Manual and Work Breakdown for GLitch - (version 2.0)

Hardware Security 2018

Introduction:

GLitch is a GPU-accelerated Rowhammer attack that exploits RH bit flips to escape the Firefox JavaScript VM on Android platforms. To build this attack the required primitive is to be able to trigger bit flips from the GPU. In this project you will start by getting acquainted with the dreaded OpenGL library on a native environment and eventually port these bit flips to JavaScript through WebGL.
At the end of these 4 weeks you will be able to trigger bit flips from JS to break ASLR, and who knows maybe you will impress us during demo day with an end-to-end exploit 🙃**.**
Not only, on top of learning how to trigger these bit flips, you will learn the necessary knowledge to port such attack to other mobile platforms.

This projects is divided in 4 blocks:
- **(Week 1) GLitch toolchain**. you will get familiar with OpenGL and you will build the necessary tooling for the next steps of the project.
- **(Week 2) Understand the GPU architecture**. Here you will play around with the GPU performance counters that you will later use to detect your memory accesses.
- **(Week 3) Native bit flips**. Once you have all the tooling ready and you understand the GPU you will build your hammering OpenGL shader that will trigger the bit flips for you from a native environment.
- **(Week 4) Porting to JS**. After you have everything working in a native environment you will need to port this to JS and use these bit flips to break ASLR on the browser. ***IF*** everything goes well this should be easier than it sounds

Week 1 - GLitch Toolchain (Deadline 20.11.2018, 23:59)

The first week is a week of preliminaries. By the end of the week you will have your working GLitch toolchain that you will need to extract infos from the textures in the next weeks.
The goals for this week are mainly 2:

1) Get to learn OpenGL.
2) Extract infos about the textures you allocate from the kernel.

Implicit requirements of the week are: (i) read (carefully) the paper, and (ii) install the Android NDK on your pc to build native Android applications. Two links that can be useful to set this up are getting started and building command line apps.

OpenGL Debugger:

This task has the purpose of getting you familiar with the necessary OpenGL functionalities while building some tools that you can later reuse to debug your OpenGL programs. This has the purpose of allowing you to snoop into the execution of the shader and, most importantly, to check the content of the textures you will be reading from the shader (this will be required to scan for bit flips). For this reason your OpenGL program needs to extract data from a texture and write it to the framebuffer.
In this program you will use all the fundamental building blocks required later on to trigger bit flips and understand the GPU caches (i.e., textures and uniforms). As a consequence it is important that you fully understand how this works. Here you can get some code to setup your egl environment,

The complete program needs to do the following:

1. Allocate a new Framebuffer backed by a 2D texture (you need this to properly export the data from the shader).
2. Allocate a 2D texture of size 32x32 (i.e., 4KB == 1 page). *(For the next task of this week allocate multiple textures)
3. Fill up a texture with a marker value (e.g., `0x41414141`).
4. Set the texel at (**rndX**, **rndY**) coordinates with (**rndX+rndY**), where **rndX** and **rndY** are random `uint32_t` values generated at startup (check the `glTexSubImage2D()` function).
5. Pass the texture to the shader using a uniform
6. Pass these coordinates to the shader using a uniform.
7. Run the GPU program (i.e., two shaders) that reads the texel at (**rndX**, **rndY**) from the active texture and writes it to the framebuffer (check the `texelFetch()` function)
8. Read the framebuffer to extract the value sampled from the texture (check the `glReadPixels()` function).

Warnings:

Before running OpenGL programs you need to set up the EGL context. EGL acts as bridge between the OpenGL API and the underlying Android windowing system. This function is uploaded in the eglSetup.c file on Canvas. You can just plug it in your program.

Every texture you allocate needs to enable the following filters to allow you to control memory read/writes from the shaders.

GL_TEXTURE_WRAP_S	GL_CLAMP_TO_EDGE
GL_TEXTURE_WRAP_T	GL_CLAMP_TO_EDGE
GL_TEXTURE_MIN_FILTER	GL_NEAREST
GL_TEXTURE_MAG_FILTER	GL_NEAREST

Material:

Here is a list of some useful material to start working with OpenGL:

- Hello Triangle: OpenGL 101 explaining the basics of the library
- Shader tutorials: OpenGL and WebGL2. The WebGL2 version uses the same OpenGL Shading Language of OpenGL ES 3 (i.e., GLSL300 ES) therefore it might be more accurate.
- Textures: We suggest again the WebGL2 shader tutorial which provides you with the necessary concepts of texture sampling.
- OpenGL ES 3 reference page
- OpenGL ES Shading Language 3

Grading:

(*10 points*) You need to provide us with a screenshot of a run of your program showing that you are able to retrieve the data that you passed as input to the shaders (i.e., **rndX+rndY**). Furthermore, you need to upload the source code of your program to Canvas along with Makefile(s) – see this for makefiles.

Kernel infos:

As described in the paper we do not have the possibility to directly allocate contiguous memory from the GPU. Therefore, you need to allocate a great amount of textures to deplete buddy chunks of order < 6 and start using allocations of higher orders – usually in the order of hundreds of MBs.

In the paper, we use a timing side-channel to detect these contiguous allocations. We do not ask you to do so due to time constraints. For this reason we provide you with a pre-rooted phone running a custom kernel. This solves two challenges: (i) you need to know the virtual address of the textures you will use for hammering since they're allocated and handled by the KGSL driver; (ii) you need access to `/proc/{pid}/pagemap` to figure out the mapping between virtual and physical addresses to detect contiguous memory.

You need to build an interface that allows you to combine the information from the `debugfs` entry of the KGSL driver with the `/proc/{pid}/pagemap` file in order to be able to detect contiguous memory regions. You can modify the code from the previous task to allocate multiple textures to verify you are actually able to detect contiguous memory.

The KGSL file allows you to extract the virtual address of GPU allocations. This file has 8 attributes (`<gpuaddr, useraddr, size, id, flags, type, usage, sglen>`) you are interested in 3 of them, namely `useraddr`, `id` and `usage`. Your interface should do the following:

1. Use the `usage` attribute to filter texture allocations which are the ones you are interested in.

2. Match these virtual addresses with your OpenGL textures using the `id` field — `id`*s* grow incrementally therefore they follow the index of your texture array.
3. Recover the PFNs of the allocated textures. Which means you need to use the `useraddr` to query the `/proc/{pid}/pagemap` file and extract the PFN.
4. Use the data you just extracted from `pagemap` to detect contiguous allocations by identifying incremental PFNs of subsequent textures allocations.
5. Identify group of textures allocated from chunks of order ≥ 6.
6. For each of these groups generate a list containing the following infos for each textures:
    - `group_id` (this is only needed for evaluation I need it to keep track of different groups)
    - `id` (this is needed to identify the index in your textures array)
    - `useraddr`
    - `pfn`
    - `alloc_order` (buddy allocation order)

For evaluation purposes, you should dump the `pagemap` and `kgsl` files for the current process along with the exported arrays — export this data in csv including the header following the naming convention reported above.

Grading:

You need to upload to canvas:

- (10 *points*) The code used to export the list of contiguous textures. Plus the dumps of a run of your code containing pagemap, kgsl and result. We will check your code and evaluate the test run to grade this part of the assignment.

Week 2 - Understanding the GPU (Deadline 27.11.2018, 23:59)

This week you will learn about the GPU architecture and you will build a secondary part of the GLitch toolchain to extract performance counters measurements to later understand the behaviour of your access pattern when hammering the memory.

Performance counters:

This week we ask you to identify the necessary performance counters to reverse engineer the GPU caches. These need to measure L1 hits and misses as well as UCHE hits and misses[1] — hint the VBIF is the memory controller. Performance counters are accessible directly from your OpenGL program via the `GL_AMD_performance_monitor` extension. The extension' functions are defined in the `gl2ext.h` file. However, you need to dynamically load the extension's functions using `eglGetProcAddress()`.

To pinpoint the correct counters we provide you with a list of all the available ones for the Adreno 330 GPU[2] and a detailed diagram of its architecture in Figure 1. This diagram is accompanied by a Legend of the acronyms to aid your analysis. In order to validate your hypothesis about the chosen counters, you should create a shader that sequentially accesses memory in order to identify L1 and UCHE misses. By playing around with the amount of these accesses you can understand if the counters you chose are correct.

Once you identify the correct performance counters you're required to build a plot similar to Figure 3a in the paper reporting the size of L1. To do so you simply need to iterate twice over the same access pattern while increasing the maximum value as shown in Listing 1 of the paper. However, remember you're using OpenGL ES 3 and not 2.

Tips & Warnings:

1) The GLSL compiler tries to optimize the shaders. Therefore, dead code gets removed from the shader. Which means that unused reads may be skipped at compile time and therefore may not be performed by the GPU. Try to make sure your code gets actually executed by adding dependencies on the different loads.
2) You don't care about the number of polygons you draw. As a consequence, from now on you can always call `glDrawArrays( GL_POINTS, 0, 1)` to execute your shaders.

Grading:

(*10 points*) You are required to provide us with the list of performance counters necessary to detect L1 hits and misses and UCHE hits and misses. You should provide us with an explanation of your choices.
(*10 points*) You're required to upload the code used to build the plots for the size of L1 along with the plot itself.

[1] Note that these might not be explicit and might require the combination of multiple counters.
[2] Credits to the Freedreno community for providing this list. You can extract it yourself by using the `GL_AMD_performance_monitor` extension.

Legend:

ARB	arbitrators	RBBM	register backbone management
AXI	advanced extensible interface	SP	shader processor
CP	command processor	TP	texture processor
GRAS	graphics rasterizer	TSE	triangle setup engine
HLSQ	high level sequencer	UCHE	unified cache
MARB	memory arbitration block	VBIF	memory interface block
PC	primitive controller	VFD	vertex fetch decoder
PWR	power	VPC	vertex parameter cache
RAS	rasterizer	VSC	visibility stream composer
RB	render backend		

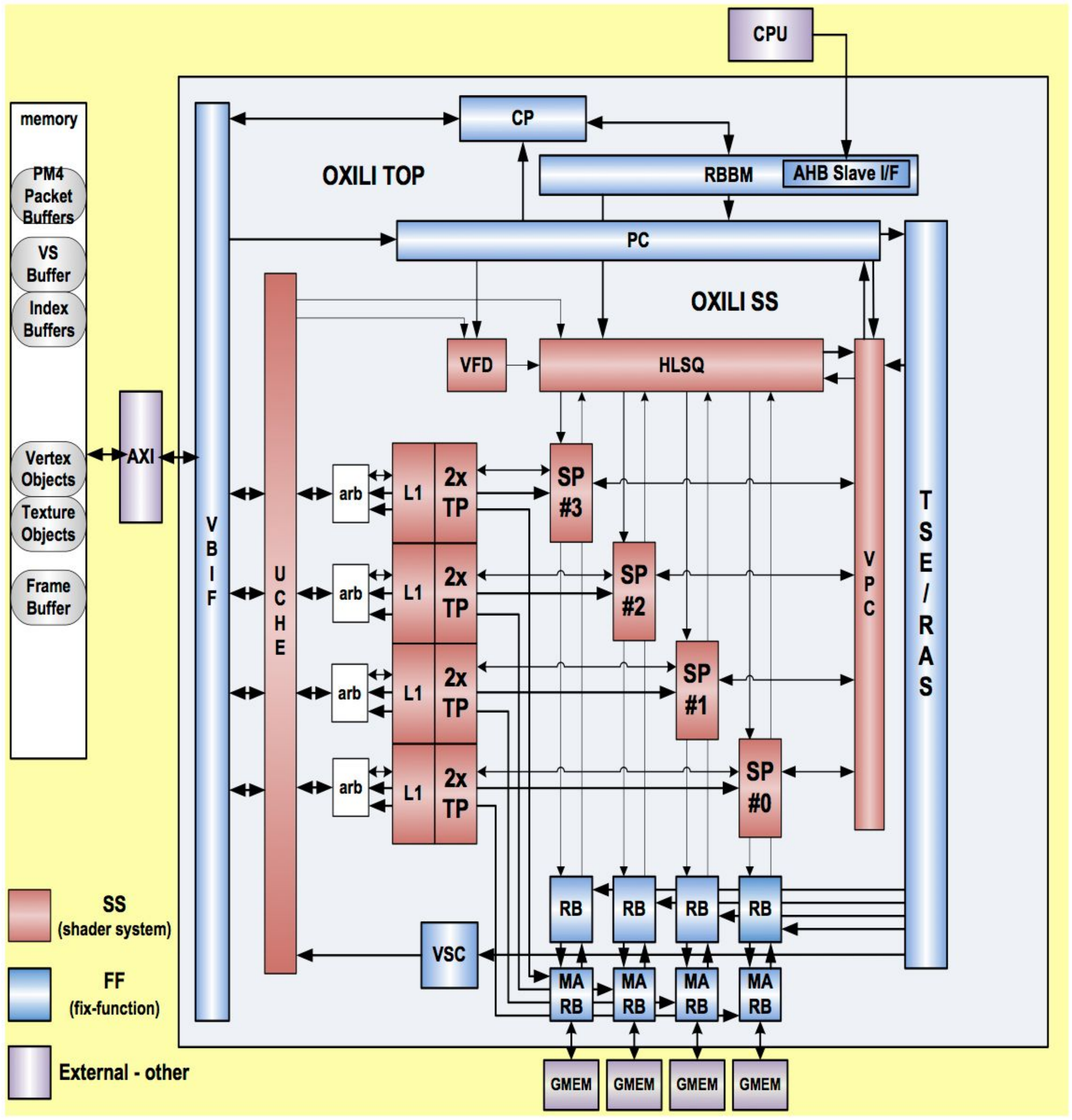

Figure 1. Adreno 330 GPU architecture. Do not share this diagram

Week 3 - Native Bit Flips (Deadline 04.12.2018, 23:59)

From this week we start working on the actual code you will use to obtain your first GPU-accelerated bit flip. We split this week work in two tasks that represents the two fundamental requirements to trigger bit flips: (i) fast memory access and (ii) knowledge of physical location.
We strongly suggest you to carefully use the information about the Snapdragon 800/801 DRAM mapping provided in the Appendix of the paper.

Hammer-pattern

ASSUMPTION: For this task you have to theorize an efficient access pattern that allows you to trigger bit flips while evicting the caches. You build this access pattern under the assumption of dealing with contiguous memory that allows you to carry out double-sided rowhammer (i.e., at least 3 contiguous rows). You will eventually rely on your GLitch toolchain to understand which textures' allocations are backed by contiguous memory.

The hammer-pattern is a twofold access pattern. This pattern has the purpose of evicting the two-level caches while in the meantime performing accesses that can be used to trigger bit flips. While describing this pattern we will frequently use the terms *hammering*-access and *idle*-access. These are respectively accesses performed with the purpose of triggering bit flips (*hammering*-access) and accesses performed with the purpose of simply filling up a cache set (*idle*-access) to evict hammering addresses. Since you are trying to carry out double-sided Rowhammer you want to access rows *n-1* and *n+1* to induce bit flips in row *n*. Every access to these two rows is an *hammering*-access. All the other accesses, are *idle*-accesses that they do not aid the hammering process.
Minimizing the access time between two *hammering*-accesses is paramount to maximize the number of bit flips. As a consequence, you need to identify an efficient access pattern that does so while in the meantime evicting the caches to allow new accesses to rows *n-1* and *n+1*.

Tips:

1) You need to beware of the stride of memory addresses mapping to the same cache set. By knowing this and the size of a DRAM row you can identify how many memory addresses you can map from a specific row.
2) Take into account the difference between *row hits* and *row conflicts* when choosing the addresses you want to add to your *hammer*-pattern. This will increase the number of bit flips.
3) Think in terms of addresses and offsets instead of textures. 4KB textures make it easy to convert from one to the other.

Grading:

(*8 points*) You need to provide us with an explanation of your hammer-pattern. This needs to contain an array with the relative offsets of your memory accesses (e.g. [0KB, 64KB, 36KB, …, 8KB]) accompanied by a brief explanation about your choice. You can use a diagram if you think this can help your description.

Trigger bit flips

Now that you have your ideal *hammer*-pattern in mind you can use it to trigger bit flips.
The hammering process follow these steps:

1. Deplete (*partially*) the memory
2. Identify the textures backed by contiguous allocations (KGSL-pagemap interface)
3. Fill the textures with a pattern that aids bit flips
4. Hammer the contiguous areas
5. Look for bit flips

We now expand on these key points.
For steps (1) and (2) you should use your GLitch toolchain to extract these infos. You (1) need to allocate multiple textures to exhaust the allocations coming from order < 6 and start using allocation of order ≥ 6 and then (2) use your KGSL-pagemap interface to extract the textures backed by these contiguous allocations.

Step (3) is a key component to trigger bit flips. Rowhammer appears to be highly data dependent. As a consequence, if you aim to trigger a 1-to-0 bit flips you want to fill the two rows *n-1* and *n+1* with 0s to have a stronger influence on the charges of the hammered row (i.e., row *n*).
This means that you need to fill all the textures residing in these rows with 0s while filling row *n* with 1s (i.e., 0xff).

Then you can finally start hammering.
The *hammering* shader is the core of the whole project. This shader is the responsible for triggering bit flips by carrying out your *hammer*-pattern. The number of accesses dramatically influence the number of bit flips you may trigger. Therefore, you will need to iterate multiple times over this access pattern. We empirically discovered 2 millions to be a good number of hammering accesses.

Finally, after running your *hammering* shader you need to read back the content of the texture you just hammered to look for bit flips. To do so, you need to use the `glReadPixels()` function. However, before doing that you need to bind the the hammered texture to the framebuffer (see `glFramebufferTexture2D()`). Then you can finally check the output for bytes different from `0xff`.

Tips:

1) While reads are performed at row level within a bank, you may want to save some time when filling the textures and fill the whole *Row* spanning over all the banks. That is you can fill every 64KB-aligned chunks from where the contiguous memory starts.

2) Order 6 allocation spans over 4 rows while double-sided Rowhammer requires only 3 rows. This means that for every order 6 allocation you can hammer 2 rows (i.e., row 1 and row 2 on all the 8 banks). Therefore, you will need to move your start offset of 64 KB after hammering and checking the memory for bit flips. Which also means that you will need to refill your textures.

Grading:

(*12 points*) You need to upload the source code of your final program and show us a proof of a triggered bit flip with a screenshot of your program.

Week 4 - Port to JavaScript (Deadline 14.12.2018, 17:00 - **hard**)

JavaScript bit flips

As a first step you need to port your bit flips to JavaScript. WebGL2 is the Web version of OpenGL ES 3. As a result the two libraries are extremely similar. This means that your hammer-shader can be used "as is" in the browser. It also means that most of the function calls you currently have in your native code simply need to be converted into a more high-level and object oriented programming language (e.g., glBindTexture(...) == gl.bindTexture(...)).

Since you can't request Kernel information from JS you will need to set up a small web server that will communicate with the kernel on your behalf. To do this we suggest you to build a Python server running on the client side (i.e., DAS-4) which when requested from your JS code will read through adb the KGSL and pagemap files. You can extract the contiguous chunk on the web server side. You're not requested to perform these computation in JS. As we already mentioned this is a simple way to get the information you will retrieve through the timing side channel.
Since the browser is a more complex program than the command line application you used so far, you may want to extract the status of the KGSL allocation right before allocating your textures to filter out the already existing ones when looking for contiguous memory. This helps when matching your array index with the allocation `id` retrieved from KGSL.

The suggested scheduling should be:

- Load page
- GET request to extract ground truth
- Answer callback to allocate textures
- GET request to look for contiguous memory
- Answer callback providing the contiguous chunks
- Start hammering from the start of these chunks.

Exploitation

Once you have your bit flips you need to reuse them for the exploit. To do this there are two fundamental steps required: (1) you need to know if your bit flip is exploitable, (2) you need to release the exploitable page in order to place your target in the same memory location.

1) The first part relies on the type-flipping primitive we explain on the paper. You need to make sure that the bit flip you identified appears in the 25 MSBs of the 64-bit double – you can read the GLitch web page for more details about this.

2) This part is clearly probabilistic. You won't be 100% sure the memory is going to get reused by your objects. However, it is very likely if you allocate enough of these.

In order to release the vulnerable texture you're actually required to first deplete an allocation pool (internal to the KGSL driver) which we discovered to be of 2048 pages. This means that before you release your vulnerable texture(s) you should release 2048 other textures. Be careful, **DO NOT** release the textures on row *n-1* and *n+1* that you need for hammering.
Afterwards you can trigger the allocation of lots of JS arrays which you need to fill with object pointers (e.g., new ArrayBuffer(0x30) or "hello world").
Eventually simply trigger the bit flip and hope your allocation was in the right place:)

Grading:

You should upload the source code of your final program along with a picture of a bit flip and a picture (or a short video) reporting the pointer leak.
(*10 points*) You need to reproduce your bit flips from JavaScript
(*10 points*) You should be able to leak a pointer using these bit flips

Demo

You should use the extra few days in the last week to tie the loose ends from the previous weeks, add some reliability to your attack, and prepare a demo to show how your attack works to the class. You can record a video or make a live demo and if you have time, why not try to build the full GLitch exploit?:)
This choice is completely up to you.

[**IMPORTANT!**] Demoing your attackis not mandatory, but it is a chance to impress everybody with your work and become eligible for the best demo award given on the last Friday of the course at 17:00 in S111. We will hold a borrel on the same day to celebrate the end of the course! Please do join either way, if you are presenting a demo or not!