Practical Microarchitectural Attacks from Integrated GPUs

Master's Thesis

Pietro Frigo

THESIS COMMITTEE:

Prof.dr.ir. J.C.A van der Lubbe
Prof.dr.ir. J.S.S.M. Wong
Assist.Prof.dr C. Doerr

SUPERVISORS:

Dr. K. Razavi
Assist.Prof.dr.ir C. Giuffrida
Prof.dr. H.J. Bos
Assist.Prof.dr C. Doerr

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 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.

Contents

List of Figures

Chapter 1

Introduction

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 [1, 2, 3]. Examples include integrated cryptographic accelerators, audio processors, radio processors, network interfaces, FPGAs, and even tailored processing units for artificial intelligence [4]. 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 [5, 6], but entirely ignores the threat of more advanced microarchitectural attacks. Among this plethora of specialized units that are being included on the System-on-Chip, currently the most widely spread integrated processors are the *Graphics Processing Units* (GPUs). Most laptops and almost all mobile phones [7] are sold today with a *System-on-Chip* (SoC) comprising this integrated unit. This is due to the always higher demand for better graphics even on lower-end commodity platforms. These systems are known as *heterogeneous system architectures* (HSAs) and are able to balance their workload in a more efficient way among the two co-processors.

Even though modern GPUs are evolving always more towards the goal of *general purpose* computing, these systems do not necessarily provide such advanced parallel programming features, and on some low-end hardware they are still restricted to very limited graphics acceleration. These more limited capabilities of such systems drew researchers to the misguided conclusion of an even more limited attack surface. Throughout this research we demonstrate how such assumption is fundamentally flawed by showing how the (*supposedly*) limited features of such system do not inhibit an attacker from carrying out advanced microarchitectural attacks on commodity systems. On the contrary, we demonstrate that GPUs, already widely employed in almost all mobile platforms 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 APIs, allowing side-channel and Rowhammer attacks from within the JavaScript sandbox on mobile ARM platforms.

1.1 Microarchitectural attacks

Microarchitectural attacks are a family of attacks that arose as a way to circumvent defenses protecting cryptographic algorithms [8, 9, 10]. These attacks base their primitives on intrinsic hardware properties such as execution time [8], power consumption [11] and even DRAM charges [12]. Since they base their effectiveness on physical properties of the system, researchers initially assumed a threat model that limited such attacks to attackers with physical access to the devices. However, this assumption was proven erroneous in recent years when different studies demonstrated the possibility of carrying out microarchitectural attacks not only from native software[9, 10, 12, 13, 14, 15, 16, 17, 18] but also remotely from malicious JavaScript websites [19, 20, 21, 22, 23, 24].

These attacks are implemented with two different purposes: either *steal* data, through a variety of side-channels, or *corrupt* data, using fault injection attacks.

Side-channels: Side-channels are a category of microarchitectural attacks that have the goal of stealing secrete data — or transmit in the case of covert-channels. This type of attacks was first described in 1996 by Kocher [8] who showed how it was possible to leak information by measuring timing differences when executing different operations. After this first study on timing side-channels many other researchers followed this path [10, 13, 14, 15, 16, 21, 25, 26, 27, 28] showing the strength of timing attacks. These attacks base their effectiveness on the possibility of measuring time differences while executing different operations. This timing difference is usually consequence of (*undocumented*) hardware optimization such as usage of caches to store recently used data or branch predictors [26]. The most common targeted component when implementing timing side-channels are CPU caches. However, any level of the memory hierarchy can be the target of these attacks.

Fault injection: Fault attacks are a second category of microarchitectural attacks that exploit hardware glitches to trigger data corruptions. These attacks usually target undocumented states such as out-of-range voltage [29, 30] to inject glitches and modify data that should not be accessible by the attacker, hence bypassing applications' trust boundaries. The most dramatic and wide spread example of software-based fault injection attack is Rowhammer. Rowhammer exploits the transient nature of DRAM capacitors to trigger bit flips on specific memory rows by performing particular access patterns that influence the capacitor charges on adjacent rows. Previous work shows that it is possible to compromise cloud VMs [18, 31] and browsers [23, 24] and even to gain kernel privileges [17, 32].

While their native implementation already represents a significant threat, the relevance of microarchitectural attacks escalated after Oren et al. [19] unveiled the

possibility of implementing remote attacks. To counter this threat, a number of recently proposed defenses aim to protect against these attacks. Defenses against timing side-channels eradicate the problem from its roots proposing to reduce the resolution [33, 34, 35, 36], fuzz [37] or make deterministic [38] the value returned by explicit and implicit timers. Defenses targeting Rowhammer, on the other hand, usually propose to either target memory allocations isolating different privileged contexts [39] or to monitor memory accesses to enforce preventive measures [40].

All these defenses, however, share the common implicit assumption of a restricted threat model that limits the attacker's capabilities to what is provided by the CPU cores.

1.2 Research Goal

As of now microarchitectural attacks are considered to be implementable just from CPU cores. The goal of our research is to revisit this current assumption to demonstrate that it is insufficient to protect only against attacks that originate from the CPU. Therefore, we need to show the possibility of building these attacks with another exploitation vector: the GPU. To do so, we need to identify the common primitives an attacker needs when performing microarchitectural attacks and determine their possible equivalent when adopting integrated GPUs as exploitation vector. We then need to define a new threat model that takes these integrated processors into account and we need to investigate the efficacy of current state-of-the-art defenses protecting from microarchitectural attacks against this new threat model. This last paragraph can be summarized in the following research question:

What means does the GPU provide to an attacker for building microarchitectural attacks? And how do currently proposed defenses against traditional implementations cope with this new exploitation vector?

The investigation of such questions can unfold by answering the following subquestions:

- What are the primitives that an attacker traditionally needs to carry out microarchitectural attacks?
- What means does an integrated GPU provide to an attacker trying to build such primitives?
- What is the new threat model defined by the introduction of this new exploitation vector?
- How do currently proposed defenses respond to this new attack paradigm?
- How can we cope with this new threat model?

To the best of our knowledge, currently no research has been carried out on the field of microarchitectural attacks from integrated GPUs. Since these systems

are embedded on almost every commodity platform such as smartphones and laptops [7] we chose to address such knowledge gap in order to provide new and valuable information to the body of science.

1.3 Contributions

In this thesis, we show that we can build all necessary primitives for performing powerful microarchitectural attacks directly from the GPU. Dramatic is the fact that we can obtain these primitives, hence 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 [37, 38]. 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 [23]. 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-based Rowhammer strategies were found too inefficient to trigger bit flips [32]. 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 proof of concept end-to-end attack, uses all these GPU primitives in orchestration to reliably compromise the browser on a mobile device using only microarchitectural attacks in a matter of few minutes. In comparison, even on PCs, all previous Rowhammer attacks from JavaScript require non default configurations (such as reduced DRAM refresh rates [24] or huge pages [23]) and often take such a long time that some researchers have questioned their practicality [39].

Our 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 [39]. 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.

We can summarize our contributions as follow:

- The first study of the architecture of integrated GPUs, the functionalities that they provide, and how they are exposed from JavaScript using the standardized WebGL library.
- A series of novel attacks that are executed directly on the GPU, compromising existing defenses and showing new possibilities for microarchitectural attacks.
- The first end-to-end remote Rowhammer exploit on mobile phones using the GPU-based primitives in orchestration, compromising mobile browsers on commodity smartphones.
- Directions for containing GPU-based attacks on integrated systems.

1.4 Thesis Outline

Due to the broad nature of this project it is hard to follow the standard structure of a thesis. As a consequence, we laid out our manuscript following the conventional approach used when building our attack.

We start in Chapter 2 providing the fundamental knowledge to understand future architectural analyses and microarchitectural attacks. This consists in a thorough description of the first two layers of the memory hierarchy: caches and DRAM. We then proceed with recovering the architecture of our target system in Chapter 3. We initially give a description of a general GPU architecture, describing how an attacker can gain control over it, and we conclude with a more detailed analysis of the GPU embedded in our test system (i.e., the Adreno 330) showing how this processor has access to resources shared with the rest of the system. After we have all the required knowledge about our system we then need to identify the necessary primitives required to build microarchitectural attacks. We start by describing these primitives making a distinction between side-channels and Rowhammer attacks and we then proceed in recovering each of these primitives from an integrated GPU. After recovering all the necessary primitives to build complete microarchitectural attacks we make such attacks concrete in Chapter 5 showing how we can combine all the primitives defined in the previous chapter in order to create an end-to-end exploit that compromises a mobile browser without relying on any software bug. We complete this chapter discussing possible mitigations against such attacks. We draw our final conclusions in Chapter 6 discussing directions for future works.

Chapter 2

The Memory Hierarchy

As mentioned in Chapter 1 microarchitectural attacks aim at (i) stealing or (ii) corrupting *data*. This data needs to be stored on one of the many different levels of the memory hierarchy. Comprehensive knowledge about this hierarchy is required in order to understand the GPU architecture we present in Chapter 3 and the primitives we introduce in Chapter 4 We start by giving a broad overview of the memory ladder on modern systems. We then focus on the two rungs usually targeted by microarchitectural attacks, namely caches and DRAM.

2.1 The memory pyramid

Modern systems are equipped with different layers of storages. The distinction between these layers is usually made based on *access time* to the requested resources. There is a direct correlation between performances, size and cost. Usually high performances storages are very limited in size due to their high costs, while low performances storages have usually sizes millions of times larger compared to the former due to their affordable price.

Figure 2.1 depicts a simplified pyramid of the memory hierarchy on modern architectures. Climbing up this pyramid we decrease the size while increasing speed and cost. On top of the pyramid there are *caches*. These are the fastest storages available to a processor — after registers. However, due to their price are usually very limited in size. Below the caches, resides *DRAM*. DRAM chips are relatively fast and provide a good compromise between price and performances. While caches are usually in the order of KBs or MBs, modern systems, even lower end smartphones, are usually equipped with GBs of storage space in DRAM chips providing the system with enough memory to run complex applications and storing recently used items for short- and mid- term operations. At the bottom of this hierarchy there are long-term storages such as flash drives and hard drives. Among the two, flash drives are significantly faster and more expensive. Nonetheless, they usually serve the same purpose of long-term storage.

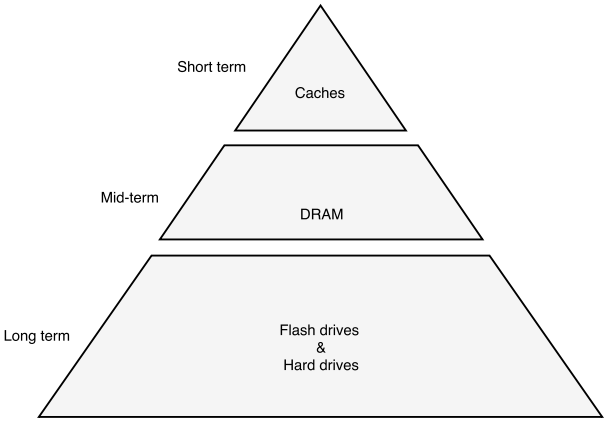


Figure 2.1: Memory Hierarchy

2.2 Caches

Processors' caches have been introduced to overcome the limitation imposed by the slow memory bus accessing DRAM. To bypass this bandwidth bottleneck and speed up processors' computations a smaller and faster storage solution was introduced closer to this unit. These fast storage units, if present, intercede almost every memory access requested by the processors. If the content is found in the caches (*cache hit*) then the data gets fetched faster. If the data is not stored on the caches (*cache miss*), then the processor has to request such data to the memory controller who fetches it from (slower) DRAM.

When designing a cache there are 2 main principles involved: *spatial* and *temporal* locality. Spatial locality means that data — or instructions — stored close to recently fetched data is likely to be fetched too. Temporal locality, instead, means that data that has been recently used is likely to be re used in a short timespan. The goal of the caches is this of optimizing these two principles. Depending on the purpose of the cache, they might be engineered to maximize one or the other. We can ascertain these choices by analyzing their architectural designs and sizes.

Cache organization

Depending on the system requirements these caches can follow different designs. The usual architecture follows one of these 3 organization:

1. Direct mapping
2. Fully associative
3. Set associative

We now describe how each of these organizations uses the memory address to identify if the requested content is currently stored in the cache.

Direct Mapping: Direct-mapped caches use a one-to-one mapping to store data into the cache. This means that a memory address can be stored into one, and only one, *cacheline* (i.e., the minimum storage unit in the cache). The mapping is computed directly from the memory address. This can be either virtual or physical depending on the system design. We describe memory addressing in Section 2.3. Of this memory address the lowest k bits are the so-called *offset*. This allows the processor to query the data stored in the cache at byte granularity. Hence, 2^k *bytes* of data are stored into a cache line. The following n bits are used as *index*. This identifies the index of the cacheline. The remaining bits represent the *tag*. The tag is used to identify if the cacheline stored at the specific index is the one belonging to that address. Addresses mapping to the same cacheline are known as *congruent*. Consecutive requests to congruent addresses generate cache misses for every access.

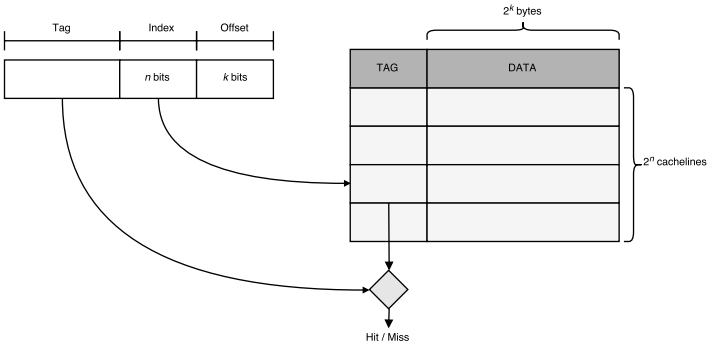


Figure 2.2: Direct-mapped cache

Fully associative: Fully associative caches do exactly the opposite of what direct mapping does. While in direct-mapped caches each memory address can occupy just a single index in fully associative caches there is no index at all and every memory address can reside in any available slot (i.e., *way*). This type of architecture solves the problem of congruent addresses present with direct mapping. However, memory lookups get more expensive with bigger cache sizes due to the necessity of checking multiple ways in the cache to retrieve the requested data — all the ways in the case of a cache miss. This mapping is usually adopted in very small and fast caches such as *Translation Lookaside Buffers* (TLBs).

Set associative: This architecture is a good compromise between direct-mapped and fully associative caches. In this case the cache gets split into 2^n sets and each set contains a variable number m of ways. The n bits used as index in this architecture are used to identify the cache set, contrarily to direct-mapped caches. Once identified the cache set the cache behaves as a fully associative cache within the cache set. Therefore, it compares the Tag extracted from the memory address against all the ways within the identified cache set.

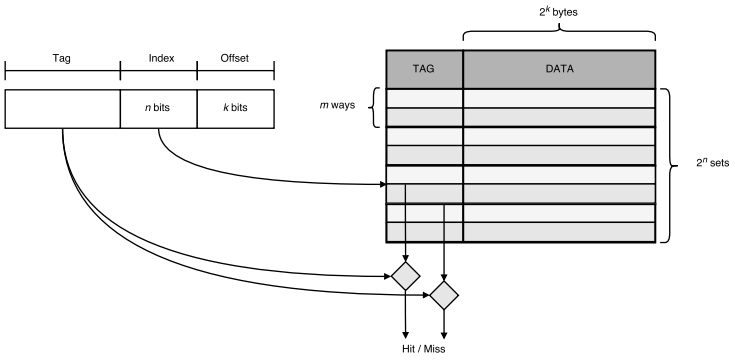


Figure 2.3: Set associative cache

Cache replacement policy

Modern caches usually employ a set-associative organization. Cachelines within a specific cache set then need to be handled in an efficient way in order to optimize spatial and temporal locality. Depending on their usage, caches can enforce replacement policies of multiple natures. On modern CPU architectures the most widespread replacement policy is *least recently used* (LRU[1]). This replacement policy keeps track of the last usage of a cacheline within a cache set evicting every time the least recently used. This scheme highly optimizes temporal locality, keeping in the cache cachelines that have recently being used.

While this replacement policy is enforced in most of x86 systems, On ARM architectures, instead, a pseudo random replacement policy is usually adopted. This is easy to implement in hardware [41] and is highly energy efficient compared to other policies [42] making it perfectly suitable for mobile ARM platforms.

Multi level caches

Modern systems deploy multi level caches to further optimize their performances. These systems follow the overall trend of "size vs performances" applied to the whole memory hierarchy. They usually embed very little and faster L1 caches and relatively bigger and slower L2 caches. The number of levels and their coherency strategies is a design choice. Most of Intel CPUs are provided with 3 level caches with a private per-core L1 and L2 and a larger L3 shared among the different cores. Whereas, ARM architectures usually have just two level caches where L2 is shared among the different cores due to space limitations on the SoC. The coherency policy between the different caches can then be inclusive, exclusive or non-inclusive. *Inclusive* caches require data to be stored on a lower level if present in a higher level. *Exclusive* caches do the opposite keeping objects just in one of the two levels. *Non-inclusive*, on the other hand, do not take any of this into consideration. Hence, a cacheline can be stored in one of the two levels, both of them or none.

[1] when referring to LRU we means both LRU and pseudo LRU

2.3 DRAM

DRAM is placed just below the caches and represents the next rung in the memory ladder. DRAM is a mid-term solution that represents a great compromise between (expensive) fast caches and slow flash and hard drives.

DRAM organization

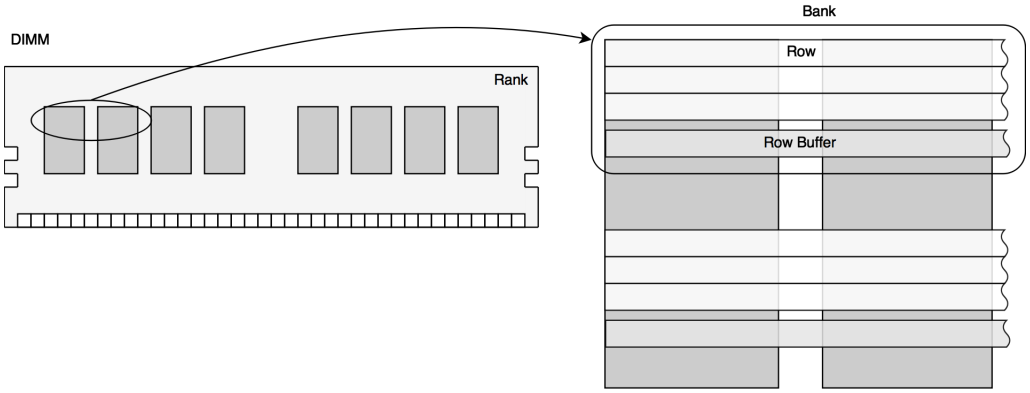


Figure 2.4: DRAM architecture

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). DIMMs are the typical module available in the stores (Figure 2.4). These modules are commonly partitioned in either one or two *ranks* which they usually correspond to the physical front and back of the DIMM. Each rank is then divided into separate *banks*. It is frequent to find 8 banks on DDR3 chips and 16 on DDR4. Finally every bank contains the memory array that is arranged in *rows* and *columns*. The memory array contains the building blocks of a DRAM chip: *cells*. A cell is made of a *transistor* and a *capacitor*. Each cell stores the information for a single bit. Based on its capacitor's charge and the specific encoding the value then gets translated to 0 or 1. Capacitors' charge is transient. Therefore, the memory controller is responsible of recharging them in order to preserve the stored values. This recharging operation is known as *refreshing* and it usually happens at 64 *ms* intervals.

DRAM internally performs reads at row granularity. When the system wants to access a specific byte, the DRAM chip *activates* a row and stores it in the *row buffer*. The row buffer resembles a direct mapped cache within the DRAM chip. This buffer loads a full row when a word is requested from it optimizing temporal and spatial locality also at DRAM level. If the following request falls within the boundaries of the currently active row this gets fetched directly from the row buffer

(*row hit*). Otherwise, the active row needs to be restored before activating a new row and loading it into the row buffer (*row conflict*).

Memory addressing

When the system is running applications are oblivious of all these details and work with memory addresses that later get mapped to specific rows and columns. As mentioned above in modern systems these memory addresses can be either *virtual* or *physical*. The virtual space represents a wrapper of the physical address space and it was introduced to solve the security issue of process isolation. By implementing this system each process is provided with a specific address space independently from the other processes running on the system. Modern systems employ *paging* to simplify the translation between the two address spaces. This means that the minimum unit they are able to physically allocate is a so-called *page* usually of the size of 4 KB. And the translation between virtual to physical addresses is then made on a page basis. Virtual addresses are translated to physical addresses from the *Memory Management Unit* (MMU). This conversion can take place before querying the caches, as for most of the x86 systems, or even after, as we will see later in Chapter 3. Regardless of when this conversion happens, DRAM is always queried using the physical addresses. Every time the system wants to read a byte from system memory the physical address eventually needs to be translated into a <channel, DIMM, rank, bank, row, column> hextuple. This mapping is usually undocumented and has been subject of different studies [16, 43].

Chapter 3

A Primer on integrated GPUs

While CPU architectures have been subject of multiple studies[44, 45, 46], inner details of integrated GPUs on mobile (ARM) platforms are currently undocumented. Since our final goal is this of performing microarchitectural attacks from these processors we need to perform a detailed analysis of their architecture understanding how they interact with the rest of the system.

Even though our analysis of the required primitives for microarchitectural attacks is subject of Chapter 4, in this chapter, while providing a thorough analysis of the integrated GPU architecture, we also investigate our fundamental prerequisite for any microarchitectural attack: *access to shared resources*. De facto, every microarchitectural attack needs access to resources shared with other victim processes. For instance, on a cache attack the attacker needs to share a cache set with the victim and in a Rowhammer attack the attacker and the victim need to share neighboring rows within a DRAM bank. As a consequence, the ability of sharing resources with other (distrusting) processes is imperative for any type of microarchitectural attack. We define this as our primitive *P0* and throughout this chapter we proceed to examine how an integrated GPU shares resources, not only with other victim processes, but also with other victim processors.

We start our analysis by looking at the general architecture of integrated GPUs understanding how these systems accelerate the rendering pipeline. Afterwards, we describe how developers, and therefore attackers, have native access to these processors through the OpenGL API and remote access through its JavaScript wrapper WebGL. To make our architectural analysis more concrete we then study a specific GPU implementation of an ARM SoC: Adreno 330. We present a novel reverse engineering technique that makes usage of OpenGL shaders to reconstruct the architecture of the GPU caches and we show how these caches differ from the typical model described in Chapter 2. Eventually, we conclude by discussing the advantages of our approach.

3.1 The GPU architecture

A *Graphics Processing Unit* is a specialized circuit conceived with the purpose of accelerating image rendering. We now discuss how the GPU architecture implements the rendering pipeline.

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. The pipeline starts from *vertex* shaders that perform 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 typically using external data obtained in the form of *textures*. This output of the pipeline is what is then displayed to the user.

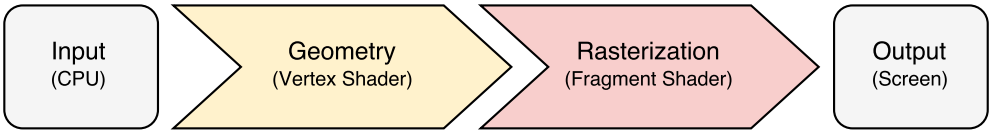


Figure 3.1: Stages of the Rendering Pipeline

Processing units: Figure 3.2 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, modern GPUs are designed following the SIMD paradigm (Same Instruction/Multiple Data), which consists in running the same shader with multiple inputs (i.e., different veritces or fragments). For this reason, GPUs include multiple SPs, each incorporating multiple Arithmetic Logical Units (ALUs) to further parallelize the computations. Shaders running on the SPs can then query the *texture processors* (TPs) to fetch additional input data used during their execution. This data, as mentioned above, is typically in the form of textures to which TPs apply filters of different natures (e.g., anti-aliasing).

Caches: GPUs, to speed up their computations, exploit different type of caches to optimize data transfers — both in input and output. Shaders, during their run, make use of external data (e.g., vertices and textures). All this data is stored on DRAM due to its large size. Since fetching data from DRAM is slow and can cause pipeline stalls, the GPU includes a multi level cache to speed up accesses to vertices and textures. These levels are usually two (i.e., L1 and L2). While the larger L2 is used by multiple units of the system (e.g., SPs, to store vertices, and TPs, to store textures), TPs make also use of a faster (but smaller) Texture L1 cache to further

speed the inner execution of the shader. We later discuss the architecture of these caches in the Adreno 330 GPU (Section 3.3).

Finally, in order to increase also the output performances when writing to *framebuffers*, integrated GPUs are usually equipped with smaller chunks of faster on-chip memory (*OCMEM*) that let them store portions of the render target and asynchronously transfer them back to DRAM, as shown in Figure 3.2.

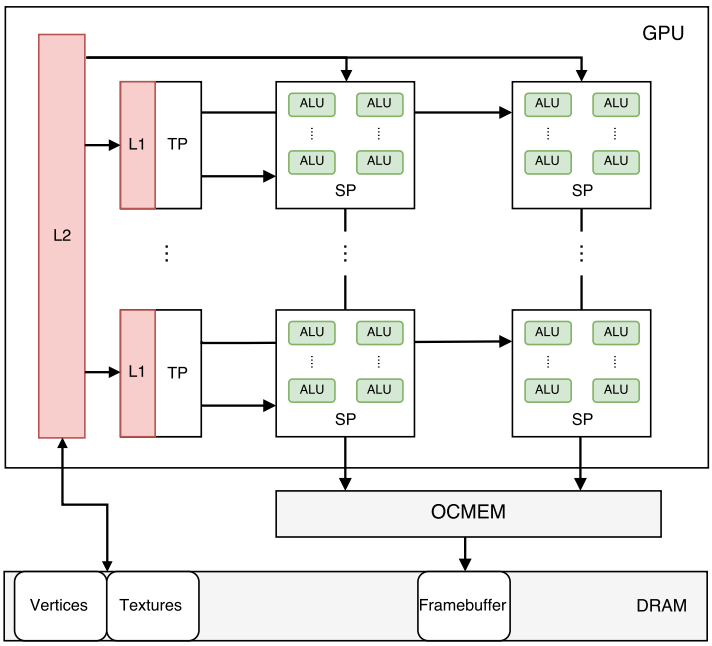


Figure 3.2: Building blocks of an integrated GPU

3.2 Programming the pipeline

Now that we know how how GPUs are able to accelerate the rendering pipeline, we still need a bridge interface to get control over the latter. This interface is usually provided by Graphics Libraries such as Direct3D and OpenGL that allow developers to communicate with these processors. While all these provide similar functionalities, OpenGL is the most widespread due to its cross-platform support.

OpenGL: OpenGL is a graphics API that exposes GPU hardware acceleration to developers that seek higher performances for graphics rendering. This API is supported by every major operating system and architecture. For this reason, image editing applications, video games and graphically intensive applications such as CAD have been adopting it for decades in order to improve their performances. Developers employing OpenGL write their shaders using the OpenGL Shading Language (GLSL). This is a C-like programming language part of the specification. Depending

on the version of the API supported from the underlying hardware different functionalities are made available to the developer. Most of modern computers support the latest major revision of the API (i.e., OpenGL ≥ 4). On mobile platforms, however, the standard API is usually not directly accessible, even if supported. Instead, the (stripped) OpenGL ES is exposed to the developer. OpenGL ES 2 is the first version of the API targeting programmable GPUs, hence making use of shaders. This version, is support by every modern platform. Therefore, it is a fundamental toolset throughout all our research.

WebGL: In recent years, with the raise of HTML5, users became more demanding and expressed the wish of using high performance games and applications, not only locally, but also on-line. Browsers, on the other hand, were not capable of directly handling these applications completely in software. As a consequence, the Web development paradigms underwent a complete revision which headed towards the design of more powerful interfaces that balance applications' workload among different components of the system. WebGL is an example of this transformation.

The WebGL API exposes the GPU-accelerated rendering pipeline to the Web to bolster the development of these Web Apps. Currently supported by every major browser[47] it provides most of the functionalities accessible from its backbone, namely OpenGL ES 2.0. Since it was conceived with the purpose of porting native graphics applications to the Web, the anatomy of these two APIs is almost equivalent. Shaders complying to the GLSL 100 standard (i.e., Shading Language available in OpenGL ES 2.0) can be seamlessly compiled and run from both the environments. Furthermore, the function calls required to deploy these shaders to the GPU are almost identical in native and JavaScript applications. In Listing 3.2.1 we show the code necessary to run a basic shader such as the one we present in Listing 3.3.1 for both the environments. We can see how the two snippets are identical with very minor syntax changes dictated by the different natures of the programming languages (e.g. statically typed vs. dynamically typed).

This flawless commutativity was a design choice to help current OpenGL developers porting their applications to the Web bringing 3D accelerated content online. In Chapter 4 we demonstrate how this design allows us to build equally powerful

```
1
2 var texLocation =
3  gl.getUniformLocation(prog, "tex");
4
5 gl.uniform1i(texLocation, 0);
6 gl.activeTexture(gl.TEXTURE0 + 0);
7 gl.bindTexture(gl.TEXTURE_2D, tex);
8 gl.drawArrays( gl.POINTS, 0, 1);
```

a: JavaScript code.

```
1
2 GLuint texLocation =
3  glGetUniformLocation(prog, "tex");
4
5 glUniform1i(texLocation, 0);
6 glActiveTexture(GL_TEXTURE0 + 0);
7 glBindTexture(GL_TEXTURE_2D, tex);
8 glDrawArrays( GL_POINTS, 0, 1);
```

b: C code.

Listing 3.2.1: Comparison between native and JavaScript APIs.

microarchitectural attacks from both JavaScript and native applications.

3.3 The Adreno 330: A case study

To better understand the GPU architecture, we analyze a concrete example of a modern system: the Adreno 330. The Adreno 330 is the integrated 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 UCHE cache (i.e., L2 cache using the proprietary terminology). This essentially means that the GPU operates on virtual memory rather than physical memory (Section 2.3), unlike the CPU cores.

P0. **Access to shared resources:** While, on some newer architectures, it is possible for the CPU and GPU to share various levels of caches in the memory hierarchy[48], this is not a guarantee in every system. However, at the very least, every processor residing on the SoC needs to eventually share system memory with the other co-processor. This provides us with our target: *system memory*.

Considering the architecture in Figure 3.2, an attacker has 3 main access points to DRAM. She can access memory: (i) by inputing vertices to the vertex shaders, (ii) fetching textures within the shaders themselves or (iii) 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 (i) buffers containing vertices are lazily instantiated making it more difficult to obtain predictable allocations. Furthermore, since the synchronization among the parallel executions of the same shader over multiple SPs (i.e., SIMD) is implicit, it makes it difficult for an attacker to achieve predictable behaviors, necessary when carrying out any type of microarchitectural attack. Accessing memory through OCMEM (iii) is also tricky given its large size and asynchronous transfers to system memory. We hence opted for implementing our primitive *P0* through the *texture fetching* functionality (ii). Texture fetching takes place within the boundaries of a shader, providing strong control over the order of memory accesses. Moreover, textures' allocations are easy to control, making it possible to obtain more predictable memory layouts as we explain in Section 4.3.

The remaining obstacles are (i) dealing with L1 and L2 in between shaders and DRAM, and (ii) 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 will then use this information to selectively flush the GPU caches in order to reach DRAM in Chapter 4.

3.3.1 Texture addressing

While the developer, when programming the shaders does not care about memory location and can directly design them taking into consideration the coordinate systems of her 3D model, we, on the other hand, do care about the structure of the data. This allows us to understand how the GPU caches handle their requests and, to a certain extent, where the data we are accessing is positioned in memory.

Integrated GPUs partition textures in order to maximize spatial locality when fetching them from DRAM [49]. This process is known as *tiling*, or *binning*, and it 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 (i.e., $16_{px} * 4 = 64$ 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:

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

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 *P0* targets DRAM, we need to address them in the physical address space. Luckily, textures are page-aligned objects. Hence, their virtual and physical addresses share the lowest 12 bits[1].

3.3.2 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. The 4 attributes we need to recover are: (a) cacheline size, (b) cache size, (c) associativity and (d) replacement policy. We describe our novel reverse engineering technique and how we used it to obtain information about these attributes.

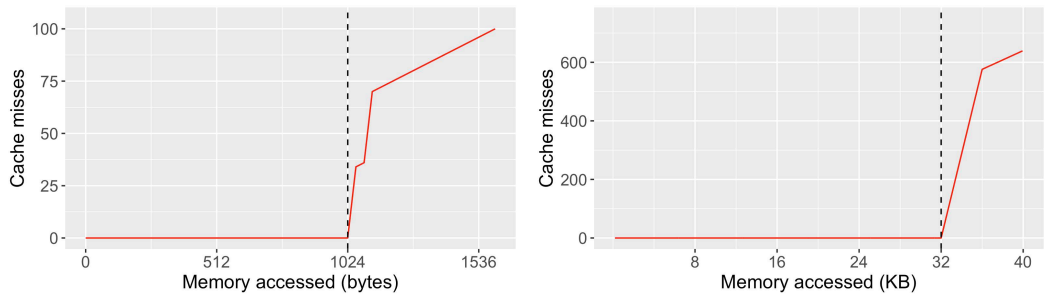
[1] On most modern architectures, a memory page is 4 KB. Hence, 12 bits for the offset within the page ($2^{12} = 4096$)

```
1  #define MAX max // max offset
2  #define STRIDE stride // access stride
3
4  uniform sampler2D tex;
5  attribute coord;
6  void main() {
7          vec2 texCoord;
8          vec4 val;
9          // for (a) we can omit the external loop
10         for (int i = 0; i < 2; i++) {
11                 for (int x = 0; x < MAX; x += STRIDE) {
12                         texCoord = offToPixel(x);
13                         val += texture2D(tex, texCoord);
14                 }
15         }
16 gl_Position = val;
17 }
```

Listing 3.3.1: Vertex shader used to measure the size of the GPU caches. To obtain the cacheline size (a) we run the same code performing a single loop over the accesses.

Reversing primitives: To gain the aforementioned details we (ab)use the functionalities provided by the GLSL code that runs on the GPU. Listing 3.3.1 presents the code of the shader we used to obtain (b). We use similar shaders to obtain the other attributes. The `texture2D()` function[50] 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* (NDC). These coordinates represent the 3D model to the 2D *screen space* and they are limited to the [-1, 1] range. For this reason, we need to perform an additional conversion between pixel coordinates and NDC. This merely consists in dividing the pixel (x, y) by (W, H).

With this code, we gain access to memory with 4 bytes granularity (dictated by the size of `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). These performance counters are directly accessible from OpenGL via the `GL_AMD_performance_monitor` extension. This exposes performance counters that can be used to monitor L1 requests (`TPL1_TPPERF_L1_REQUESTS`) and misses (`TPL1_TPPERF_TP0_L1_MISSES`) as well as read requests to the memory controller (`AXI_READ_REQUESTS_TOTAL`).



(a) L1 size (b) UCHE size

Figure 3.4: The two plots show the results obtained by the performance counters when running the shader in Listing 3.3.1 with stride equal to the cacheline size.

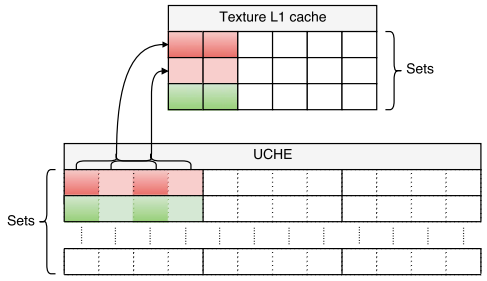


Figure 3.5: Mapping of a UCHE cacheline into multiple L1 cache sets

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

Table 3.1: Details of the two caches.

Size: We can identify (a) and (b) with the shader in Listing 3.3.1 – we need to apply minor changes to obtain (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. We identify this value when $C_{miss} = 2$. This means we overflowed the content of a single *cacheline*, hence a new cacheline is loaded. Then we set the `STRIDE` to the cacheline size and run the shader until $C_{miss} - C_{req}/2 \neq 0$. We run the same experiment for both L1 and UCHE. Figure 3.4 shows the outcome of (b) for both of the caches. We report the retrieved values in Table 3.1.

Associativity and replacement strategy: The non-perpendicular rising edge in both of the plots in Figure 3.4 confirms they are set-associative caches and it suggest a LRU or FIFO replacement policy. Based on the hypothesis regarding the replacement policy we retrieved associativity (c) using *dynamic eviction sets*. We depict this technique in the diagram in Figure 3.6. This consists in filling up the cache with addresses belonging to a set S_0 plus an additional evicting address E_0, and then sequentially adding the addresses evicted from S_0 to the eviction set. Once the address added to the eviction set is equal to E_0 we have evicted, hence recovered, every element inside the cache set. Finally, to identify the mapping function, we

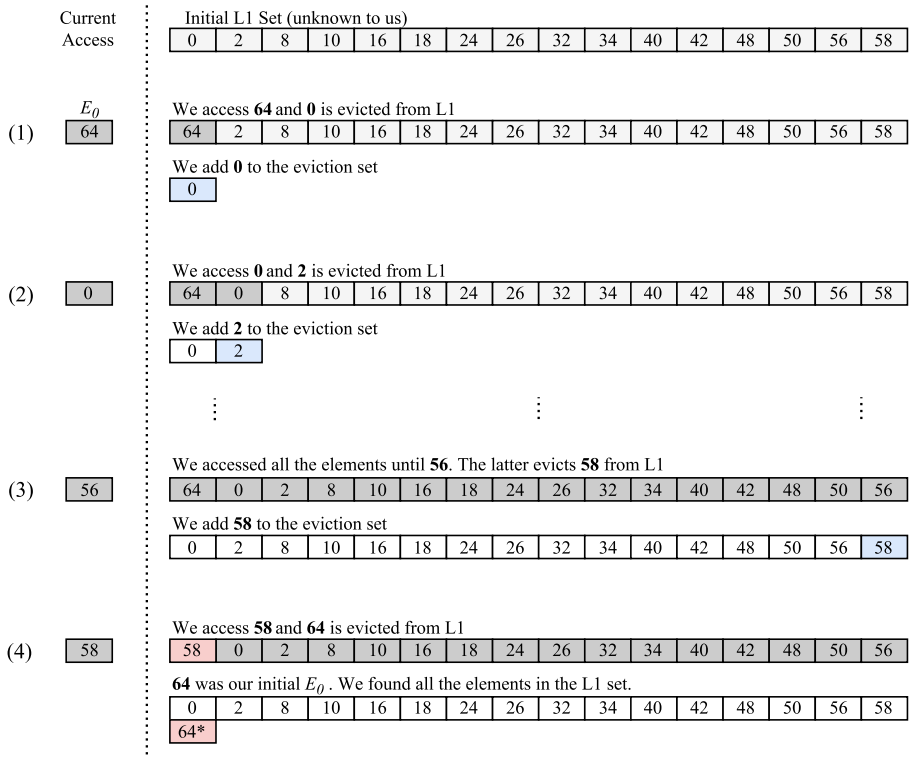


Figure 3.6: Dynamic eviction set to reconstruct the associativity and mapping function of the cache.

need to identify the common bits among the addresses belonging to the eviction set (excluding the offset bits as explained in Section 2.2). These, are responsible for indexing the cache set.

While UCHE resulted in a normal 8-way set associative cache behaving as the caches described in Section 2.2, we identified 16 ways mapped with an unconventional function for L1. Given a virtual address `0x9d142a14`, considering just its 10 least significant bits ...`100|001|0100`, bits [0,3] are used for the offset ($log_2(cacheline)$), while bits 4 and 6 are used to identify the cache set. The remainder creates the tag. This mapping, as we will explain in the next paragraph, fills L1 with shuffled values from UCHE.

Once identified the associativity we can recover the replacement strategy (d) by filling up a cache set and accessing again the first element before performing the first eviction. We use this technique, due to the prior assumption of a LRU or FIFO replacement policy. Since the first element gets evicted even after a *recent use* in both of the caches we deduce a FIFO replacement policy. This replacement policy is quite unusual and not commonly seen on CPU caches.

Synopsis: All the details about these two caches are summarized in Table 3.1. 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 and these 4 are then shuffled over two different L1 cache sets by the mapping function explained above. We shown this mapping in Figure 3.5. We will exploit this property when building efficient eviction sets in Section 4.3. Finally,the only missing detail about these caches is their coherency policy. We discovered L1 and UCHE to be non-inclusive caches. This was to be expected considering that L1 has more ways than UCHE.

3.4 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. Jiang et al. [51, 52] demonstrated two instances of timing attacks on GPGPUs to recover AES keys. While Naghibijouybari et al. [53] demonstrated a more generalized outcome demonstrating the possibility of building covert-channels between two processes concurrently running on a GPU. All these attacks, however, focus on general purpose discrete GPUs which are usually adopted on cloud systems, whereas we targeted HSAs on commodity hardware which are not as powerful and do not necessarily provide support for these libraries.

If confined to the OpenGL API, some newer versions provide other, more powerful, means to gain access to memory such as image load/store, which supports memory qualifiers (e.g., `volatile` usually employed for non-coherent memory accesses when performing Rowhammer attacks) , or SSBOs (*Shader Storage Buffer Objects*), which would have given us linear addressing instead of the tiled addressing explained in Section 3.3.1. However, they confine the threat model to local attacks carried out from a malicious application. As we will show in Chapters 4 and 5, we decided to restrict our abilities to what provided by the OpenGL ES 2.0 API in order to relax our threat model to remote WebGL-based attacks.

Finally, the reverse engineering technique we described in Section 3.3.2 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: PerfKit [54].

In the case of a restricted access to the PMU, or no performance counter providing the necessary data (i.e., cache misses), we could have obtained the same outcome performing a timing attack. This consists in running the shader in Listing 3.3.1 and

measuring the time it took after every iteration. We can identify the size of the cache by identifying a spike on the time taken to access (twice) the elements in the access pattern. Even if this makes the process slightly more complex, it does not prevent an attacker from performing the analysis.

Chapter 4

Attacker Primitives

Every type of attack base itself on a set of primitives. Microarchitectural attacks are no exception. As mentioned in Chapter 1 we can divide microarchitectural attacks in two macro families. Those that aim at *stealing* data and those that aim at *corrupting* it. We restrict our analysis to the two most common types of attacks within these two families, namely (i) timing side-channels, to steal data, and (ii) Rowhammer attacks to corrupt it. In this chapter we define and recover the necessary primitives to conduct these attacks from an integrated GPU, exploring why the latter "*accelerates*" these attacks; i.e., makes them more effective than their CPU counterpart.

4.1 Defining the primitives

We start our investigation from side-channels, describing the current state of the art in terms of attacks and defenses and identifying our necessary primitives to carry out such attacks. Then we proceed in doing the same for Rowhammer attacks, giving first an exhaustive introduction about the Rowhammer vulnerability and eventually discussing required primitives and defenses.

4.1.1 Timing side-channels

A primary mechanism for leaking data using microarchitectural attacks is to time operations over resources shared with a victim process (*P0*). For instance in any type of cache attack, the attacker measures the difference between a *cache hit* and a *cache miss* to reveal information about a secret operation. This hints at our first primitive: *timers*.

Even though the most common family of timing side-channels are the so-called cache attacks. Since our primitive *P0* targets system memory we do not address this family[1], whereas we thoroughly describe the state of the art in timing side-channels targeting DRAM. These attacks are known as DRAMA attacks.

[1]For an exhaustive analysis of this topic we suggest [55]

DRAMA Attacks: As for the caches also DRAM chips present a similar concept of cache hits and cache misses. This is exhibited by the *row hits* and *row conflicts* generated by the row buffer. The time difference between row hit and row conflict has been exploited for different purposes in the literature. While this difference was already known [43], Pessl et al. [16] first described a full class of attacks based on this property, coining the term *DRAMA* attacks. Pessl et al. [16] used the difference between row hit and row conflicts to develop a generalized reverse engineering technique that can be used to retrieve the usually undocumented DRAM addressing function on different architectures. Furthermore, they presented a cross-CPU covert channel with high bandwidth and low error rate that exploits *row conflicts* in a `PRIME+PROBE`[2] fashion where the attacker repeatedly access a row while measuring the access time. Bhattacharya et al. [56] exploited this time difference to identify the DRAM bank of an RSA exponent to later corrupt it using a Rowhammer attack. And finally, Schwarz et al. [20], following the trend of remote JavaScript attacks, demonstrated the possibility of detecting this time difference even from a webpage making it possible to build DRAM-based remote covert channels.

***P1*. Timers:** Having access to high-resolution timers is a primary requirement for building any type of timing side-channel. For example, in case of DRAMA attacks, the attacker must be able to tell the difference between a row hit and a row conflict with a difference of tens of nanoseconds. While there are many examples of timing attacks executed natively[9, 10, 13, 15, 16, 57] — on both caches and DRAM — recent work shows that it is also possible to perform these attacks in JavaScript from the browser [19, 20, 21, 23, 24], extending their threat to the Internet users.

In response browser vendors reduced the resolution of the `performance.now()` timer [33, 34, 35, 36] to $5\,\mu s$ — 100 ms in the case of the Tor browser — in order to make it harder for attackers to perform timing attacks in JavaScript. However, recent work shows that simply reducing the timers resolution is not enough. It is possible to use another core as a timing source by exploiting the `SharedArrayBuffer` [21] extension that atomically shares data among two threads. Or it is possible to use two enhancing techniques known as (i) clock-edging or (ii) edge-thresholding to improve the resolution of a coarse timer [20, 37]. *Clock-edging* (i) recovers higher resolution from the coarsened `performance.now()` by calling this function in a tight loop until the value changes. This allows an attacker to obtain a resolution of $\sim 500\mu s$ from a $5\,\mu s$ timer. *Edge-thresholding* (ii), on the other hand, uses a more advanced padding technique to gain even higher resolution from limited resolution timers. In this case the attacker uses a constant-time padding function (f_P) that allows her to fill the gap between the execution of the secret function (f_S) she is trying to measure and the edge of the new clock (i.e., change of the `performance.now()` value from t to $t + 5\,\mu s$). The attacker can then leak information about f_S, with $2\,ns$ precision, by requesting the value of `performance.now()` after executing $f_S + f_P$ and checking if

[2] `PRIME+PROBE` is a generalized attack that consists in (*i*) filling a cache set with attacker controlled data (`PRIME`), (*ii*) waiting for a victim operation to run, and (*iii*) refilling the cache set while measuring the time of this operation (`PROBE`).

the new measurement has exceeded the new clock edge.

Defenses: As a consequence to these new timing techniques, more fundamental approaches propose to restructure the architecture of the browser in order to limit, or even eradicate, the possibility to leak data. Kolhbrenner et al. [37] designed Fuzzyfox, a browser that introduces randomness in the JavaScript event loop to add noise to timing measurements performed by an attacker destroying the reliability of the measurement taken with the two aforementioned implementations and therefore reducing the bandwidth of the side-channel. Cao et al. [38], instead, propose an antithetical solution named DeterFox that 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 making it completely impossible to build any side-channel.

We show in Section 4.2 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.

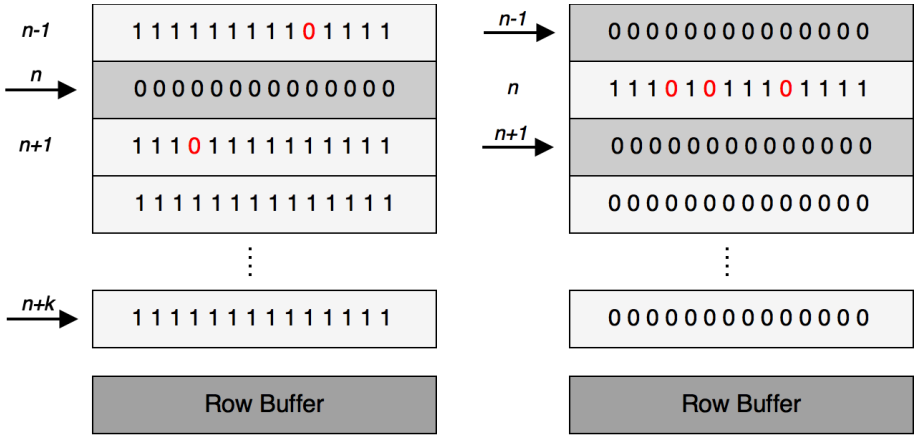
4.1.2 Rowhammer attacks

Rowhammer is a prime example of an attack that corrupts data by abusing a software-based fault injection attack. In Section 2.3, we described the organization of a DRAM chip explaining the concept of rows. These rows are composed of *cells* where each cell stores 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 an fault 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 [12].

There are two main variants of Rowhammer: (i) single-sided Rowhammer and (ii) double-sided Rowhammer. (i) Single-sided Rowhammer access a specific aggressor row n — and a random row $n + k$ needed just to cause a row conflict — triggering bit flips on the two adjacent rows $n - 1$ and $n + 1$ (Figure 4.1a). (ii) Double-sided Rowhammer, instead, amplifies the power of single-sided Rowhammer by reversing the roles of these rows. Therefore, the attacker quickly accesses rows $n-1$ and $n+1$ (i.e., aggressor rows) in order to impose higher pressure on row n's (i.e., victim row) capacitors triggering more bit flips (Figure 4.1b).

The first recorded exploitation of this vulnerability is attributed to Seaborn and Dullien who managed to gain kernel privileges by triggering bit flips on page tables[17]. The same exploitation technique was reproposed in Rowhammer.js to gain root privileges from a browser [23] and on Drammer to obtain the same on ARM Android devices[32]. The main obstacles in performing these kind of attacks are (i) knowing the physical location of the targeted row and (ii) fast memory accesses. These define our next two primitives:



(a) Single-sided Rowhammer (b) Double-sided Rowhammer

Figure 4.1: Two flavors of the Rowhammer attack. Single-sided hammers from a single row causing bit flips in the adjacent rows. Double-sided hammers two rows to trigger bit flips in the "sandwich" row.

***P2*. 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 either exploit information from interfaces such as `/proc/pagemap`, or 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 [17, 18, 23], 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 [32] abuses the physically contiguous memory provided by the Android ION allocator. This remains a fundamental requirement even when approaching this from the GPU.

***P3*. 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 [17, 18, 31]. On most ARM platforms, flushing the CPU cache is a privileged operation. Drammer [32] 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 [23] and Dedup Est Machina [24] rely on eviction buffers to flush the caches. While this works on x86, flushing CPU caches on ARM is too slow to trigger bit flips [32]. Hence, it remains an open question whether it is possible to perform Rowhammer attacks from the browser on most mobile devices.

Defenses: Current state-of-the-art defenses propose two main solutions to address the Rowhammer issue:
– *Monitor access patterns*: This solution is based on the fundamental nature of Rowhammer that requires to frequently access a specific *aggressor* row — or two in the case of double-sided Rowhammer. By exploiting performance counters to monitor these access patterns the system can perform targeted refreshes to recharge the victim rows [40, 58].
– *Physical memory isolation*: By isolating domains with different privileges (i.e., kernel- and user- space), the attacker is (*theoretically*) incapable of obtaining privilege escalation [39].

In section 4.3 we address primitive *P2* and we discuss how we can use a novel timing side-channel executed from the GPU that mixes the knowledge of the DRAM architecture [16] and low-level memory management to find contiguous physical regions of memory from the browser. Afterwards, in Section 4.4, we reuse this primitive to enable the more efficient double-sided Rowhammer and report on the first successful Rowhammer bit flip in the browser on ARM devices.

4.2 The Timing Arms Race

To implement timing side-channel attacks, attackers need the ability to time a secret operation (*P1*). In this section, we present explicit and implicit GPU-based timing sources, demonstrating how state of the art defenses such as Fuzzyfox and DeterFox are fundamentally 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.

4.2.1 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 [59]. 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.

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.

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 μs, reducing the granularity of the actual measurements.

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. As a consequence, techniques such as clock-edging and edge-thresholding can both be applied to this timer too.

Like `TIME_ELAPSED_EXT`, this timer is driver- and browser-dependent. Firefox supports it, while Chrome disables it due to compatibility issues [60].

4.2.2 WebGL2-based timers

The timers introduced in the previous section are made available through a WebGL extension. This, similarly to the `SharedArrayBuffer` timer [21], can be argued to be easily amendable, by simply disabling the 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()`.

clientWaitSync

This function waits until either the sync object gets *signaled* , 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 [37]. 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:

1. Flush the command buffer.
2. Dispatch the command to the GPU.
3. Issue the WebGLSync fence.
4. (*Optional*) Execute CPU operation.
5. Count the loops of `clientWaitSync(0)` until it is signaled.

Whether Step 4 is needed, depends on the secret the attacker is trying to leak, and if this is a CPU or GPU operation. If measuring a secret GPU operation we use the

	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`	$60\,ns$	$0.4\,ns$	$0.4\,ns$	$1.8\,ns$

Table 4.1: Results on different browsers for the two families of timers. With † we indicate driver dependent values.

CPU as ground truth, but if 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 show in Table 4.1.

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 is exactly analogous 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.

4.2.3 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 [37] and DeterFox [38]. 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 4.3. However, since we were not able to compile Fuzzyfox and DeterFox for Android platforms, we did not carry out our evaluation on such system to maintain consistency over the measurements.

Table 4.1 shows the results of our experiments. The two explicit timers, as mentioned before, are driver-/browser-dependent, but if available, return unam-

biguous 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 [37, 38]. Chrome rounds up the value for `TIME_ELAPSED_EXT` to $1\,\mu s$ and returns 0 for `TIMESTAMP_EXT`. As mentioned earlier, the granularity of the measurements obtained from `TIMESTAMP_EXT` can be further improved by implementing clock-edging or edge-thresholding.

The evaluation of the two WebGL2-based timers, on the other hand, demonstrated them to be 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.

4.3 Side-Channel Attacks from the GPU

Now that we have access to high-precision timers (*P1*) and we described how an attacker can gain access to shared resources (*P0*) through the texture fetch functionality exposed to the shaders, we can exploit these two primitives to build a timing side-channel. 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 (*P2*) directly in JavaScript, mandatory requirement for building effective Rowhammer attacks.

We start by describing 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.

4.3.1 Cache Eviction

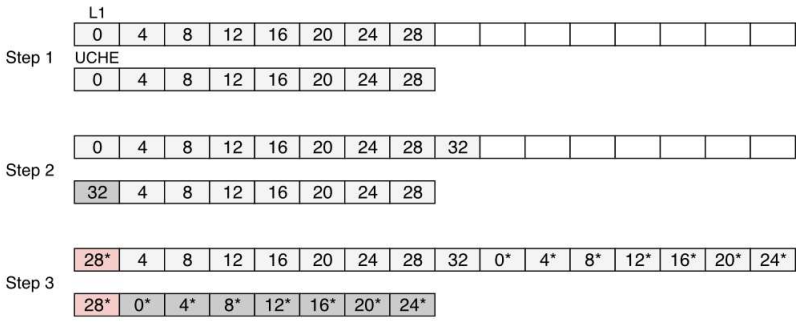


Figure 4.2: Efficient cache eviction strategy. The * represents an access to an address $n[4\text{K} * i + 32]$.

Considering the GPU architecture presented in Section 3.3, 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. Recalling the analysis in Section 3.3: UCHE and L1 have different cacheline size with $UCHE_{line} = 4 * L1_{line}$, L1 has 16 ways while UCHE just 8 and they are non-inclusive caches. This last property, combined with the different number of ways, makes it more difficult to identify canonical eviction sets. However, as shown in Figure 3.5, 32 of the 64 bytes in a UCHE cacheline map to two 16-byte cachelines in the same L1 set. We exploit this property to build an efficient eviction strategy.

UCHE sets have 8 ways, a FIFO replacement policy and they load addresses at the stride of 4 KB (i.e., addresses $n[0]$ and $n[0 + 4K]$ map to the same UCHE cache set). Hence, we need at least 9 memory addresses that are 4 KB away from each other to evict a cacheline (i.e., $n[4\text{K} * i]$ with $i \in [0, 8]$). Figure 4.2 demonstrates

how we use these 9 addresses to evict both of the caches. We start by accessing 8 memory addresses $n[4K*i]$ with $i \in [0, 7]$ (1). These will load 8 64-byte cachelines on UCHE and as many 16-byte cachelines on L1. Then, (2) we access the last memory address $n[32K]$ of our access pattern. This evicts one cacheline from UCHE while loading a new one in L1. Finally, (3) we access again the initial 8 memory addresses $n[4K * i]$, however, this time by shifting them of 32 bytes (i.e., $n[4K * i + 32]$). This loads the same cachelines to UCHE while loading new cachelines in L1. While the first 7 accesses cause evictions in UCHE and not in L1. The last access of this sequence, apart from causing another cache miss in UCHE, will be a cache miss also in L1 causing the first L1 eviction. Then, by iteratively alternating these 9 accesses between $n[4K * i]$ and $n[4K * i + 32]$, we are able to load the same cacheline into UCHE while loading a different cacheline into L1. This strategy allows us to evict L1 by using the same eviction set for UCHE without incurring additional misses using textures fetches within the shader. We hence can ignore L1 when mentioning memory accesses from now on. Every memory access, however, iteratively represents accessing both $n[4K * i]$ and $n[4K * i + 32]$ as part of UCHE eviction.

4.3.2 Allocating contiguous memory

As describe in Section 3.3, our test chipsets (i.e., Snapdragon 800/801) operate on virtual addresses due to the presence of an IOMMU. This means that the GPU is capable of dealing with physically non-contiguous memory and it allows the GPU driver to allocate it accordingly. The Android GPU driver for the Adreno GPU allocates memory using the `alloc_page()` macro in the Linux kernel which queries its Buddy allocator for single pages [61].

The Buddy Allocator: The Buddy allocator manages free memory in chunks of *power-of-two* number of pages [62]. The exponent of this expression is known as *order*. When requesting a block of memory to Buddy this gets allocated from the smallest order it can fit in. For instance, if 7 contiguous pages are requested, they fit in an order 3 allocation ($2^3 = 8$ pages). If no chunk of order n is vacant a chunk from the next first available order (i.e., $n + 1$) is split in two halves. These are the so-called *buddies* and they are both of order n. If there are no slots available at order $n + 1$, the operation is repeated recursively at order $n + 2$ and higher. All the allocated memory coming from order $n + k$ is physically contiguous. Therefore, all the consecutively allocated n-order chunks will be contiguous too. This property is the backbone of primitive *P2*.

In order to obtain physically contiguous memory starting from order 0 allocations, we need to exhaust all the memory until we reach an order n that satisfies the mandatory constraint of 3 consecutive rows to perform double-sided Rowhammer. Pessl et al. [16] reverse engineered the function mapping physical addresses to DRAM location for the Snapdragon 800/801 chipsets. Row n stores two consecutive pages (i.e., 8 KB) which are sub-split among two different ranks. Having 8 KB per row in each of the 8 banks translates to 64 KB row alignment (Figure 4.3). This converts

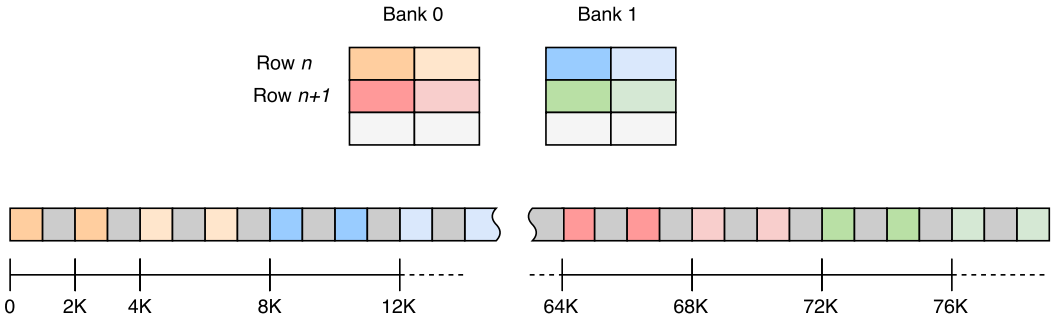


Figure 4.3: Snapdragon 800/801 Memory addressing layout. At the bottom the physical address space. This maps two half pages to a row. The other half pages are mapped to the same row but on a different rank.

to 16 pages. As described in Section 4.1.2, to carry out double-sided Rowhammer attacks we need at least three rows. This translates to order $\lceil log_2(16 * 3) \rceil = 6$ allocations. Order 6 allocations cover $2^6 = 64$ contiguous pages which spans over 4 complete rows.

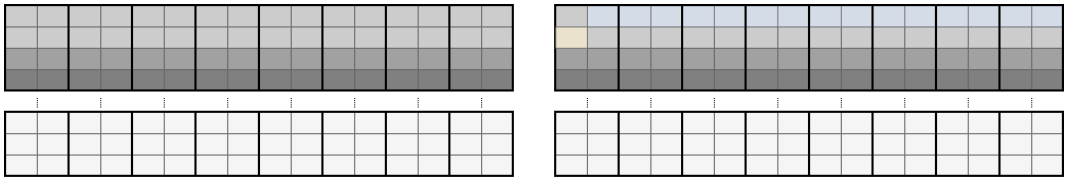
4.3.3 Detecting contiguous memory

Due to the predictable behavior of the Buddy allocator just described, we know with high probability that once we allocate some memory at one point we will allocate some contiguous areas. Now we need to identify these contiguous areas to perform double-sided Rowhammer attacks. In order to do so, we build a timing side-channel that measures the time difference between *row hits* and *row conflicts* directly from the GPU.

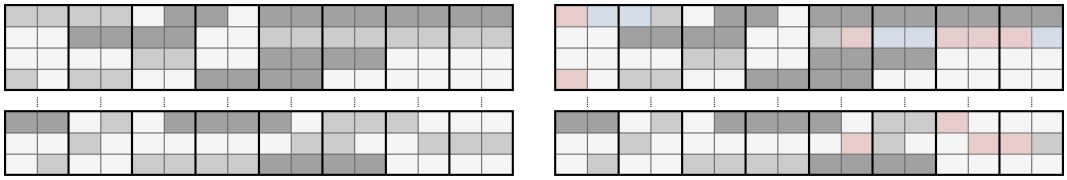
Design

With order ≥ 4 allocations covering one, or more, rows across every bank, we can test for row hits accessing addresses mapping to the 16 pages in row n. Assuming the first byte of row n to be $n[0]$, we know that memory accesses at $n[0]$ will cause a row hit with memory accesses at $n[0 + 4\text{K}]$, a row conflict with memory accesses at $n[0 + 64\text{K} * i]$ and $n[4\text{K} + 64\text{K} * i]$ with $i \geq 0$.

We previously explained that to obtain a block of order $n + 1$ from the Buddy allocator we need to exhaust every n-order. This implies that allocations of order n are likely to be followed by other allocations of the same order(left diagram in Figure 4.4a). Since every allocation of order ≥ 4 will span over a full row across every bank, accesses at $64 * i$ KB are likely to generate row conflicts for every allocations of order ≥ 4. Therefore, there is no point in triggering row conflicts to detect contiguous allocations. Furthermore, this property of the Buddy allocator degrades the granularity of our side-channel to a maximum detectable order of 4,



(a) Allocations of order ≥ 4. Memory layout on the left and access to the *hit*-pattern on the right. We see how the *hit*-pattern does not generate any row conflict (blue pages). The yellow page represents the 16 access we removed from the access pattern.



(b) Allocations of order < 4. Memory layout on the left and access to the *hit*-pattern on the right. We see that allocations, contrarily to what happens in Figure 4.4a, do not follow a structured pattern. As a consequence they are very likely to generate row conflicts (red pages) when accessing the *hit*-pattern.

Figure 4.4: The diagrams depict the 16 pages per row split among 8 banks of the Snapdragon 800/801 architecture. It demonstrates why accesses following the *hit*-pattern allow us to identify between allocations of oder ≥ 4 from allocations of order < 4.

since every allocation of order ≥ 4 will behave identically. Nonetheless, we can still adopt this side channel to gain an insight of the Buddy allocation trend. We can discern between the two populations (i.e., order ≥ 4 and < 4) by measuring the time it takes to access memory following a *hit*-pattern that tries to cause row hits among its memory accesses (Figure 4.4).

Implementation

A full row spanning across every bank covers 16 pages of the physical address space (i.e., order 4 allocation). In order to test for row hits we want to touch all these pages and measure the time it took to perform this access pattern (i.e., $n[0 + 4\text{K} * i]$ with $i \in [0, 15]$). However, for allocations of order ≥ 4, access patterns with $n[0]$ that is not row aligned (i.e., $n[0]$ not accessing the first page of a complete row) generate a row conflict between the first ($n[0]$) and last ($n[0 + 4\text{K} * 15]$) access — yellow page in the right diagram of Figure 4.4a. Since the alignment of any access pattern is unknown, we then limit the *hit*-pattern to 15 addresses (i.e., $n[0 + 4\text{K} * i]$ with $i \in [0, 14]$).

In order to obtain a single measurement, we test this sequence for multiple offsets (i.e., 4 KB stride) within a 512 KB area of the address space and we then compute the median to identify if the underlying block is backed by contiguous allocations.

This allows us to maximize the number of row conflicts for allocations of order < 4 while filtering out the noise from allocations of order ≥ 4. To obtain these measurements, we employ the `TIME_ELAPSED_EXT` asynchronous timer presented in Section 4.2 due to its explicit nature. Nonetheless, we can use any of the homebrewed timers introduced in that section.

Evaluation

We evaluate our side channel to show how it can detect allocation trends of the Buddy allocator. Figure 4.5 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. We can discern between the two different populations by testing the median of the measurements taken within the boundaries of a 512 KB sample against a preset threshold value. Furthermore, we can filter out false-positives by observing the traits of the two different distributions where allocations of order < 4 are more spread than those of orders ≥ 4. This is due to the varying possibility of generating row conflicts depending on the offset under test.

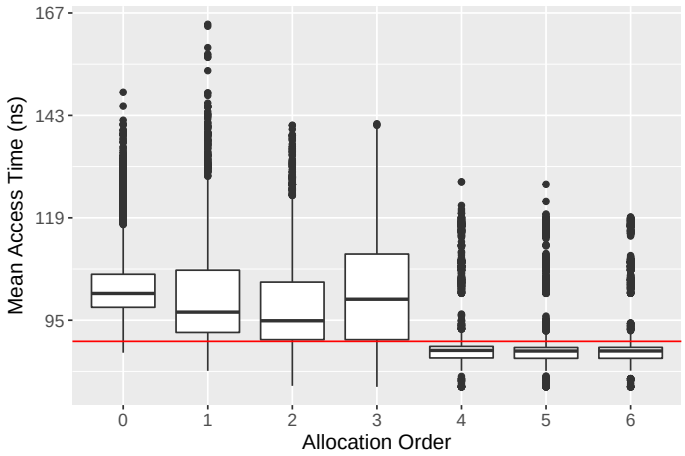


Figure 4.5: Evaluation of contiguous memory timing side-channel.

4.4 Rowhammer Attacks from the GPU

Now that we have access to contiguous physical memory directly in JavaScript using our GPU-based side-channel attack discussed in Section 4.3 we can demonstrate how we can remotely trigger Rowhammer bit flips on this contiguous memory by exploiting the texture fetching functionality from a WebGL shader. We then evaluate the results of our implementation.

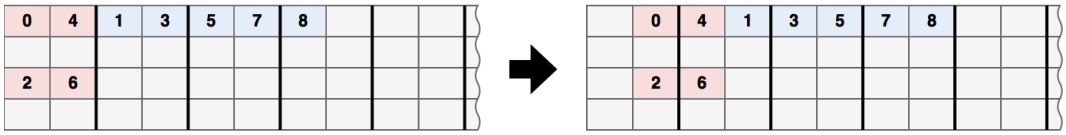
4.4.1 Eviction-based Rowhammer on ARM

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 us to optimize the access time to the aggressor rows. In Section 4.3, 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.

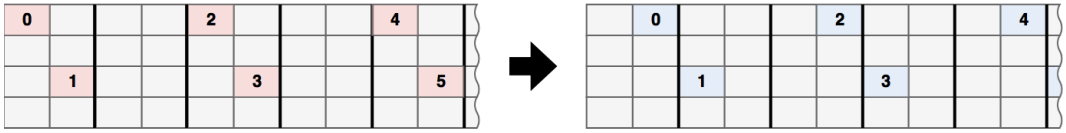
Since DRAM rows cover 8 KB areas (split among the ranks) of the virtual address space and each UCHE set stores addresses at 4 KB of stride we can map at most two addresses from each row to a cache set. Recalling the description in Section 4.1.2, when performing double-sided Rowhammer we access rows $n-1$ and $n+1$ to trigger bit flips in row n. Which means that with two aggressor rows, and two pages per row, we have at most 4 *hammering*-accesses in our access pattern. 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. This access pattern is depicted in Figure 4.6a.

Since we currently have no knowledge about the row alignment among the different allocations we need to indiscriminately hammer every 4 KB offset. However, not all of these alignments are effective in order to trigger bit flips. For instance, Step 2 in Figure 4.6a, splits the *hammering*-accesses over 2 banks, dramatically incrementing the mean time between these.

We can optimize this by building a timing side-channel similar to the one presented in Section 4.3. Recalling the description of the *hit*-pattern, we removed $n[4\text{K} * 15]$ from the access pattern since it caused row conflicts for every sequence with $n[0]$ not row aligned. Now we can exploit this very property to detect the alignment of the two pages. We can build an access pattern that aims at detecting this row conflict. This follows the pattern presented in Figure 4.6b. As you can see



(a) Hammer Pattern. In red the *hammering*-accesses that cause row conflicts. In blue the *idle*-accesses that cause row hits on other banks.



(b) Access Pattern to detect row alignment. By measuring the time difference between shifting this access pattern of 4 KB we can identify where the row is starting and cut in half the number of offsets we need to hammer.

Figure 4.6: Access patterns used for Rowhammer attacks. The numbers in the pages represent the order of the accesses.

this access pattern allows a user to detect which alignments start at the beginning of the row. By identifying this alignment we can reduce in half the number of offsets we need to hammer increasing the stride after every execution from 4 KB to 8 KB. Thus, we can skip Step 2 of Figure 4.6a.

4.4.2 Evaluation

Drammer [32] 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 $\sim 260\,ns$ for each memory access. We computed the mean execution time over the 9 memory accesses following our *hammer*-pattern. The mean access time for *hammering*-accesses is on average $\sim 180\,ns$, which means that our GPU-accelerated 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[3].

We compare our implementation against a native eviction-based implementation running on the CPU adopting cache eviction strategies proposed by in Rowhammer.js [23]. 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 flip. 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, compared to the threshold value of $\sim 260\,ns$, is far too slow

[3]Out of these 3 smartphones, however, the HTC One M8 and LG G2 revealed a limited number of vulnerable cells, hence triggering less bit flips. We verified this even with a native Drammer implementation.

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

Table 4.2: Ability to trigger bit flips from a native application (left) and remotely (right) on ARM test platforms. * implements eviction-based Rowhammer.

to trigger Rowhammer bit flips. Table 4.2 summarizes our findings showing the effectiveness of the currently known techniques to trigger bit flips on ARM. Our GPU-based Rowhammer attack is the only known technique that can produce bit flips with eviction-based Rowhammer on ARMv7 architecture. Moreover, due to the seamless commutativity between OpenGL and WebGL our technique is the first one to prove the possibility of performing remote Rowhammer attacks on mobile platforms.

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. Time to first bit flip includes in the measurements the time required to detect contiguous memory via our side-channel attack presented in Section 4.3). This is to demonstrate the practicality of such attack disproving the validity of the assumption made by Brasser et al. [39] that remote Rowhammer attacks are too time consuming to be relevant in a real scenario.

We run the experiment 15 times looking for 1-to-0 bit flips. After each experiment, we restart the browser to bring the phone to its initial state. It took us between 13 to 40 seconds to find our first bit flip with an average of 26 seconds. 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. De facto, 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 $flips/min$.

Chapter 5

Exploitation & Mitigations

In this chapter, we describe how we managed to harmoniously coordinate all the primitives presented in Chpater 4 to build a reliable exploit compromising a mobile browser. We first exploit the timing side-channel presented in Section 4.3 to retrieve the physical location of the memory and we afterwards exploit the Rowhammer bit flips described in Section 4.4 to compromise a JavaScript object and obtain an arbitrary memory read/write primitive. Before discussing our exploitation technique we introduce our threat model. This model is not restricted to the exploit we present in this chapter but has the goal of showing the extent of this far-reaching threat answering the research question we initially posed in Chapter 1. We conclude this chapter presenting our suggestions on how to mitigate, in software, such threat.

5.1 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 [37, 38] and protect kernel memory from Rowhammer attacks [39].

5.2 Exploitation

In order to compromise the JavaScript sandbox of the Firefox browser we need one more primitive used in any standard exploitation technique. This is the arbitrary memory read/write primitive. An attacker in possession of this primitive can read and write any mapped region of the process virtual address space allowing her to obtain remote code execution by overwriting a method pointer of a JavaScript object

with arbitrary content [63]. While usually this is obtained by exploiting multiple software bugs, we want to show that we can obtain such primitives solely by exploiting our GPU-accelerated microarchitectural attacks. To do so we need to identify a data structure that gives us such power if compromised by our bit flips. We call this object *hammer* target and we define it in Section 5.2.2.

The exploitation consists in 4 main steps: (i) we first need to identify contiguous memory, then (ii) we need to hunt for exploitable bit flips, afterwards (iii) we need to land our *hammer* target on the vulnerable cell and finally (iv) we need to trigger the bit flip to compromise our object in order to gain the arbitrary read/write primitive. We describe each of these steps in a separate section.

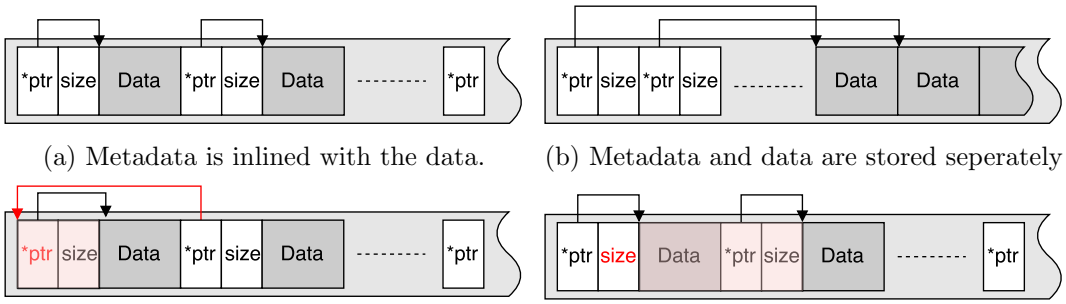
5.2.1 Allocate and detect physically contiguous memory

First step to carry out our attack is what we described in Section 4.3, namely we need to know the (*relative*) physical address of memory allocations to reliably mount the more effective double-sided Rowhammer attacks. For this purpose, we exploit our timing side-channel to coarsely identify contiguous memory up to 64 KB. Then, to discover allocations of order ≥ 6 we use the first bit flip as a side-channel. Considering the predictable behavior of the Buddy allocator we know that allocations of order n are likely to be followed by allocations of order $\geq n$. Since, with our timing side-channel, we have recovered a block of the address space backed by an allocation of order ≥ 4 we can now heuristically hammer the memory allocated after this area while looking for bit flips to detect allocations of order ≥ 6. The first bit flip will likely represent an allocation of this order. While this methodology is based on a heuristic, we demonstrated its effectiveness in Section 4.4 with a mean time to first bit flip of 26 s.

5.2.2 Bit flips hunt

Once we have obtained physically contiguous memory, we need to find exploitable bit flips to later reuse to gain control over the memory (Figure 5.3a). This process resembles what we explained in Section 4.4. The number of exploitable bit flips, however, completely depends on the *hammer* target.

Hammer Target: The most common JavaScript objects used to retrieve an arbitrary read/write primitive are the `ArrayBuffer` objects. These buffers were introduced to aid the usage of libraries such as WebGL which seek high performances and need full control over raw data making it unfeasible to employ normal JavaScript `Array` objects. With byte control over the stored data they implicitly provide arbitrary read/write. However, this is scoped to attacker's controlled data. While initially these objects were stored in the heap with medatadata and data inlined (Figure 5.1a), browsers' developers quickly realized that it represented a terrible design in terms of security since every overflow could allow an attacker to overwrite the metadata of the object following in the heap. As a consequence, in newer browsers



(a) Metadata is inlined with the data.

(b) Metadata and data are stored seperately

(c) 1-to-0 bit flip inside the data pointer allows the attacker to pivot the pointer towards other data structures.

(d) 0-to-1 bit flip inside the size field allows the attacker to increase the size to cover the metadata of the following object.

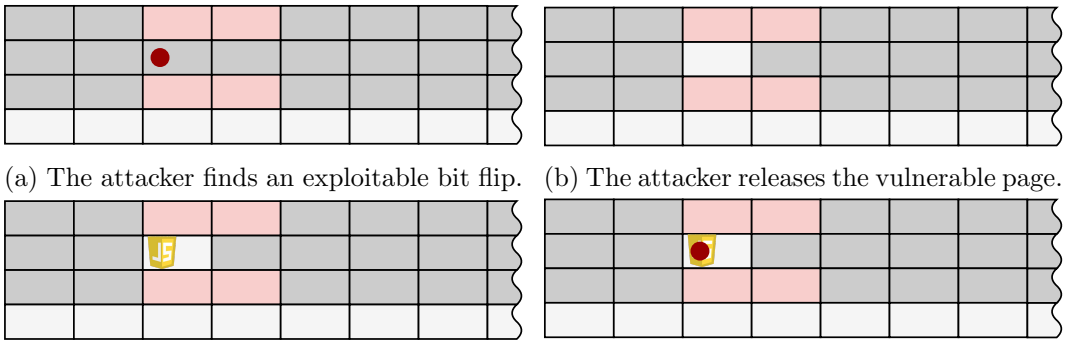
Figure 5.2: Heap layouts and exploitation.

versions these two parts of the objects are separated hardening their exploitation (Figure 5.1b).

Bosman et al. [24] exploited these objects in their Dedup est Machina attack to craft a fake object within the `ArrayBuffer` over which they had full control and later performed pointer pivoting (i.e., changing the value of a pointer to point to your fake object) by exploiting a Rowhammer bit flip, in order to gain access to that object. This technique, however, relies on the assumption that the attacker is in possession of an information leak to break ASLR and craft the fake object. We decided to avoid this step to demonstrate that it is possible to build the same primitive in a zero-knowledge scenario.

While we mentioned that all modern browsers currently separate `ArrayBuffer`'s metadata and data, SpiderMonkey (i.e., Firefox JavaScript engine) still enforces the inline design for `ArrayBuffer` objects that are initialized with size smaller than 96 bytes. When the size of these objects gets increased over this threshold the objects get relocated and split between metadata and data. However, if we manage to gain such overflow by exploiting our Rowhammer bit flips we can bypass these checks and avoid to relocate the data buffer. The metadata of these objects is of 48 bytes and it contains two possible targets for our bit flips. We can either (i) directly hammer the size field triggering a 0-to-1 bit flip (Figure 5.1d), or we can (ii) trigger a 1-to-0 bit flip on the least significant bits of the data pointer (Figure 5.1c).

Exploitable bit flips: Now that we know the objects we want to hammer we need to understand how many bit flips are exploitable within a page. SpideMonkey, allocates these buffers in 3 sizes: 32, 64, 96 bytes. We discovered that each of these sizes gets allocated in separate heaps in memory making it more difficult to obtain effective memory layouts. Nonetheless, we analyzed the number of possible arrangements and we discovered that a bit flip is exploitable in more than 22% of locations within a page. This minimum value takes into consideration, not only the layout of the objects, but also a conservative scenario where just 8 bits are



(a) The attacker finds an exploitable bit flip.

(b) The attacker releases the vulnerable page.

(c) The attacker triggers multiple JavaScript allocations in order to reuse the vulnerable page.

(d) The attacker execute the Shader and trigger the bit flip on the *hammer* target.

Figure 5.4: Bit flip exploitation.

exploitable from the data pointer (i.e., pointer pivoting of at most 256 bytes, hence within the page) and 25 most significant bits of the size field.

5.2.3 Memory reuse

Once we have identified our target exploitable bit, we need to land sensitive data on the vulnerable location. We can accomplish this in two stages. First, we need to free the vulnerable texture. Since the backing OpenGL library adopts some pooling techniques to gain better performances, we need to force this release by releasing multiple objects in order to fill up this pool. We discovered the pool to contain 2048 pages. By releasing this amount of memory we can systematically release our vulnerable page back to the Buddy allocator (Figure 5.3b).

Then we need to trigger the JavaScript engine to request such page. We obtain so by allocating multiple instances of our *hammer* target. This exhausts the current memory reserved by the SpiderMonkey heap and requests newer pages from the Buddy allocator (Figure 5.3c).

5.2.4 Data corruption

Now that our `ArrayBuffer` is landed on the vulnerable page we need to spawn again our shader to the GPU in order to trigger the bit flip on the `ArrayBuffer`'s metadata (Figure 5.3d). Depending on the field we are trying to hammer (i.e., (i) size or (ii) data pointer) we then gain access either to the consecutive object (i) or to another object stored at 2^n bytes away, where n is the bit position we manged to flip.

We can finally identify the compromised object by iterating over the array of allocated `ArrayBuffer` objects and checking the `byteLength` in case (i) or the content of the buffer in case (ii). Now we have control over the metadata of another

object allowing us to corrupt this gaining control over the data pointer and the size field, hence gaining an arbitrary read/write primitive.

5.3 Mitigations

Now that we have demonstrated how we can compromise a mobile browser sandbox using only microarchitectural attacks, in this section we discuss possible mitigations against GPU-accelerated attacks. We divide the discussion in two parts: 1. defending against side-channel attacks, and 2. possible solutions against browser-based Rowhammer attacks.

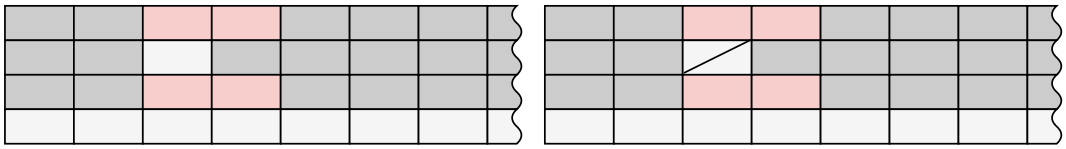
5.3.1 Timing side channels

To protect the system against side-channel attacks, currently the only practical solution in the browser is disabling all the 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 4.2.

First, we recommend disabling the explicit timers provided by the WebGL extension `EXT_DISJOINT_TIMER_QUERY`. If disabling them does not represent a viable option, we at least suggest reducing their granularity similar to what browser vendors have done with `performance.now()` [33, 34, 35, 36]. As mentioned in Section 4.2, Chrome already limits the precision of the `TIMESTAMP_EXT` timer to $1\mu s$. While this does not stop an attacker from building the side-channel, it at leasts reduces the bandwidth of the channel making it more difficult to exploit it in real world scenarios.

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 – or transmit in the case of covert channels – 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 [37] and DeterFox [38]. WebGL functions that directly expose sensitive information to the JavaScript context such as `getSyncParameter()`, if they cannot be disabled, should be at least coarsened to report completion only after a certain amount of time has passed, similar to `performance.now()`.

We are currently following responsible disclosure and discussing these mitigations with major browser vendors.



(a) The attacker releases the vulnerable page.

(b) The page released from the WebGL context gets tagged and cannot be reused until the context is active.

Figure 5.6: Page tagging mechanism to enforce conditional reuse.

5.3.2 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 allow efficient detection of Rowhammer events as shown by previous work [40]. 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 in the browser.

Page Tagging: As discussed in Section 4.4, to exploit Rowhammer bit flips, an attacker needs to ensure that the victim rows are reused to store sensitive data (e.g., `ArrayBuffer`). 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 G-CATT [39] 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 (Figure 5.6). By isolating pages that are tagged by an active WebGL context using guard rows [39], 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.

Chapter 6

Conclusion

Research on microarchitectural attacks has shown their applicability in different scenarios and with different purposes. Research on side-channels has shown how these attacks can be used to compromise cryptographic algorithms [9, 13], break ASLR [15, 45] and even reverse engineer systems' microarchitectures [16, 53], while research on Rowhammer has shown the possibility of compromising both cloud VMs [18, 31] and browsers [23, 24], and also obtain privilege escalation [17, 32]. While these attacks cover a broad spectrum of goals, they all rely on the common implicit assumption of a CPU as exploitation vector. As a consequence, all the state-of-the-art defenses proposed in recent studies [37, 38, 39, 40] try to address these issues placing only CPUs in the scope of their threat model.

The goal of our research was to question the validity of this assumption by trying to build microarchitectural attacks on commodity platform employing a different exploitation vector: integrated GPUs.
At the beginning of our study we posed the following research question:

What means does the GPU provide to an attacker for building microarchitectural attacks? And how do currently proposed defenses against traditional implementations cope with this new exploitation vector?

We now discuss our findings keeping the structural separation between side-channels and Rowhammer attacks that we enforced throughout the manuscript. We then conclude with some final remarks.

6.1 Side-channel attacks

One of the goals of this research was to identify how an integrated GPU aids the process of building side-channel attacks. During our analysis in Chapter 4 we defined "*timers*" as our necessary primitive for building timing side-channels. We showed how the GPU directly provides an attacker with explicit timing sources and further aids the process of crafting new homebrewed timers — allowing attackers to bypass state-of-the-art defenses from both industry and academia. Moreover, the very high

precision of the resulting timers can help crafting fast and reliable side-channel attacks, minimizing the impact of noise. To substantiate this claim, we reported on the first instance of a remote GPU-based side-channel attack that relies on (i) the deterministic behavior of the GPU caches and (ii) the predictable nature of the Buddy allocator to detect contiguous memory allocations (*P2*).

By presenting multiple examples of WebGL-based timers we aimed at demonstrating 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. Furthermore, the intent of crafting homebrewed timers based on the WebGL2 API was this of avoiding criticism resulting from the insecure nature of the `EXT_DISJOINT_TIMER_QUERY` WebGL extension. In fact, while this extension, as already proposed for the timing-prone browser extension `SharedArrayBuffer` [38], can be argued to be easily amendable by simply disabling it, WebGL, and its revision WebGL2, are deeply embedded into many Web applications and disabling them is hardly a pain-free option.

We strongly believe that as long as the JavaScript context will synchronously interact with external contexts such as WebWorkers [21], WebGL and potentially others (e.g., audio), a diligent attacker will always be able to craft new timing sources and our homebrewed timers are a demonstration of this. By targeting state-of-the-art defenses (i..e, Fuzzyfox and DeterFox) we further wish to demonstrate a more structural redesign of modern browser is required in order to implement security-by-design in parallel with the main focus on performances.

Finally, while the aim of our side-channel was to obtain primitive *P2* in order to reuse it to build effective Rowhammer attacks, we believe that other remote side-channel attacks through the GPU may be possible given that they can potentially share more resources with the CPU cores [48]. We recommend this as a possible topic for future studies. We hope with this work to spur further research in this area to highlight the security implications of side-channel attacks from integrated GPUs.

6.2 Rowhammer

Van der Veen et al. [32] reported eviction-based Rowhammer on ARMv7 to be too slow to trigger bit flips from the CPU. We confirmed their findings with the experiment we conducted in Section 4.4. With our GPU-accelerated attack we demonstrate how, by shifting the attack paradigms and employing an integrated GPU as exploitation vector, we are able to overcome current limitations obtaining the first eviction-based Rowhammer bit flip on this architecture. More dramatic is the fact that the WebGL API allows us to trigger these bit flips from a website without experiencing any complication, nor performance issue (i.e., difference in number of bit flips), compared to a native environment.

With 26 s to the first bit flip and 23.7 $flips/min$ following the prime, we establish a powerful primitive for remote mobile Rowhammer attacks, demonstrating how

dangerously erroneous is the assumption that remote Rowhammer attacks are not practical [39], hence they do not need to be addressed.

Our GPU-accelerated Rowhammer attack serves as a proof of how we need to extend our threat model to cover exploitation vectors other than CPUs, not only in the context of side-channels, but also when building Rowhammer attacks. This issue needs to be addressed in future studies. We believe that the best solution to thwart this threat resides in software-based mitigations that rely on performance counter such as ANVIL [40]. However, since these resources are currently not available for these processors, we aim at investigating possible software-based defenses targeting these types of attacks in our future work.

6.3 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. Our exploit represents the first instance of a Web-based microarchitectural attack on ARM platforms and aims at demonstrating how browsers' GPU acceleration, while brining faster graphics rendering, also paves the way to more powerful microarchitectural attacks than possible before. While current research addressing heterogeneous system architecture security merely suggests memory access control [6], we show how these policies already enforced in different platforms, such as our test system Nexus 5, do not protect the users from advanced microarchitectural attacks. While we have plans for mitigations against these attack and we are currently following responsible disclosure to discuss these with browser vendors, we hope our efforts will make processor vendors more careful when embedding the next specialized unit into our commodity processors.

Bibliography

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

[2] 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," ASPLOS'10.

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

[4] 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: 13-09-2017.

[5] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 50–53, 2016.

[6] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 470–481, ACM, 2015.

[7] G. Key, "ATX Part 2: Intel G33 Performance Review." `https://www.anandtech.com/show/2339/23`. Accessed: 13-09-2017.

[8] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

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

[10] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers Track at the RSA Conference*, pp. 1–20, Springer, 2006.

[11] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*, vol. 31. Springer Science & Business Media, 2008.

[12] 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 *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 361–372, IEEE Press, 2014.

[13] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.," in *USENIX Security Symposium*, pp. 719–732, 2014.

[14] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, Springer, 2016.

[15] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 368–379, ACM, 2016.

[16] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.," in *USENIX Security Symposium*, pp. 565–581, 2016.

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

[18] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack.," in *USENIX Security Symposium*, pp. 1–18, 2016.

[19] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1406–1418, ACM, 2015.

[20] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *Proceedings of the 21th International Conference on Financial Cryptography and Data Security (FC17)*, p. 11, 2017.

[21] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," *NDSS (Feb. 2017)*, 2017.

[22] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *European Symposium on Research in Computer Security*, pp. 191–209, Springer, 2017.

[23] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 300–321, Springer, 2016.

[24] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 987–1004, IEEE, 2016.

[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches.," in *USENIX Security Symposium*, pp. 897–912, 2015.

[26] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[27] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "Cain: silently breaking aslr in the cloud," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[28] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 380–392, ACM, 2016.

[29] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," 2017.

[30] N. Timmers and C. Mune, "Escalating Privileges in Linux using Voltage Fault Injection," 2017.

[31] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation.," in *USENIX Security Symposium*, pp. 19–35, 2016.

[32] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1675–1689, ACM, 2016.

[33] B. Zbarsky, "Clamp the resolution of performance.now() calls to 5us." `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`. Accessed: 16-08-2017.

[34] Chromium, "window.performance.now does not support sub-millisecond precision on windows." https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110. Accessed: 16-08-2017.

[35] A. Christensen, "Reduce resolution of performance.now." https://bugs.webkit.org/show_bug.cgi?id=146531. Accessed: 16-08-2017.

[36] M. Perry, "Bug: Reduce precision of time for JavaScript." https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517. Accessed: 16-08-2017.

[37] D. Kohlbrenner and H. Shacham, "Trusted Browsers for Uncertain Times.," in *USENIX Security Symposium*, pp. 463–480, 2016.

[38] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic Browser," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, ACM, 2017.

[39] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Practical and generic software-only defenses against rowhammer attacks," *arXiv preprint arXiv:1611.08396*, 2016.

[40] 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 *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[41] A. Seznec, "A case for two-way skewed-associative caches," in *ACM SIGARCH Computer Architecture News*, vol. 21, pp. 169–178, ACM, 1993.

[42] K. Pagiamtzis and A. Sheikholeslami, "Using cache to reduce power in content-addressable memories (cams)," in *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pp. 369–372, IEEE, 2005.

[43] 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: 16-08-2017.

[44] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 48–65, Springer, 2015.

[45] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP'13, 2013.

[46] S. M., "L3 cache mapping on Sandy Bridge CPUs." http://lackingrhoticity.blogspot.nl/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html. Accessed: 16-08-2017.

[47] "WebGL current support." http://caniuse.com/#feat=webgl. Accessed: 16-08-2017.

[48] I. Corporation, "The Compute Architecture of Intel Processor Graphics Gen9." https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf. Accessed: 18-08-2017.

[49] L.-Y. Wei, "Tile-based texture mapping on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 55–63, ACM, 2004.

[50] K. Group, "OpenGL ES Shading Language version 1.00." https://www.khronos.org/files/opengles_shading_language.pdf. Accessed: 18-08-2017.

[51] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 394–405, IEEE, 2016.

[52] Z. H. Jiang, Y. Fei, and D. Kaeli, "A Novel Side-Channel Timing Attack on GPUs," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pp. 167–172, ACM, 2017.

[53] H. Naghibijouybari, K. Khasawneh, and N. Abu-Ghazaleh, "Constructing and Characterizing Covert Channels on GPGPUs," pp. 22–25, 2017.

[54] N. Corporation, "Nvidia perfkit." https://developer.nvidia.com/nvidia-perfkit. Accessed: 13-09-2017.

[55] D. Gruss, "Software-based microarchitectural attacks," *arXiv preprint arXiv:1706.05973*, 2017.

[56] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: flipping secret exponent bits using timing analysis," in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 602–624, Springer, 2016.

[57] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 490–505, IEEE, 2011.

[58] J. Corbet, "Defending against Rowhammer in the kernel.." https://lwn.net/Articles/704920/. Accessed: 13-09-2017.

[59] V. Shimanskiy, "EXT_disjoint_timer_query." https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_disjoint_timer_query.txt. Accessed: 18-08-2017.

[60] I. Ewell, "Disable timestamps in WebGL.." https://codereview.chromium.org/1800383002. Accessed: 13-09-2017.

[61] 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: 18-08-2017.

[62] M. Gorman, "Chapter 6: Physical Page Allocation." https://www.kernel.org/doc/gorman/html/understand/understand009.html. Accessed: 13-09-2017.

[63] argp, "OR'LYEH? The Shadow over Firefox." http://phrack.org/issues/69/14.html#article. Accessed: 13-09-2017.