GPU.zip: On the Side-Channel Implications of Hardware-Based Graphical Data Compression

Yingchen Wang*, Riccardo Paccagnella†, Zhao Gang*, Willy R. Vasquez*,
David Kohlbrenner‡, Hovav Shacham*, Christopher W. Fletcher§

*University of Texas at Austin, †Carnegie Mellon University,
‡University of Washington, §University of Illinois Urbana-Champaign

***Abstract*—Compression is a widely-deployed optimization that reduces data movement throughout modern computing stacks. Unfortunately, it is also a well-known source of side-channel leakage capable of leaking (potentially) fine-grained functions of the underlying data. There has, however, been a saving grace. Compression is typically software visible. Thus, software can "opt out" of harm's way by disabling compression when sensitive data is involved, and tailor mitigations to known, public compression algorithms.**

This paper challenges the above conventional wisdom by demonstrating the existence of, and exploiting, *software-transparent* uses of compression. Specifically, we find that integrated GPUs from Intel and AMD vendors compress graphical data in vendor-specific and undocumented ways — even when software does not specifically request compression. Compression induces data-dependent DRAM traffic and cache utilization, which can be measured through side-channel analysis. We show the efficacy of this side channel by performing cross-origin SVG filter pixel stealing attacks through the browser.

1. Introduction

It is well known that moving data is often costlier than computing on said data. Some data patterns occur more often than others, so compression helps address the memory wall problem by eliminating redundancies within data, thereby encoding the same data with fewer bits. Compression has seen significant uptake throughout the stack (e.g., in operating systems [1, pp. 362–64], Internet protocols [2], [3], [4], databases [5] and graphics software [6]).

Unfortunately, besides its well-recognized performance benefits, compression is also a known source of side-channel data leakages. For example, a line of work [7], [8], [9] (starting with Kelsey [10]) has shown that a chosen-input attacker can leak secrets in HTTP/HTTPS requests and responses bit-by-bit by exploiting how compressibility is often secret-dependent [10].

Fortunately, several properties of compression help mitigate leakages. First, the compression is software visible. Thus, software can at least in principle "opt out" of harm's way by disabling compression when sensitive data is present. Moreover, the compression algorithm, detailing the process of eliminating redundancies in data, is publicly available. This enables defenders to focus their attention, mitigations, and analysis on a relatively small set of public compression algorithms (e.g., DEFLATE).

This paper questions the applicability of these properties, in modern systems, by demonstrating that compression of graphical data can be *software-transparent* and *vendor-specific*. A GPU may compress a texture generated by a graphical API even when software does not request compression (e.g., by using a compressed texture format). Further, the underlying compression algorithms used are vendor-specific and undocumented. To confirm the existence of and understand the details of such subtle compression, one has to carefully design experiments, trace software stacks, and find and reverse-engineer a sea of implementations.

The technical core of the paper is a detailed investigation of integrated GPU (iGPU)-based software-transparent lossless compression schemes deployed on Intel and AMD processors. To pinpoint the component responsible for compression, and determine when it is invoked relative to frame rendering, we trace the software stack from OpenGL through its backend Mesa, the Linux kernel, and finally the iGPU. Our analysis reveals that the compression is performed by an iGPU-side component in a software-transparent manner. We then perform an in-depth reverse-engineering of the compression algorithm used by Intel 8th through 11th Gen iGPUs and AMD 5th Gen GCN iGPU, and show that compression algorithms a) are changing across generations and b) vary between vendors.

We demonstrate that an attacker can exploit the iGPU-based compression channel to perform cross-origin pixel stealing attacks in the browser using SVG filters (the latest version of Google Chrome as of April 2023), even though SVG filters are implemented as constant time [11], [12], [13], [14]. The reason is that the attacker can create highly redundant or highly non-redundant patterns depending on a single secret pixel in the browser. As these patterns are processed by the iGPU, their varying degrees of redundancy cause the lossless compression output to depend on the secret pixel. The data-dependent compression output directly translates to data-dependent DRAM traffic and data-dependent cache occupancy. Consequently, we show that, even under the most passive threat model — where an attacker can only observe coarse-grained redundancy information of a pattern using a coarse-grained timer in the browser

and lacks the ability to adaptively select input — individual pixels can be leaked. Our proof-of-concept attack succeeds on a range of devices (including computers, phones) from a variety of hardware vendors with distinct GPU architectures (Intel, AMD, Apple, Nvidia). Surprisingly, our attack also succeeds on discrete GPUs, and we have preliminary results indicating the presence of software-transparent compression on those architectures as well.

In summary, this paper contributes the following.

1) We investigate the security implications of software-transparent, vendor-specific, and undocumented iGPU graphical data compression schemes.
2) We perform a detailed root-cause analysis on how the compression schemes induce data-dependent DRAM traffic and cache footprints. We isolate when and where compression occurs, and reverse-engineer the compression algorithms used in recent Intel and AMD iGPUs.
3) We develop an end-to-end cross-origin pixel stealing attack in the latest version of Google Chrome as of April 2023 that works on multiple hardware platforms.

Disclosure. In March 2023 we disclosed our findings to GPU vendors (AMD, Apple, Arm, Intel, Nvidia, and Qualcomm) and our Chrome attack to Google. The GPU vendors largely declined to act; one said the side channel was outside their threat model, another that it was the responsibility of software to mitigate. As of August 2023, Apple and Google were still deciding whether and how to mitigate.

2. Background

2.1. GPUs, surfaces, and compression

Many GPU operations have two-dimensional pixel buffers, called *surfaces*, as inputs or outputs. For example, texture maps applied to three-dimensional shapes are inputs to rendering; the framebuffer shown on the screen is an output of rendering; and programmatically generated textures are both inputs and outputs.

Graphical operations on surfaces exhibit locality of reference in (x, y)-space, but GPUs have memory hierarchies similar to CPUs, where locality is governed by physical memory address.[1] This observation motivates the use of *tiling*: a sophisticated arrangement of surface pixels in memory that allows efficient random access to arbitrary (x, y) coordinates while exhibiting memory locality along both axes. Different GPUs use different tiling schemes. For example, on Intel GPUs each 64-byte cacheline holds a 4×4 pixel window (Section 4.2), and on AMD GPUs four consecutive cachelines together hold an 8×8 pixel window, each cacheline having a 2×8 pixel subwindow (Section 4.4).

Increasing graphical complexity, screen resolution, and refresh rates mean that, even with the memory locality afforded by tiling, GPUs are constrained by memory bandwidth. Compression is a natural response to memory bandwidth pressure, but general-purpose compression algorithms are incompatible with the random access and locality requirements on surfaces.

GPU programming interfaces have long supported compressed textures as a way of reducing GPU memory bandwidth usage and game graphical asset size. To support locality, texture compression algorithms have a fixed compression ratio as a design parameter. For example, BC7 compresses each 4×4 pixel window to 16 bytes, a $4 : 1$ ratio. (See Pranckevičius [15] for a survey of formats.) As a result, these algorithms are *lossy*: They cannot be reversed to recover the original pixel values.

It is fine for developers to opt in to lossy compression for some assets, but a GPU that applied it to *every* surface would risk unacceptable visual degradation on some workloads, especially those with many intermediate render targets where error can accumulate. To be applied transparently to every surface, compression must be of a special kind: It must preserve locality and be lossless, yet allow low-entropy regions to be described using fewer cachelines than their uncompressed representation. The result is not a reduction in memory footprint — indeed, the opposite, since out-of-band metadata is allocated to identify compressed surface regions — but a reduction in memory bandwidth usage when reading and writing a compressible surface.

Blog posts, whitepapers, developer guides, and tech talks from every major GPU vendor make clear that they have developed and deployed lossless compression schemes of this kind: AMD [16], Apple [17, at 2:51], Arm [18], Imagination [19], Intel [20], Nvidia [21, pp. 12–14], and Qualcomm [22, Section 2.3.5].

Operating systems' graphics software stacks need modifications to enable the use of these lossless compression schemes, for example to allocate a buffer that holds compression metadata, but graphics software stacks do not implement compression or decompression. Indeed, GPU vendors treat their lossless compression schemes as proprietary information. Reverse-engineering and documenting the compression schemes implemented by Intel and AMD GPUs is a major contribution of this paper. We show that Intel GPUs try to represent two cachelines (a 4×8 pixel window) using one cacheline (Section 4.3) and that AMD GPUs try to represent four cachelines (an 8×8 pixel window) using one, two, or three cachelines (Section 4.5).

2.1.1. A note about OpenGL and coordinate systems.

GPUs can be programmed using many APIs. The mechanism we study is not API-specific, so we expect to find the same result regardless of which API we use. For concreteness, we implement our experiments using OpenGL, and adopt OpenGL's terminology to describe them.

In OpenGL, *texture* refers generically to either an input surface or output surface. A *shader* is a user-defined subroutine written in a domain-specific language that executes on the GPU. A vertex shader manipulates polygon vertices, whereas a fragment shader computes the color (and other properties) of the pixels bounded by a polygon's vertices. A *program object* combines shaders to render parts of a frame.

1. Indeed, *integrated* GPUs share DRAM and, in some cases, a last-level cache, with the CPU.

Table 1. SoCs and integrated GPUs tested in our leakage channel experiment.

SoC model	iGPU model	Operating system and kernel	LLC size	Memory module	Display
Intel i7-8700 (Coffee Lake)	Intel UHD 630	Ubuntu 22.04 (Linux 5.15)	12 MB	DIMM DDR4-2666	1920×1080 @ 60 Hz
Intel i5-1135G7 (Tiger Lake)	Intel Iris Xe	Debian 11.6 (Linux 5.10)	8 MB	DIMM DDR4-3200	1920×1080 @ 60 Hz
Intel i7-12700K (Alder Lake)	Intel UHD 770	Ubuntu 22.04 (Linux 5.19)	25 MB	DIMM DDR4-3200	1920×1080 @ 60 Hz
AMD Ryzen 7 4800U (Zen 2)	AMD Radeon RX Vega 8	Ubuntu 20.04 (Linux 5.19)	8 MB	DIMM DDR4-3200	1920×1080 @ 60 Hz

A *pipeline* invokes one or more such program objects to render a frame.

Graphics APIs, including OpenGL, put the origin at the bottom left, with the y-axis pointing up, and tiling formats follow suit, with the first pixel in the first tile in a surface being the one at $(0, 0)$. But GPU documentation illustrates surface tiling formats with the origin at the *top* left, with the y-axis pointing down towards higher memory addresses. In this paper, figures that illustrate GPU memory layouts (Figs. 6, 9, 11, and 13) follow the GPU documentation convention. To understand how they apply to pixels displayed on screen, these figures should be interpreted with their y-axis flipped.

2.2. Browsers, the SOP, and pixel stealing

Pixel stealing attacks arise when one webpage can use a side channel to infer the values of individual pixels that the page does not have access to programmatically.

Generally, a browser is willing to load visual content requested by one page, but belonging to another distrusting page, at the same time in the form of iframes. As part of the same-origin policy (SOP), iframes guarantee that even if the framing page is malicious, it cannot inspect either the source code for the iframed page, or the final visual product of the iframe. Since a page may contain visual data that is considered secret, e.g., usernames, email content, etc., it is important that this safety property is upheld.

However, some web standards allow for the framing page to apply additional visual effects, e.g., CSS filters, to that content. These include dozens of different filters, from complex visual effects (3D lighting) to simple convolutions, many of which are borrowed from the SVG standard. Unfortunately, allowing a framing page to apply semi-arbitrary computation to secret visual content has proven to be fertile ground for side-channel attacks.

Paul Stone's original attack [23] in 2013 took advantage of source-code fast paths in browser SVG filter implementations to induce rendering time differences between black and white pixels and then distinguish between them using that rendering time. This requires the use of an iframe containing a victim page, then specific combinations of CSS and SVG filter constructs to isolate, binarize, and then finally render a timing-variable `feMorphology` filter over a target pixel. This attack approach was mitigated by re-writing filter implementations to remove pixel-value dependent branches and emulate the constant-time cryptographic coding style.

Most, but not all, modern sites that display sensitive content like usernames disallow cross-origin framing via features such as X-Frame-Options and the frame-ancestors Content-Security-Policy (CSP) directive.

3. iGPU Graphical Data Compression Exists

In this section, we provide evidence supporting the existence of software-transparent graphical data compression on the iGPU of multiple processors. This evidence undergirds both the case studies in Section 4 and the cross-origin pixel stealing attack in Section 5. Specifically, we create an iGPU workload that constructs textures with different patterns and uses them as inputs or outputs for iGPU graphical manipulation. We investigate how DRAM traffic and bandwidth vary with the graphical patterns being processed, and how pattern-dependent DRAM traffic is reflected in rendering times and in memory hierarchy contention.

3.1. Experimental setup

The experiments we present in this section are tested on machines with Intel and AMD processors. The characteristics of these machines are summarized in Table 1. Our devices have the latest microcode patches as of April 2023.

Metrics. Our main measurement metrics are DRAM traffic per frame and rendering time per frame. We also use DRAM bandwidth to help explain results.

Rendering time per frame (or simply *rendering time* where unambiguous) is the amount of time it takes for a frame to refresh. A caveat is that modern graphics stacks cap the frame rate at the monitor's refresh rate. This puts a floor on measured rendering time per frame of 16.7 ms in our setup. In many of our experiments, we artificially increase the complexity of the graphical workloads we ask the iGPU to perform so rendering time exceeds this floor.

DRAM traffic per frame is the amount of data transmitted across the DRAM data bus during the rendering of one frame. Since our measurement process samples the amount of data across the DRAM data bus every 1 ms, we use the processor's cycle counter to synchronize the rendering process and the sampling process. Synchronization allows us to identify and aggregate the measurements that belong to the same frame in computing DRAM traffic per frame.

DRAM bandwidth is the amount of data transmitted across the DRAM data bus between consecutive performance counter readings, divided by the time between those consecutive readings. Compared to the DRAM traffic per frame metric, the DRAM bandwidth metric exhibits higher variability. The reason is that GPU memory utilization is not uniform within a frame, and for each frame, we collect only one data point for the DRAM traffic but sample the DRAM bandwidth every 1 ms.

Algorithm 1: OpenGL graphical workload.

```
Input: pattern, iterations
1 for i ← 1 to iterations do
2 |   texture ← GPU-WRITE_pattern();
3 |   framebuffer ← GPU-READ(texture);
4 end
```

Experiment Design. We carefully design our workloads and experiments so that the above metrics are not impacted by known effects described in prior work that are not related to compression [24], [25]. Firstly, we write all workloads to follow constant-time programming principles. Secondly, we sample iGPU frequency when running each experiment to ensure that any pattern-dependent observation of any metric is not due to iGPU frequency variations.

Measurement Tools. On Intel, we follow the IGT GPU tools.[2] We use perf events `uncore_imc/data_reads` and `uncore_imc/data_writes` to measure the amount of data written to, or read from, DRAM, and `i915/actual-frequency` to measure iGPU frequency. Where there is more than one IMC (e.g., 12700K), we sample the associated perf events for all and report the sum. We measure IMC perf events every 1 ms and i915 perf events every 5 ms.

On AMD, we use the `hwmon` interface exposed by the Linux AMD iGPU driver to measure iGPU frequency. To measure traffic on the DRAM data bus, we use the AMD data fabric performance counters [26]. We sample both the data fabric events and the iGPU frequency every 1 ms.

Plotting. Before reporting results, we exclude any data points that are identified as outliers and verified to be caused by non-systematic effects. We then calculate the average and standard deviation. All plots include error bars; a dot apparently alone is due to tight distribution of data.

3.2. Graphical workload

We design a workload that attempts to trigger iGPU lossless compression with an iGPU-created texture. Recall from Section 2.1 that a texture is a data buffer holding 2D pixel data that can be used as the input or output of GPU rendering. In our workload, we study whether the GPU compresses textures losslessly "under the hood" in a software-transparent manner.

Our workload defines two program objects: GPU-READ and GPU-WRITE. GPU-READ includes a trivial fragment shader that copies from an input texture to the program object output. GPU-WRITE includes a more complicated fragment shader that computes one of the four patterns illustrated in Figure 1 depending on a configuration parameter.[3]

2. Online: https://gitlab.freedesktop.org/drm/igt-gpu-tools.

3. This configuration parameter is conveyed to the GPU-WRITE program object in an OpenGL uniform and to the fragment shader in a vertex color.

Table 2. RESULTS OF RUNNING OUR GRAPHICAL WORKLOAD WITH DIFFERENT PATTERNS ON PROCESSORS PRESENTED IN TABLE 1.

SoC (iGPU)	Pattern	DRAM traffic per frame (MB)	Rendering time (ms)
Intel i7-8700 (UHD 630)	BLACK	524.5 ± 8.6	70.5 ± 0.3
	RANDOM	1063.0 ± 11.8	78.1 ± 0.3
	GRADIENT	563.1 ± 11.0	70.7 ± 0.3
	SKEW	1050.6 ± 9.6	78.2 ± 0.3
Intel i5-1135G7 (Iris Xe)	BLACK	451.0 ± 47.0	27.6 ± 0.03
	RANDOM	893.0 ± 83.5	29.4 ± 0.04
	GRADIENT	452.8 ± 46.6	27.6 ± 0.03
	SKEW	453.7 ± 46.5	27.5 ± 0.03
Intel i7-12700K (UHD 770)	BLACK	491.0 ± 25.7	46.3 ± 0.6
	RANDOM	787.3 ± 32.3	47.4 ± 0.7
	GRADIENT	493.7 ± 27.8	46.6 ± 0.4
	SKEW	490.7 ± 26.1	46.6 ± 0.4
AMD Ryzen 7 4800U (Radeon RX Vega 8)	BLACK	9.1 ± 2.7	27.9 ± 0.1
	RANDOM	1214.2 ± 17.7	37.4 ± 0.2
	GRADIENT	314.0 ± 5.6	30.3 ± 0.1
	SKEW	526.4 ± 10.8	32.0 ± 0.1

We construct a 3000 × 3000 pixel 2D texture (large enough to fill the LLC and force DRAM accesses for all devices under test), which we call *texture* below. (We stress that we do *not* ask that this texture have a software-compressed (`GL_COMPRESSED_*`) layout.) We arrange for the GPU-WRITE program object to write to *texture*. We arrange for the GPU-READ program object to read from *texture* and write to the framebuffer.

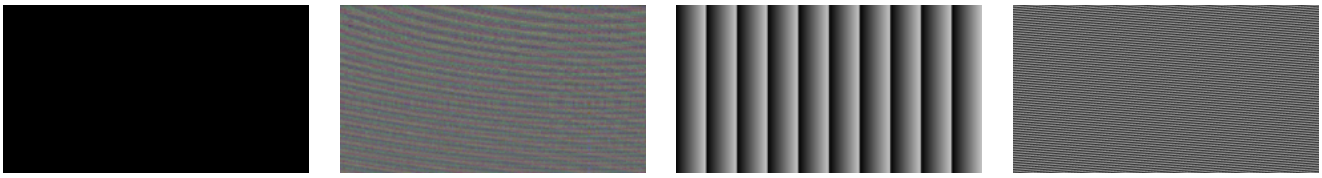
Our workload pseudocode is presented in Algorithm 1. The workload is executed on each frame refresh. The *iterations* parameter controls how many times the workload executes its inner loop. Within each iteration, we execute GPU-WRITE at Line 2 first to create write traffic to DRAM, then GPU-READ at Line 3 to create read traffic from DRAM.

3.3. Evidence for iGPU graphical data compression

We run the workload from Section 3.2 with each of the four patterns in Figure 1 on the systems listed in Table 1, while measuring rendering time per frame and DRAM traffic per frame. We report and interpret the experiment results. We argue that the data are consistent with graphical data loss compression whose implementation varies by platform.

As described in Section 3.2, our workload has a complexity scaling parameter, *iterations*. For this experiment we set *iterations* = 20, which was sufficient to ensure that rendering time exceeded the 16.7 ms floor on all devices under test. We render each pattern for 400 seconds while sampling performance counters as described in Section 3.1. In Table 2, we report the average and standard deviation for rendering time per frame and DRAM traffic (read + write) per frame for each experimental condition.

First, a note about DRAM traffic. A naive implementation of our workload calls for the GPU to write a first texture and then read a second texture for each loop iteration; with parameters as described above and assuming a pixel takes 32 bits, such a naive implementation would transfer 1373 MB per frame. The actual observed DRAM traffic per frame



(a) BLACK (b) RANDOM (c) GRADIENT (d) SKEW

Figure 1. The four patterns with which we instantiate our graphical workloads. The BLACK is all black. Pixels in the RANDOM have their color channels chosen as a pseudorandom function of the pixel location (with fixed seed). The GRADIENT has a repeated grayscale gradient in the x direction. The SKEW like GRADIENT, has a grayscale gradient, but aligned such that the colors in each row are shifted compared with the row above.

is somewhat lower than this upper bound on all platforms, perhaps reflecting savings due to caching.

Importantly, on each device under test, the observed DRAM traffic per frame varies with the pattern rendered; this is consistent with lossless compression. Furthermore, *which* patterns are apparently compressible, and how much, varies by device; this is consistent with different devices implementing different compression algorithms.

In particular, on the Intel i7-8700, the four patterns fall into two categories: BLACK and GRADIENT are apparently compressible whereas RANDOM and SKEW are not. On the Intel i7-12700K and i5-1135G7, only RANDOM is apparently non-compressible. On the AMD Ryzen 7 4800U, the compression algorithm is apparently more sophisticated; we observe a distinct DRAM traffic per frame for each pattern ranging from less than 9 MB to more than 1200 MB.

On each device, the observed rendering time also varies with the pattern rendered, and is higher when the DRAM traffic per frame is higher. This is consistent with rendering slowdowns due to contention on the memory subsystem.

We give additional evidence to support all these inferences in Sections 3.4 and 4.

1. Rendering different patterns on iGPUs from different vendors results in varying DRAM traffic, consistent with graphical data compression.

3.4. The Intel i7-8700 iGPU: A case study

We now expand on the experiment in Section 3.3 for a single iGPU: that of the Intel i7-8700.

Is there evidence that lossless compression is applied to graphical data used as both *inputs* and *outputs* of GPU rendering? What can we conclude about the compression parameters? Can we attribute rendering time differences, where present, to memory bus contention? And does compression leave data-dependent footprint in the cache?

To answer these questions, we refine our experimental setup. In the workload of Algorithm 1, each loop iteration first executes GPU-WRITE that writes to a texture and then executes GPU-READ that reads from a texture. In the experiments below, we split this workload into two: a *read workload* whose main loop executes only GPU-READ, and a *write workload* whose main loop executes only GPU-WRITE.

Like the combined workload, the read and write workloads are parameterized by texture pattern. We arrange for the read workload to read the same textures as in the write workload. In the read workload we "pre-bake" the input texture in OpenGL code executing before the main render loop and excluded from instrumentation. We observe identical results for the BLACK and GRADIENT patterns (both apparently compressible) and for the RANDOM and SKEW patterns (both apparently non-compressible). For the following experiments, we run our workloads with all four patterns, but report the collected data grouped by compressible and non-compressible patterns.

Like the combined workload, the read and write workloads are also parameterized by *iterations*, the number of times the main loop is run in rendering a frame. For the following experiments, we vary *iterations* to understand how GPU behavior changes as rendering complexity increases.

Compression ratio. First, we study the compression ratio of iGPU reading and writing independently. We run the read workload and write workload described above with our four patterns (grouped as compressible and non-compressible), varying *iterations*. For each experimental condition we gather data from 400 seconds of execution.

On the read workload (Figure 2a), the DRAM read traffic dominates, and DRAM traffic differences manifest only from reads (not writes), reflecting the fact that the GPU-READ program object reads a texture. However, on the write workload (Figure 2b), DRAM write traffic dominates, and DRAM traffic differences mainly come from writes, reflecting the fact that the GPU-WRITE program object writes to a texture. For each experimental condition, the DRAM read traffic generated by the read workload is equal to the DRAM write traffic generated by the write workload.[4] Comparing non-compressible to compressible patterns, we observe a traffic ratio of 2 in reads (for the read workload) and writes (for the write workloads) across settings of *iterations*.

2. A compression ratio of 2 on both iGPU reading and writing is observed on an Intel i7-8700 processor.

Root cause of rendering time difference. Next, we study how compression-induced traffic reduction affects rendering time. For sufficiently large *iterations* settings, the frame rendering time for our workloads exceeds the 16.7 ms floor.

4. The read workload has non-negligible DRAM write traffic, probably because it writes to the framebuffer in rendering to the screen.

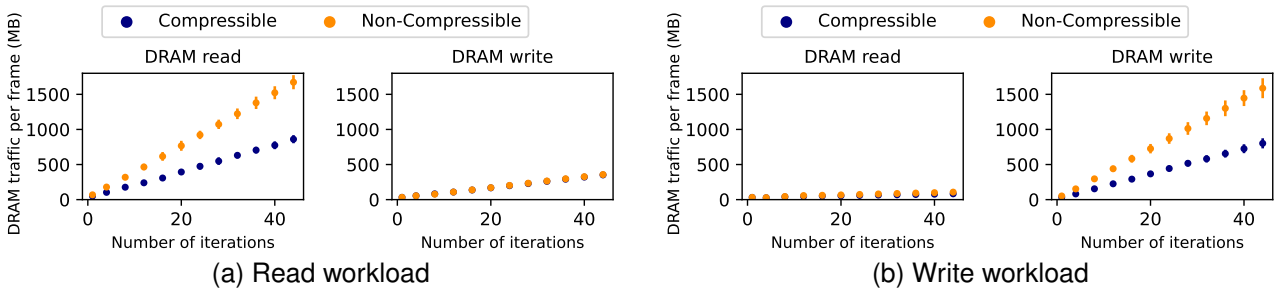


Figure 2. DRAM read and write traffic per frame (MB) vs. *iterations* on our Intel i7-8700 when running the read workload (subfigure a) and write workload (subfigure b) on a compressible or non-compressible pattern. We observe a stable compression ratio of 2 on both DRAM read traffic and write traffic.

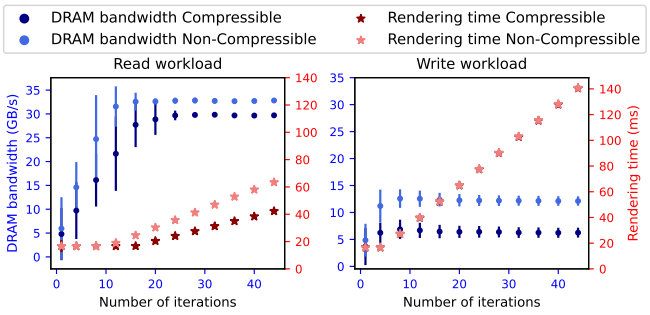


Figure 3. DRAM bandwidth (GB/s) and rendering time (ms) vs. *iterations* on our Intel i7-8700 when running the read workload (left) and write workload (right) on compressible and non-compressible patterns. The read workload generates enough traffic to saturate the iGPU memory subsystem. As a result, traffic reduction due to compression leads to a proportional reduction in the rendering time of compressible patterns. The write workload does not saturate the iGPU memory subsystem. The rendering time delay is due to running out of iGPU computation resources. Therefore, DRAM bandwidth differences are not reflected in the rendering time.

In this regime, observed rendering time is a function of how fast the iGPU can render each frame, and we can analyze where the bottleneck is.

We find that, when the iGPU saturates the iGPU memory subsystem, rendering time per frame correlates with the DRAM traffic per frame, which (as shown above) varies according to the compressibility of the rendered pattern.

In Figure 3, we plot the DRAM bandwidth (GB/s) and rendering time (ms) for compressible and non-compressible patterns as we increase *iterations* in the read workload and write workload. When *iterations* is sufficiently high, the DRAM bandwidth stops increasing for both workloads.

In that regime, for the read workload, we observe an almost two-fold difference in rendering time between compressible and non-compressible patterns, all else equal. Note, the apparent compression rate of 2 implies that the arithmetic intensity (in operations/byte) of compressible patterns is actually about twice that of non-compressible. Since rendering time is changing proportionally with arithmetic intensity, the workload is memory-bandwidth bound. This is consistent with both compressible and non-compressible patterns having almost-equal DRAM bandwidths (which we hypothesize corresponds to the iGPU memory subsystem's peak bandwidth). We conclude that the read workload saturates the iGPU memory subsystem, and that the reduction in DRAM traffic due to compression is a cause of reduced rendering time for compressible patterns.

The situation is different for the write workload. Initially, the compressible and non-compressible patterns display different DRAM bandwidth, and this trend persists even after the bandwidth stops increasing at *iterations* $= 4$ (where the difference in bandwidth between non-compressible and compressible patterns settles to a factor of 2, in line with the apparent compression ratio). We observe no rendering time difference between the compressible and non-compressible patterns. This means that the write workload is either memory-latency or compute bound. There is evidence to suggest that it is the latter: GPU-WRITE is computationally heavy in creating textures from scratch and mainly issues non-blocking write operations that are off the critical path. As the iGPU memory subsystem remains under-utilized (relative to the documented maximum bandwidth), the differences in DRAM traffic resulting from compression does not trigger any notable differences in rendering time.

As established above, our write workload cannot saturate the iGPU memory subsystem. We now introduce extra memory contention alongside the write workload. This allows us to saturate the iGPU memory subsystem and confirm that the write workload's pattern-dependent DRAM bandwidth differences can manifest rendering time differences.

We use the `memcpy` workload from the `stress-ng` benchmark suite[5] to add system memory contention. With *iterations* fixed to 10 and the texture pattern varied, we measure the rendering time of the write workload as the number of `memcpy` stressors increases on our Intel i7-8700. For each combination of pattern and number of `memcpy` stressors, we render the write workload for 10 000 frame updates.

Figure 4 illustrates that when the number of `memcpy` stressors increases, both compressible and non-compressible patterns experience delays in rendering time. Notably, when a sufficient number of `memcpy` stressors are launched, we observe a difference in rendering time between compressible and non-compressible patterns. Furthermore, this difference increases with the number of `memcpy` stressors.

5. Online: https://github.com/ColinIanKing/stress-ng.

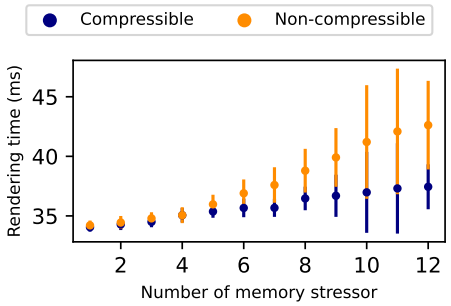


Figure 4. Rendering time (ms) vs. the number of `memcpy` stressors on our Intel i7-8700 when the write workload loads a compressible or non-compressible pattern in a loop of *iterations* equals to 10.

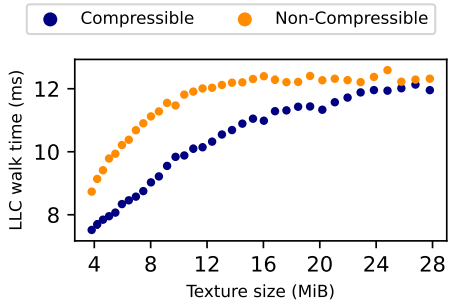


Figure 5. The LLC walk time (ms) of compressible and non-compressible patterns vs. nominal texture size on our Intel i7-8700. The maximum difference in LLC walk times occurs when the texture size is the size of our LLC (12 MB).

3. When the iGPU memory subsystem becomes saturated, memory bus contention variations caused by graphical data compression lead to rendering time differences on an Intel i7-8700 processor.

Last level cache walk time. On our Intel i7-8700, the iGPU and CPU share the last level cache (LLC). We find that the compression unit sits above the LLC and causes data-dependent LLC access patterns, which can be captured by CPU processes accessing their own data in memory.

Following the approach of Shusterman et al. [27], we define an *LLC walk time* metric for contention over the LLC. First, we allocate an LLC-sized buffer of 12 MB. Second, we access the whole buffer repeatedly until it occupies the entire LLC. Third, we run the write workload to create and load a texture into the LLC. Finally, we record the time to access the LLC-sized buffer again as our LLC walk time. We access the buffer elements in a fixed but nonconsecutive order to avoid triggering the prefetcher. When the entire buffer is resident in the cache, we find that the walk time is 6 ms; when the entire buffer has been evicted from cache, the walk time is 14 ms.

We investigate how the texture size affects the LLC walk time of compressible and non-compressible patterns. For each combination of pattern and texture size, we collect 2000 LLC walk time data points. In Figure 5, we plot the LLC walk time (ms) of compressible and non-compressible patterns as we increase the texture size. Note that the x axis plots *nominal* texture size, assuming 32 bits per pixel and no compression. There are three regimes of interest:

- When the nominal texture size is smaller than 12 MB, the write workload does not fully utilize the LLC for either the compressible or non-compressible patterns. Here, we observe that the LLC walk time increases linearly with nominal texture size, and that the gap in LLC walk time between compressible and non-compressible patterns widens with nominal texture size. This is expected, as the magnitude of size difference (between the original and compressed texture) increases with nominal texture size.
- When the texture size equals 12 MB (the LLC size), the write workload should fully utilize the LLC for the non-compressible pattern. This matches our observations: at 12 MB and beyond, the LLC walk time saturates for non-compressible patterns (but not for compressible patterns).
- Finally, when the texture size equals 24 MB, the write workload should fully utilize the LLC for even the compressible patterns (due to our previous finding of a compression ratio of 2). This again matches our observations: at 24 MB and beyond, the LLC walk time saturates for compressible patterns.

4. When a compressed texture is read from DRAM into the last-level cache on an Intel i7-8700 processor, it remains compressed. Compression thus induces pattern-dependent cache utilization, which can be captured by the LLC walk time metric.

Intra- vs inter-page HW compression. We validate our findings by measuring the number of memory *pages* accessed by our read workload and write workload. On those experiments in which DRAM traffic per frame differs between compressible and non-compressible patterns, the number of memory pages accessed per frame is *equal* between compressible and non-compressible patterns. We conclude that the iGPU compression reduces the number of cachelines accessed within a given memory page.

4. Reverse-Engineering Intel and AMD iGPU Graphical Data Compression

The previous section provides evidence supporting the existence of iGPU graphical data compression on multiple processor platforms. However, it does not indicate exactly when or where compression takes place. In this section, we trace the entire software stack from OpenGL through its backend Mesa, the Linux kernel and the iGPU. For three separate processor platforms from Intel and AMD, we demonstrate that the observable compression effects are fully attributable to a mechanism on the iGPU that is transparent to software on the CPU. Furthermore, we fully reverse-engineer, for the first time, the undocumented

compression algorithms used by the iGPUs of Intel 8th through 11th generation Core processors.

4.1. A minimal OpenGL example

Throughout this section we use a simple OpenGL program to cause predicable behavior on the CPU and iGPU. Our program defines a 3000×3000 pixel array with color values with specified RGBA patterns described below. It creates a texture from the pixel array. On each frame, it executes a trivial program object that outputs this texture to the screen. By tracing the execution of this simple program through userspace and the kernel, we single out the GPU as responsible for compressing the texture in memory.

For the GRADIENT pattern, our program fills the pixel array row by row with a repeated set of 150 colors ranging from $(0, 0, 0, 0)$ to $(149, 149, 149, 0)$. For the SKEW pattern, our program instead uses 151 colors ranging from $(0, 0, 0, 0)$ to $(150, 150, 150, 0)$, causing an offset striping pattern not aligned to the image dimensions.

4.2. Intel iGPU tracing

We use the minimal OpenGL program from Section 4.1 to trace the interaction between Mesa and i915 and dump the compressed texture. We further find evidence in Mesa's documentation supporting software-transparent, iGPU-based graphical data compression on Intel platforms, although Mesa is not aware of the exact compression algorithm. The compressed texture we dump aligns with hints from Mesa's documentation and our measurements.

Tile-Y format. On both our 8th Gen and 11th Gen Intel iGPUs, Mesa configures the tiling format of a texture as the "tile-Y" format.[6] As shown in Figure 6, the texture is first broken into 4 KiB tiles, where each tile represents a 32×32 pixel region.[7] When stored in memory, each tile forms an 8×8 grid of 64 B cachelines arranged in a Y-major configuration. Each cacheline corresponds to a 4×4 pixel region.[8] The tiled surface stored in memory is an upside-down reflection of the visual content displayed on the screen.

Color Control Surface. On Intel GPUs, the Color Control Surface (CCS) is an auxiliary surface that contains the compression status of cacheline pairs. For the i7-8700, the compression state is specified by 2 bits in the CCS [29, p. 159], and for the i5-1135G7 it is specified by 4 bits [30, p. 291].

According to Mesa documentation, in the tile-Y formatted main surface paired cachelines are horizontally adjacent in image space, but have starting addresses that differ by 512 B or 8 cachelines. For example, in Figure 6, cachelines 1 and 9 in the tile are a pair.

6. For other tiling formats supported, see Intel's reference manual [28].

7. A physical page is of size 4 KiB. A tile is 4 KiB such that moving within a tile does not leave the same physical page in memory.

8. Online: https://docs.mesa3d.org/isl/tiling.html.

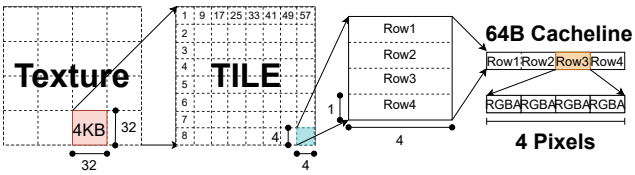


Figure 6. Intel tile-Y formatted texture. Each tile has a size of 4 KiB and is an 8×8 grid of 64 B cachelines arranged Y-major. Each cacheline corresponds to 4×4 pixels.

When creating a texture in OpenGL on our Intel iGPUs, Mesa sets up a tile-Y formatted main surface and an auxiliary CCS placed after the main surface at the next 4 KiB page-aligned boundary. Mesa allocates but does not fill the surface and its CCS. Notably, the Mesa developers have reverse-engineered how the compression state of a cacheline pair is encoded in 2 bits in the CCS, relevant for the i7-8700.[9] For a cacheline pair, 0b00 represents valid color data, 0b01 represents compressed color data that fits in one cacheline, and 0b11 represents the clear color (i.e., that the cacheline pair should be ignored). The iGPU compresses cacheline pairs in the main surface with a software-transparent, undocumented lossless compression algorithm and stores the compression states in the CCS. Since the compression is data-dependent, different patterns may exhibit different memory access patterns.

Mesa and i915 texture creation. We run our minimal OpenGL program under GDB and trace the Linux kernel driver (i915) using KProbes. Most interaction between Mesa and i915 are via Iris, the Mesa OpenGL driver for Intel.

Figure 7 presents our trace of Mesa and i915 when creating a texture with specified color values in OpenGL.

To summarize our findings: Mesa initially owns two header objects (`y_bo` and `linear_bo`) which specify format properties for two texture buffers that are allocated by the kernel and mapped into Mesa's address space at `iris_bo_mesa_addr` and `iris_bo_linear_mesa_addr` (steps 1–4 in Figure 7). These two headers (and their associated buffers) will encode the same texture, but in two different formats (`y_bo`: tile-Y with an auxiliary CCS for compression states, `linear_bo`: linear RGBA without a CCS). At this point, `iris_bo_mesa_addr` is filled with 0s and `iris_bo_linear_mesa_addr` is filled with the linearly RGBA-encoded pixel array data defined by our program above.

Next, Mesa issues a non-blocking command to the iGPU (using the above four objects as arguments) to preprocess the buffers (steps 5, 6). We find that this command causes the iGPU to asynchronously — and without any further intervention by the CPU — fill `iris_bo_mesa_addr` with a color-compressed tile-Y formatted texture, derived

9. Online: https://gitlab.freedesktop.org/mesa/mesa/-/blob/6d37f7f5ac9dbfd28874c24bbb67d14e932b2dac/src/intel/isl/isl.h.

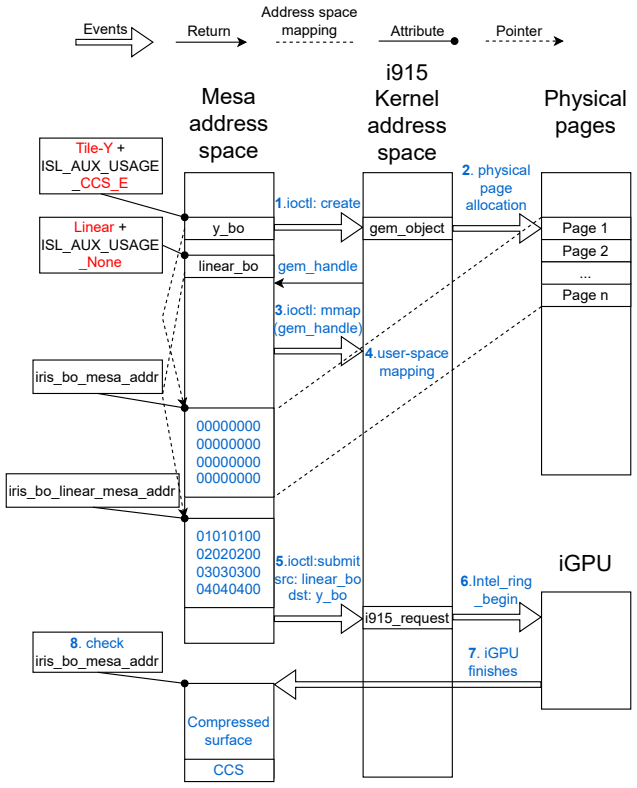


Figure 7. Mesa and i915 trace of texture creation. For the texture created in our minimal OpenGL program, i915 allocates two buffer objects, `linear_bo` at address `iris_bo_linear_mesa_addr` and `y_bo` at address `iris_bo_mesa_addr`, that correspond to the same texture in linear format and tile-Y format respectively (1, 2). i915 then maps both buffers into Mesa's address space (3, 4). After step 4, `iris_bo_linear_mesa_addr` is filled with linearly RGBA-encoded texture data and `iris_bo_mesa_addr` is filled with 0s. Later, Mesa issues a non-blocking command to the iGPU to fill `iris_bo_mesa_addr` (5, 6). The iGPU asynchronously — and without any further intervention by the CPU — updates `iris_bo_mesa_addr` with a color-compressed tile-Y formatted texture (7, 8).

from `iris_bo_linear_mesa_addr`, and correspondingly update the CCS for `y_bo` (steps 7, 8).

Dumping compressed and uncompressed textures. Because textures are mapped into the Mesa address space, we can easily dump them for analysis. In Figure 8, we show the hexadecimal representation of the first 128 B of the tile-Y formatted main surface and the first 16 B of the auxiliary CCS for GRADIENT (compressible) and SKEW (non-compressible) on our 8th Gen Intel iGPU.

With SKEW, the main surface contains pixels following the tile-Y format, where the first 16 B contain the 4 pixels in the first row and the second 16 B contain the 4 pixels in the second row, etc. Its auxiliary CCS is all `0x00` suggesting all the cacheline pairs contain valid color data. However, with GRADIENT, the main surface contains uninterpretable pixels that do not correspond to the input data in any obvious way. Furthermore, only 64 out of the 128 B in the main surface are non-zero. On the other hand, its auxiliary CCS contains `0x55` (0b01010101), suggesting a cacheline pair is compressed into one of the cachelines. The two dumped CCS encodings are consistent with the observed compression ratio of 2 in Section 3.4.

4.3. Intel GPU compression, reverse-engineered

We now showcase a complete reverse engineering of the Intel graphical data compression/decompression algorithm. We wrote a separate program to experiment with the iGPU's compression algorithm. Using this program, we reverse-engineer the (previously undocumented) algorithm used in 8th, 9th, and 10th generation Intel SoCs ("UHD Graphics 630") and the more sophisticated algorithm used in 11th generation Intel SoCs ("Iris Xe Graphics"). Additionally, we implemented our reverse-engineered decompression algorithms and tested their correctness.

Tooling. We use our program to define a pixel array of a specified size with color values specified as functions of the pixel's row, column, and array index (using a `dc`-like syntax). We create a texture from the pixel array (following the steps in Figure 7), and save the corresponding userspace memory mapping (created by the Linux kernel i915 driver) to a file.

We use our program to mount a chosen-plaintext attack on the compression algorithm by observing if and how a cacheline pair or a specified 4×8 pixel window was compressed.

Though we do not rely on this capability in our reverse engineering, we confirm that a variant of our program would also allow us to mount a chosen-ciphertext attack. The userspace mappings are writable; if we modify the compressed representation of a pixel window and then use an OpenGL call to recover the pixels corresponding to the texture, the GPU decompresses the *modified* representation.

Notation. Compression acts on a 4×8 pixel window. Absent compression, the left 4×4 pixels are stored in the first 64 B cacheline, the right 4×4 pixels are stored in the second cacheline. Within each cacheline the pixels are stored in row-major order starting from top left; the four color channels for a pixel are stored together in RGBA order; each channel holds a value between 0 and 255.

We write R_i, G_i, B_i, and A_i for the red, green, blue, and alpha values of the ith pixel ($1 \le i \le 32$), interpreted as elements of $\mathbb{Z}/256\mathbb{Z}$. When describing an operation applied to all four channels, we will let X stand for one of R, G, B, or A.

Eighth generation compression. The key idea behind Intel's algorithm is to make a *prediction* about pixel values and to transmit, for each pixel, the *residual* or error between the prediction and the actual pixel value. For a color channel X, the prediction $\hat{X}$ is chosen to be the minimum observed value: $\hat{X} \leftarrow \min_i X_i$; for the ith pixel, the X-channel residual x_i is $x_i \leftarrow X_i - \hat{X}$. The number of bits needed

	Gradient	Skew
The tile-Y formatted main surface	08 00 00 00 20 09 00 40 12 20 09 6C 03 00 40 12 20 09 6C 03 00 40 12 20 09 6C 03 00 40 12 20 09 6C 03 24 41 5B 60 1B FC 07 24 41 5B 60 1B FC 07 24 41 5B 60 1B FC 07 24 41 5B 60 1B FC 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 01 01 01 00 02 02 02 00 03 03 03 00 8D 8D 8D 00 8E 8E 8E 00 8F 8F 8F 00 90 90 90 00 83 83 83 00 84 84 84 00 85 85 85 00 86 86 86 00 79 79 79 00 7A 7A 7A 00 7B 7B 7B 00 7C 7C 7C 00 6F 6F 6F 00 70 70 70 00 71 71 71 00 72 72 72 00 65 65 65 00 66 66 66 00 67 67 67 00 68 68 68 00 5B 5B 5B 00 5C 5C 5C 00 5D 5D 5D 00 5E 5E 5E 00 51 51 51 00 52 52 52 00 53 53 53 00 54 54 54 00
The auxiliary CCS	55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 8. The first 128 B cacheline pair in the main surface and the first 16 B auxiliary CCS of GRADIENT and SKEW on an 8th Gen Intel iGPU. The CCS of SKEW is all `0x00`, which suggests that its main surface contains valid color following the tile-Y format. The CCS of GRADIENT is all `0x55`, which suggests that a cacheline pair in its main surface is compressed into one of the cachelines.

to encode the residual is $\ell(X) = \lceil \lg(\max_i X_i - \min_i X_i) \rceil$ if the X_i are not all equal, or $\ell(X) = 0$ otherwise.

The compressed encoding begins with a 48-bit header: 1 bit per channel to encode a skip bit, set to 1 if $\ell(X) = 0$ (i.e., all "no X residual is needed"), or 0 otherwise; 8 bits per channel for the prediction $\hat{X}$; and 3 bits per channel to encode $\ell(X) - 1$ (or 0 if the corresponding skip bit is set). Because the encoding's goal is to fit the information about 32 pixels into a single 64 B cacheline, the 48-bit header leaves room for $\lfloor (8 \cdot 64 - 48)/32 \rfloor = 14$ bits of residual per pixel. Whether this budget suffices to encode the residuals, i.e., whether $\ell(R) + \ell(G) + \ell(B) + \ell(A) \leq 14$, is what determines whether a cacheline pair is compressed.

When compression is applied, each pixel is always allocated 14 bits: $\ell(R)$ bits to encode $r_i = R_i - \hat{R}$, $\ell(G)$ bits to encode g_i, $\ell(B)$ bits to encode b_i, $\ell(A)$ bits to encode a_i, and 0 padding if necessary.

The idea behind the compression algorithm is described in an Intel patent [31], but the encoding details are different.

Eleventh generation compression. Compression in 11th generation SoCs is built on the same prediction-and-residual idea as in previous generations, but with a few minor differences and three substantial new features that make it meaningfully more capable.

We begin by listing the minor differences. First, the fixed header has a different format: a 1-bit color transform flag (described below); 8 bits that are usually all 0 (except in a special compression mode described below), followed by 4-bit encodings of $\ell(R)$, $\ell(G)$, and $\ell(B)$; and, finally, the 32-bit prediction, for a total of 53 bits. The parameter $\ell(A)$ is not explicitly encoded; instead, whatever bits in the per-pixel field are not used for the color channel are used as the alpha-channel residual. Second, pixels are described not in row-major order as in 8th generation compression but in 2×2 blocks, as shown in Figure 9. This is even though uncompressed cachelines in the legacy tile-Y format continue to list pixels in row-major order.

In the compression setting most similar to the 8th generation format, 32 pixels from two cachelines are compressed to fit a single cacheline, with each pixel allotted 14 bits. However, this is just one of several settings supported by

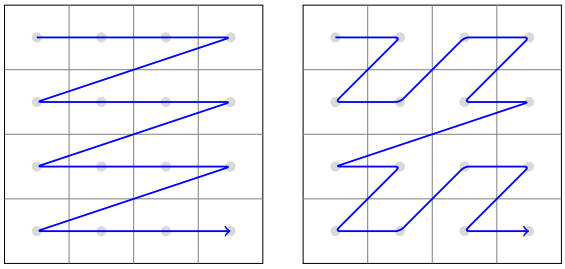


Figure 9. Encoded pixel order within a 4×4 pixel window in eighth generation and eleventh generation compression. Eighth generation compression pixel order is row-major as in uncompressed legacy tile-Y. Eleventh generation compression pixel order follows a Z-order curve.

11th generation GPUs. Whereas in previous generations each cacheline pair had 2 bits of metadata, with one value to signify compression, the 11th generation reserves 4 bits of metadata; different metadata values signify different compression settings. The 128-to-64 setting just described corresponds to metadata value `0x6`.

Metadata value `0x1` corresponds to a more aggressive 128-to-32 setting that fits information about all 32 pixels into the first half of one cacheline, with a per-pixel budget after the header of 6 bits. Metadata value `0x2` corresponds to compression of the first cacheline only (the left 4×4 pixel half of the window) in place from 64 B to under 32 B, leaving the second cacheline unchanged, with a per-pixel budget of 12 bits; metadata value `0x8` corresponds to compression of the second cacheline only, leaving the first cacheline unchanged, again with a per-pixel budget of 12 bits; and metadata value `0xa` corresponds to separate 64-to-32 compression of each cacheline, like `0x2` and `0x8` combined. The metadata values, but not the compression algorithm itself, are documented in Intel's manuals [32, pp. 128–130].

We do not know what benefit is obtained from packing the information in a 64 B cacheline into 32 B.

A major limitation of the 8th generation compression format is that each color channel is encoded independently,

and compression cannot take advantage of correlation between channels. For example, compressed grayscale pixel windows (where $R_i = G_i = B_i$) will store the same residual three times for each pixel. The 11th generation compression algorithm improves on this by supporting a color transformation to a colorspace an Intel patent application calls "BCoCg" [33] that decorrelates the color channels.[10]

To transform colors in the BCoCg colorspace back to RGB, the decompression algorithm computes:

$$R_i \leftarrow Co_i + B_i \qquad G_i \leftarrow Cg_i + \left\lfloor (B_i + R_i)/2 \right\rfloor \ . \qquad (1)$$

Note that (n, n, n) in the RGB colorspace corresponds to $(n, 0, 0)$ in the BCoCg colorspace. This is an example of decorrelation made possible by the transform.

The first bit in a compressed cacheline specifies the colorspace used: 1 for BCoCg, 0 for RGB. When BCoCg is used, the header encodes $\ell(B)$, $\ell(Co)$, and $\ell(Cg)$, in that order; the fixed-width field encoding a pixel's residual likewise encodes the residuals for B, Co, Cg and, with whatever bits remain, A, in that order.

Finally, in 128-to-64 (`0x6`) mode, the GPU can take advantage of uniform 2×2 blocks, i.e., ones in which all four pixels have the same color values and could therefore in principle be described by a single residual. A cacheline pair describes eight 2×2 blocks; if four of these blocks are uniform, the number of residuals needed drops from 32 to 20, and per-residual bit budget rises from 14 to $\left\lfloor (8 \cdot 64 - 53)/20 \right\rfloor = 22$. Bits 2 through 9 of the header (otherwise all 0) identify the uniform blocks: A 1 bit means the corresponding block (in Z order) is uniform. In a particularly Intel touch, the 22-bit residual fields are split. The first 14 bits of all twenty residuals are encoded like in usual mode `0x6`, for a total of 280 bits; then the last 8 bits of all twenty residuals, for a total of 160 additional bits.

One subtle point concerns the handling of arithmetic wraparound and negative color values. In the RGB colorspace, neither 8th generation nor 11th generation GPUs will take advantage of wraparound for compression. So, for example, color values 1, 2, 254, and 255 will have a prediction of 1 and 8-bit residuals rather than a prediction of 254 and 2-bit residuals. By contrast, equation (1) for colorspace transformation to Co and Cg is evaluated by embedding $\mathbb{Z}/256\mathbb{Z}$ in $\mathbb{Z}$. So, for example, and $R_i = 1$ and $B_i = 3$ will produce $Co_i = -2$, whereas $R_i = 255$ and $B_i = 1$ will produce $Co_i = 254$.

4.4. AMD iGPU tracing

We perform a similar tracing on an AMD iGPU (Vega 8 in an AMD Ryzen 7 4800U). As before, we trace Mesa and the relevant driver (amdgpu) to demonstrate how a texture is formatted in memory and compressed.

Mesa and amdgpu texture creation tracing. The process of texture creation on the AMD iGPU follows a path like that of Intel's. We observe that the iGPU is responsible for compression on AMD, as with Intel. Indeed, the compressed texture does not have a Mesa address space mapping or a Linux kernel mapping throughout its lifecycle. To be able to read the texture, we developed a kernel module that traces the buffer object `amdgpu_bo` representing the tiled texture in the kernel. After the iGPU writes to the texture, we update the buffer object to remove the `NO_CPU_ACCESS` attribute, then force a kernel address space mapping by invoking `amdgpu_bo_kmap()`. We are then able to dump the tiled texture via the kernel address space mapping. Figure 10 illustrates this process.

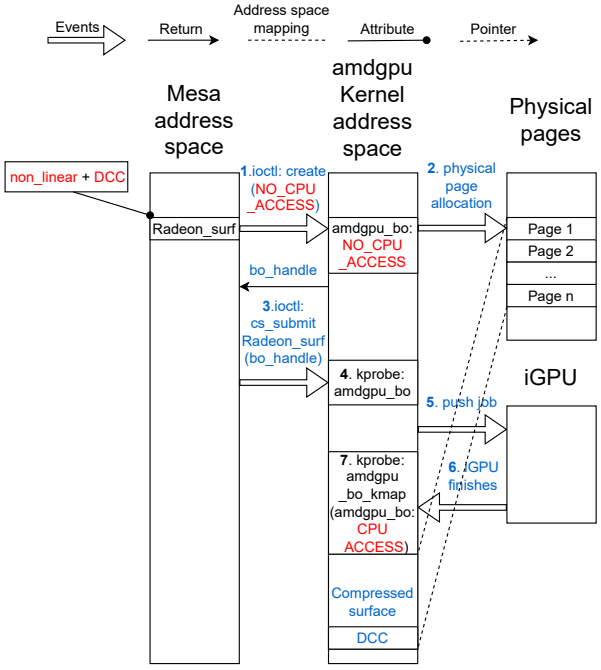


Figure 10. Mesa and amdgpu trace of texture creation. For the texture created in our minimal OpenGL program, amdgpu allocates a buffer object `amdgpu_bo` with a `NO_CPU_ACCESS` flag (1, 2). Mesa submits a non-blocking command to the iGPU for filling it (3–5). After the iGPU writes to the `amdgpu_bo`, we remove the `NO_CPU_ACCESS` flag and invoke `amdgpu_bo_kmap()` to force a kernel mapping (6, 7) that we can dump.

AMD tiling format, reverse-engineered. We use our minimal OpenGL program to create a texture with the RANDOM pattern in Figure 1. Since RANDOM is not compressible on our AMD iGPU, we use the tiled surface and the linear pixel values to reverse-engineer the tiling format. We present our result in Figure 11. Like Intel iGPUs, our AMD iGPU divides a texture into 4 KiB tiles, which are 32×32 pixel regions. Unlike on Intel iGPUs, each tile is broken down into 256 B blocks of dimension 8×8. The pixels in a 256 B block are rearranged in 4 cachelines and compressed together.

Delta Color Compression. AMD GPUs implement what AMD documentation calls Delta Color Compression (DCC) [16]. Each 256 B (8×8 pixels) block is accompanied

10. Intel's colorspace is related to, but not the same as, the YCoCg-R colorspace used in the HEVC screen content coding extensions [34].

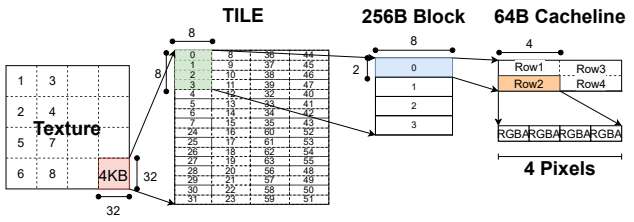


Figure 11. AMD tile formatted texture. Each tile has size of 4 KiB and is a 4×4 grid of 256 B blocks. Each block consists of 4 cachelines and corresponds to 8×8 pixels. Each cacheline is a region of 2×8 pixels. All cachelines are arranged in an undocumented combination of Y-major and X-major formats. We reverse-engineer the format and document here.

by 8 bits of DCC metadata. Similar to Intel CCS, the DCC metadata is placed right after the tiled main surface at the next 4 KiB page-aligned boundary.

The compressed and uncompressed texture on an AMD iGPU based on the 5th Gen GCN architecture. We dump the tile formatted main surface and the DCC metadata for both the GRADIENT and SKEW patterns on our AMD iGPU (5th Gen GCN). Both GRADIENT are SKEW are compressible on our AMD iGPU. However, they result in different DCC metadata encodings: the GRADIENT encoding consists solely of `0x28`, whereas the SKEW encoding is a combination of `0x28` and `0xcc`.

As explained in Section 4.5, `0x28` metadata means 256 B to 64 B compression, achieving a compression ratio of 4, whereas `0xcc` means (one configuration of) 256 B to 128 B compression, resulting in a compression ratio of 2. This is consistent with the compression ratios observed for GRADIENT and SKEW in Section 3.3 (Table 2). Partial dumps of GRADIENT and SKEW are shown in Figure 12.

4.5. AMD GPU compression, reverse-engineered

As noted in Section 4.4, AMD compression applies to 8×8 pixel tiles made up of four cachelines and described by 8 bits of metadata.

In fact, the AMD compression primitive applies separately to each of four 2×8 pixel subtiles. The length of the compressed representation of a subtile can vary between 2 B and (in principle) 74 B.

The GPU's strategy is to pack k consecutive subtile representations into one cacheline, leaving the next $k-1$ cachelines unused. Let L_i be the length (in bytes) of the compressed representation of the ith subtile. If $L_1 + L_2 + L_3 + L_4 \leq 64$, then the first cacheline will encode all four subtiles, leaving the other three cachelines unused (a situation we can describe as *4:0:0:0*). Otherwise, if $L_1 + L_2 + L_3 \leq 64$ then the first cacheline will encode the first three subtiles, the next two cachelines will be unused, and the fourth cacheline will hold the last subtile uncompressed (*3:0:0:1*). Other possible arrangements are *1:3:0:0*, *2:0:2:0*, *2:0:1:1*, *1:2:0:1*, *1:1:2:0*, and *1:1:1:1*, the last of these meaning no part of the tile is compressed.

Codewords are concatenated to make eight bits of metadata that describe the subtile arrangement into cachelines: codeword 0b11 stands for *1*, 0b0110 for *2:0*, 0b001100 for *3:0:0*, and 0b00101000 for *4:0:0:0*. Because this code is prefix-free when read from the lsb, metadata can be parsed unambiguously. For example, metadata `0xcc` (0b11001100) means *3:0:0:1*. (An AMD patent describes a similar but more flexible metadata encoding [35, Fig. 4].) In addition, metadata 0b01000000 indicates an all-black tile, and metadata 0b00010000 indicates a solid color other than black.

Knowing how many compressed subtiles a cacheline holds, one can parse the cacheline unambiguously. The subtile representations are interleaved: the first header bytes for all subtiles, the second header bytes for all subtile windows, the deltas for all subtile windows. There can be unused bytes at the end of a cacheline; these are apparently left uninitialized.

Whereas uncompressed pixels are stored in the usual RGB colorspace, compression is always attempted in a GCrCb colorspace that, like Intel's BCoCg colorspace, decorrelates the color channels. To translate from GCrCb to RGB, one computes $R_i \leftarrow Cr_i + G_i$ and $B_i \leftarrow Cb_i + G_i$. The GCrCb colorspace is documented in an AMD patent [35].

The fundamental object of compression is a 2×4-element *window*, consisting of the values in a single color channel of the pixels on the left or right half of a subtile. Each 2×8 pixel subtile includes eight such windows.

The *first header* for a subtile is 16 bits that specify, first, whether or not each window is described by a byte in the second header, and then whether or not each window is constant, i.e., has all entries equal. The *second header* for a window includes descriptive bytes whose interpretation is explained below. Finally, for each non-constant window, *deltas* describe how individual window color values relate.

In AMD's compression format, each window entry is computed as a delta from a neighboring entry, rather than all deriving from the same base value. The neighbor relationships are illustrated in Figure 13.

Deltas are represented as a sign bit followed by up to seven value bits; a delta with value x and sign bit set is interpreted as $255 - x$. There is one exception: When a window has a second header byte, the first delta is nonnegative and its sign bit is instead treated as the least-significant value bit. A window's deltas are stored bitsliced: All eight sign bits, then all eight least significant value bits, and so on.

The window's second-header byte, when present, specifies both the base X_0 to which the first delta is added and the number of value bits for delta representation. Specifically, there are as many delta value bits as the header byte has trailing 0 bits, and X_0 is equal to the base after the least-significant 1 bit is flipped to a 0. So, for example, a `0x70` header specifies deltas encoded using (a sign bit and) four value bits, and $X_0 = 96$.

Every entry in a constant window is equal to that window's second-header byte, if present; to 0 for a left-side window without a second-header byte; and to the top right value of the corresponding left-side window for a right-side window without a second-header byte.

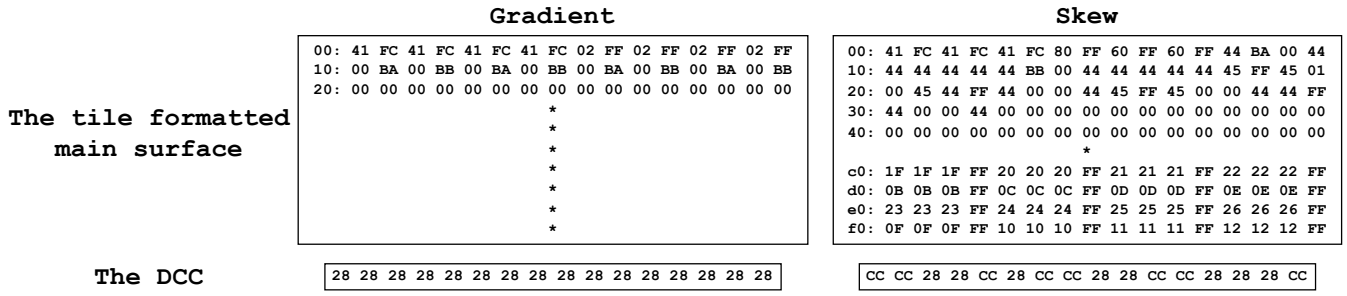


Figure 12. The first 256 B block in the main surface and the first 16 B chunk of DCC metadata of GRADIENT and SKEW on an AMD iGPU based on the 5th Gen GCN architecture. Three cachelines in the first 256 B block of SKEW are compressed into one cacheline, leaving the fourth cacheline uncompressed, and resulting in a compression ratio of 2. The first 256 B block of GRADIENT is compressed into one cacheline, resulting in a compression ratio of 4. The compression ratio is consistent with our observation in Section 3.3 (Table 2). The DCC of GRADIENT is all `0x28`, whereas the DCC of SKEW is a combination of `0x28` and `0xcc`.

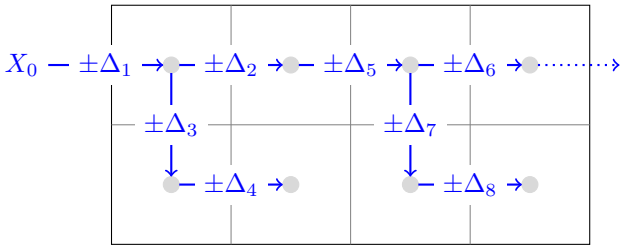


Figure 13. Interpretation of encoded deltas for a 2×4 pixel window in AMD GPU compression. If a header byte for the window is present, it explicitly encodes the base X_0 value for Δ_1. If a header byte for the window is absent, X_0 for a left-side window is 0 and for a right-side window is the value of the top right pixel of the window to the left. Deltas are sign-and-magnitude encoded except that, in the presence of an explicit base X_0, Δ_1 is encoded as a nonnegative value, with the sign bit interpreted as the lsb.

5. Side-Channel Attacks Exploiting Graphical Data Compression

In this section, we demonstrate proof-of-concept (PoC) cross-origin pixel stealing attacks that compromise Chrome's same-origin policy by utilizing the software-transparent GPU graphical data compression leakage channel. Our PoCs can steal pixels from a cross-origin iframe by either measuring the rendering time difference due to memory bus contention or by using the LLC walk time metric to infer the GPU-induced CPU cache state changes. We demonstrate successful pixel stealing on iGPUs and (surprisingly) on discrete GPUs. Finally, we include an end-to-end PoC in a real-world scenario, stealing Wikipedia usernames to deanonymize users (Section 5.4).

5.1. Experimental setup

We run experiments to confirm that Chrome (version 112) can trigger iGPU compression on our Intel i7-8700 (cf. Table 1). To monitor the DRAM traffic, we use the same methodology as detailed in Section 3.1. However, from JavaScript we can only access coarse-grained timers.

For the feasibility experiments in Section 5.2, we build Chrome with debug symbols (`v8_symbol_level = 2`), and use uprobe to expose the CPU time-stamp counter to Chrome. The time-stamp counter allows us to synchronize the rendering process and the sampling process and pinpoint the block of sampled data that belongs to the same frame.

We stress that the attacks evaluated in Sections 5.3 and 5.4 target the official release of Chrome without any customization and call only the usual JavaScript timers.

5.2. Measuring iGPU compression from Chrome

Similar to prior work [23], [36], [37], [25], [24], we build a web-page framework for violating the same-origin policy on cross-origin iframes. Specifically, the same-origin policy disallows a containing web page from accessing the content inside a cross-origin iframe. By applying SVG filters to the iframe, we can extract its content pixel by pixel due to iGPU compression side effects.

Our framework isolates, binarizes, and expands a selected single cross-origin pixel to a 2000×2000 pixel iframe that is still cross-origin. At this point, the enlarged iframe is either all black or all white depending on the target pixel. We then render an SVG filter stack on top of the iframe that creates compressible or non-compressible textures depending on the target pixel color.

Compression-centric SVG filter stack. We design an SVG filter stack that maximizes the difference in DRAM traffic between black and white target pixels in the presence of compression. We use `feTurbulence` to create 2000×2000 pixels of deterministically generated random noise. Then we use `feBlend` in `multiply` mode to combine the noise with the 2000×2000 expanded and binarized cross-origin pixel. This is a mux that outputs all black if the iframe is black, or noise if the iframe is white. We chain up many layers of $\texttt{feBlend}_{\texttt{multiply}}$ with the noise and the expanded pixel iframe as inputs.

Stated algorithmically, we ask the browser to perform Algorithm 2, where *iframe* is the 2000×2000 expanded

Algorithm 2: SVG filter stack.

Input: *iframe*, *layers*

$noise \leftarrow \text{TURBULENCE}()$;
$x_1 \leftarrow \text{BLEND}_{\text{MULTIPLY}}(noise, iframe)$;
for $i \leftarrow 2$ ***to*** *layers* **do**
  $x_i \leftarrow \text{BLEND}_{\text{MULTIPLY}}(noise, x_{i-1})$;
end

and binarized cross-origin pixel, and *layers* is a parameter we can vary to increase the workload complexity.

When the target pixel iframe is all black, all intermediate x_is are also all black, and are compressible on our tested platforms. Conversely, when the iframe is all white, all intermediate x_is become *noise*, which is non-compressible on our tested platforms. A compressible or non-compressible x_i induces a different amount of DRAM traffic due to iGPU compression. The malicious containing webpage can then infer the target pixel color by measuring the traffic difference through rendering time or CPU LLC cache side effects.

Compression SVG filter stack: DRAM traffic. To confirm that our SVG filter stack induces our leakage channel in Section 3, we increase the complexity of our workload by increasing the number of filter *layers*. We render our filter stack in an infinite loop of `requestAnimationFrame` for 400 seconds and measure the induced DRAM traffic and individual frame rendering times.

Figure 14a plots the DRAM read and write traffic per frame versus *layers* and Figure 14b plots the DRAM bandwidth and rendering time from this experiment. We observe a compression ratio of 2 in both DRAM read and write traffic, reflecting that the intermediate x_is are compressible when the binarized target pixel is black, and the filter stack writes to and reads from intermediate values in a chain. Our Chrome workload exhibits a rendering time difference between targeting a black and white cross-origin pixels, despite not saturating the iGPU memory subsystem. This suggests our Chrome workload is a combination of the read and write workloads in Section 3.4, with the workload being computationally bounded but the contention on the memory bus being significant enough to influence the rendering time.

With a black pixel, DRAM bandwidth average and standard deviation decrease as *layers* increases. This is because of amortization of a large I/O transfer that happens at the start of every frame refresh. As *layers* increases, the contribution to average DRAM bandwidth from our SVG filter stack eventually outweighs the initial I/O transfer, and the DRAM bandwidth average and standard deviation decrease.

Compression SVG filter stack: LLC walk time. Since our SVG filter stack reads from and writes to the x_is, the LLC walk time metric in Section 3.4 should apply here as well.

To maximize the LLC walk time difference between compressible and non-compressible x_is, we set the size of our enlarged, binarized target pixel iframe to match the size of our Intel i7-8700 LLC. To avoid having the iGPU be idle, and allow other system activities disrupt the induced LLC access pattern, we adjust *layers* to ensure that the SVG filter stack takes more than 16.7 ms to render. We measure the LLC walk time from Chrome using the same methodology as Shusterman et al. [27] immediately after rendering.

In Figure 15, we show the LLC walk time distribution when the binarized cross-origin pixel is black or white. We observe distinct distributions because iGPU compresses black intermediate x_i layers, causing fewer LLC evictions.

Correctness check. To further confirm we are observing the iGPU compression effect, we experimentally disabled Vulkan in Chrome to force OpenGL usage.[11] We then attached GDB to the Chrome GPU process and extracted the starting all-black or all-white textures for the iframe, as well as the compressed-black or uncompressed-turbulent results of SVG filter stack rendering.

5.3. Proof-of-Concept pixel stealing attack

Having validated the side-channel effects of our compression-centric SVG filter stack, we build end-to-end pixel stealing attacks on stock Chrome. We tested both the LLC walk time and rendering time channels for this PoC. The PoC first profiles the channel on the iGPU with attacker controlled black or white cross-origin pixels. Our target is a cross-origin 48-pixel square checkerboard, which allows us to observe stability as well as transitions.

We have a 1-pixel square `div` scroll over the checkerboard column by column. For each cross-origin pixel, we use the same methodology in Section 5.2 to expand it to an iframe and render our SVG filter stack on the iframe for a tunable duration, and then collect frame rendering times during execution or the LLC walk time afterwards. Finally, we exclude outliers by removing the top and bottom 5 percent of collected data and comparing the mean average with the thresholds from profiling to classify the target pixel.

As we have noted, the PoC design is similar to that used in prior work [23], [36], [37], [25], [24], but with a novel side channel. For completeness, we illustrate the pipeline of our PoC in Figure 16.

PoC results. We test our PoC on heterogeneous hardware, tweaking the iframe size, the filter stack layers, and per-pixel rendering duration for each platform.[12] Our PoC can successfully reconstruct the cross-origin checkerboard on integrated GPUs from Intel, AMD, Apple, and Arm, and on two discrete GPUs from Nvidia. We have preliminary results that confirm that the amount of memory traffic varies based on the target pixel color for our Mali iGPU and our Nvidia discrete GPUs. The accuracy and throughput vary across platforms due to the underlying compression algorithm, hardware specifications, and our tuning of parameters. We observe the best PoC throughput on the platform

11. We observe the same compression effect in Chrome with Vulkan enabled and disabled.

12. All platforms except for the Google Pixel have a 60 Hz frame rate, and the Google Pixel has a 90 Hz frame rate.

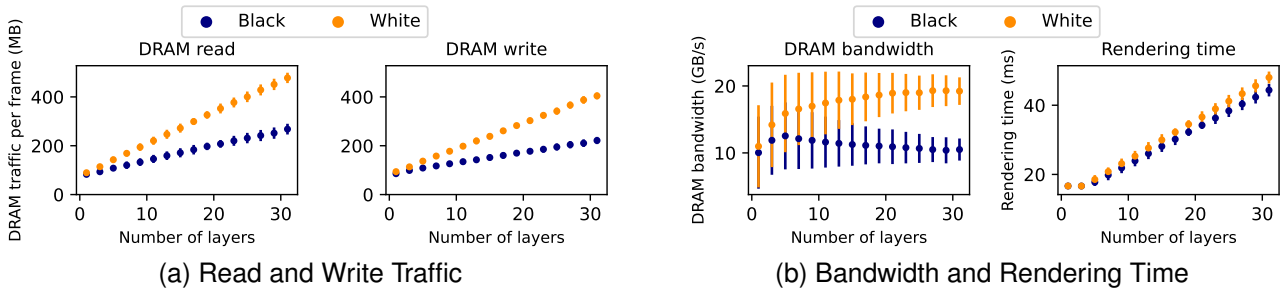


Figure 14. DRAM read and write traffic per frame (MB) vs. *layers*, and DRAM bandwidth (GB/s) and rendering time (ms) vs. *layers* on our Intel i7-8700 when Chrome renders the compression SVG filter stack on an iframe expanding a binarized cross-origin pixel. We observe our Chrome workload as a combination of the read and write workloads in Section 3.4.

Table 3. PoC RESULTS VIA BOTH RENDERING TIME AND LLC WALK TIME SIDE CHANNEL ON HETEROGENEOUS HARDWARE PLATFORMS.

SoC & iGPU	Chrome version	Operating system	Screen resolution	Rendering time side channel		LLC walk time side channel	
				Throughput (pixels/second)	Accuracy	Throughput (pixels/second)	Accuracy
Intel i7-8700 (Desktop) Intel UHD 630	112 64-bits	Ubuntu 22.04 (kernel 5.15)	1920 × 1080	2.0	99.6%	2.0	98.3%
Intel i7-12700K (Desktop) Intel UHD 770	111 64-bits	Ubuntu 22.04 (kernel 5.19)	1920 × 1080	0.5	93.8%	2.7	96.2%
Intel i7-10610U (Laptop) Intel UHD 620	112 64-bits	Windows 11 Pro	1920 × 1080	1.0	98.3%	0.5	94.3%
Intel i7-10510U (Laptop) Intel UHD 620	111 64-bits	Ubuntu 22.04 (kernel 5.19)	1920 × 1080	1.2	95.6%	1.4	96.9%
AMD Ryzen 7 4800U (Desktop) AMD Radeon Vega 8	111 64-bits	Ubuntu 22.04 (kernel 5.19)	1920 × 1080	6.2	93.4%	2.1	97.5%
AMD Ryzen 5 7600X (Desktop) NVIDIA GeForce RTX 2080 Super	109 64-bits	Windows 10 Pro	3440 × 1440	0.5	99.6%	N/A	N/A
Intel i7-11800H (Laptop) NVIDIA GeForce RTX 3060 Laptop	112 64-bits	Windows 11 Home	3840 × 1600	0.5	96.9%	N/A	N/A
Apple M1 Mac Mini (Desktop) Apple 8-core GPU	109 64-bits	Ventura 13.1	1920 × 1080	0.2	96.8%	N/A	N/A
Google Tensor (Google Pixel 6) Arm Mali G78 MP20	112 64-bits	Android 13	1080 × 2040	0.2	68.6%	N/A	N/A

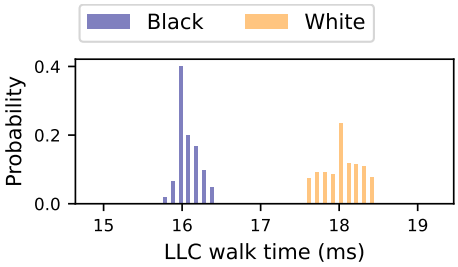


Figure 15. LLC walk times on our Intel i7-8700 when Chrome renders the compression SVG filter stack on an iframe targeting a binarized cross-origin pixel.

under test with the most aggressive implementation of GPU compression: the AMD Ryzen 7 4800U. On platforms with powerful DRAM controllers, e.g., Intel i7-12700K, the LLC walk time channel gives significantly better results than the rendering timing channel. On all platforms except on the Google Pixel, our accuracy is higher than 90% with a throughput ranging from 0.2 to 6.2 pixels per second.

This throughput is much faster than prior works that target SVG filters rendered on the GPU [24], [25]. The throughput from Taneja et al. ranges from 0.04 to 0.12 pixels per second, with an accuracy of around 70%. The throughput from Wang et al. ranges from 0.3 to 1.2 pixels per second, with an accuracy of around 95%.

5.4. Wikipedia username stealing example

As a proof-of-concept for a realistic attack, we demonstrate stealing a username. In this scenario the target iframe is Wikipedia, which shows the user's username in the top corner. We run this PoC with multiple browser windows open, with one playing a YouTube video during the attack. Figure 17 shows the results of our attack on an Intel i7-8700 and an AMD Ryzen 7 4800U. We calculate the accuracy based on the ground truth after color binarization. Our attack is unoptimized, but completes in 30 minutes on the Ryzen

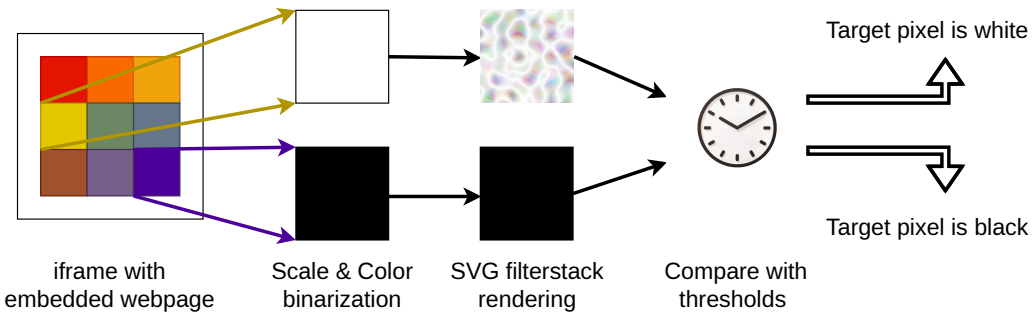


Figure 16. The pixel stealing Proof-of-Concept attack pipeline. We embed a cross-origin webpage in an iframe. We isolate and binarize a single cross-origin pixel from that iframe, and then expand it. We apply our compression-centric SVG filter stack on top of expanded pixel to create compressible or non-compressible textures depending on the target pixel's color. We use either the rendering time or the LLC walk time to infer the target pixel's color.

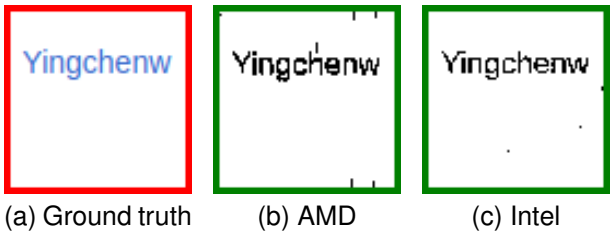


Figure 17. Pixel stealing PoC for deanonymizing a user, run with other tabs open playing video. "Ground Truth" is the victim iframe (Wikipedia logged in as "Yingchenw"). "AMD" is the attack result on a Ryzen 7 4800U after 30 minutes, with 97% accuracy. "Intel" is the attack result for an i7-8700 after 215 minutes with 98% accuracy.

with 97.0% accuracy. The Intel attack is significantly slower, at 215 minutes with 98.3% accuracy. Unlike previous works that are sensitive to noise in DVFS oscillation [24], [25], our PoC succeeds even in the presence of system noise, showcasing robustness.

6. Discussion

As display resolutions and frame rates increase, so too do GPU memory bandwidth demands. Graphical data compression is a natural response to these increasing bandwidth demands and, as we have documented, one deployed by vendors across the industry. It is natural for GPU vendors to choose software-transparent compression. The software stack of graphical rendering is complex and has many stakeholders; the less that the software stack must be modified to enable compression, the more easily can compression schemes be deployed and improved with new GPUs.

But data-dependent optimizations made by hardware and transparent to software create side channels that software is ill-positioned to mitigate. Making matters worse, while this paper considered compression applied to 2D textures, there is ample evidence (e.g., [35], [38]) suggesting that vendors are "dreaming bigger", i.e., considering applying compression to other forms of graphical data and to other places in the graphics pipeline (e.g., the frame buffer [18]). At worst, this creates a dire situation where *all* graphical data is vulnerable, regardless of what we do to harden GPU software (e.g., rewriting for constant time doesn't help).

What is to be done?

Disabling compression across the board is likely unworkable. Likewise, security through obscurity is clearly ineffective; as we have shown, with modest reverse-engineering effort, it is possible to reconstruct the compression algorithms implemented by Intel and AMD iGPUs. Using visually-lossy (*data/pattern-independent*) compression as a fallback when lossless compression fails in order to maintain a data-independent compression ratio, as proposed by some vendors (e.g., [39]), would close the specific side channels we exploit but may leave others, such as Hertzbleed [40], open; in addition, lossy compression applied at each step of a complicated rendering stack may degrade the output image unacceptably. Finally, one could consider exploiting existing implementation idiosyncrasies. For example, we found that Intel GPUs support compression on tile-Y surfaces but not on tile-X surfaces. This enables users to unilaterally opt a buffer out of compression on today's GPUs. But such approaches are inherently brittle and could stop working on future hardware or even after a stray microcode patch.

We believe, instead, that the right solution is to broaden the contract between (GPU) hardware and (graphics) software. For example, by exposing a policy such as "disable data-dependent compression" for a specific buffer, variable, etc. Due to the nature of compression, there seems to be an interesting design space governing such policies. Indeed, a crucial difference between GPU compression and TLS and HTTP compression (as exploited in the CRIME attack and its successors [7], [8], [9]) is the presence of metadata governing compression of small blocks of graphical data. One could imagine a future contract augmenting such metadata to allow software to opt sensitive regions out of compression.

Importantly, the above type of abstract but explicit (software-visible) contract would carry benefits for multiple

stakeholders in the GPU ecosystem. For example, it need not expose vendor secrets (e.g., the specific compression algorithms used).[13] Moreover, it trivially generalizes beyond compression (think "disable data-dependent *optimizations*") and could likely leverage existing techniques (e.g., from the information flow literature) to facilitate deployment.

7. Related Work

7.1. Security implications of lossless compression

Kelsey et al. pioneered the notion of secret-dependent compression ratios, and how to exploit them to build compression-ratio attacks in the context of CPU-side software-visible lossless compression algorithms [10]. Follow-up works showed how to apply said attacks to extract secrets from secure message transport protocols like HTTP and HTTPS [7], [8], [9] and, more recently, databases [41]. Beyond compression ratio, Schwarzl et al. [42] show how to exploit (de-)compression algorithm secret-dependent execution time to launch remote timing attacks on PHP applications and PostgreSQL. Finally, Tsai et al. [43] and Vicarte et al. [44] study compression ratio attacks stemming from proposed (but not known to be implemented) software-transparent compression schemes applied to processor hardware caches, register files and other pipeline elements.

Relative to the above work, this paper is the first to study the security implications of, and build compression-ratio attacks underpinned by, *software-transparent* lossless compression *in the wild*. Software-transparent lossless compression is an insidious new threat that opens new attack scenarios beyond what is available to an attacker exploiting software-visible compression. For example, it fundamentally breaks constant-time programming (which would presumably disable any software-visible compression, should it be present). It also dramatically complexifies the design of software defenses; as we have found, different vendors use different undocumented compression algorithms that vary even across generations of the same product line.

We are also the first to study the security of lossless compression in the context of code run on a GPU, and how that setting broadens the attack scenarios enabled by compression (e.g., cross-origin pixel stealing attacks).

7.2. Cross-origin pixel stealing attacks

There is a line of work [23], [37], [45] (starting with that of Paul Stone [23]) that exploits how SVG filters rendered on the CPU can be used to create pixel-dependent microarchitectural side channels and enable pixel stealing. These are ineffective/inapplicable when SVG filters are rendered on the GPU (which is the common deployment today).

Two works have since shown how to 'revive' cross-origin pixel stealing in the GPU setting [24], [25]. They work by exploiting analog effects. Specifically, by exploiting how attacker-chosen filters can create pixel-dependent GPU power consumption differences, which impacts CPU or GPU clock frequency, which in turn impacts metrics such as rendering time per frame.

Relative to these works, ours is the first to demonstrate a *purely digital* channel for pixel stealing on an iGPU. We show how a microarchitectural optimization—lossless data compression/decompression—likewise enables pixel stealing. Even after disabling access to the analog domain (e.g., through power, frequency, temperature [24], [25]), our attacks still go through. This is relevant for defenders, showing that our work will require separate (and likely different) defenses relative to prior art.

13. Although, if vendors are willing to provide better documentation, we would not object.

8. Conclusion

This paper shows that *software-transparent vendor-bespoke* compression exists *in the wild*, and can be used to launch novel side-channel attacks. These findings undermine the existing security posture related to compression-ratio attacks. For one, they show that software "does not simply" turn off compression. And in lieu of completely disabling compression, they call into question whether it will ever be appropriate to make assumptions about what compression algorithm will be in use.

More broadly, we note the rich literature on hardware-based compression for *other* parts of the system (e.g., the processor cache) [46]. In fact, there has been preliminary work analyzing the theoretical security implications of several of these proposals [43], [44]. We are unaware of these being implemented today. That said, our work (combined with today's scaling trends) suggests they may well be on the horizon and deserve early security-centric attention.

Acknowledgment

We thank our anonymous reviewers for their valuable feedback. This work was funded by NSF grants 1942888, 1954521, 2153388, and 2154183 and gifts from Google, Mozilla, and Qualcomm.

References

[1] J. Levin, **OS Internals, Volume II: Kernel Mode*. Technologeeks.com, 2019.

[2] S. Hollenbeck, "Transport layer security protocol compression methods," RFC 3749, May 2004.

[3] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The secure real-time transport protocol (SRTP)," RFC 3711, Mar. 2004.

[4] B. Monsour, W. R. Stevens, R. Pereira, and A. Shacham, "IP payload compression protocol (IPComp)," RFC 3173, Sep. 2001.

[5] M. D. D. Team, "Data compression," https://learn.microsoft.com/en-us/sql/relational-databases/data-compression/data-compression?view=sql-server-ver16, 2022.

[6] A. Pranckevicius, "Texture compression in 2020," 2020, online: https://aras-p.info/blog/2020/12/08/Texture-Compression-in-2020/.

[7] J. Rizzo and T. Duong, "The CRIME attack," Presented at Ekoparty 2012, Sep. 2012, slides online: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit.

[8] T. Be'ery and A. Shulman, "A perfect CRIME? only TIME will tell," Presented at Black Hat Europe 2013, Mar. 2013, whitepaper online: https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf.

[9] Y. Gluck, N. Harris, and A. Prado, "BREACH: Reviving the CRIME attack," Presented at Black Hat USA 2013, Aug. 2013, whitepaper online: https://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf.

[10] J. Kelsey, "Compression and information leakage of plaintext," in *FSE*, 2002.

[11] "W3c filter effects module level 1," https://www.w3.org/TR/filter-effects-1/#priv-sec, accessed on 12 02, 2022.

[12] "Canvas composite operations and css blend modes leak cross-origin data via timing attacks," https://bugs.chromium.org/p/chromium/issues/detail?id=699028, accessed on 12 02, 2022.

[13] "Svg filter timing attack," https://bugzilla.mozilla.org/show_bug.cgi?id=711043, accessed on 12 02, 2022.

[14] "Pixelstealing and history-stealing through floating-point timing side channel," https://bugzilla.mozilla.org/show_bug.cgi?id=1131288, accessed on 12 02, 2022.

[15] A. Pranckevičius, "Texture compression in 2020," Online: https://aras-p.info/blog/2020/12/08/Texture-Compression-in-2020/, Dec. 2020.

[16] C. Brennan, "Getting the most out of Delta Color Compression," Online: https://gpuopen.com/learn/dcc-overview/, Mar. 2016.

[17] D. Roberts and K. Hinson, "Discover advances in Metal for A15 Bionic," Online: https://developer.apple.com/videos/play/tech-talks/10876/, Sep. 2021.

[18] "Arm frame buffer compressions (AFBC)," Online: https://www.arm.com/technologies/graphics-technologies/arm-frame-buffer-compression.

[19] R. Britton, "Reducing memory bandwidth with PVRIC," Online: https://blog.imaginationtech.com/reducing-bandwidth-pvric/, Jul. 2018.

[20] B. Widawsky, "Framebuffer modifiers: Supporting end-to-end graphics compression," Presented at the 2017 Linux Plumbers Conference, Sep. 2017, slides online: https://blog.linuxplumbersconf.org/2017/ocw/system/presentations/4694/original/Framebuffer%20modifiers.pdf.

[21] "Whitepaper: NVIDIA GeForce GTX 1080," Online: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016.

[22] *Qualcomm Adreno Vulkan Developer Guide*, Qualcomm Technologies, Aug. 2017, archived: https://web.archive.org/web/20171201150710/https://developer.qualcomm.com/qfile/34706/80-nb295-7_a-adreno_vulkan_developer_guide.pdf.

[23] P. Stone, "Pixel perfect timing attacks with HTML5," Context Information Security, White Paper, 2013.

[24] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, "Hot pixels: Frequency, power, and temperature attacks on GPUs and ARM SoCs," in *USENIX Security*, 2023.

[25] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data," in *S&P*, 2023.

[26] *Processor Programming Reference (PPR) for AMD Family 17h Model 60h, Revision A1 Processors*, Sep. 2020.

[27] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website fingerprinting through the cache occupancy channel and its real world practicality," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2042–60, 2021.

[28] *Intel Open Source HD Graphics, Intel Iris Graphics, and Intel Iris Pro Graphics Programmer's Reference Manual: Volume 5: Memory Views*, May 2016.

[29] *Intel Open Source HD Graphics, Intel Iris Graphics, and Intel Iris Pro Graphics Programmer's Reference Manual: Volume 12: Display*, May 2016.

[30] *Intel Iris Xe and UHD Graphics Open Source Programmer's Reference Manual: Volume 12: Display Engine*, Dec. 2021.

[31] A. R. Appu, P. Surti, and H. Akiba, "Method and apparatus for multi format lossless compression," Patent US10453169B2, 2019.

[32] *Intel Iris Xe and UHD Graphics Open Source Programmer's Reference Manual: Volume 5: Memory Data Formats*, Dec. 2021.

[33] S. Kothandaraman, K. Vaidyanathan, A. R. Appu, K. Szerszen, and P. Surti, "Unified memory compression mechanism," Patent application publication US20220084156A1, 2022.

[34] L. Zhang, X. Xiu, J. Chen, M. Karczewicz, Y. He, Y. Ye, J. Xu, J. Sole, and W.-S. Kim, "Adaptive color-space transform in HEVC screen content coding," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 446–59, 2016.

[35] C. Brennan and T. T. Paltashev, "Method and apparatus for compressing randomly accessed data," Patent US10062143B2, 2018.

[36] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using CSS filters," in *CCS*, 2013.

[37] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P*, 2015.

[38] T. G. Akenine-Moller, P. Surti, A. Koker, D. Puffer, and J. Nilsson, "Cache and compression interoperability in a graphics processor pipeline," Patent WO2018057109A1, 2018.

[39] B. Har-Even, "Introducing PVRIC4—taking image compression to the next level," Online: https://blog.imaginationtech.com/introducing-pvric4-taking-image-compression-to-the-next-level/, Oct. 2018.

[40] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning Power Side-Channel Attacks Into Timing Attacks on x86," in *USENIX Security*, 2022.

[41] M. Hogan, Y. Michalevsky, and S. Eskandarian, "Dbreach: Stealing from databases using compression side channels," in *S&P*, 2023.

[42] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss, "Practical timing side channel attacks on memory compression," in *S&P*, 2023.

[43] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking secrets through compressed caches," in *ASPLOS*, 2020.

[44] J. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data," in *ISCA*, 2021.

[45] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security*, 2017.

[46] D. R. Carvalho and A. Seznec, "Understanding cache compression," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 3, pp. 36:1–27, 2021.

Appendix
Mesa and i915 trace of texture creation

In Figure 18, we show the complete Mesa and i915 (Linux kernel driver for Intel integrated graphics) trace of texture creation in our minimal OpenGL program from Section 4.1.

① `glTexImage2D` creates the texture with an input pixel array that specifies each pixel's RGBA value. ② - ⑤ A series of Mesa functions are invoked to create an `iris_resource` object with a tile-Y formatted main surface and an auxiliary CCS surface. Upon creation, the main surface is configured with parameters including height, width, and size. The auxiliary CCS is configured with parameters including size and relative offset from the main surface starting address. ⑥ - ⑦ Mesa issues an ioctl call to i915, which requests the kernel to create a buffer object `drm_i915_gem_object` that manages this `iris_resource`. i915 creates the buffer object and returns a `gem_handle` to Mesa for Mesa to reference the same buffer object in the future. ⑧ Mesa then issues another ioctl call with the returned `gem_handle` as an argument, which requests the kernel to allocate physical pages for the buffer object. ⑨ - ⑪ Finally, after `drm_i915_gem_object` has physical pages backing it up, Mesa issues a mmap ioctl call with the same `gem_handle` as an argument. i915 returns an mmap offset to Mesa, which then obtains a Mesa address space mapping of `drm_i915_gem_object` at address `iris_bo_mesa_addr`. Later, when the iGPU finishes writing to `drm_i915_gem_object` (described below), we can find the compressed tile-Y formatted main surface at the address `iris_bo_mesa_addr`, and the associated CCS at the address `iris_bo_mesa_addr` plus the relative offset.

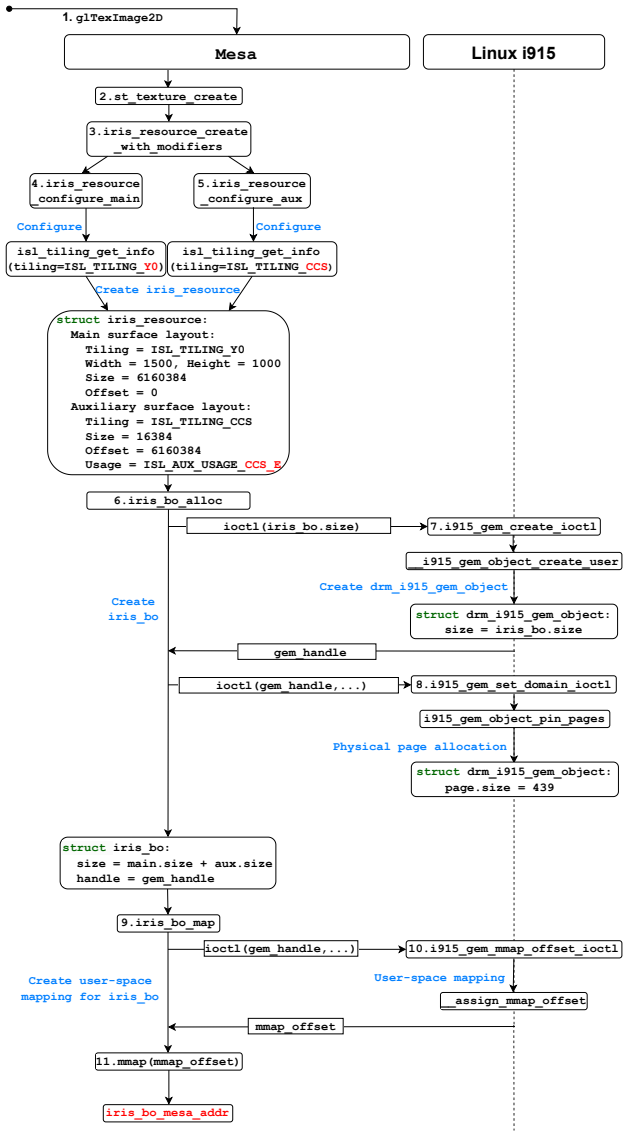


Figure 18. Complete Mesa and i915 trace of tile-Y formatted texture creation in our minimal OpenGL program from Section 4.1. Upon texture creation (`glTexImage2D`), Mesa configures a data structure `iris_resource` that records the necessary information of a tile-Y formatted main surface and its CCS auxiliary surface. Mesa issues ioctl calls to i915 to create and allocate a buffer object `drm_i915_gem_object` that represents the texture in the kernel space. Then, Mesa requests a userspace mapping of the `drm_i915_gem_object` and maps it at address `iris_bo_mesa_addr`. Once the iGPU completes writing to `drm_i915_gem_object`, `iris_bo_mesa_addr` contains the compressed tile-Y formatted main surface, and `iris_bo_mesa_addr` plus an offset contains the auxiliary CCS.

Appendix
Mesa and i915 trace of batch creation

Above we illustrate the process of texture creation in our minimal OpenGL program from Section 4.1. We show how to find the compressed tile-Y formatted surface and its auxiliary CCS. A natural question to ask is: How does Mesa interact with the iGPU to create the compressed surface?

In Figure 19, we show the complete Mesa and i915 trace of batch creation, where a batch is a job that Mesa submits to the iGPU. ① Above we mention that the function `glTexImage2D` creates an `iris_resource` object with a tile-Y formatted main surface and an auxiliary CCS. On top of that, `glTexImage2D` also creates another linearly formatted main surface with no CCS. The creation of this linear surface follows a similar process as that of the tiled one. We use `linear_iris_bo` to denote the Mesa address space mapping of the linear surface and `y0_iris_bo` to denote that of the tile-Y surface. Upon creation, `y0_iris_bo` is all 0s, and `linear_iris_bo` contains linearly formatted RGBA pixel values. ② - ④ To submit a rendering task to the iGPU, Mesa creates an `iris_batch` object that collects all necessary information from commands, and vertex buffers, to surfaces. The batch adds `linear_iris_bo` as the source with the

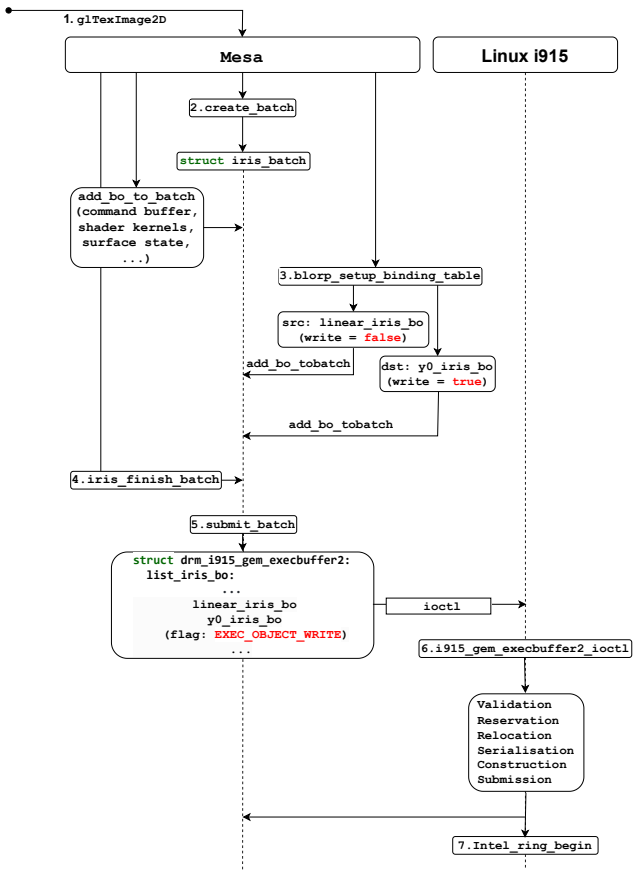


Figure 19. Complete Mesa and i915 trace of batch creation in our minimal OpenGL program from Section 4.1. `glTexImage2D` creates a linearly formatted main surface (with no CCS) mapped at address `linear_iris_bo` and a tile-Y formatted main surface (with CCS) mapped at address `y0_iris_bo`. Initially, `y0_iris_bo` is all 0s, and `linear_iris_bo` contains linearly formatted RGBA pixel values. For the iGPU to execute a rendering task, Mesa prepares an `iris_batch` with `linear_iris_bo` as the source and `y0_iris_bo` as the destination. Mesa submits the batch to i915, which validates and submits it to the iGPU. In the end, the iGPU fills `y0_iris_bo` with the compressed tile-Y formatted surface using `linear_iris_bo` as the source.

write attribute set to false, and adds `y0_iris_bo` as the destination with the write attribute set to true. ⑤ Mesa then submits this `iris_batch` to i915 via an ioctl call. ⑥-⑦ i915 verifies the validity of all pointers in the batch and then submits the job to the iGPU. The iGPU asynchronously fills `y0_iris_bo` with the compressed tile-Y formatted surface using `linear_iris_bo` as the source.