.org/p/chromium/issues/detail?id=784817.
[8] 2017. National Vulnerability Database. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd.
[9] 2017. OpenGL ES Benchmark 1. https://github.com/googlesamples/android-ndk/tree/master/hello-gl2.
[10] 2017. SELinux. https://wiki.centos.org/HowTos/SELinux.
[11] 2017. The Common Vulnerability Scoring System version 2. https://www.first.org/cvss/v2/.
[12] 2017. WebGL Security. http://www.khronos.org/webgl/security/.
[13] 2018. Android NDK. https://developer.android.com/ndk/index.html.
[14] 2018. ANGLE: From OpenGL to Direct3D and back again. https://docs.google.com/presentation/d/1CucIsdGVDmdTWRUbg68IxLE5jXwCb2y1E9YVhQo0thg/pub?slide=id.g26efd2cf6_0178.
[15] 2018. AppArmor. https://wiki.ubuntu.com/AppArmor.
[16] 2018. Client-Side Vertex Arrays. https://www.khronos.org/opengl/wiki/Client-Side_Vertex_Arrays.
[17] 2018. 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/.
[18] 2018. glTexImage2D specification – OpenGL ES 3.0. https://www.khronos.org/registry/OpenGL-Refpages/es3.0/html/glTexImage2D.xhtml.
[19] 2018. Google Play Instant. https://developer.android.com/topic/google-play-instant/.
[20] 2018. GPU Command Buffer - The Chromium Projects. https://www.chromium.org/developers/design-documents/gpu-command-buffer.
[21] 2018. OpenGL ES Benchmark 2. https://github.com/googlesamples/android-ndk/tree/master/gles3jni.
[22] 2018. OpenGL ES Benchmark 3. https://github.com/learnopengles/Learn-OpenGLES-Tutorials (Lesson 5).
[23] 2018. OpenGL ES Benchmark 4. https://github.com/learnopengles/Learn-OpenGLES-Tutorials (Lesson 7).
[24] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005. Control-Flow Integrity. In *Proc. ACM CCS*.
[25] A. Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*.
[26] 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).
[27] ARM. 2007. Architecture Reference Manual, ARMv7-A and ARMv7-R edition. *ARM DDI* 0406A (2007).
[28] ARM. 2013. Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. *ARM DDI* 0487A.a (ID090413) (2013).
[29] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. USENIX Security Symposium*.
[30] 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*.
[31] A. Bittau, P. Marchenko, M. Handley, and B. Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proc. USENIX NSDI*.
[32] D. Brumley and D. Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium*.
[33] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)* (2017).
[34] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*.
[35] Ding, R. and Qian, C. and Song, C. and Harris, B. and Kim, T. and Lee, W. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proc. USENIX Security Symposium*.
[36] K. Elphinstone and G. Heiser. 2013. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?. In *Proc. ACM SOSP*.
[37] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension. In *Proc. USENIX Security Symposium*.
[38] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *Proc. IEEE Security and Privacy (S&P)*. bibtex: frigo2018.
[39] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. 2008. The Design and Implementation of Microdrivers. In *Proc. ACM ASPLOS*.
[40] T. Ho, D. Dean, X. Gu, and W. Enck. 2014. PREC: Practical Root Exploit Containment for Android Devices. In *Proc. ACM CODASPY*.
[41] T. C. Hsu, K. Hoffman, P. Eugster, and M. Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proc. ACM CCS*.
[42] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. 2018. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proc. ACM MobiSys*.
[43] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proc. USENIX OSDI*.
[44] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. IEEE Symposium on Security and Privacy (S&P)*.
[45] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. 2008. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM EuroSys*.
[46] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. 2011. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS*.
[47] M. Payer, E. Kravina, and T. R. Gross. 2013. Lightweight Memory Tracing. In *Proc. USENIX ATC*.
[48] 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)*.
[49] N. Provos. 2003. Improving Host Security with System Call Policies. In *Proc. USENIX Security Symposium*.
[50] N. Provos, M. Friedl, and P. Honeyman. 2003. Preventing Privilege Escalation. In *Proc. USENIX Security Symposium*.
[51] A. Razeen, A. R. Lebeck, D. H. Liu, A. Meijer, V. Pistol, and L. P. Cox. 2018. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *Proc. ACM MobiSys*.
[52] F. Roesner and T. Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Proc. USENIX Security Symposium*.
[53] Seccomp 2018. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
[54] S. Shekhar, M. Dietz, and D. S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. USENIX Security Symposium*.
[55] R. Strackx, P. Agten, N. Avonds, and F. Piessens. 2015. Salus: Kernel Support for Secure Process Compartments. *EAI Endorsed Transactions on Security and Safety* (2015).
[56] M. M. Swift, B. N. Bershad, and H. M. Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*.
[57] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proc. ACM CCS*.
[58] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. 2014. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proc. ACM/IEEE ISCA*.
[59] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proc. ACM SOSP*.
[60] J. Wang, X. Xiong, and P. Liu. 2015. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *Proc. USENIX ATC*.
[61] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proc. USENIX Security Symposium*.
[62] 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. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. ACM/IEEE ISCA*.
[63] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran. 2018. Sugar: Secure GPU Acceleration in Web Browsers. In *Proc. ACM ASPLOS*.
[64] 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)*.
[65] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proc. ACM MobiSys*.
[66] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proc. USENIX OSDI*.
[67] H. Zhang, D. She, and Z. Qian. 2015. Android Root and its Providers: A Double-Edged Sword. In *Proc. ACM CCS*.
[68] 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*.
[69] S. Zhu, L. Lu, and K. Singh. 2016. Case: Comprehensive Application Security Enforcement on COTS Mobile Devices. In *Proc. ACM MobiSys*.