An Exploration of ARM System-Level Cache and GPU Side Channels

Patrick Cronin
University of Delaware
United States of America
ptrick@udel.edu

Xing Gao
University of Delaware
United States of America
xgao@udel.edu

Haining Wang
Virginia Tech
United States of America
hnw@vt.edu

Chase Cotton
University of Delaware
United States of America
ccotton@udel.edu

ABSTRACT

Advanced RISC Machines (ARM) processors have recently gained market share in both cloud computing and desktop applications. Meanwhile, ARM devices have shifted to a more peripheral based design, wherein designers attach a number of coprocessors and accelerators to the System-on-a-Chip (SoC). By adopting a System-Level Cache, which acts as a shared cache between the CPU-cores and peripherals, ARM attempts to alleviate the memory bottleneck issues that exist between data sources and accelerators. This paper investigates emerging security threats introduced by this new System-Level Cache. Specifically, we demonstrate that the System-Level Cache can still be exploited to create a cache occupancy channel to accurately fingerprint websites. We redesign and optimize the attack for various browsers based on the ARM cache design, which can significantly reduce the attack duration while increasing accuracy. Moreover, we introduce a novel GPU contention channel in mobile devices, which can achieve similar accuracy to the cache occupancy channel. We conduct a thorough evaluation by examining these attacks across multiple devices, including iOS, Android, and MacOS with the new M1 MacBook Air. The experimental results demonstrate that (1) the System-Level Cache based website fingerprinting technique can achieve promising accuracy in both open (up to 90%) and closed (up to 95%) world scenarios, and (2) our GPU contention channel is more effective than the CPU cache channel on Android devices.

ACM Reference Format:
Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An Exploration of ARM System-Level Cache and GPU Side Channels. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3485902

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '21, December 6–10, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485902

1 INTRODUCTION

While Advanced RISC Machines (ARM) processors have dominated the mobile device market over the past decade, recently they have also gained market share in both cloud computing and desktop applications. Enterprises like Apple and Samsung have announced plans to develop ARM based laptop devices that function with the complete MacOS and Windows operating systems. Apple has already released its M1 ARM chip to power its newest laptop and desktop devices. Spurring this rapid expansion of ARM devices into new markets is the adoption of a more peripheral based design that attaches a number of coprocessors and accelerators to the System-on-a-Chip (SoC). ARM has also adopted a System-Level Cache to serve as a shared cache between the CPU-cores and peripherals. This design works to alleviate the memory bottleneck issues that exist between data sources and the accelerators, allowing higher speed communication and increased performance.

If the marketshare of ARM processors in desktop and laptop systems continues to increase, it is expected that attackers will devote more resources to attacking the ARM architecture. While extensive research has been conducted on exploring and securing microarchitectural side channels on Intel's x86 systems, far less research has been focused on the ARM architecture. Furthermore, as mobile OSes tend to deny low level control over the hardware, most vulnerabilities are usually within non-essential APIs [5, 9, 21, 27, 54, 55] and are rapidly patched. ARM designers must be careful to ensure that their designs are not vulnerable to malicious attacks when exposed to a full fledged operating system, where OS developers are able to exert far fewer restrictions on potential attacker activities.

In this paper, we present an in-depth security study on recent personal computing devices (e.g., mobile phones and laptops) equipped with ARM processors with the recent DynamIQ [34] design. Unlike previous designs that only share cache within core clusters, these devices contain multiple levels of cache and share the last-level cache with other core clusters and accelerators (e.g., graphics processing unit). Unlike x86 processors, these ARM devices utilize heterogeneous core architectures, different caching policies, and advanced energy aware scheduling to increase performance and battery life. We endeavor to examine whether those advancements (e.g., new cache architectures, the tight integration of accelerators, etc.) make the ARM platform more difficult to attack compared with with x86 platforms.

Specifically, we focus on investigating cache occupancy channels [50], which continually monitor shared cache activities, to fingerprint websites. We design a series of microbenchmarks to better understand how ARM system behaviors (e.g., energy aware scheduling, core selection, and different browsers) affect the cache occupancy channels. Based on our preliminary study, we further optimize the attack for these new ARM cache designs and consider multiple different browsers, including Chrome, Safari, and Firefox. The redesigned attack significantly reduces the attack duration while increasing accuracy over previous cache occupancy attacks. Furthermore, we introduce a novel GPU contention channel in mobile devices, which can achieve similar accuracy as the cache occupancy channel. To evaluate the proposed attacks, we conduct a thorough evaluation across multiple devices, including iOS, Android, and MacOS with the new, ARM-based, M1 MacBook Air. The experimental results show that the System-Level Cache based website fingerprinting technique can achieve promising accuracy in both open (up to 90%) and closed (up to 95%) world scenarios.

Overall, the main contributions of this work are summarized as followed:

- An examination of the system-level cache within new ARM SoCs that utilize the DynamIQ design principle, especially how different components and software scheduling affect cache behaviors.
- A thorough evaluation of the cache occupancy side channel attack on Android, iOS, and MacOS platforms implemented in both native and JavaScript attack vectors.
- An analysis of JavaScript engine memory management and how it impacts attack effectiveness.
- The discovery of a new GPU side channel attack that can be utilized to fingerprint user behaviors on MacOS and Android.

The rest of this paper is organized as follows: Section 2 provides necessary background information. Section 3 presents the threat model and discusses the unique challenges that the ARM architecture creates for attackers in a shared cache occupancy attack. Section 4 details our system design and Section 5 describes our experimental setup. Section 6 analyzes our findings and Section 7 surveys related works. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 Caching and Side-Channel Attacks

Modern computer systems utilize a tiered memory system to enhance their performance, from the smallest and fastest (i.e., L1) to larger and slower (e.g., L2 and L3). Two important distinctions in caching are exclusive and inclusive caching. Inclusive caching guarantees that any memory address that is included in a cache tier is also present in the cache tiers below it. For example, a value in the L1 cache is also present in the L2 and L3 caches. By contrast, an exclusive caching policy ensures that items are only present in one level of the cache (e.g., an item in the L1 cache is not present in the L2 or L3 cache). While there are various pros and cons to both caching policies, Intel x86 processors mostly employ inclusive caching, but recent ARM processors tend to utilize exclusive caching policies.

As portions of the cache are shared between all processes, it has been widely exploited for side channel attacks. By determining

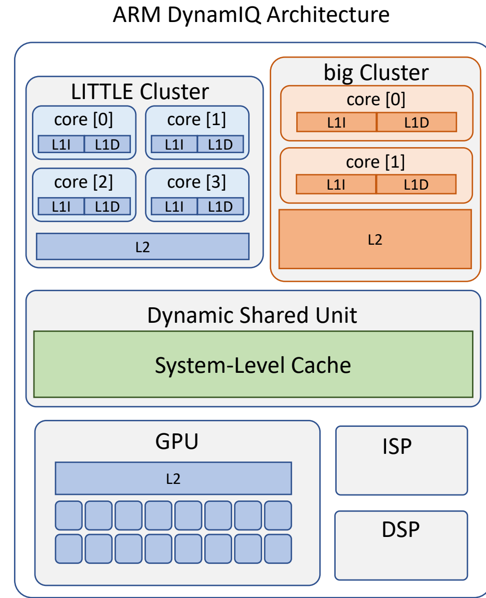


Figure 1: Overview of ARM's DynamIQ architecture featuring heterogeneous processor cores organized into high (big) and low (LITTLE) performance clusters. The CPU clusters and accelerators (GPU, ISP, and DSP) are all connected to a shared system-level cache.

whether specific memory is in the cache (e.g., timing its access time), attackers can infer the information of the victim. The 'prime+probe' attack [30, 40] attempts to identify vulnerable data locations that indicate specific program flows. With a high resolution timer and a predictable program, cache-based side channel attacks allow attackers to extract private information such as encryption keys.

Cache Occupancy Channel. Shusterman et al. [50] suggested two versions of the cache occupancy channel, *cache occupancy* and *cache sweeping*. In cache occupancy, they designated a sample rate (every 2ms) and accessed the entire buffer. If the buffer is accessed faster than 2ms, the total time to access the buffer is recorded. If the access takes longer than 2ms, a miss is recorded. In cache sweeping, the cache buffer is continually accessed and the number of full 'sweeps' in each sampling period is recorded. At the beginning of each sample period, the system starts accessing the cache from the first location. They demonstrated that such techniques can be used for robust website fingerprinting in x86 systems.

2.2 Consumer ARM System Design

Unlike x86 systems which utilize homogeneous core designs in their processors, consumer ARM devices (as opposed to ARM based server platforms which are out of the scope of this work) differ greatly and utilize a heterogeneous architecture.

ARM big.LITTLE and DynamIQ. In ARM, the big.LITTLE design was first developed to overcome the battery limitation in mobile devices. The big.LITTLE architecture consists of a SoC made from two discrete computing clusters, one low power group of cores and one high power group [31]. With a number of new scheduling techniques, the architecture allows the mobile OS to utilize high

and low power cores for different tasks to extend battery life. In ARM, the cache system is also redesigned. Instead of having a private and shared cache architecture with an identical size across all cores, big.LITTLE utilizes differently sized caches, wherein the high performance cores have access to larger L1/L2 caches than their lower performance counterparts. As the L2 caches of the different core clusters are not shared between clusters, a large amount of cache coherency traffic is necessary to facilitate switching tasks between the high and low performance cores, resulting in suboptimal performance.

To overcome this performance limitation, a newer system 'DynamIQ' [34] was developed for ARM. The DynamIQ system allows greater modularity and design freedom than the original big.LITTLE system. DynamIQ allows the processor designers to create multiple clusters of heterogeneous processors (instead of just two), and employs a shared L3 cache to improve computational performance between processor clusters, as shown in Figure 1. Our work explores the potential security vulnerabilities in this shared cache architecture.

Accelerators. Due to the explosive popularity of machine learning applications in image and signal processing domains, mobile devices have begun to require a low power method for executing neural network inference functions. To resolve this issue, current mobile devices make use of a number of accelerators or co-processors to enable advanced functionalities within their energy budget. Recent versions of Apple's custom A series chips, Qualcomm's Snapdragon, and Samsung's Exynos chips have begun to increase their reliance on accelerator peripherals. Those chips include dedicated digital signal processors, image signal processors, motion co-processors, neural processing units, and graphics processing units.

The inclusion of numerous accelerators creates a major system design issue. To utilize a co-processor, it must be supplied with a set of instructions and data to operate on. The co-processor must then complete its calculations and return the data to the main processor. In a non-integrated SoC, communication with co-processors must take place over a bus, and this can severely limit any performance speedup. Nvidia has attempted to resolve part of this problem on x86 with GPUDirect [15], allowing for direct transfer of data to the GPU without the CPU. To speed up co-processor performance in ARM, the DynamIQ system utilizes a system-level cache that is shared with these accelerators. ARM calls this technology *cache stashing* [32], which allows tightly coupled accelerators (such as GPUs) to directly access the shared L3 cache and in some cases directly access L2 caches.

2.3 Website Fingerprinting and Timer Restrictions

Website fingerprinting attacks identify the websites that a user visits. Usually this involves training a classification system to distinguish a series of sensitive websites that the attacker is interested in. The motivations for website fingerprinting can range from a desire of learning information about a target (e.g., political views, health issues, and gambling activity) to the construction of a user profile for advertisement tracking. Typically, website fingerprinting attacks involve an attacker that observes encrypted network traffic and attempts to classify the user's activities through features extracted from the packet stream (e.g., timing, packet size, and packet order) [4, 14, 20, 41, 43]. However, such attacks require access to the network traffic of the victim. To sidestep this requirement, researchers have identified that the action of downloading and rendering a website inevitably leaves a trace in the CPU and cache activities of the victim system, which can be monitored via local side channel to identify the victim's website visiting activities [36, 50].

Motivated by high profile side channel attacks like Spectre [23] and Meltdown [29] that utilize the JavaScript `performance.now()` command to perform nanosecond resolution timing measurements, browser and mobile operating system designers have worked to limit access to system APIs and high resolution timer resources. Specifically, in response to the Spectre and Meltdown attacks, browser manufacturers have greatly reduced the precision of the `performance.now()` counter [42] to between 50 microseconds and 1 millisecond. With the typical difference between cache misses and hits being defined in 10s of nanoseconds, this resolution is insufficient to successfully launch most side channel attacks.

3 THREAT MODEL AND CHALLENGES IN ARM

3.1 Threat Model

This work studies the ability of an attacker to fingerprint a user's website browsing activity via a low frequency contention channel in either the shared cache or the GPU of an ARM SoC. The attacker is motivated to track the user's web activity for some malicious purposes, such as to better identify the victim's interests for targeted advertising or to covertly determine sensitive information (e.g., medical condition, sexual/political preferences, etc.) for the purpose of discrimination or blackmail. We consider two typical scenarios in website fingerprinting: (1) closed world, where the victim only visits websites from the list of sensitive websites; and (2) open world, where users might also visit some non-sensitive websites. To accomplish the fingerprinting task, the attacker can pre-profile a list of sensitive websites and build a model based on specific browsers (e.g., Chrome/Firefox/Safari) and devices (e.g., MacBook/Smartphone).

To evaluate the potential threat from this attack, we mainly examine a web-based attacker who is only capable of delivering JavaScript from a website. We also conduct an investigation of an app based attacker who is able to trick a user into installing malware, but impose additional limits, analyzing how well the attack would function if the OS clock functions were similarly limited to those of web browser[1].

Web-Based Attacker. The web-based attacker attempts to exploit the cache occupancy channel in the context of the web browser, delivering a JavaScript file to the user via a malicious advertisement on a legitimate page or by tricking the user into visiting a malicious web page. We assume that the attacker is unable to exploit any vulnerabilities in the browser. Instead, (s)he attempts to create a cross tab attack scenario, wherein the user leaves the tab with the

[1]Researchers have demonstrated that the high precision timers available to native programs can produce very accurate attacks. OS developers may move to reduce the attack surface by reducing the granularity of available timers in the future

malicious JavaScript open and continues to browse other websites in a different tab. The malicious JavaScript in the background tab continues to run and attempts to monitor the user's activity. This is reasonable as all current web browsers enable users to visit multiple websites at the same time in different browser tabs. While tabs are isolated from each other in software, they are not necessarily segregated in hardware. Furthermore, the web-based attacker is restricted by the privileges granted to JavaScript, and are subject to the reduced precision timers, memory management, and scheduling constraints that the browser enforces.

App-Based Attacker. We assume that the app-based attacker is capable of tricking the user into installing an application or program onto their device that contains the malicious observation code. The code can be integrated into a benign application such as a music player, fitness tracker, or social media application, and is therefore capable of running a disguised process to monitor user activities. Unlike the web-based attacker, the app-based attacker is not restricted to only JavaScript and has access to the APIs provided by the operating system, allowing better control over memory management and scheduling. However, the attacker is not granted any super-user privileges and does not utilize any exploit to access privileged commands.

Note that, in both scenarios, the application/JavaScript does not necessarily need to be sourced from a purely malicious entity. Such a tracking service could be deployed in social media applications to better identify and profile user activities. Large ad-supported companies like Google or Facebook could also greatly benefit from deploying a similar script on their webpages, continually monitoring users browsing activities to better target advertisements.

3.2 Cache Occupancy Challenges in ARM

Exploiting the occupancy statistics of the last-level cache has been studied with varying degrees of success across x86 systems [6, 44, 50]. In parallel to this work, Shusterman et al. [49] performed a cursory proof that the cache occupancy could also be applied to ARM systems. We greatly expand their work, investigating a number of different configurations and optimizations across multiple browsers and devices. To motivate these optimizations, we first describe unique challenges that the ARM ecosystem presents to the cache occupancy channel.

ARM Cache Contention. ARM systems differ from common x86 architectures in multiple aspects. ARM offers exclusive and inclusive caching at different levels, and utilizes heterogeneous architectures in which multiple different core architectures and cache layouts may be present on the same chip. Also, each type of core may run at different frequencies. Those factors increase the difficulty of exploiting the cache occupancy channel in the ARM architecture. Since the system-level cache is the only cache level shared by all processor cores in ARM, if the scheduler moves the spy and victim processes between different core types, it can greatly affect the observed cache profile.

Due to the exclusive nature of the last-level cache in ARM, when a process migrates the data in its L1/L2 caches, the data will not be present in the last-level cache, but in the L1/L2 caches of its previous location. Upon migrating a process from one core type to another, some ARM processors invalidate the entirety of the previous cores' caches, while others may allow that data remain until it is evicted. In either case, in an exclusive cache setup, any reads to locations that were in the L1/L2 cache of the previous location will be serviced from the L1/L2 and have no impact on the L3 cache. This greatly hinders the cache occupancy channel: while in an inclusive cache, one could reliably observe L3 occupancy (if the value were removed from L3, it would be removed from all higher levels), the exclusive cache can serve the value from either the previous L1/L2 or main memory, giving no indication as to the status of the L3 cache[2].

Exclusive caching also has drawbacks with respect to buffer size. In an x86 system with inclusive caching, the spy process evicting the entire L3 cache would also remove any data in the L1/L2 caches. Thus, when the victim process accesses data, it always causes activities in the L3 cache[3]. However, in an ARM system, if a victim process accesses a buffer small enough to fit in the L1/L2 cache, a spy process that is monitoring the entirety of the L3 cache will never see this activity. While this behavior might be unnoticed, and even preferable, to a program under normal circumstances, it is not ideal for the cache occupancy channel. The cache occupancy channel assumes that continually accessing a large buffer in cache will completely evict any data of the victim process from the L3. Also, it assumes that any access to memory will bring data back into the L3, making it observable. Thus, to better suit ARM processors, the access patterns and buffer sizes for the cache occupancy channel should be carefully considered.

Browser Differences. Further complicating the applicability of the cache occupancy channel is the memory management of a web browser. The web-based attacker must work within the constraints of the JavaScript engine within each web browser. Today's popular web browsers, including Google Chrome, Apple Safari, and Mozilla Firefox, utilize different JavaScript engines. Furthermore, these JavaScript engines must interact with the system scheduler. Different OSes (e.g., Google's Android, Apple's iOS, and MacOS) likely utilize carefully tuned schedulers to maximize the performance. Finally, the JavaScript engines of the major browsers will manage memory in different ways, and the garbage collector of each JavaScript engine will handle memory management in a way that is not accessible to the attacker. Thus, a one size fits all approaches to cache occupancy fingerprinting is certainly not ideal as each browser may act very differently, even on the same hardware.

4 UNDERSTANDING ARM CACHE OCCUPANCY

We first design a series of microbenchmarks to better understand ARM system behaviors. In particular, we investigate how energy aware scheduling, core selection, and different browsers impact the cache occupancy channel.

[2]The L3 cache on ARM also maintains the ability to be selectively inclusive if an item is utilized by more than one core [35], however, the cache occupancy JavaScript channel does not utilize shared memory and should not experience this behavior.

[3]In some x86 server CPUs (specifically Skyake-X CPUs from Intel, the L3 is 'non-inclusive', meaning that it is neither fully inclusive or exclusive. Consumer CPUs from Intel have not yet adopted this layout.

Table 1: Devices and High Power (HP) and Low Power (LP) core configurations utilized in this work.

Device	Core Configuration	High Power L1/L2	Low Power L1/L2	System Level Cache
iPhone SE 2	2x Lightning (HP) 4x Thunder (LP)	128KB L1i / 128KB L1D / Core 8MB L2 Shared	Unknown L1i / 48KB L1D / Core 4MB L2 Shared	16MB
Android	4x Kryo 385 Gold (HP) 4x Kryo 385 Silver (LP)	64KB L1i / 64KB L1D / Core 256KB L2 / Core	64KB L1i / 64KB L1D / Core 128KB L2 / Core	2MB
MacBook Air	4x FireStorm (HP) 4x IceStorm (LP)	192KB L1i / 128KB L1D / Core 12MB L2 Shared	128KB L1i / 64KB L1D / Core 4MB L2 Shared	16MB

4.1 Test Devices

We select three commonly utilized devices: an iPhone SE 2 to test iOS, a Google Pixel 3 for Android, and a MacBook Air 2020 with M1 chip for MacOS. The detailed information about each device is included in Table 1. In the case of the Apple devices where specific cache sizing values are not provided by Apple, we rely on community microbenchmarking [11, 12] to provide detailed cache level size analysis.

We employ a Node.JS server to serve HTML and JavaScript resources to our test devices. As JavaScript is single threaded, our JavaScript microbenchmarks run within a web worker context[4].

4.2 Cache Access Pattern

Modern ARM processors utilize cache prefetchers to learn data access patterns and bring data into the cache beforehand. To accurately measure the cache performance of a device, we must develop cache access patterns to defeat the most common prefetching algorithms, next line and stride prefetching.

The next line prefetcher exploits spatial locality, fetching the next cache line of memory into the cache on every access. The stride prefetcher actively learns a pattern in data access and fetches the data based on the pattern.

It has been demonstrated that the stride prefetcher is limited in recognizing patterns within memory pages and can only keep track of a certain number of patterns before the hardware pattern matching is exhausted [7]. To evade the two prefetchers, we follow a similar access pattern to that of [7]. We create a large array of buffers which spans multiple memory pages. We then access the first, third, fifth, etc. line from every page. Accessing every other line avoids any impact of the next line prefetcher and accessing one item from each buffer before looping back to the first exhausts the ability of the stride prefetcher to learn a pattern.

4.3 Foreground vs. Background Activity

We next design a microbenchmark to identify cache behavior differences between foreground and background activity. Specifically, we seek to understand whether the scheduler treats foreground and background browser tabs differently.

To this end we create a large buffer and access increasingly large portions in the prefetcher thwarting manner described previously, normalizing the buffer access times with respect the number of memory accesses to better understand the impact of the scheduler and its use of the different core designs. To assess background activity, we run the script in a background tab while the foreground tab is set to `www.google.com`. We also find that writing to the accessed buffer (e.g., increment a counter stored at each array location) increases the consistency of experiments. This can be attributed to a more complex instruction stream reducing the amount of optimization and/or reordering that can occur, and thereby better exposing the cache sizes.

[4]Web workers were designed to facilitate long computations in a different process, while allowing the main UI thread to remain responsive.

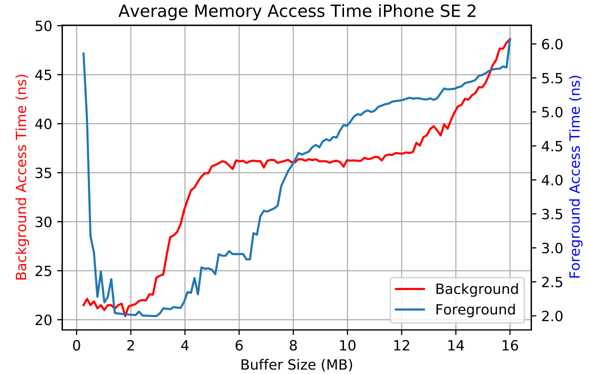


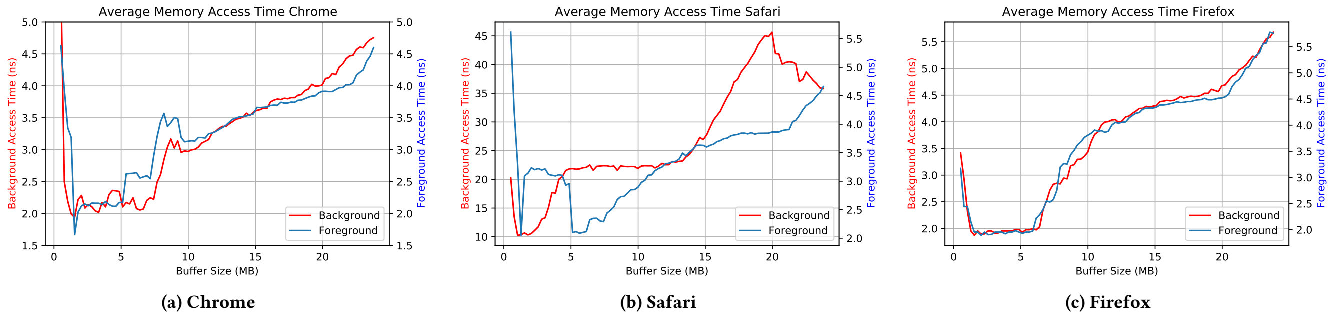
Figure 2: iPhone SE 2 Cache Average Memory Access Time

Testing this on the iPhone SE 2 demonstrates very different foreground and background cache behaviors, as shown in Figure 2. Background accesses are nearly 10x slower than foreground accesses, and the background memory access time curve is significantly different from the foreground access curve. The foreground curve experiences multiple sharp increases in cache access time, indicating that the multiple levels of cache are present (e.g., L1, L2, and L3) while the behavior of the background process shows far less distinguishable increases in timing. Similar behavior is observed on the Google Pixel 3.

4.4 Browser Memory Management

In a desktop operating system like MacOS, major browsers (e.g., Google Chrome, Apple Safari, and Mozilla Firefox) typically utilize their own rendering and JavaScript engine. Also, the M1 Macbook Air is the first device running a desktop/laptop operating system utilizing a heterogeneous processor. We therefore examine cache behaviors on the M1 MacBook Air across these major browsers.

Figure 3 shows the results for different browsers. Most notably, Apple's Safari is the only browser that seems to take advantage of the heterogeneous cores with a 10x slowdown in access speed



(a) Chrome **(b) Safari** **(c) Firefox**

Figure 3: MacBook Air M1 Cache Average Memory Access Times with Different Browsers.

and a noticeably different timing pattern for the background tab. This indicates that the foreground and background processes were impacting different caches (the different cache architectures of the high vs. low power cores). Both Google Chrome and Mozilla Firefox seem to maintain the same access speed for their respective foreground and background processes, indicating that background tabs are not relegated to the low power cores.

We also observe that the overall shape of the timing curves for cache accesses is unique to each browser, indicating that even though the access pattern was the same, the memory allocation algorithms for each JavaScript engine are vastly different. Thus, understanding how these allocation strategies affect cache timing can greatly increase the accuracy of a potential cache occupancy attack. Furthermore, different compiler optimizations and code differences could further impact the memory access differences that we observe across platforms.

5 ATTACKS ON ARM

In this section, we present our optimizations to the cache occupancy channel for various ARM devices. To determine the effectiveness of various modifications to the channel, we design a robust data collection system employing the Appium [1] and Selenium [47] frameworks to control our iOS, Android, and MacOS devices.

5.1 Setup

Data Sets. To monitor the accuracy of the cache occupancy channel, we utilize an abbreviated open world dataset, which consists of multiple accesses to sensitive and non-sensitive websites, marking all non-sensitive websites as a single class, regardless of domain. Particularly, we utilize a dataset similar to that of [50] to enable better comparison with the x86 version of the cache occupancy channel. Our dataset consists of 1,500 website accesses, containing 100 accesses to the top 10 Alexa websites (i.e., sensitive websites) and 1 access to 500 other websites not within the Alexa top 100 (i.e., non-sensitive websites). To prevent any ordering bias, we generate a random order for these 1,500 accesses and then utilize the same order for every experiment. We believe this randomization is important, and previous works do not discuss the access order.

Unlike network based fingerprinting attacks, the CPU cache may retain some of its state between website accesses, causing the machine learning system to identify incorrect features and falsely boost the accuracy of the test if websites are repeatedly accessed in the same order. Note that this abbreviated dataset is used in this section to optimize the side channel attacks on ARM. In the next section, we conduct a thorough evaluation using a significantly larger dataset.

Machine Learning Approaches. To evaluate the performance of our optimizations we utilize the Rocket [8] transform paired with ridge regression. The classifiers are trained and tested with a cross validation strategy, wherein we utilized 90% of the data for training, and 10% of the data for testing. We report the average of 5 rounds of training and testing.

5.2 Optimizing Cache Occupancy Attack

We have demonstrated in Section 4.3 that, unlike previous studies in homogeneous CPU architectures, cache accesses on low power cores can be nearly 10x slower. Combining this with the fact that browser manufacturers may continue to decrease the granularity of their timing sources to prevent attacks, it is necessary to re-examine the best way to measure cache occupancy on ARM.

Examining the Snapdragon 845 processor in the Google Pixel 3, we find that the low power cores are based on the Cortex A55 design from ARM and that the Snapdragon 845 processor has been configured to utilize 2MB of the system-level cache. Using the information from our previous microbenchmarks on the Google Pixel 3, it takes about 60ns to access a single cache value at a 2MB buffer size. Since the Snapdragon 845 employs a 64 Byte cache line size, to avoid prefetching, we should access every 32nd integer in our 2MB buffer. As the buffer can hold ≈500,000 integers, this results in $\approx 16,000$ accesses. At 60ns per access, this equates to just under 1ms. While the Snapdragon 845 has configured the system-level cache to be 2MB, the Cortex A55 supports up to 4MB of shared cache [33], and the accesses may take almost 2ms with no background activity, and will almost certainly take more than 2ms if the processor is performing another task. Thus, if the system described in [50] is used without modification, every trace would be nearly identical with only overlong accesses, and hence no identification would be possible. To this end, we propose a series of modifications that work for devices, regardless of their access speed to cache. This enables attackers to adjust the buffer size for the device and be worry-free about adjusting the sample rate if the device happens to be very slow.

Modifications. The first modification entails recording the number of cache accesses within a set time frame, instead of the time to complete accesses. This system is far less affected by changes in the accuracy of clock. The system will always record the number of actual cache accesses, a number that is far more fine-grained than the time to access the whole cache. To enhance system performance on slower devices, we also increase the access time window to 4ms to increase the number of possible accesses. With these initial modifications, we achieve 75% open world accuracy in the abbreviated 10-site test (Section 5.1) on the Google Pixel 3.

With the first enhancement, the system checks the number of total cache accesses in the time period. It then needs to frequently check the clock to see if the time period expires. We find that the Android system only completes about 2,500 accesses per 4ms window, which is far lower than the original predicted value of about 16,000 accesses per 1ms window. Upon profiling the page, we discover that the vast majority of the code runtime is consumed by the `performance.now()` call to check whether the time window is elapsed. Since the ARM last-level caches are exclusive, the attack might have several issues if the cache occupancy system continually accesses the same beginning elements of the buffer without ever accessing the entirety of the buffer. In the worst case, if the number of accesses can fit in the L1 and L2 caches, the script may never impact the L3 cache, providing minimum useful information for the task of website fingerprinting.

We thus further employ two enhancements. The first enhancement accesses the buffer in a circular fashion: if the script only completes 2,500 accesses in the time window, it will access the 2,501st element at the beginning of the next window. It only returns to the first element once all elements have been visited. This ensures that the buffer eventually fills the L3 cache and that sequential observations cover different parts of the cache. We find that this technique increases the accuracy of the 10-site open world dataset to about 83%. The next enhancement is to decrease the amount of time that the script spends on checking the time. Instead of checking after every access, we check after every 20 cache accesses. This enhancement (without circular accesses) increases the accuracy to 84%. We then combine both enhancements and further increase the accuracy to 86%. We present a thorough evaluation in Section 6.

5.3 Novel GPU Channel

The DynamIQ CPU design not only adds the L3 shared cache among all of the processing cores within a cluster, but also allows for the L3 cache to be shared with any other peripherals contained within the SoC. This means that peripherals/accelerators like the Graphics Processing Unit (GPU), Digital Signal Processor (DSP), and Image Signal Processor (ISP) are all able to impact the shared cache. In particular, the GPU is heavily utilized to display a web page to users. Newer web browsers employ hardware acceleration when rendering and displaying web pages. Elements like HTML5 Canvas, WebGL or WebGL2 animations, and videos are also usually hardware accelerated. Thus, we endeavor to explore whether the GPU and shared cache architecture of current ARM DynamIQ can be exploit to create a website fingerprinting side channel.

It is challenging to construct a GPU cache occupancy channel. WebGL2 and basic HTML5 canvas elements only update at a low frequency of 60Hz. While these sampling rates can be increased, working with the canvas element in a background tab further increases the complexity and overhead. Also, it is not straightforward to determine the amount of memory that a GPU process consumes. GPU programming within JavaScript is mainly designed around graphical interfaces and smooth animations. An ideal attack should instead perform minimal useless image display, but focus primarily on exploiting the side channel. Therefore, we utilize a JavaScript library called GPU.js [16], which is designed to enable the creation and deployment of GPU computational kernels from JavaScript to WebGL compatible code. It can reduce the amount of boilerplate code and other timing elements for an attacker.

We thus create a two-dimensional buffer of data and repeatedly utilize the GPU to process this buffer with different mathematical kernels. Unlike our improved cache occupancy channel, accelerator based channels cannot provide us with high granularity measurements. The accelerator based workload requires that the CPU should first declare the work, pass it to the accelerator (GPU), and wait until the GPU completes its task. This means that the sizing and complexity of the kernel task must be tuned for the optimal fingerprinting performance.

To understand the performance of different settings, we create a spy script similar in nature to the cache occupancy spy script. The GPU script reports the number of kernel executions that it can complete in the monitoring time period. We conduct experiments using multiple kernels, including matrix multiplication and computing the dot product. We find that the kernel that sums each row of the input array delivers far superior performance. This might be due to massively decreased complexity and time in this GPU kernel: the reduced complexity enables more possible kernel executions, which in turn leads to better observability of GPU usage. We also check the optimal size for the computation. The time taken to compute a small kernel might provide minimally useful information as the GPU startup overhead would dominate the timing, while a large kernel would take too much time and decrease the observation granularity. We find that an overall array computation of between 20KB (Android) and 40KB (MacOS) organized into 5x4KB or 10x4KB arrays works best. Finally, we examine the observation window, but limit our experiments to a maximum 10 second duration to maintain a realistic approach. Again, we find disparate sizes depending on platforms. The Google Pixel 3 provides the best performance with 500 20ms observations and the M1 MacBook Air achieves its best results with 1,000 10ms observations. We believe this is caused by the speed of the processors: the SnapDragon 845 functions much slower and thus requires more time to manifest observable differences in computation performance as opposed to simply observing GPU overhead.

6 EVALUATION

In this section, we provide detailed performance results for the cache occupancy and GPU contention channels. Here we utilize a much larger dataset containing 100 accesses to 100 sensitive sites (Alexa Top 100), and 1 access to 5,000 other websites. We report both closed (only the sensitive websites) and open (all websites) world accuracy results. As before, to remove any bias from the experimentation, the collection process is conducted using Appium

or Selenium automation of the target platform. The list of 15,000 total website accesses is randomized to ensure that there are no unintentional ordering effects and the same random access order is utilized for each experiment for better comparison.

To compute the accuracy of the fingerprinting, we utilize 10-fold cross validation with a 90/10 train/test split. We report accuracy for two machine learning algorithms, a ridge regression with a minirocket [8] transform and a minirocket transform with a 1D CNN (configuration presented in Appendix Table 5). The ridge regression with minirocket transform is a recent advancement in time series machine learning and is able to achieve results close to those of the 1D CNN in less than a minute.

6.1 Web-Based Attack Results

Table 2 presents the accuracy of the web based cache occupancy fingerprinting experiments. Our approach can achieve promising results across all of the devices in the closed world scenario (i.e., 100 sensitive websites), with accuracy ranging from 80% to 95% when utilizing the ridge regression classifier.

The open world scenario (i.e., 100 sensitive websites and 5,000 non-sensitive websites) also demonstrates high accuracy. In the open world cases, we find that in most cases the 1D CNN performs better than the ridge regression classifier. This behavior is expected as the 1D CNN utilizes multiple convolutional and pooling layers to extract features from the dataset and learn both spatial and temporal patterns.

We notice that the cache occupancy channel performs the best on the Macbook Air, and the worst on the iPhone SE 2. This is likely related to the design of both the cache systems and schedulers. The CPU core designs in the MacBook Air are one generation newer, and the M1 chip is designed specifically for desktop/laptop workloads, and is likely tuned for multi-process scenarios. Also, the M1 chip contains features to prevent single cores from dominating the cache [12], and the A13 has been discovered to use part of the shared high performance L2 cache as an extra L2 cache for the low performance cores [11]. Apple also changes the amount of the cache that the high and low power cores have access to, depending on the DVFS states of the cores [11].

To analyze these effects, we conduct experiments with different buffer sizes. The Google Pixel 3 reports 2MB of shared cache, and we find that a 2MB buffer performs the best in the fingerprinting task. While the iPhone SE2 is unclear about the actual amount of shared cache provided to the low power cores, we find that a 4MB buffer performs the best in both tested configurations. Interestingly, this 4MB buffer seems to indicate that the cache occupancy channel is solely utilizing the L2 cache of the low power cores, potentially implying that Apple schedules foreground browser rendering processes to these low power cores or that the 'extra' L2 cache that is shared with the high performance cores is not exclusively owned by either core type. The Macbook Air, however, demonstrates vastly different behaviors. Specifically, we find that a 4MB buffer performs the best for Google Chrome, a 10MB buffer for Mozilla Firefox, and a 24MB buffer for Apple's Safari. As previously mentioned, these differences may be caused by a number of reasons, including different renderers and JavaScript engines. In general, attackers need to adjust attack strategies based on various factors to achieve high overall performance.

Another possible factor in the reduced performance of mobile devices vs. laptops could be the trend of websites to deliver different pages to different devices. When a laptop visits a website, it views the entire site that usually contains much more detailed content than the corresponding mobile website. The vastly simplified mobile websites may appear more similar to the cache occupancy channel, resulting in the decreased accuracy.

6.2 App-Based Attack Results

We next evaluate the performance of the cache occupancy channel if the attacker can run in a background process on the device. We continue to employ a 4ms sample period to provide the most fair comparison between the browser- and native-based channels and develop native applications for each platform to enable this testing.

We create applications for the iOS and Android systems featuring two processes, one drives a 'webview'[5] and another acts as the spy process. This method has been used to study native side channel performance in website fingerprinting before [36]. On the Macbook Air, a spy process written in C is launched alongside the web browser to monitor traffic. The results are listed in Table 3. Overall, for the Macbook, our method can achieve about 90% accuracy for the closed world dataset, and more than 80% accuracy for the open world scenario. For other devices, we can also achieve about 70% accuracy for the open world case.

We also notice an interesting trend, in all but the Firefox browser, the channel generally performs worse in the native setting. We infer that this reduced performance can potentially be caused by the idiosyncrasies of the OS scheduler. The scheduler of a mobile phone aims to provide the best performance to the foreground process and imposes strict limitations on background processes. On the other hand, the scheduler of a laptop/desktop should ensure more equal scheduling of background processes as they are important to user satisfaction (severely diminishing the performance of background file sync, application updates/installs, etc. would be unacceptable). MacOS, specifically, offers a number of different process priorities that have recently been shown to greatly affect which cores a specific task is executed on [38] and thus mixing native and web browser processes may result in unexpected scheduling. While the process in a background tab is very likely to end up on the low power cores, the native process may be scheduled on either core, depending on how the operating system interprets its priority/whether it is a user-facing process.

6.3 Comparison to Prior Work

The cache occupancy channel has been studied from the perspective of website fingerprinting attacks before, however, those attacks utilize variable timing windows and gather data over the course of 30 seconds [49, 50]. Instead, our work improves upon both the amount of data and attack duration dramatically, utilizing 4ms attack periods over the course of only 8 seconds. Reducing the total attack time by 75% improves the practicality of the attack, as it is

[5]Both iOS and Android provide a mechanism called a webview to display web content to users within an application. The webview functions as a web browser without the navigation controls. Both iOS and Android webview components are nearly identical to the system web browser.

Table 2: Accuracy for web-based cache occupancy website fingerprint on multiple ARM devices

Device	CPU	Browser	Closed World		Open World	
			Ridge Regression	CNN	Ridge Regression	CNN
Macbook Air	Apple M1	Chrome 89	95.6	92.2	88.1	89.8
Macbook Air	Apple M1	Safari 14	94.3	89.4	78.4	85.1
Macbook Air	Apple M1	Firefox 88	88.1	83.9	68.2	77.8
iPhone SE 2	Apple A13	Safari 14	80.2	75.3	65.8	72.7
iPhone SE 2	Apple A13	Chrome 87	80.2	75.9	65.0	73.3
Google Pixel 3	Snapdragon 845	Chrome 90	88.0	81.8	66.0	75.9

Table 3: Accuracy for native application cache occupancy website fingerprint on multiple ARM devices

Device	CPU	Browser	Closed World		Open World	
			Ridge Regression	CNN	Ridge Regression	CNN
Macbook Air	Apple M1	Chrome 89	92.5	85.7	84.1	84.3
Macbook Air	Apple M1	Safari 14	91.1	87.0	72.4	81.7
Macbook Air	Apple M1	Firefox 88	90.3	87.1	70.5	81.3
iPhone SE 2	Apple A13	WebKit View	71.5	68.7	64.0	69.1
Google Pixel 3	Snapdragon 845	WebView	81.9	76.3	67.7	74.1

Table 4: Accuracy for GPU based website fingerprinting on ARM devices

Device	GPU	Browser	Closed World		Open World	
			Ridge Regression	CNN	Ridge Regression	CNN
Macbook Air	Apple 7 Core	Chrome 89	90.5	85.3	76.6	81.4
Google Pixel 3	Adreno 630	Chrome 89	88.2	82.6	67.6	77.3

unlikely that a user will navigate to a page and not interact with it for 30 seconds. Furthermore, this work conducts the most extensive study of the cache occupancy channel on ARM to date, examining both native and web based attacks, providing an in depth discussion of cross-platform accuracy enhancements. We also study multiple MacOS and iOS browsers and this is the first work to explore such a cache cache occupancy channel on iOS.

The only direct comparison that can be made is the performance of the closed world attack for the Chrome browser on the M1 chip, wherein this work performs 6.5% better in Top-1 accuracy than the work in [49]. Our Android performance is also 4.1% better (also Top-1 accuracy) in the closed world setting, though the devices are different.

We also compare to previous works done on homogeneous x86 systems like those in [50]. Our work, with the optimizations developed for the ARM architecture, achieves better results. The performance of our open world attack on Safari is 5.7% better than their best neural network configuration, and the closed world attack is 29.9% better (Top-1 accuracy). One item that complicates comparison to [50] is their open world data. Their work claims 99% accuracy in delineating between a sensitive and non-sensitive website, which could indicate significant differences between the open and closed world datasets. By contrast, our work combines and randomizes the order of the collection of the open and closed world datasets to ensure that there are no cross-sample ordering artifacts that might artificially increase accuracy.

6.4 GPU Channel Results

We utilize the same testing setup as the cache occupancy channel to evaluate the GPU contention channel. We only modify the spy process to utilize the GPU as opposed to the CPU. While all major browsers support web workers, only Google Chrome on Android and MacOS allowed for unrestricted access to the GPU in a background web worker via GPU.js, thus limiting our experiments to these two platforms. The results are listed in Table 4. Overall, our novel GPU channel can achieve similar results as the standard cache occupancy channel. On MacOS, it can still achieve more than 90% accuracy on the closed world case, and more than 81% accuracy for the open world one. The accuracy on Android is even better: it lightly improves the accuracy in both the open and closed world scenarios compared with the standard cache occupancy channel. This difference may be related to the different system architectures that make up the Adreno GPU vs. the Apple designed GPU found on the M1 chip.

These results strongly indicate that the GPU/cache contention channel is capable of mounting website fingerprinting attacks and should serve as a red flag to device manufacturers. As more accelerators are tightly integrated into the standard ARM SoC and web technologies rush to enable access (e.g., WebGPU [52]), special attention should be taken to ensure that these additions do not jeopardize user privacy.

6.5 Countermeasures

There are several approaches to potentially protect an ARM system from those contention based side channels. For example, the system can introduce noise to the measurement channel via extra operations, or manipulate timers and array accesses via obfuscation such as in Chrome Zero [45]. However, introducing extra noise has been shown ineffective [50] and leads to increased energy usage, which is unacceptable for mobile devices. Also, Shusterman et al. [49] demonstrated that the protections of Chrome Zero is largely ineffective and impose significant performance penalties. Furthermore, browser based defenses cannot thwart App-based attacks.

Another defensive approach for energy restricted devices is to remove process contention via hardware segmentation. This can guarantee that the processes are unable to interact with one another. However, it requires complete redesigns of the operating system scheduler and hardware. In our future work, we plan to develop effective defensive solutions to detect significant contention and large swings in cache occupancy (similar to [2]) for ARM devices.

7 RELATED WORK

In this section, we briefly survey the research efforts in related areas. Specifically, we conduct a detailed comparison with previous cache occupancy fingerprinting techniques.

Cache Occupancy. The most similar work is Shusterman's cache occupancy fingerprinting work [49, 50]. Their work is the first attempt to exploit a cache occupancy channel for website fingerprinting. While previous work [18] examined the individual eviction sets, these fine-grained attacks can be mitigated by modern browsers by limiting time resolution. Shusterman et al. [50] proposed that the contention of the entire cache may provide enough information to fingerprint websites within the x86 platform. In parallel to our work, Shusterman et al. [49] performed a cursory investigation of the cache occupancy channel on the Apple M1 and a Samsung S21 with the Chrome browser in a closed world scenario.

Our work provides a much deeper investigation of the cache occupancy channel in ARM devices. In addition to Android and MacOS, we also study the iOS platform. Furthermore, our approach differs from Shusterman's in that we develop a vastly different method for cache accesses (Section 5.2), which increases accuracy on budget devices with slower processors. We also study the effect of different browsers and their memory management, demonstrating that simply sizing the eviction buffer based on the shared cache provides suboptimal results in different browser engines on the same hardware (Section 6.1). Besides, we increase the attack effectiveness, utilizing only 8 seconds of observation time to identify a website unlike the previously required 30 seconds in both [49, 50]. Even with nearly 75% less sampling time, our approach outperforms the Shusterman's work by more than 6% in Top-1 accuracy in testing on the M1 MacBook Air with Google Chrome. Finally, we propose and evaluate the novel GPU based contention channel and demonstrate that it is nearly as effective as the cache occupancy channel in ARM SoC devices, raising the alarm on continued access to SoC accelerator components from JavaScript.

Website Fingerprinting Website fingerprinting has long been used to track user web surfing behaviors. As desktop browsers are the original way to surf the web, many website fingerprinting techniques focus on breaking privacy enhancing technologies like HTTPS and ToR, which leverage features extracted from the packet streams [4, 14, 20, 41, 43]. With the rise of mobile devices, more efforts have been spent examining mobile web surfing. MagneticSpy [36] examines both JavaScript and app based CPU activity channels by employing the magnetometer. They perform similar open and closed world investigations (albeit with fewer websites), and demonstrate high fingerprinting accuracy. However, the JavaScript APIs that allow access to these sensors have since been removed from support in Firefox and Safari [48]. Furthermore, iOS requires that users explicitly grant permissions to a website before it is allowed to access their accelerometer data [24]. Several previous works [26, 53] explore power based website fingerprinting on smartphones, however they require much higher sampling frequencies and cannot perform the fingerprinting from a JavaScript platform. Jana et al. [21] studied the memory allocations of website traffic, but required privileged access to process memory data (now removed from standard user access). Spreitzer et al. [51] utilized the data usage API within Android to fingerprint websites, but this must be done from a native application.

ARM Attacks Gulmezoglu et al. [18] built a similar contention based channel in ARM devices, but mainly focused on finding contention among specific sets within the cache ways of the device. Their attack is limited to the Google Pixel 5, and only utilizes native APIs within the system. While the work presents impressive results, their system relies upon identifying eviction sets within the cache. With a high resolution timer, this can take a few seconds; however, the low resolution timer available from JavaScript [42] would make the time cost of the task prohibitively long. Lipp et al. [28] and Gruss et al. [17] similarly constructed memory based JavaScript attacks, but require either privileged system calls or higher resolution timers than those currently available in modern browsers [42].

Timing Attacks from JavaScript Genkin et al. [13] executed encryption side channel attacks from a browser but utilized web assembly and shared array buffers to construct a high frequency timer. Oren et al. [39] similarly demonstrated that eviction sets could be created via JavaScript timers for website fingerprinting (not on ARM). Bosman et al. [3] demonstrated page deduplication attacks from JavaScript. Each of these attacks requires high resolution timers that have since been removed from JavaScript [42]. Schwarz et al. [46] demonstrated some interesting methods to achieve high resolution timing, but many of these techniques have been disabled or hindered within major browsers.

GPU Attacks Lee et al. [25] proposed to exploit the shared memory within the GPU for website fingerprinting. Frigo et al. [10] executed several side channel attacks from a mobile GPU. However, these attacks require timing primitives that have been removed. He et al. [19] uncovered a register leakage within Intel GPUs and exploited it to identify websites. Naghibijouybari et al. [37] utilized GPU memory allocation APIs within CUDA or OpenGL to track memory allocations and fingerprint websites. They did not explore ARM integrated GPUs or execution from a JavaScript environment and instead employed a spy program that ran as a native process with full access to CUDA/OpenGL. Karimi et al. [22] proposed a

side channel attack against an ARM SoC GPU and extracted AES keys by exploiting cache behaviors; however, the attack requires a long execution time and a stable system that does not run other tasks. Moreover, the study was not conducted from a JavaScript perspective.

8 CONCLUSION

This paper investigates whether the new ARM DynamIQ system design, specifically the inclusion of a shared last-level cache between all CPU cores and accelerators, poses a security threat to individuals. We examine the information leakage in the context of a website fingerprinting attack, demonstrating that a cache occupancy side channel can be constructed to reliably fingerprint user website activities. We reveal this security threat on Android, iOS, and MacOS, delving into how the channel responds to different browser environments and proposing enhancements over previous works. In addition, we unveil an accelerator based website fingerprinting channel, showing that the SoC GPU can be exploited in a contention based side channel from JavaScript. Our evaluation results indicate that both channels can achieve high website fingerprinting accuracy on different browsers in Android, iOS, and MacOS systems in both open and closed world scenarios.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful and constructive comments, which helped us to improve the quality of this paper. This work was supported in part by the National Science Foundation (NSF) grants DGE-1821744 and CNS-2054657 and the Office of Navy Research (ONR) grant N00014-20-1-2153.

REFERENCES

[1] appium [n. d.]. Appium. https://github.com/appium/appium.

[2] M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J. Menaud. 2018. Cache-based side-channel attacks detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In *Third International Conference on Fog and Mobile Edge Computing*.

[3] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy (SP)*.

[4] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. 2012. Touching from a distance: Website fingerprinting attacks and defenses. In *Computer and Communications Security (CCS)*.

[5] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*.

[6] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The last mile: An empirical study of timing channels on seL4. In *Computer and Communications Security (CCS)*.

[7] Patrick Cronin and Chengmo Yang. 2019. A Fetching Tale: Covert Communication with the Hardware Prefetcher. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[8] Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. 2020. MINIROCKET: A Very Fast (Almost) Deterministic Transform for Time Series Classification. *arXiv:2012.08791* (2020).

[9] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 414–432. https://doi.org/10.1109/SP.2016.32

[10] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy (SP)*.

[11] Andrei Frumusanu. 2019. The Apple iPhone 11, 11 Pro & 11 Pro Max Review: Performance, Battery, & Camera Elevated. https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/3.

[12] Andrei Frumusanu. 2020. The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test. https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested.

[13] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *Applied Cryptography and Network Security*, Bart Preneel and Frederik Vercauteren (Eds.). Springer International Publishing.

[14] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. 2012. Website Detection Using Remote Traffic Analysis. In *Privacy Enhancing Technologies Symposium (PETS)*.

[15] gpudirect 2021. GPUDirect. https://developer.nvidia.com/gpudirect.

[16] gpujs [n. d.]. GPU.js. https://github.com/gpujs/gpu.js.

[17] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *Computer Security – ESORICS 2015*, Günther Pernul, Peter Y A Ryan, and Edgar Weippl (Eds.). Springer International Publishing.

[18] Berk Gulmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining User Privacy on Mobile Devices Using AI. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. Association for Computing Machinery.

[19] Wenjian HE, Wei Zhang, Sharad Sinha, and Sanjeev Das. 2020. IGPU Leak: An Information Leakage Vulnerability on Intel Integrated GPU. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[20] Andrew Hintz. 2002. Fingerprinting websites using traffic analysis. In *Workshop on Privacy Enhancing Technologies*.

[21] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy (SP)*.

[22] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2018. A Timing Side-Channel Attack on a Mobile GPU. In *IEEE 36th International Conference on Computer Design (ICCD)*.

[23] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P)*.

[24] Andy Kong. 2020. Accessing the iPhone Accelerometer with Javascript in iOS 14 and 13. https://kongmunist.medium.com/accessing-the-iphone-accelerometer-with-javascript-in-ios-14-and-13-e146d18bb175.

[25] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE Symposium on Security and Privacy*.

[26] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. 2018. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies* (2018).

[27] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets. In *21st Annual Network and Distributed System Security Symposium, NDSS*.

[28] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security*. USENIX Association.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[30] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*.

[31] Arm Ltd. [n. d.]. big.LITTLE. https://www.arm.com/why-arm/technologies/big-little.

[32] ARM Ltd. [n. d.]. Cache Stashing. https://developer.arm.com/documentation/100453/0401/functional-description/l3-cache/cache-stashing.

[33] ARM Ltd. [n. d.]. Cortex A-55. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a55.

[34] Arm Ltd. [n. d.]. DynamIQ. https://www.arm.com/why-arm/technologies/dynamiq.

[35] ARM Ltd. [n. d.]. L3 Cache Allocation Policy. https://developer.arm.com/documentation/100453/0002/functional-description/l3-cache/l3-cache-allocation-policy.

[36] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. 2019. MagneticSpy: Exploiting Magnetometer in Mobile Devices for Website and Application Fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*. Association for Computing Machinery.

[37] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

Association for Computing Machinery.
[38] Howard Oakley. 2021. How M1 Macs feel faster than Intel models: it's about QoS. https://eclecticlight.co/2021/05/17/how-m1-macs-feel-faster-than-intel-models-its-about-qos/.
[39] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery.
[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, David Pointcheval (Ed.). Springer Berlin Heidelberg.
[41] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. 2016. Website Fingerprinting at Internet Scale. In *Network and Distributed Systems Symposium (NDSS)*.
[42] Filip Pizlo. 2018. What Spectre and Meltdown Mean For WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.
[43] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. 2018. Automated website fingerprinting through deep learning. In *Network and Distributed Systems Symposium (NDSS)*.
[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Computer and Communications Security (CCS)*.
[45] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *Network and Distributed System Security Symposium*.
[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Aggelos Kiayias (Ed.). Springer International Publishing.
[47] selenium [n. d.]. Selenium. https://github.com/SeleniumHQ/selenium.
[48] sensor [n. d.]. Sensor - Web APIs: MDN. https://developer.mozilla.org/en-US/docs/Web/API/Sensor.
[49] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium (USENIX Security)*.
[50] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. https://www.usenix.org/conference/usenixsecurity19/presentation/shusterman
[51] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. 2016. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '16)*.
[52] webgpu [n. d.]. Feature: WebGPU. https://www.chromestatus.com/feature/6213121689518080.
[53] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S Balagani. 2016. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Transactions on Information Forensics and Security* (2016).
[54] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *25th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[55] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*.

Appendices

Table 5: 1D Convolutional Neural Network Configuration

Fingerprinting Classification Network		
Layer	Operation	Kernel Size
1	Input	10000x1
2	Convolution	256x8
3	MaxPool	8
4	Convolution	256x8
5	MaxPool	8
6	Convolution	256x8
7	MaxPool	8
8	Flatten	-
10	Dropout	0.2
11	Dense	Number of Classes