Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU

Pietro Frigo
Vrije Universiteit Amsterdam
p.frigo@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Kaveh Razavi
Vrije Universiteit Amsterdam
kaveh@cs.vu.nl

Abstract—**Dark silicon is pushing processor vendors to add more specialized units such as accelerators to commodity processor chips. Unfortunately this is done without enough care to security. In this paper we look at the security implications of integrated Graphical Processor Units (GPUs) found in almost all mobile processors. We demonstrate that GPUs, already widely employed to accelerate a variety of benign applications such as image rendering, can also be used to "accelerate" microarchitectural attacks (i.e., making them more effective) on commodity platforms. In particular, we show that an attacker can build all the necessary primitives for performing effective GPU-based microarchitectural attacks and that these primitives are all exposed to the web through standardized browser extensions, allowing side-channel and Rowhammer attacks from JavaScript. These attacks bypass state-of-the-art mitigations and advance existing CPU-based attacks: we show the first end-to-end microarchitectural compromise of a browser running on a mobile phone in under two minutes by orchestrating our GPU primitives. While powerful, these GPU primitives are not easy to implement due to undocumented hardware features. We describe novel reverse engineering techniques for peeking into the previously unknown cache architecture and replacement policy of the Adreno 330, an integrated GPU found in many common mobile platforms. This information is necessary when building shader programs implementing our GPU primitives. We conclude by discussing mitigations against GPU-enabled attackers.**

I. Introduction

Microarchitectural attacks are increasingly popular for leaking secrets such as cryptographic keys [38], [49] or compromising the system by triggering bit flips in memory [41], [44], [46], [48]. Recent work shows that these attacks are even possible through malicious JavaScript applications [7], [18], [20], [37], significantly increasing their real-world impact. To counter this threat, the research community has proposed a number of sophisticated defense mechanisms [8], [9], [28]. However, these defenses implicitly assume that the attacker's capabilities are limited to those of the main CPU cores.

In this paper, we revisit this assumption and show that it is insufficient to protect only against attacks that originate from the CPU. We show, for the first time, that the Graphical Processing Units (GPUs) that manufacturers have been adding to most laptops and mobile platforms for years, do not just accelerate video processing, gaming, deep learning, and a host of other benign applicatons, but also boost microarchitectural attacks. From timers to side channels, and from control over physical memory to efficient Rowhammer attacks, GPUs offer all the necessary capabilities to launch advanced attacks. Worse, attackers can unlock the latent power of GPUs even from JavaScript code running inside the browser, paving the way for a new and more powerful family of remote microarchitectural attacks. We demonstrate the potential of such attacks by bypassing state-of-the-art browser defenses [9], [28], [43] and presenting the first reliable GPU-based Rowhammer attack that compromises a browser on a phone in under two minutes.

We specifically focus on mobile platforms given that, on such platforms, triggering Rowhammer bit flips in sandboxed environments is particularly challenging and has never been demonstrated before. Yet, mobile devices are particularly exposed to Rowhammer attacks given that catch-all defenses such as ANVIL [4] rely on efficient hardware monitoring features that are not available on ARM.

Integrated Processors While transistors are becoming ever smaller allowing more of them to be packed in the same chip, the power to turn them all on at once is stagnating. To meaningfully use the available dark silicon for common, yet computationally demanding processing tasks, manufacturers are adding more and more specialized units to the processors, over and beyond the general purpose CPU cores [12], [14], [47]. Examples include integrated cryptographic accelerators, audio processors, radio processors, network interfaces, FPGAs, and even tailored processing units for artificial intelligence [42]. Unfortunately, the inclusion of these special-purpose units in the processor today appears to be guided by a basic security model that mainly governs access control, while entirely ignoring the threat of more advanced microarchitectural attacks.

GPU-based Attacks One of the most commonly integrated components is the Graphical Processing Unit (GPU). Most laptops today and almost all mobile devices contain a programmable GPU integrated on the main processor's chip [26]. In this paper, we show that we can build all necessary primitives for performing powerful microarchitectural attacks directly from this GPU. More worrying still, we can perform these attacks directly from JavaScript, by exploiting the WebGL API which exposes the GPU to remote attackers.

More specifically, we show that we can program the GPU to construct very precise timers, perform novel side channel attacks, and, finally, launch more efficient Rowhammer attacks from the browser on mobile devices. All steps are relevant.

Precise timers serve as a key building block for a variety of side-channel attacks and for this reason a number of state-of-the-art defenses specifically aim to remove the attackers' ability to construct them [9], [28], [43]. We will show that our GPU-based timers bypass such novel defenses. Next, we use our timers to perform a side-channel attack from JavaScript that allows attackers to detect contiguous areas of physical memory by programming the GPU. Again, contiguous memory areas are a key ingredient in a variety of microarchitectural attacks [20], [46]. To substantiate this claim, we use this information to perform an efficient Rowhammer attack from the GPU in JavaScript, triggering bit flips from a browser on mobile platforms. To our knowledge, we are the first to demonstrate such attacks from the browser on mobile (ARM) platforms. The only bit flips on mobile devices to date required an application with the ability to run native code with access to uncached memory, as more generic CPU cache eviction were found too inefficient to trigger bit flips [46]. In contrast, our approach generates hundreds of bit flips directly from JavaScript. This is possible by using the GPU to (i) reliably perform double-sided Rowhammer and, more importantly, (ii) implement a more efficient cache eviction strategy.

Our end-to-end attack, named GLitch, uses all these GPU primitives in orchestration to reliably compromise the browser on a mobile device using only microarchitectural attacks in under two minutes. In comparison, even on PCs, all previous Rowhammer attacks from JavaScript require non default configurations (such as reduced DRAM refresh rates [7] or huge pages [20]) and often take such a long time that some researchers have questioned their practicality [8].

Our GLitch exploit shows that browser-based Rowhammer attacks are entirely practical even on (more challenging) ARM platforms. One important implication is that it is *not* sufficient to limit protection to the kernel to deter practical attacks, as hypothesized in previous work [8]. We elaborate on these and further implications of our GPU-based attack and explain to what extent we can mitigate them in software.

As a side contribution, we report on the reverse engineering results of the caching hierarchy of the GPU architecture for a chipset that is widely used on mobile devices. Constructing attack primitives using a GPU is complicated in the best of times, but made even harder because integrated GPU architectures are mostly undocumented. We describe how we used performance counters to reverse engineer the GPU architecture (in terms of its caches, replacement policies, etc.) for the Snapdragon 800/801 SoCs, found on mobile platforms such as the Nexus 5 and HTC One.

Contributions We make the following contributions:

- The first study of the architecture of integrated GPUs, their potential for performing microarchitectural attacks, and their accessibility from JavaScript using the standardized WebGL API.
- A series of novel attacks executing directly on the GPU, compromising existing defenses and uncovering new grounds for powerful microarchitectural exploitation.
- The first end-to-end remote Rowhammer exploit on mobile platforms that use our GPU-based primitives in orchestration to compromise browsers on mobile devices in under two minutes.
- Directions for containing GPU-based attacks.

Layout We describe our threat model in Section II before giving a brief overview of the graphics pipeline in Section III. In Section IV, we discuss the high-level primitives that the attackers require for performing microarchitectural attacks and show how GPUs can help building these primitives in Section V, VI, VII and VIII. We then describe our exploit, GLitch, that compromises the browser by orchestrating these primitives in Section IX. We discuss mitigations in Section X, related work in Section XI and conclude in Section XII. Further information including a demo of GLitch can be found in the following URL: https://www.vusec.net/projects/glitch.

II. Threat Model

We consider an attacker with access to an integrated GPU. This can be achieved either through a malicious (native) application or directly from JavaScript (and WebGL) when the user visits a malicious website. For instance, the attack vector can be a simple advertisement controlled by the attacker. To compromise the target system, we assume the attacker can only rely on microarchitectural attacks by harnessing the primitives provided by the GPU. We also assume a target system with all defenses up, including advanced research defenses (applicable to the ARM platform), which hamper reliable timing sources in the browser [9], [28] and protect kernel memory from Rowhammer attacks [8].

III. GPU rendering to the Web

OpenGL is a cross-platform API that exposes GPU hardware acceleration to developers that seek higher performances for graphics rendering. Graphically intensive applications such as CAD, image editing applications and video games have been adopting it for decades in order to improve their performances. Through this API such applications gain hardware acceleration for the rendering pipeline fully exploiting the power of the underlying system.

The rendering pipeline: The rendering pipeline consists of 2 main stages: *geometry* and *rasterization*. The geometry step primarily executes transformations over polygons and their vertices while the rasterization extracts fragments from these polygons and computes their output colors (i.e., pixels). *Shaders* are GPU programs that carry out the aforementioned operations. These are written in the OpenGL Shading Language (GLSL), a C-like programming language part of the specification. The pipeline starts from *vertex* shaders that performs geometrical transformations on the polygons' vertices provided by the CPU. In the rasterization step, the polygons are passed to the *fragment* shaders which compute the output color value for each pixel usually using desired *textures*. This output of the pipeline is what is then displayed to the user.

WebGL: WebGL is the result of the increasing demand of porting the aforementioned graphically intensive applications to the Web. This API exposes the GPU-accelerated rendering pipeline to the Web to bolster the development of such applications. Currently supported by every major browser [2] it provides most of the functionalities accessible from the OpenGL ES 2.0 API. Since it was conceived with the purpose of porting native graphics application to the Web, the anatomy of these two APIs is almost equivalent. This means that the aforementioned shaders can be compiled and run seamlessly from both the environments providing a fast lane to hardware acceleration to every JavaScript-enabled developer.

While these APIs were designed with the purpose of accelerating image rendering we will show through out this paper how this *acceleration* acquires another meaning while we exploit it to build the necessary primitives to carry out microarchitectural attacks.

IV. Attacker Primitives

"*Microarchitectural attacks*" aim to either (*a*) steal data using variety of side channels or (*b*) corrupt data using hardware vulnerabilities such as Rowhammer.

In this section we analyze the two aforementioned attacks' families identifying the required primitives that the attackers need to conduct them. We further explore why GPU "*accelerates*" these attacks; i.e., makes them more effective than what is possible when malicious code runs on the CPU.

A. Leaking data

A primary mechanism for leaking data using microarchitectural attacks is to time operations over resources shared with a victim process. For example, in a `FLUSH+RELOAD` cache attack [49], the attacker checks whether accessing a shared memory page with a victim is suddenly faster, which reveals that the victim has accessed the shared page, bringing it to the cache. In `FLUSH+RELOAD` and many other popular variants [38], the attacker needs a mechanism that can tell whether a certain memory operation on the memory hierarchy executed fast or slow. This hints at our first primitive:

***P1*. Timers:** Having access to high-resolution timers is a primary requirement for building timing side-channel attacks. There are many examples of these attacks executed natively [6], [21], [38], [40], [49], but more recently Oren et al. [37] showed that it is possible to mount such attacks from JavaScript, extending the threat model to remote exploitation.

In response, browser vendors immediately reduced the resolution of JavaScript timers in order to thwart these attacks [10], [11], [18], [50], but recent studies demonstrated other possibilities to construct timers in the browser [18], [28]. In response, more advanced defenses such as FuzzyFox [28] introduces randomness in the JavaScript event loop to add noise to timing measurements performed by an attacker. Similarly, DeterFox [9] attempts to make all interactions to/from browser frames that have a secret deterministic in order to stop an attacker to time operations that involve secret data.

We show in Section V how WebGL can be used for building high-precision timing primitives that are capable of measuring both CPU and GPU operations, bypassing all existing, even advanced defenses. Moreover, the very high precision of the resulting timers helps craft fast and reliable side-channel attacks, minimizing the impact of noise. Finally, while it has been argued that timing-prone browser extensions such as `SharedArrayBuffer` objects have relatively limited applicability and can be easily disabled for security reasons [9], [43], WebGL is deeply embedded into many Web applications and disabling it is hardly a pain-free option.

***P2*. Shared resources:** Another fundamental requirement in a side-channel attack is having access to resources shared with other (distrusting) processes. For example, in a cache attack used to leak information from a victim process, the cache should be shared by the attacker process. Previous work shows variety of ways for leaking information over shared resources, such as CPU data caches [18], [38], [49], the translation lookaside buffer [23] and memory pages [7], [40]. Co-processors, such as (untrusted) GPUs, may share various resources with the CPU cores, but at the very least, they share memory pages with the rest of the system.

We discuss how the integrated GPU of a modern ARM processor can get access to the system memory in Section VI, allowing an attacker to perform a side-channel attack directly from the GPU. To do this, an attacker needs to bypass multiple levels of *undocumented* GPU caches which we reverse engineer and report on for the first time as part of this work. Unlike CPU caches that are large and optimize for a general-purpose workload by implementing either random [30] or non-deterministic [20] replacement policies, we show that GPU caches are small and follow a deterministic replacement policy. This allows an attacker to reason about cache hits or misses with great precision, paving the way for fast and reliable side-channel attacks with little noise, as we will show in Section VII.

B. Corrupting data

Rowhammer is a prime example of an attack that corrupts data by abusing a hardware fault. Previous work shows that it is possible to corrupt page tables for privilege escalation [44], [46], compromise the browser [7], [20] and cloud VMs [41], [48]. The main obstacles in performing these attacks are *(I)* knowing the physical location of the targeted row and *(II)* fast memory access [46].

***P3*. Knowledge of the physical location:** Knowing the physical location of allocated memory addresses is a requirement in order to understand which rows to *hammer*. The typical approach is to exploit physically contiguous memory in order to gain knowledge of *relative* physical addresses. Previous work abuses the transparent huge page mechanism that is on-by-default on x86_64 variants of Linux [20], [41], [44], which provided them with 2 MB of contiguous physical memory. Huge pages are off-by-default on ARM. To address this requirement, the Drammer attack [46] abuses the physically contiguous memory provided by the Android ION allocator.

This remains a fundamental requirement even when approaching this from the GPU. We discuss how we can use a novel timing side-channel executed from the GPU that mixes the knowledge of the DRAM architecture [40] and low-level memory management to find contiguous physical regions of memory from the browser in Section VII.

***P4*. Fast memory access:** Accessing memory quickly is a necessary condition when performing Rowhammer attacks. In order to be able to trigger bit flips, in fact, the attacker needs to quickly access different DRAM rows. The CPU caches, however, absorb most, if not all, of these reads from DRAM. On the x86 architecture, flushing the CPU cache using the unprivileged `clflush` instruction is a common technique to bypass the caches [41], [44], [48]. On most ARM platforms, evicting the CPU cache is a privileged operation. Drammer [46] hence relies on uncached DMA memory provided by the Android ION allocator for hammering.

In the browser, there is no possibility for executing cache flush instructions or conveniently accessing DMA memory through JavaScript. Rowhammer.js [20] and Dedup Est Machina [7] rely on eviction buffers to flush the cache. While this works on x86, flushing CPU caches on ARM is too slow to trigger flips [46]. Hence, it remains an open question whether it is possible to perform Rowhammer attacks from the browser on most mobile devices.

In Section VIII, we report on the first successful Rowhammer bit flips in the browser on ARM devices. This is now possible from the GPU by (i) enabling more efficient double-sided Rowhammer variant with physical memory layout information leaked by *P3*, and, more importantly, (ii) implementing an efficient cache eviction (and thus hammering) strategy due to the small size and deterministic behavior of the GPU caches.

We use these bit flips to build GLitch in Section IX, our reliable end-to-end exploit that orchestrates all the GPU-based primitives we described to compromise the browser running on a mobile phone in less than two minutes by relying on microarchitectural attacks alone.

V. The Timing Arms Race

To implement timing side-channel attacks, attackers need the ability to time a secret operation (*P1*). These attacks often require fine-grained timing. For example, in case of timing attacks on the caches, the attacker must be able to tell the difference between a cache access and a memory access with a difference of tens of nanoseconds. Recent work shows that it is possible to perform these attacks in JavaScript from the browser [7], [18], [20], [37], extending their threat to the Internet users. In response, different defenses have been proposed to protect browsers from these attacks. Browser vendors reduced the resolution of the `performance.now()` timer [10], [11], [39], [50] in order to make it harder for attackers to perform timing attacks in JavaScript. However, recent work shows that simply reducing the timer resolution is not enough: it is possible to use another core as a timing source [18] or use other techniques such as clock-edging or edge-thresholding to improve the resolution of a coarse timer [28]. More fundamental approaches either introduce noise in the measurements [28] or make interaction among browser frames deterministic to avoid secret-dependant timings [9].

In this section, we present explicit and implicit GPU-based timing sources, demonstrating how such defenses are flawed due to their incomplete threat model that does not take the GPU into account. We start by presenting two explicit timing sources showing how these allow us to time both GPU's and CPU's operations. We then present two other commutable implicit timers based on the second revision of the WebGL API. We test all these timers against major browsers as well as the state of the art defenses mentioned above (i.e., FuzzyFox and DeterFox) and discuss the implications of these timing sources.

A. Explicit GPU timing sources

`EXT_DISJOINT_TIMER_QUERY` is an OpenGL extension developed to provide developers with more detailed information about the performance of their applications [45]. This extension, if made available to the system by the GPU driver, is accessible from both WebGL and WebGL2, and provides the JavaScript runtime with two timing sources: (*1*) `TIME_ELAPSED_EXT` and (*2*) `TIMESTAMP_EXT`. Such timers allow an attacker to measure the timing of secret operations (e.g., memory accesses) performed either by the CPU or the GPU.

***T1*. TIME_ELAPSED_EXT:** This functionality allows JavaScript code to query the GPU asynchronously to measure how much time the GPU took to execute an operation. While there are different instances of JavaScript-based side-channels on the CPU [7], [18], [37], there are no current examples of remote GPU-based attacks. In Section VII, we will show how we can use the timer we are now presenting to implement a the first timing side-channel targeting DRAM executed directly on a remote GPU.

Since `TIME_ELAPSED_EXT` is based on a WebGL extension that requires the underlying OpenGL extension to be accessible, its availability and resolution are driver and browser dependent. The specification of the extension requires the return value to be stored as a `uint64` in a nanosecond variable as an implementation dependent feature, it does not guarantee nanosecond resolution, even in a native environment. Furthermore, when adding the browser's JavaScript engine on top of this stack the return value becomes browser-dependent as well. Firefox limits itself to casting the value to an IEEE754 double in accordance to the ECMAScript specification which does not support 64 bit integers, while Chrome rounds up the result to $1\,\mu s$, reducing the granularity of the actual measurements.

***T2*. TIMESTAMP_EXT:** Besides the asynchronous timer, the extension also provides a synchronous functionality for measuring CPU instructions. Specifically, by activating the extension the OpenGL context acquires a new parameter, `TIMESTAMP_EXT`, which the code can poll using the WebGL `getParameter()` function. The result is a synchronous timestamp returned from the GPU that can be used in lieu of the well-known `performance.now()` to measure CPU operations.

Like `TIME_ELAPSED_EXT`, this timer is driver- and browser-dependent. Firefox supports it, while Chrome disables it due to compatibility issues [15].

B. WebGL2-based timers

The timers introduced in the previous section are made available through a WebGL extension. We now demonstrate how WebGL represents a more fundamental issue in the timing arms race, by showing how an attacker can craft homebrewed timers using only standard WebGL2 functions. WebGL2 is the latest version of the API and, while not as widely available as WebGL1 yet, it is supported by default in major browsers such as Chrome and Firefox.

The API provides two almost commutable timing sources based on WebGLSync, the interface that helps developers synchronize CPU and GPU operations. GLSync objects are fences that get pushed to the GPU command buffer. This command buffer is serialized and accepts commands sequentially. WebGL2 provides the developer with several functions to synchronize the two processors, and we use two of them to craft our timers: `clientWaitSync()` and `getSyncParameter()`.

***T3*. clientWaitSync:** This function waits until either the sync object receives a signal, or a timeout event occurs. The attacker first sets a threshold and then checks the function's return value to see if the operation completed (`CONDITION_SATISFIED`) or a timeout occurred (`TIMEOUT_EXPIRED`) Unfortunately, the timeout has an implementation-defined upper bound (`MAX_CLIENT_WAIT_TIMEOUT_WEBGL`) and therefore may not work in all cases. For instance, Chrome sets this value to 0 to avoid CPU stalls. To address this problem, we adopted a technique which we call *ticks-to-signal* (TTS) which is similar to the clock-edging proposed by Kolhbrenner and Shacham [28]. It consists of calling the `clientWaitSync()` function in a tight loop with the timeout set to 0 and counting until it returns `ALREADY_SIGNALED`. The full timing measurement consists of several smaller steps: first ① flush the command buffer, and ② dispatch the command to the GPU, then ③ issue the WebGLSync fence, and finally ④ count the loops of `clientWaitSync(0)` until it is signaled.

If measuring a secret CPU operation we execute the secret between steps ③ and ④. Whether the CPU or the GPU acts as ground truth depends on the secret the attacker is trying to leak. However, when measuring a secret CPU operation, we require the GPU operation to run in (*relatively*) constant time. Since the measurement requires a context change it can be more noisy to the timers based on `EXT_DISJOINT_TIMER_QUERY`. Nonetheless, this technique is quite effective, as we will show in Section V-C.

TABLE I: Results on different browsers for the two families of timers. With † we indicate driver dependent values.

	Chrome	Firefox	FuzzyFox	DeterFox
`TIME_ELAPSED_EXT`	$1\,\mu s$	$100^{\dagger}\,ns$	-	-
`TIMESTAMP_EXT`	-	$1.8^{\dagger}\,\mu s$	-	-
`clientWaitSync`	$60\,ns$	$0.4\,ns$	$0.4\,ns$	$1.8\,ns$
`getSyncParameter`	$60\,ns$	$0.4\,ns$	$0.4\,ns$	$1.8\,ns$

***T4*. getSyncParameter:** This function provide an equivalent solution. If called with `SYNC_STATUS` as parameter after issuing a fence, it returns either `SIGNALED` or `UNSIGNALED`, which exactly anaologous to . `clientWaitSync(0)`.

The timers we build using both these functions work on every browser that supports the WebGL2 standard (such as Chrome and FireFox). In fact, in order to comply with the WebGL2 specification none of these functions *can* be disabled. Also, due to the synchronous nature of these timers, we can use them to measure both CPU and GPU operations.

C. Evaluation

We evaluate our timers against Chrome and Firefox, as well as two Firefox-derived browsers that implement state-of-the-art defenses in effort to stop high-precision timing: FuzzyFox [28] and DeterFox [9]. We use a laptop equipped with an Intel Core i5-6200U processor that includes an integrated Intel HD Graphics 520 GPU for the measurements. We further experimented with the same timers on an integrated Adreno 330 GPU on an ARM SoC when developing our side-channel attack in Section VII

Table I shows the results of our experiments. The two explicit timers, as mentioned before, are driver-/browser-dependent, but if available, return unambiguous values. So far, we found that the extension is available only on Firefox. Both Fuzzyfox and DeterFox disable it, without any mention of it in their manuscripts [9], [28]. Chrome rounds up the value for `TIME_ELAPSED_EXT` to $1\,\mu s$ and returns 0 for `TIMESTAMP_EXT`.

The two WebGL2-based timers, however, are effective in all four browsers. On Chrome, we get a precision of $60\,ns$—enough to distinguish a cache from a memory access. On Firefox, Fuzzyfox and deterfox we managed to get ns and even sub-ns precision—close to the frequency of our target processor. We further tested our WebGL2-based timers against Chrome Zero [43], a Chrome plugin developed to protect users against side channels. This did not affect their resolution.

D. Discussion

We showed how the GPU provides an attacker with explicit timing sources directly and aids the crafting of new timers—allowing attackers to bypass state-of-the-art defenses from both industry and academia. As long as the JavaScript context can synchronously interact with external contexts such as

WebWorkers [18], WebGL and potentially others (e.g., audio), a diligent attacker can craft new timing sources. Our home-brewed timers are a demonstration of how tackling the threat posed by timing side-channels by besieging timing sources does not represent a viable and long term solution to the issue.

VI. A Primer on the GPU

Modern SoCs accommodate multiple co-processors within the same chip to gain better performances while saving space and power. In order to fully exploit the advantages of this design, these co-processors usually share resources over the memory hierarchy. In this section, we look at the general architecture of integrated GPUs before studying a concrete GPU implementation on a commonly deployed ARM SoC. Like many similar implementations, this integrated GPU shares DRAM at the bottom of the memory hierarchy with the CPU (P2) However, to reach DRAM from the GPU, we need to evict two specialized GPU caches with an entirely different architecture than that of modern CPUs. We present a novel reverse engineering technique that makes usage of OpenGL shaders to reconstruct the architecture of these GPU caches. We use this knowledge in Section VII for building a DRAM-based side channel that leaks information about the layout of data in physical memory.

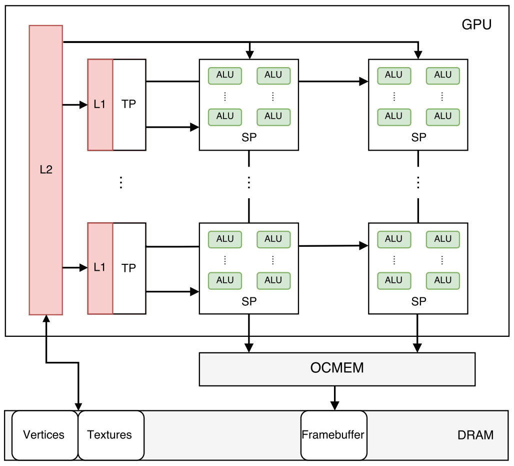


Fig. 1: Building blocks of an integrated GPU

A. The GPU architecture

A *Graphical Processing Unit* is a specialized circuit conceived with the purpose of accelerating image rendering. As mentioned in Section III, this system aids the rendering pipeline by executing the shaders provided by the developer. We now discuss how the GPU architecture implements this pipeline.

Processing units: Figure 1 shows the general architecture of a GPU. The *Stream Processors* (SPs) are the fundamental units of the GPU that are in charge of running the shaders. To maximize throughput when handling inputs, GPUs include multiple SPs, each incorporating multiple ALUs to further parallelize the computation. Shaders running on the SPs can then query the *texture processors* (TPs) to fetch additional input data used during their computations. This data is typically in the form of textures used to compute the fragment colors to which TPs apply filters of different natures (e.g., anti-aliasing).

GPU caching: During their execution, shaders can request external data in the form of textures by querying the TPs. All this data is stored on DRAM due to its large size. Since fetching data from DRAM is slow and can cause pipeline stalls, GPUs often include a two-level private cache (i.e., L1 and L2) to speed up accesses to vertices and textures. While the larger L2 is used by both SPs, to store vertices, and TPs to store textures, the latter makes use of a faster (but smaller) L1 cache to further speed the inner execution of the shader. We later discuss the architecture of these caches in the Adreno 330 GPU.

In order to increase performances when writing to framebuffers, integrated GPUs are usually equipped with smaller chunks of faster on-chip memory (*OCMEM*) that allows them to store portions of the render target and to asynchronously transfer them back to DRAM, as shown in Figure 1.

B. The Adreno 330: A case study

To better understand the architecture of integrated GPUs, we analyze the Adreno 330, a GPU found in the common Snapdragon 800/801 mobile SoCs. These SoCs are embedded in many Android devices such as LG Nexus 5, HTC One, LG G2 and OnePlus One.

The A330 exposes a similar architecture to what we described earlier in this section. Main peculiarity of this system, however, is the presence of an IOMMU in between DRAM and the L2 cache (known as *UCHE*). This essentially means that the GPU operates on virtual memory rather than physical memory, unlike the CPU cores.

Considering the architecture in Figure 1, an attacker can access memory either by inputing vertices to the vertex shaders or fetching textures within the shaders themselves for building a P2 primitive. Another possibility for accessing memory is by writing to the framebuffer. All these operations, however, need careful access patterns that avoid the caches or the OCMEM in order to reach memory. We found that buffers containing vertices are lazily instantiated and the implicit synchronization between parallel executions of the same shader on different SPs makesa it difficult for an attacker to achieve predictable behavior when accessing memory. Accessing memory through OCMEM is also tricky given its larger size and asynchronous transfers. We hence opted for texture fetching. Texture fetching takes place within the boundaries of a shader, providing strong control over the order of the memory accesses. Moreover, textures' allocations are easy to control, making it possible to obtain more predictable memory layouts as we explain in Section VII.

The remaining obstacle is dealing with L1 and L2 in between the shaders and the DRAM, and the less obvious *texture addressing* necessary for converting from pixel coordinates to (virtual) memory locations. We start by analyzing this mapping function which allows us to access desired memory addresses before analyzing the cache architecture in A330. We then use this information to selectively flush the GPU caches in order to reach DRAM.

1) Texture addressing: Integrated GPUs partition textures in order to maximize spatial locality when fetching them from DRAM [5], [16]. Known as *tiling*, this is done by aggregating data from close *texels* (i.e. texture pixels) and storing them consecutively in memory so that they can be collectively fetched. Tiling is frequently used on integrated GPUs due to the limited bandwidth available to/from system memory.

These tiles, in the case of the A330, are 4×4 pixels. We can store each pixel's data in different internal formats, with `RGBA8` being one of the most common. This format stores each channel in a single byte. Therefore, a texel occupies 4 bytes and a tile 64 bytes.

Without tiling, translation from (x, y) coordinates to virtual address space is as simple as indexing in a 2D matrix. Unfortunately tiling makes this translation more complex by using the following function to identify the pixel's offset in an array containing the pixels' data:

$$f(x,y) = \left(\frac{y}{T_H} * \frac{W + T_W - 1}{T_W} + \frac{x}{T_W}\right) * (T_W * T_H) + (y \bmod T_H) * T_W + x \bmod T_W$$

Here W is the width of the texture and T_W, T_H are respectively width and height of a tile.

With this function, we can now address any four bytes within our shader program in the virtual address space. However, given that our primitive *P2* targets DRAM, we need to address in the physical address space. Luckily, textures are page-aligned objects. Hence, their virtual and physical addresses share the lowest 12 bits given that on most modern architectures a memory page is 4 KB.

2) Reverse engineering the caches: Now that we know how to access memory with textures, we need to figure out the architecture of the two caches in order to be able to access DRAM through them. Before describing our novel reverse engineering technique and how we used it to understand the cache architecture we briefly explain the way caches operate.

Cache architecture: A cache is a small and fast memory placed in-between the processor and DRAM that has the purpose of speeding up memory fetches. The size of a cache usually varies from hundreds of KBs to few MBs. In order to optimize spatial locality the data that gets accessed from DRAM is not read as a single word but as blocks of bigger size so that (*likely*) contiguous accesses will be already cached. These blocks are known as *cachelines*. To improve efficiency while supporting a large number of cachelines, caches are often divided into a number of sets, called *cache sets*. Cachelines, depending on their address, can be place in a specific cache set. The number of cachelines that can simultaneously be placed in a cache set is referred to as the *wayness* of the cache and caches with larger *ways* than one are known as *set-associative* caches.

```
#define MAX max // max offset
#define STRIDE stride // access stride

uniform sampler2D tex;

void main() {
vec4 val;
vec2 texCoord;
// external loop not required for (a)
for (int i=0; i<2; i++) {
for (int x=0; x < MAX; x += STRIDE) {
texCoord = offToPixel(x);
val += texture2D(tex, texCoord);
}
}
gl_Position = val;
}
```

Listing 1: Vertex shader used to measure the size of the GPU caches.

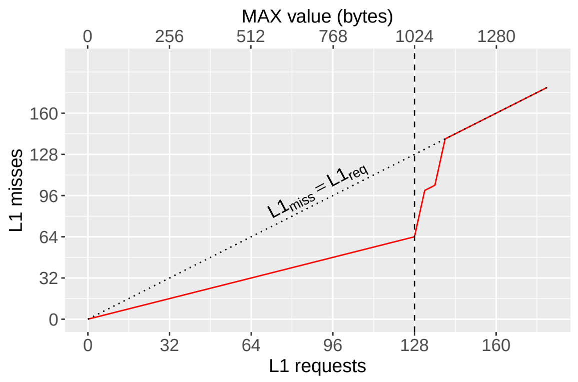
When a new cacheline needs to be placed in a cache set another cacheline needs to be *evicted* from the set to make space for the new cacheline. A predefined *replacement policy* decides which cacheline needs to be evicted. A common replacement is LRU or some approximation of it.

From this description we can deduce the four attributes we need to recover, namely (*a*) cacheline size, (*b*) cache size, (*c*) associativity and (*d*) replacement policy.

Reversing primitives: To gain the aforementioned details we (ab)use the functionalities provided by the GLSL code that runs on the GPU. Listing 1 presents the code of the shader we used to obtain (*b*). We use similar shaders to obtain the other attributes. The OpenGL's `texture2D()` function [19] interrogates the TP to retrieve the pixels' data from a texture in memory. It accepts two parameter: a *texture* and a bidimensional vector (`vec2`) containing the pixel's coordinates. The choice of these coordinates is computed by the function `offToPixel()` which is based on the inverse function $g(off) = (x, y)$ of $f(x, y)$ described earlier. The function `texture2D()` operates with *normalized device coordinates*, therefore we perform an additional conversion to normalize the pixel coordinates to the [-1,1] range. With this shader, we gain access to memory with 4 bytes granularity (dictated by the `RGBA8` format). We then monitor the usage of the caches (i.e., number of cache hits and misses) through the performance counters made available by the GPU's *Performance Monitoring Unit* (PMU).

Size: We can identify the cacheline size (*a*) and cache size (*b*) by running the shader in Listing 1 – with a single loop for (*a*). We initially recover the cacheline size by setting `STRIDE` to the smallest possible value (i.e., 4 bytes) and sequentially increasing `MAX` of the same value after every iteration. We recover the cacheline as soon as we encounter 2 cache misses ($C_{\text{miss}} = 2$). This experiment shows that the size of cacheline in L1 and UCHE are 16 and 64 bytes respectively.

We then set `STRIDE` to the cacheline size and run Listing 1



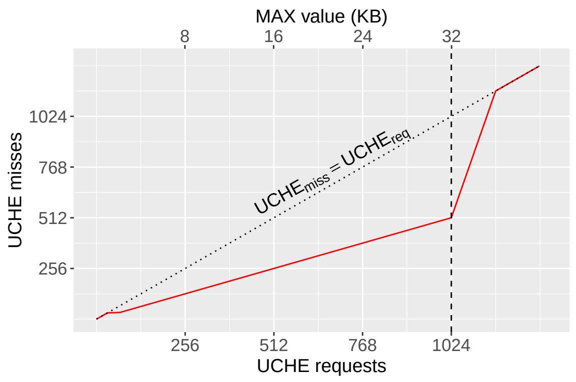


Fig. 2: Cache *misses* over cache *requests* for L1 and UCHE caches. The results are extracted using the GPU performance counter after each run of the shader in Listing 1 with `STRIDE` equal to cacheline size and increasing `MAX` value.

until the number of cache misses is not half of the requests anymore ($C_{\text{miss}} \neq C_{\text{req}}/2$). We run the same experiment for both L1 and UCHE. Figure 2 shows a sharp increase in the number of L1 misses when we perform larger accesses than 1 KB and for UCHE after 32 KB, disclosing their size.

Associativity and replacement strategy: The non-perpendicular rising edge in both of the plots in Figure 2 confirms they are set-associative caches and it suggest a LRU or FIFO replacement policy. Based on the hypothesis of a deterministic replacement policy we retrieved the details of the cache sets (c) by means of *dynamic eviction sets*. This requires two sets of addresses, namely S, a set that contains the necessary amount of elements to fill up the cache, and E, an eviction set that initially contains only a random address $E_0 \notin S$. We then iterate over the sequence $\{S, E, P_i\}$ where P_i is a probe element belonging to $S \cup E_0$. We perform the experiment for increasing i until P_i generates a cache miss. Once we detect the evicted cacheline, we add the corresponding address to E and we restart the process. We reproduce this until $P_i = E_0$. When this happens we have evicted every cacheline of the set and the elements in E can evict any cacheline that map to the same cache set (i.e., an *eviction set*). Hence, the size of E is *associativity* of the cache.

Once we identified the associativity of the caches, we can recover the replacement strategy (d) by filling up a cache set and accessing again the first element before the first eviction. Since this element gets evicted even after a recent use in both of the caches, we deduce a FIFO replacement policy for both L1 and UCHE.

Synopsis: All the details about these two caches are summarized in Table II. As can be seen from this table, there are many peculiarities in the architecture of these two caches and in their interaction. First, the two caches have different cacheline sizes, which is unusual when comparing to CPU caches. Then, L1 presents twice the ways UCHE has. One UCHE cacheline is split into 4 different L1 cachelines. The mapping function explained above shuffles these 4 subunits over two different L1 cache sets as shown in Figure 3. We will exploit this property when building efficient eviction sets

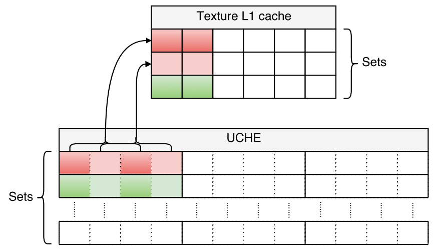


Fig. 3: Mapping of a 64-byte UCHE cacheline into multiple L1 cacheline over two different L1 sets.

TABLE II: Summary of the two level caches.

	L1	UCHE
Cacheline (*bytes*)	16	64
Size (*KB*)	1	32
Associativity (*#ways*)	16	8
Replacement policy	*FIFO*	
Inclusiveness	*non-inclusive*	

in Section VII. Finally, we discovered L1 and UCHE to be non-inclusive. This was to be expected considering that L1 has more ways than UCHE.

C. Generalization

Parallel programming libraries, such as CUDA or OpenCL, provide an attacker with a more extensive toolset and have already been proven to be effective when implementing side-channel attacks [24], [25], [33]. However, we decided to restrict our abilities to what is provided by the OpenGL ES 2.0 API in order to relax our threat model to remote WebGL-based attacks. Newer versions of the OpenGL API provide other means to gain access to memory such as image load/store, which supports memory qualifiers, or SSBOs (*Shader Storage Buffer Objects*), which would have given us linear addressing instead of the tiled addressing explained in Section VI-B1. However, they confine the threat model to local attacks carried out from a malicious application.

Furthermore, the reverse engineering technique we described in Section VI-B2 can be applied to other OSes and architectures without much effort. Most of the GPUs available nowadays are equipped with performance counters (e.g. Intel, AMD, Qualcomm Adreno, Nvidia) and they all provide a userspace interface to query them. We employed the `GL_AMD_performance_monitor` OpenGL extension which is available on Qualcomm, AMD and Intel GPUs. Nvidia, on the other hand, provides its own performance analysis tool called PerfKit [13].

VII. Side-Channel Attacks from the GPU

In Section VI, we showed how to gain access to remote system memory through the texture fetch functionality exposed from the WebGL shaders. In this section, we show how we are able to build an effective and low-noise DRAM side-channel attack directly from the GPU. Previous work [24], [25], [33] focuses on cross-GPU attacks in discrete GPU scenarios with a limited impact. To the best of our knowledge, this is the first report of a side-channel attack on the system from an integrated GPU that affects all mobile users. This attack benefits from the small size and the deterministic (FIFO) replacement policy of the caches in these integrated GPUs. We use this side channel to build a novel attack that can leak information about the state of physical memory. This information allows us to detect contiguous memory allocation (*P3*) directly in JavaScript, a mandatory requirement for building effective Rowhammer attacks.

First, we briefly discuss the DRAM architecture. We then describe how we are able to build efficient eviction sets to bypass two levels of GPU caches to reach DRAM. We continue by explaining how we manage to obtain contiguous memory allocations and finally we show how, by exploiting our timing side channel, we are able to detect these allocations.

A. DRAM architecture

DRAM chips are organized in a structure of channels, DIMMs, ranks, banks, rows and columns. *Channels* allow parallel memory accesses to increase the data transfer rate. Each channel can accommodate multiple Dual In-line Memory Modules (*DIMMs*). These modules are commonly partitioned in either one or two *ranks* which usually correspond to the physical front and back of the DIMM. Each rank is then divided into separate *banks*, usually 8 in DDR3 chips. Finally every bank contains the memory array that is arranged in *rows* and *columns*.

DRAM performs reads at row granularity. This means that fetching a specific word from DRAM activates the complete row containing that word. *Activation* is the process of accessing a row and storing it in the *row buffer*. If the row is already activated, a consecutive access to the same row will read directly from the row buffer causing a *row hit*. On the other hand, if a new row gets accessed, the current row residing in the buffer needs to be restored in its original location before loading the new one (*row conflict* [40]). We rely on this timing difference for detecting contiguous regions of physical memory as we discuss in Section VII-D.

B. Cache Eviction

Considering the GPU architecture presented in Section VI, the main obstacles keeping us from accessing the DRAM from the GPU is two levels of caches. Therefore, we need to build efficient eviction strategies to bypass these caches. From now on we will use the notation $v[\mathit{off}]$ to describe memory access to a specific *offset* from the start of an array v in the virtual address space.

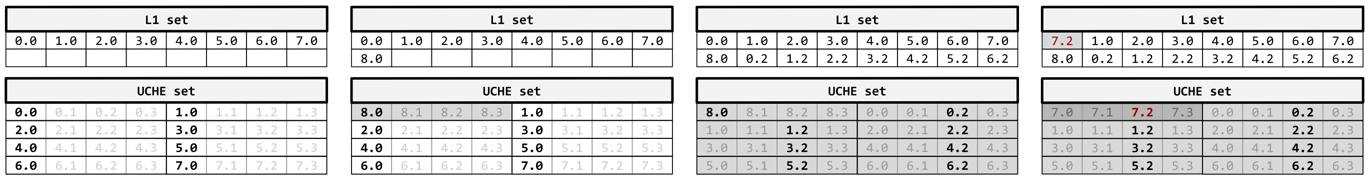
Set-associative caches require us to evict just the set containing the address $v[i]$, if we want to access $v[i]$ from memory again. Having a FIFO replacement policy allows us to evict the first cacheline loaded into the set by simply accessing a new address that will map to the same cache set. A UCHE set can store 8 cachelines located at 4 KB of stride (i.e., $v[4\text{K}\times i]$ as shown in Figure4.a). Hence, if we want to evict the first cacheline, we need at least 8 more memory accesses to evict it from UCHE (Figure 4.b). In a common scenario with *inclusive* caches, this would be enough to perform a new DRAM access. In these architectures, in fact, an eviction from the Last Level Cache (*LLC*) removes such cacheline from lower level caches as well. However, the non-inclusive nature of the GPU caches neutralizes this approach.

To overcome this problem we can exploit the particularities in the architecture of these 2 caches. We explained in Section VI-B2 that a UCHE cacheline contains 4 different L1 cachelines and that two addresses $v[64\times i]$ and $v[64\times i\text{+}32]$ map to two different cachelines into the same L1 set (Figure 3). As a result, if cacheline at $v[0]$ was stored in the UCHE and was already evicted, we can load it again from DRAM by accessing $v[0\text{+}32]$. By doing so we simultaneously load the new $v[0\text{+}32]$ cacheline into L1 (Figure 4.c). This property allows to evict both of the caches by alternating these 9 memory accesses between $v[4\text{K}\times i]$ and $v[4\text{K}\times i\text{+}32]$ (Figure 4.d). Our access patterns will exploit this to be completely oblivious of L1. As a consequence, from now on we will simply mention accesses to addresses $v[4\text{K}\times i]$. Nonetheless, every time we will use this notation it is important to remember that it implicitly conceals both of the accesses.

C. Allocating contiguous memory

Before discussing how the Adreno GPUs allocates memory, we need to explain the relationship between physical memory contiguity and DRAM adjacency.

Contiguity & Adjacency: In order to carry out a reliable Rowhammer attack, we need three *adjacent* rows inside a DRAM bank. It is important to understand that $adjacency \neq contiguity$. The memory controller decides where to store the data on a DRAM location based on the given physical address. Pessl et al. [40] reversed engineered the mapping between physical addresses and DRAM locations for the Snapdragon 800/801 chipsets. For simplicity, we adopt a simplified DRAM model and assume $contiguity \approxeq adjacency$, but the interested readers can find how we relax this assumption in Appendix A using the information in [40]. In the Snapdragon 800 each row n stores two consecutive pages (i.e., 8 KB). With 2 pages per row and 8 banks within the module, rows n and $n+1$ are 16 pages apart.



(a) (b) (c) (d)

Fig. 4: The diagrams show an efficient GPU cache (set) eviction strategy. We use the notation *a.b* to abbreviate the lengthy v[4K$\times a$+16$\times b$]. The eviction happens in 4 steps: (*a*) first we fill up the 8 slots available in a cache set by accessing v[4K$\times i$]; (*b*) after the cache set is full we evict the first element by accessing v[4K$\times$8]; (*c*) then, in order to access again v[0] from DRAM we need to actually read v[32] since v[0] is currently cached in L1. The same holds for every page v[4K$\times i$] for $i \in [1, 6]$; (*d*) finally, we evict the first L1 cacheline by performing our 17th access to v[4K$\times$7+32] which replaces v[0].

The buddy allocator: The Adreno 330 GPU operates on virtual addresses due to the presence of an IOMMU. This means that it is capable of dealing with physically non-contiguous memory and it allows the GPU driver to allocate it accordingly. The Adreno android kernel driver allocates memory using the `alloc_page()` macro which queries the buddy allocator for single pages [32].

The buddy allocator manages free memory in `free_lists` containing chunks of *power-of-two* number of available pages [17]. The exponent of this expression is known as the *order* of the allocation. Figure 5a shows these `free_lists`. When the buddy allocator receives a request for a block of memory, it tries to satisfy that allocation from the smallest possible orders. Figure 5b shows an example of such process. We want to allocate two buffers, namely a with 15 pages and b with 40 pages. We start by allocating a. `alloc_page()` asks for pages one by one (i.e., order 0 allocations). The order 0 `free_list` contains one single page. Therefore, the first allocation makes it empty. The following page then needs to come from order 1 (i.e., 2^1 contiguous pages). This means that buddy needs to split the block in two separate pages and return one back to the buffer while storing the other one in the order 0 `free_list`. This process is repeated for every requested page and can be executed for every order n < `MAX_ORDER`. That is, if no block of order n is vacant, a block from the next first available order (i.e., $n + k$) is recursively split in two *halves* until order n is reached. These halves are the so-called *buddies* and all the allocated memory coming from order $n + k$ is physically contiguous. As a consequence, considering our example in Figure 5b array a will be served by blocks of order 0, 1 and 3, while b by a single block of order 6, since all the small orders are exhausted.

We use this predictable behavior of the buddy allocator for building our primitive *P3*. Due to our precondition of 3 *adjacent* rows to perform a reliable Rowhammer attack, we therefore require an allocation of order $\lceil log_2(16_{\text{pages}} \times 3_{\text{row}})\rceil = 6$.

D. Detecting contiguous memory

Now that we know that we can force the buddy allocator into providing us with contiguous memory necessary to perform our Rowhammer attack, we require a *tool* to detect these contiguous areas. Our *tool* is a timing side-channel attack. We introduce a side channel that measures time differences between *row hits* and *row conflicts* in order to deduce information about the order of the allocations.

To distinguish between contiguous and non-contiguous allocations, we can either test for *row conflicts* or *row hits*. In Figure 5b, we allocated array b of 40 pages from an order six allocation that spans over four *full rows*. In our example, a *full row* is 64 KB of contiguous physical memory that maps to the same row n over the different banks. It would be intuitive to exploit the row conflicts to detect memory located in adjacent rows. For example, accessing b[0] and b[64 K] generates a row conflict since b is backed by physically-contiguous memory. However, this solution is limited due to the way buddy allocator works. We previously explained that to obtain a block of order n+1 from buddy we need to exhaust every n-order block available. This implies that allocations of order n are likely to be followed by other allocations of the same order. Since every allocation of order $\geq$ 4 spans over a full row, every access to allocations coming from these orders following the v[64K$\times i$] pattern will always generate row conflicts. At the same time, allocations of order < 4 are also likely to generate conflicts since the blocks in the buddy's `free_lists` are not predictable (Figure 5b). To address this problem, we detect blocks of order $\geq$ 4 by testing for row hits instead. This allows us to obtain the same granularity while achieving less noisy measurements.

This access pattern, which we call *hit*-pattern, touches 15 virtually-contiguous pages. To extract a single measurement we repeatedly iterate over it in order to minimize the noise. In Figure 5c we show how the *hit*-pattern behaves when touching pages belonging to arrays a and b. As you can see, sequential accesses over pages of b generate only row hits (**green** pages) while the same access pattern over a can arbitrarily generate row conflicts (**red** pages) or row hits depending on the backing allocations.

We limit our *hit*-pattern to 15 pages instead of the 16 pages of a full row because of the unknown physical alignment of our first access v[0]. For instance, in Figure 5c, we set our v[0]=b[13] and, as you can see, v[4K$\times$(16)](=b[29]) generates a row conflict since v[0] is not row-aligned.

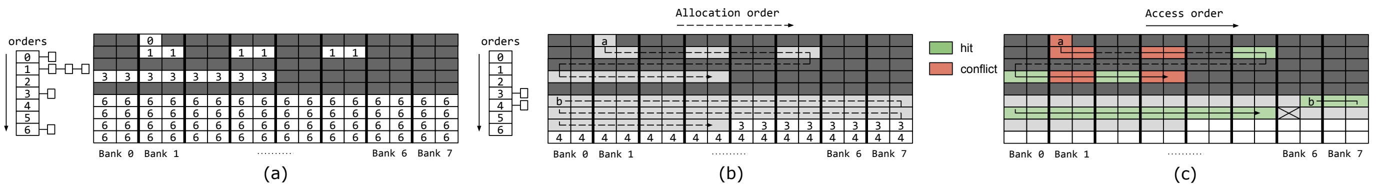


Fig. 5: The diagrams show how we can force the buddy allocator into providing us with contiguous physical memory and how we can detect this contiguous areas using our *hit*-pattern. (*a*) shows the buddy allocator keeping track of available memory in its `free_lists`. (*b*) shows the process of allocating 2 arrays namely a and b of respectively 15 and 40 pages, and the result of this process on the buddy's `free_lists`. (*c*) shows how our *hit*-pattern detects the contiguous memory backing b.

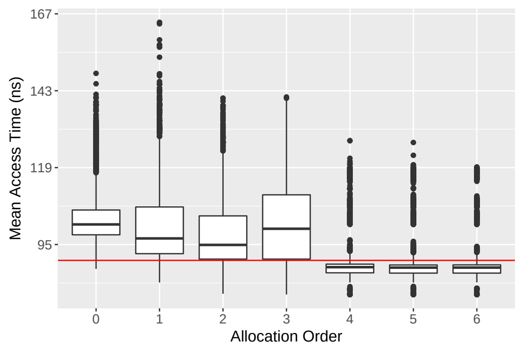


Fig. 6: Evaluation of contiguous memory timing side-channel. *Mean access time* is equal to $T_{total}/\#accesses$. Number of accesses is dependent on the resolution of the available timer.

E. Results

We evaluate our side channel to show how it can detect allocation trends of the buddy allocator. To obtain these measurements, we employ the `TIME_ELAPSED_EXT` asynchronous timer presented in Section V. We run the *hit*-pattern for $v[0]$ equal to every page within 512 KB areas. After collecting all these measurements, we use their median value to maximize the number of row conflicts for allocations of order < 4 while filtering out the noise from those of order ≥ 4. .

Figure 6 shows the *mean access time* over the allocation order. Allocations of order ≥ 4 have a lower median and are less spread compared to allocations of order < 4. Due to the deterministic replacement policy of the GPU caches, we can see how the measurements have very little noise for allocation of order ≥ 4. While the granularity of our side-channel is limited to order 4, this still provides us with valuable information regarding the allocation trend that allows us to infer the current status of the allocator. This makes it possible to heuristically identify the order of the underlying allocations.

VIII. Rowhammer Attacks from the GPU

We now have access to contiguous physical memory directly in JavaScript using our GPU-based side-channel attack discussed in Section VII. We demonstrate how we can remotely trigger Rowhammer bit flips on this contiguous memory by exploiting the texture fetching functionality from a WebGL shader running on the GPU. After a brief introduction of the Rowhammer bug, we discuss how we can trigger these bit flips from the GPU by efficiently evicting GPU caches. Finally, we evaluate the results of our implementation and discuss its implications.

A. The Rowhammer bug

In Section VII-A, we described the organization of a DRAM chip explaining the concept of rows. These rows are composed of *cells* where each cell store the value of a bit in a *capacitor*. The charge of a capacitor is transient, and therefore, DRAM needs to be recharged within a precise interval (usually 64 ms).

Rowhammer is a software-based fault injection attack that can be considered a fallout of this DRAM property. By frequently activating specific rows an attacker can influence the charge in the capacitors of adjacent rows, making it possible to induce bit flips in a victim row without having access to its data [27].

There are two main variants of Rowhammer: (*1*) single-sided Rowhammer and (*2*) double-sided Rowhammer. (*1*) Single-sided Rowhammer access a specific aggressor row n triggering bit flips on the two adjacent rows $n-1$ and $n+1$. (*2*) Double-sided Rowhammer, instead, amplifies the power of single-sided Rowhammer by reversing the roles of these rows. Therefore, the attacker quickly access rows $n-1$ and $n+1$ (i.e., aggressor rows) in order to impose higher pressure on row n (i.e., victim row) capacitors triggering more bit flips. Double-sided Rowhammer, however, requires some knowledge about physical memory in order to select aggressor rows. This information is not available in JavaScript and cannot be derived if the system does not support huge pages.

In many instances, double-sided Rowhammer is necessary for triggering Rowhammer bit flips. For example, in the Dedup Est Machina attack [7], the authors report that they could not trigger bit flips with the default refresh rate with single-sided Rowhammer given that Windows does not support huge pages. The situation is quite similar with ARM-based devices that often do not support huge pages. Fortunately, our novel GPU-based side-channel attack discussed in Section VII, provides us with information about contiguous physical memory regions in JavaScript, allowing us to perform double-sided Rowhammer on ARM devices in the browser.

B. Eviction-based Rowhammer on ARM

In order to trigger bit flips we need to be able to access the aggressor rows fast enough to influence the victim row. Therefore, we need to build an access pattern that allows

TABLE III: Ability to trigger bit flips natively (left) and remotely (right). * implements eviction-based Rowhammer.

	Drammer	Rowhammer.js*	GPU*
Nexus 5	✓/ -	- / -	✓/ ✓
HTC One M8	✓/ -	- / -	✓/ ✓
LG G2	✓/ -	- / -	✓/ ✓

us to optimize the access time to the aggressor rows. In Section VII, we demonstrated an efficient cache eviction strategy to implement our contiguous memory detection side channel. This efficient technique gains even more relevance when trying to trigger Rowhammer bit flips. The FIFO replacement policy requires us to perform DRAM accesses to evict a cacheline. This is much slower compared to architectures with the common LRU policy where the attacker can reuse cached addresses for the eviction set. Nonetheless, we can benefit again from the limited cache size and deterministic replacement policy in GPUs to build efficient access patterns.

Since DRAM rows cover 8 KB areas (split among the ranks) of the virtual address space and each UCHE set stores addresses at 4 KB of stride we can map at most two addresses from each row to a cache set. Having two aggressor rows when performing double-sided Rowhammer we can load 4 cachelines from these rows. With 8 ways per UCHE set we need to perform 5 more DRAM accesses in order to evict the first element from the cache set. We call these accesses *idle*-accesses, and we choose their addresses from other banks to keep the latency as low as possible. Our access pattern interleaves *hammering*-accesses with *idle*-accesses in order to obtain a pareto optimal situation between minimum and maximum idle time. Since we have no knowledge about the row alignment among the different allocations we need to indiscriminately hammer every 4 KB offset.

C. Evaluation

Drammer [46] studies the correlation between median access time per read and number of bit flips. The authors demonstrate that the threshold time needed to trigger bit flips on ARM platforms is ~$260\,ns$ for each memory access. We computed the mean execution time over the 9 memory accesses following our *hammer*-pattern. The distance between two *hammer*-access is on average ~$180\,ns$, which means that our GPU-based hammering is fast-enough for triggering bit flips. We tested our implementation on 3 vulnerable smartphones: Nexus 5, HTC One M8 and LG G2. All of them including the Snapdragon 800/801 chipsets. We managed to obtain bit flips on all three platforms.

We compare our implementation against a native eviction-based implementation running on the CPU adopting cache eviction strategies proposed by in Rowhammer.js [20]. Even on our most vulnerable platform (i.e., Nexus 5) and with perfect knowledge of physical addresses for building optimal eviction sets, we did not manage to trigger any bit flips. The reason for this turned out to be the slow eviction of CPU caches on ARM: Each memory access, including the eviction of CPU caches, takes $697\,ns$ which is too slow to trigger Rowhammer bit flips. Table III summarizes our findings. Our GPU-based Rowhammer attack is the only known technique that can produce bit flips remotely on mobile platforms.

Furthermore, we demonstrate the advantages of GPU-accelerated microarchitectural attacks by measuring the time to first bit flip and $\#flips/min$ on the Nexus 5. We excluded the other two platforms due to their limited number of vulnerable cells. This includes the time required to detect contiguous memory via our side-channel attack in Section VII).

D. Results

We run the experiment 15 times looking for 1-to-0 bit flips. After each experiment, we restart the browser. It took us between 13 to 40 seconds to find our first bit flip with an average of 26 seconds in the case of 1-to-0. This difference in the time that our attack takes to find its first bit flip is due to locating contiguous memory given that the browser physical memory layout is different on each execution. Finding bit flips usually takes few seconds once we detect an allocation of order ≥ 4. Moreover, after identifying the first bit flip, on average, we find $23.7\,flips/min$. We try the same experiment looking for 0-to-1 bit flips and obtained similar results. But after the first flip, on average, we find $5\,flips/min$, making them less frequent than 1-to-0 bit flips.

IX. Exploiting the GLitch

In this section, we describe GLitch, our remote end-to-end exploit that allows an attacker to escape the Firefox sandbox on Android platforms. We start with bit flips in textures from the previous section and show how they can be abused.

A. Reusing Vulnerable Textures

After templating the memory by using page sized textures, we need to release the textures containing the exploitable bit flips. To improve performance, WebGL uses a specific memory pool for storing textures. This pool contains 2048 pages. Hence, to avoid our target texture to remain in this pool, we first need to release 2048 previously-allocated textures before releasing the vulnerable texture(s). After releasing our target texture, we can start allocating `ArrayObjects` which will be containers for data that we will later corrupt. The probability that our target texture gets reused by one of our `ArrayObjects` depends on the memory status of the system. For example, our target texture gets reused by our `ArrayObjects` 65% of the time when allocating 50 MB of them and 100% of the times when allocating 300 MB `ArrayObjects`. These results are taken when the system is under normal memory conditions (i.e., Firefox together with other applications running in background).

B. Arbitrary Read/Write

We now discuss how we corrupt elements of an `ArrayObject` to escape the Firefox sandbox. Our attack consists of two steps: first, we use a 1-to-0 bit flip to leak a pointer (i.e., breaking ASLR), and then we forge a counterfeit object using the pointer which we reference later by exploiting a second 0-to-1 bit flip. This provides us with arbitrary read/write primitive. Both steps rely on *type flipping* [7].

Type flipping: Firefox employs a common technique known as *NaN-boxing* [1], [7] to differentiate between objects and IEEE-754 doubles. Every value stored in an `ArrayObject` is of 64 bits. The last 32 bits of this value are the so-called *tag* which identifies the type of the object. If the tag value is below `0xffffff80` (i.e., `JSVAL_TAG_CLEAR`) the whole 64-bit word is considered an IEEE-754 double, otherwise the first 32 bits are considered as a pointer to an object. This allows us to exploit every bit flip within the first 25 bits of the tag to turn any pointer into a double and vice versa. Such property provides us with two powerful primitives: 1) leaking any object pointer by triggering a 1-to-0 bit flip (breaking ASLR), and 2) the ability of forging a pointer to any memory location.

The exploit chain: We use type flipping to carry out both of the steps of the attack. The only difference consists in the direction of the bit flip (i.e., 0-to-1 or 1-to-0). As we mentioned earlier, we target an `ArrayObject` with our bit flips. `ArrayObject` allows us to store any type of object within its elements by using the aforementioned *NaN-boxing* technique. Once we know which locations are exploitable with a bit flip in the `ArrayObject` the attack unfold as follows. We first place the pointer of an inline `ArrayBuffer` in the 1-to-0 vulnerable location and we trigger the bit flip to leak the pointer of this object. We use inline `ArrayBuffers` since they guarantee complete control over the stored data, making it easier to craft a counterfeit object inside them. Furthermore, header and data are stored inline, allowing us to leak also the pointer of our counterfeit object since we will lay this object within the `ArrayBuffer` itself.

After leaking the pointer, we need to create a reference to our counterfeit object that will grant us arbitrary memory read/write. We use a `JSString` as a counterfeit object for two reasons: (i) the UTF-16 standard provides us with the arbitrary read/write primitive we are seeking [3], and (ii) the header of these objects does not contain any other pointers, making it easy to forge them. Our counterfeit `JSString` contains a `pointer` to the memory location we want to read from or write to. Finally, we only need to forge a double that encapsulate the pointer to the fake `JSString` in the vulnerable 0-to-1 location. Once triggered the bit flip we obtain a pointer to the fake string allowing us to access memory at the `pointer`.

C. Results

We run GLitch 17 times on the Nexus 5. Out of the 17 trials, GLitch successfully compromised the browser in 15 cases and in the remaining two cases, one of the bit flips did not trigger (i.e., no crash). The results along with a comparison of related attacks are summarized in Table IV. The end-to-end exploitation time is varied and dominated by finding exploitable bit flips. We achieved the fastest end-to-end compromise in only 47 seconds while the slowest compromise took 586 seconds. On average, GLitch can break ASLR in only 27 seconds and fully compromise the device remotely in 116 s, making it the fastest known remote Rowhammer attack.

TABLE IV: End-to-end attack time for breaking ASLR and fully compromising the system with GLitch and comparison with related attacks. We use '-' when the attack does not have that target or '*' when we did not find the exploitation time.

Attack	Full compromise	Breaking ASLR
GLitch	116 s	27 s
Dedup Est Machina [7]	823 s	743 s
Rowhammer.js [20]	*	-
AnC [18]	-	114 s

X. Mitigations

In this section we discuss possible mitigations against GPU-based attacks. We divide the discussion in two parts: 1) defending against side-channel attacks, and 2) possible solutions against browser-based Rowhammer attacks.

A. Timing side channels

To protect the system against side-channel attacks, currently the only practical solution in the browser is disabling all possible timing sources. As we discussed earlier, we do not believe that breaking timers alone represents a solid long-term solution to address side-channel attacks. However, we do believe that eliminating known timers makes it harder for attackers to leak information. Hence, we now discuss how to harden the browser against the timers we built in Section V.

First, we recommend disabling the explicit timers provided by `EXT_DISJOINT_TIMER_QUERY`. If disabling it is not a viable option, we at least suggest reducing its granularity similar to what browser vendors have done with `performance.now()` [10], [11], [39], [50]. Furthermore, we suggest impeding every type of explicit synchronization between JavaScript and the GPU context. Functions such as `clientWaitSync()` and `getSyncParameter()` allow an attacker to build implicit timers that can be used to leak sensitive data from both CPU and GPU. As a consequence, they need to be redesigned in order to disrupt these channels. A possible solution to fix the `clientWaitSync()` function could be to implement it through a callback that executes in the JavaScript event loop only when the GPU has concluded its operation. This callback will then be subject to recently proposed defenses in FuzzyFox [28] and DeterFox [9]. WebGL functions that directly expose sensitive information to the JavaScript context such as `getSyncParameter()` should be coarsened to report completion only after a certain amount of time has passed, similar to `performance.now()`.

Another mitigation possibility is introducing extra memory accesses as proposed by Schwarz et al. [43]. This, however, does not protect against the attack we described in Section VII since the attack runs from the GPU. The potential security benefits of implementing this solution on GPUs and its performance implications require further investigation.

B. GPU-accelerated Rowhammer

Ideally, Rowhammer should be addressed directly in hardware or vendors need to provide hardware facilities to address Rowhammer in software. For example, Intel processors provide advanced PMU functionalities that allows efficient detection of Rowhammer events as shown by previous work [4]. Unfortunately, such PMU functionalities are not available on ARM platforms and, as a result, detecting Rowhammer events will be very costly, if at all possible. But given the extent of the vulnerability and the fact that we could trigger bit flips in the browser on all three phones we tried, we urgently need software-based defenses against GPU-accelerated attacks.

As discussed in Section VIII, to exploit Rowhammer bit flips, an attacker needs to ensure that the victim rows are reused to store sensitive data (e.g., pointers). Hence, we can prevent an attacker from hammering valuable data by enforcing stricter policies for memory reuse. A solution may be enhancing the physical compartmentalization initiated by CATT [8] to userspace applications. For example, one can deploy a page tagging mechanism that does not allow the reuse of pages tagged by an active WebGL context. By isolating pages that are tagged by an active WebGL context using guard rows [8], one can protect the rest of the browser from potential bit flips that may be caused by these contexts.

There are trade-offs in terms of complexity, performance, and capacity with such a solution. Implementing a basic version of such an allocator with statically-sized partitions for WebGL contexts is straightforward, but not flexible as it wastes memory for contexts that do not use all the allocated pages. Dynamically allocating (isolated) pages increases the complexity and has performance implications. We intend to explore these trade-offs as part of our future work.

XI. Related Work

Olson et al. [36] provide a taxonomy of possible integrated accelerators threats classified based on the confidentiality, integrity and availability triad. They discuss that side-channel and fault attacks can potentially be used to thwart the confidentiality and integrity of the system. To the best of our knowledge, the attacks presented in this paper are the first realization of these attacks that make use of timing information and Rowhammer from integrated GPUs to compromise a mobile phone. While there has been follow up work that shields invalid memory accesses from accelerators [35], we believe further research is necessary to provide protection against microarchitectural attacks. We divide the analysis of these microarchitectural attacks in the rest of this section.

A. Side-channels

Side-channels have been widely studied when implemented natively from the CPU [6], [18], [29], [34], [38], [40], [49]. In recent years, however, researchers have relaxed the threat model by demonstrating remote attacks from a malicious JavaScript-enabled website [18], [37]. All these instances, however, are attacks carried out from the CPU.

There is some recent work on showing possibilities of executing microarchitectural attacks from the GPU, but they target niche settings with little impact in practice. Jiang et al. [24], [25] present two attack breaking AES on GPGPUs, assuming that the attacker and the victim are both executing on a shared GPU. Naghibijouybari et al. [33] demonstrate the possibility of building covert-channels between two cooperating processes running on the GPU. These attacks focus on general-purpose discrete GPUs which are usually adopted on cloud systems, whereas we target integrated GPUs on commodity hardware.

B. Rowhammer

Since Kim et al. [27] initially studied Rowhammer, researchers proposed different implementations and exploitation techniques. Seaborn and Dullien [44] first exploited this hardware vulnerability to gain kernel privileges by triggering bit flips on page table entries. Drammer uses a similar exploitation technique to root ARM Android devices [46]. These implementations, however, relied on the ability of accessing memory by bypassing the caches, either using the `CLFLUSH` instruction on x86_64 or by exploiting DMA memory [46]. Our technique does not require any of these expedient.

Dedup Est Machina [7] and Rowhammer.js [20] show how Rowhammer can be exploited to escape the JavaScript sandbox. These attacks rely on evicting the CPU caches in order to reach DRAM. On the ARM architecture, eviction-based Rowhammer is too slow to trigger Rowhammer bit flips even natively due to large general-purpose CPU caches. We showed for the first time how GPU acceleration allows us to trigger bit flips evicting the GPU caches. This allowed us to trigger bit flips from JavaScript on mobile devices.

XII. Conclusions

We showed that it is possible to perform advanced microarchitectural attacks directly from integrated GPUs found in almost all mobile devices. These attacks are quite powerful, allowing circumvention of state-of-the-art defenses and advancing existing CPU-based attacks. More alarming, these attacks can be launched from the browser. For example, we showed for the first time that with microarchitectural attacks from the GPU, an attacker can fully compromise a browser running on a mobile phone in less than 2 minutes. While we have plans for mitigations against these attack, we hope our efforts make processor vendors more careful when embedding the next specialized unit into our commodity processors.

Disclosure

We are coordinating with the Dutch Cyber Security Centrum (NCSC) for addressing some of the issues raised in this paper.

Acknowledgements

We would like to thank our shepherd Simha Sethumadhavan and anonymous reviewers for their valuable feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 SHARCS under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI Dowsing.

References

[1] "Value.h," https://dxr.mozilla.org/mozilla-central/source/js/public/Value.h, Accessed on 30.12.2017.

[2] "WebGL current support," http://caniuse.com/#feat=webgl, Accessed on 30.12.2017.

[3] argp, "OR'LYEH? The Shadow over Firefox," in *Phrack 0x45*.

[4] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS'16*.

[5] M. Balci, A. Seetharamaiah, C. Frascati, K. Nagendra, C. Sharp, and G. Garcia, "Flex rendering based on a render target in graphics processing," wO2015164397.

[6] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[7] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P'16*.

[8] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAnt Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory," in *SEC'17*.

[9] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic Browser," in *CCS'17*.

[10] A. Christensen, "Reduce resolution of performance.now," https://bugs.webkit.org/show_bug.cgi?id=146531, Accessed on 30.12.2017.

[11] Chromium, "window.performance.now does not support sub-millisecond precision on windows," https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110, Accessed on 30.12.2017.

[12] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture Support for Accelerator-rich CMPs," in *DAC'12*.

[13] N. Corporation, "Nvidia perfkit," https://developer.nvidia.com/nvidia-perfkit, Accessed on 30.12.2017.

[14] Esmaeilzadeh, Hadi and Blem, Emily and St. Amant, Renee and Sankaralingam, Karthikeyan and Burger, Doug, "Dark Silicon and the End of Multicore Scaling," in *ISCA'11*.

[15] I. Ewell, "Disable timestamps in WebGL." https://codereview.chromium.org/1800383002, Accessed on 30.12.2017.

[16] F. Giesen, "Texture tiling and swizzling," https://fgiesen.wordpress.com/2011/01/17/texture-tiling-and-swizzling/, Accessed on 30.12.2017.

[17] M. Gorman, "Chapter 6: Physical Page Allocation," https://www.kernel.org/doc/gorman/html/understand/understand009.html, Accessed on 30.12.2017.

[18] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS'17*.

[19] K. Group, "OpenGL ES Shading Language version 1.00," https://www.khronos.org/files/opengles_shading_language.pdf, Accessed on 30.12.2017.

[20] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *DIMVA'16*.

[21] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *S&P'11*.

[22] M. Hassan, A. M. Kaushik, and H. Patel, "Reverse-engineering embedded memory controllers through latency-based analysis," in *RTAS'15*.

[23] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *S&P'13*.

[24] Z. H. Jiang, Y. Fei, and D. Kaeli, "A Novel Side-Channel Timing Attack on GPUs," in *GLSVLSI 2017*.

[25] ——, "A complete key recovery timing attack on a GPU," in *HPCA'16*.

[26] G. Key, "ATX Part 2: Intel G33 Performance Review," https://www.anandtech.com/show/2339/23, Accessed on 30.12.2017.

[27] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *SIGARCH 2014*.

[28] D. Kohlbrenner and H. Shacham, "Trusted Browsers for Uncertain Times." in *SEC'16*.

[29] N. Lawson, "Side-channel attacks on cryptographic software," in *S&P'09*.

[30] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *SEC'16*.

[31] S. M., "How physical addresses map to rows and banks in DRAM," http://lackingrhoticity.blogspot.nl/2015/05/how-physical-addresses-map-to-rows-and-banks.html, Accessed on 30.12.2017.

[32] I. Malchev, "KGSL page allocation," https://android.googlesource.com/kernel/msm.git/+/android-msm-hammerhead-3.4-marshmallow-mr3/drivers/gpu/msm/kgsl_sharedmem.c#621, Accessed on 30.12.2017.

[33] H. Naghibijouybari, K. Khasawneh, and N. Abu-Ghazaleh, "Constructing and Characterizing Covert Channels on GPGPUs," in *MICRO-50*.

[34] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion," in *SOSP'17*.

[35] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *MICRO-48*.

[36] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," in *IEEE Computer Architecture Letters 2016*.

[37] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their implications," in *CCS'15*.

[38] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *RSA'06*.

[39] M. Perry, "Bug: Reduce precision of time for JavaScript," https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517, Accessed on 30.12.2017.

[40] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." in *SEC'16*.

[41] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *SEC'16*.

[42] K. Sato, C. Young, and D. Patterson, "Google Tensor Processing Unit (TPU)," https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, Accessed on 30.12.2017.

[43] M. Schwarz, M. Lipp, and D. Gruss, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *NDSS'18*.

[44] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," in *Black Hat 2015*.

[45] V. Shimanskiy, "EXT_disjoint_timer_query," https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_disjoint_timer_query.txt, Accessed on 30.12.2017.

[46] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS'16*.

[47] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *ASPLOS'10*.

[48] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation." in *SEC'16*.

[49] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *SEC'14*.

[50] B. Zbarsky, "Clamp the resolution of performance.now() calls to 5us," https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab, Accessed on 30.12.2017.

Appendix A
Snapdragon 800/801 DRAM mapping

In Section VII-C, we explained that *contiguity* differs from *adjacency*. However, we also stated that, we could assume the congruency between these two attributes for the Snapdragon 800/801 SoCs. Here we show how we can relax that assumption.

As explained in Section VII-A, DRAM is organized in channels, DIMMs, ranks, banks, rows and columns. The CPU/GPU, however, only access DRAM using virtual addresses. After a translating a virtual address to its physical addresses, the *memory controller* converts the physical address to a DRAM address consisting of the elements we mentioned above. This mapping of physical addresses to DRAM addresses is undocumented, but it has been reverse engineered for many architectures [22], [31], [48] including Snapdragon 800/801 [40], shown in Table V.

	Channel	DIMMs	Ranks	Banks
Bits	-	-	10	[13,14,15]

TABLE V: DRAM mapping

Snapdragon 800/801 does not employ multiple channels or DIMMs and as a result no bits in the physical addresses are assigned to their selection. Within the DIMM, we find two ranks and eight banks within the ranks. Bit ten of a physical address is responsible for choosing these ranks, while bits [13-15] are responsible for choosing the banks. As we show in Figure 7, this configuration translates to 1 KB aligned areas of the physical address space shuffled over the two different ranks ($2^{10} = 1$ KB) and a change of bank every 8 KB ($2^{13} =$ 8 KB). Since the division among the ranks happens at a smaller granularity than a page, it means that every row (within a bank) is 4 KB large and stores 2 half-pages as shown in Figure 7.

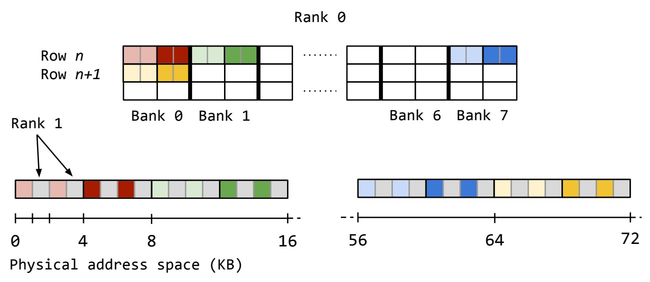


Fig. 7: Snapdragon 800/8011 DRAM mapping

Remembering the assumption we made in Section VII-C, now it should be clear why we are allowed to simplify our model considering two pages per row. Since we are only interested in touching memory at page level, due to the stride imposed by the UCHE cache (i.e., 4 KB), we can build our model completely oblivious of the ranks. Thus, we can consider rows of 8 KB.