A Survey of Cache Attacks on Mobile Devices

Matthias Grabner
Freie Universität Berlin
14195 Berlin, Germany
m.grabner@fu-berlin.de

***Abstract*—As processors have evolved over time, so have cache attacks. Today's smartphones mostly utilize multi-core ARM processors, which can have diverse architectures. These different architectures can pose unique challenges to cache attacks. If successful, this form of side-channel attacks can lead to a loss of privacy or even break encryptions. In this paper, we present the different types of attacks available, what architectures they affect and how attackers can deal with potential design variations in ARM processors. We also discuss what attacks can be executed on which type of cache organization, and lastly include a discussion and summary.**

***Index Terms*—Security, Cache Attacks, Mobile Devices, ARM**

I. Introduction & Context

Cache attacks are a form of side-channel attacks that exploit the architecture of processing units in order to access information about the execution of another program, which should not be accessible by malicious programs. Examples of information obtained through such attacks include mouse movements [1], keystrokes [1] and cryptographic keys [2], [3]. They can achieve this, by monitoring the specific memory areas they suspect the program to be accessing, which enables a plethora of conclusions about the activity of the victim program. This is made possible due to multiple programs sharing the same cache or different caches connected with cache coherence protocols. Further elaborations on how various attacks that fall in the category of cache attacks work, are given in section IV.

Cache attacks have been successfully demonstrated on both desktop environments [4] as well as mobile devices [5]. It has been shown that such attacks work under various architectures and parameters, making them a potential threat for almost all devices using some sort of shared cache [2]. Still, while cache attacks on Intel CPUs have been thoroughly researched in the past, attacks on ARM processors have not been researched on a larger scale until 2016 [5]. We were not able to locate a survey discussing the different types of cache attacks on mobile devices. Therefore, this paper will try to do just that and offer an overview of cache attacks, with a focus on mobile devices. In the last 20-years, an increase in CPU complexity has also resulted in more complex cache attacks. It is vital to understand these simpler attack patterns such as Prime+Probe and how single-core CPUs can be attacked, to understand more advanced attacks such as Invalidate+Transfer.

With over 90% of U.S. adults owning a smartphone, attacks on a fundamental part of these devices could present a serious security threat to a large group of people [6]. Moreover, the feasibility of automated cache attacks further amplifies this risk [7], making it easier for attackers to exploit vulnerabilities at scale, potentially affecting millions of users.

The threat model under which cache attacks work, does not require root access, although root access can make the attack easier. Additionally, root access by the malicious program can enable online attacks, which then do not require preparation or external computations [5]. With a few exceptions, whether cache attacks can be successfully executed, solely depends on the processor architectures. This results in Android ROMs, such as the security focused Lineage OS, being potentially vulnerable to this type of attack as well [8]. With this few limitations, many devices have been shown to be vulnerable to cache attacks in the past.

First, an overview of related work regarding cache attacks on mobile devices will be given. Then, preliminaries will be explained and the general types of cache attacks will be discussed. Afterwards, various problems that arise when executing cache attacks on the ARM architecture will be explored and existing solutions discussed.

II. Related Work

The work related to cache attacks on mobile devices can be roughly split into two groups: work discussing cache attacks on Intel platforms that may have application on ARM architectures, and work discussing cache attacks with the ARM architecture already in mind. For the purpose of this paper, works from the second category are of particular importance.

The work by Lipp, et al. shows that ARM architectures too are susceptible to many of the attacks previously discovered against Intel processors [5]. The authors experimentally show this by testing a OnePlus One, an Alcatel One Touch Pop 2 and a Samsung Galaxy S6. Existing methods for attacks against the Intel processors are modified, and the results are benchmarked by measuring cache covert channels.

Yarom and Falkner present a deep dive into the Flush+Reload attack on the Intel architecture [4]. Page sharing, RSA and fundamental cache architecture are explained and an implementation of Flush+Reload to attack GnuPG is shown, as well as potential mitigation techniques. Arguably, this attack is the most popular form of a cache attack [9] and is relevant to understand the extensions for ARM CPU's by Lipp, et al. [5].

Irazoqui et al. present an alternative attack on multiprocessor systems using CPUs with non-inclusive last-level caches, such as CPUs by AMD and ARM, called Invalidate+Transfer. As the attack relies on ARM's AMBA protocol, which is used for both cross-core and cross-cpu communication, the attacks should also translate to the multi-core systems found in smartphones [2]. Therefore, this type of attack is highly relevant for cache attacks on mobile devices and was also experimentally tested against the AES and ElGamal encryption systems on a system running four *AMD Opteron 6168* CPUs [2].

Green, et al. highlight the previously overlooked AutoLock feature of ARM processors adversely affecting Evict+Time, Prime+Probe and Evict+Reload attacks. According to the authors, this not publicly documented security measure against cache attacks is built into an unknown number of devices [10].

Su and Zeng provide a broad survey of cache-based attacks. Although manufacturer-specific countermeasures against cache-attacks are discussed, no implementations of the attack on Intel or ARM processors are shown [9]. The attacks are categorized, trends are discussed and theoretical defenses roughly analyzed.

III. Preliminaries

First, some preliminary knowledge will be given that is required to understand how cache attacks function on the ARM architecture.

Caches have the purpose of bridging the gap between fast computation and the comparatively slow main memory. They are small, fast and are supposed to store data that will be needed in the near future. To achieve this, memory is loaded from the main memory into the respective cache according to spatial and temporal conditions.

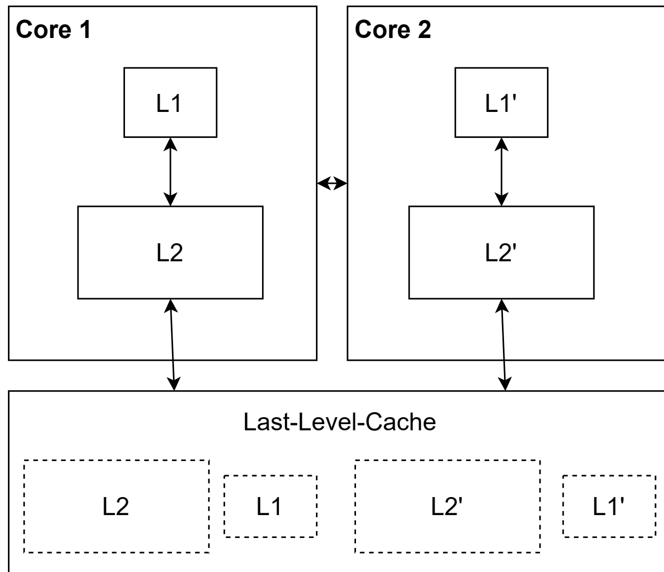

Fig. 1. Example of a CPU utilizing an Inclusive and Shared LLC. The dotted lines mark copies of the contents of the higher level caches.

A. Cache Hierarchy

Each CPU typically has multiple caches that differ from each other in size, speed and function. A layer 1 (L1) cache represents the highest cache level with only a very small amount of memory. The Cortex-A720 featured in the Snapdragon 8 Gen3 SoC, for example, features a 32KB or 64KB layer 1 cache [11]. Ideally, the L1 cache will bear a very high load of memory accesses due to its fast speed, despite its small size.

The size of the layer 2 (L2) cache is larger than that of the L1 cache, but also slower. It is the job of the L2 cache to store data that would not fit in the L1 cache, but is likely still needed for execution of the program. Also, it is responsible for bridging the gap between the main memory and the L1 cache, should no layer 3 cache exist. Recent trends show manufacturers greatly increasing L2 size, in hope of granting a boost to applications needing large amounts of code in a short period of time [12].

A layer 3 cache is often optional on processors such as the Cortex-A720 and can vary in size greatly. The Cortex-A720 for example, offers sizes between 256KB and 32MB [11]. How cache sizes should be distributed among the different levels to ensure optimal efficiency, is still actively researched [12].

B. Inclusiveness

Whether a cache is inclusive of another cache or not is defined by whether data in one cache, is also guaranteed to be in the other cache. In ARM CPUs, the caches can either be inclusive, only instruction-inclusive, only data-inclusive or non-inclusive [5]. This can be the case for various levels and not only the LLC. An example would be inclusive L2 caches, including memory from the L1.

Su and Zeng formally define this property as follows: assume m is a unit of memory and the contents of the respective cache levels are represented by L1, L2 and LLC. Then the following statements hold if a cache is inclusive throughout all levels

$$m \in L1 \longrightarrow m \in L2 \longrightarrow m \in LLC \tag{1}$$

Additionally, it is ensured that if memory is evicted from the LLC, it is also evicted from the higher levels [9]:

$$m \notin LLC \longrightarrow m \notin L2 \longrightarrow m \notin L1 \tag{2}$$

Caches can also be exclusive, ensuring that a block of memory isn't present in any of the lower levels if it is cached in any of the higher cache levels, resulting in formal definition 3:

$$m \in L1 \longrightarrow m \notin L2 \longrightarrow m \notin LLC \tag{3}$$

What design choice is made by the manufacturers depends on a variety of factors further outlined by Nori, et al. [12].

C. Cache Coherence

To make use of caches and the data contained within them, it has to be assured that we always know if their data is valid and up to date. As multiple caches can keep a local "copy"

of data that is also present in the main memory, they could also keep different values for the same memory location with no additional precautions. It is the job of cache coherence protocols to ensure that no stale values are read and interpreted as valid. Most ARM processors use either the MESI or the MOESI protocol, meaning the cache lines can be in one of the following states: Modified, Owned, Exclusive, Shared or Invalid. More information on how these protocols work exactly can be found in the documentation provided by ARM [13].

D. Replacement Policy

The replacement policy of a cache dictates what cache lines should be replaced if new memory lines are loaded into a full cache. A simple example of such a replacement policy would be least-recently-used (LRU), which evicts the cache lines with the longest "idle time". However, modern processors use increasingly complex strategies such as pseudo-random policies [5]. This means that a function suggests a pseudo-random cache line to be evicted based on certain input parameters.

IV. Types of Attacks

Various types of attacks have been documented that all allow the spy to infer information about activity of the victim program. The attacks can be categorized by the way in which they remove cache content, resulting in two types of attacks: flush-based and eviction-based attacks. Below, discussions of the most common types of attacks can be found.

A. Evict+Time

Evict+Time works by measuring the execution time of the victim program, evicting a cache-set and measuring the execution time of the victim program again. The eviction is achieved by accessing memory blocks in the attacker's address space, which are mapped to the cache-set to be evicted. As the relevant data was likely cached when we timed the execution in the first step, a longer execution of the victim program in step three means that the data had to be fetched from a lower-level cache or even the main memory. Therefore, if the execution time of the victim program has changed significantly, we can infer that a region of memory mapped to the cache-set was accessed by the victim program [5]. Still, we require an efficient eviction strategy, which is inherently connected with the replacement policy of the CPU. While this isn't a problem for a simple LRU policy, it can present a significant problem if the CPU implements a pseudo-random replacement policy, as discussed in section V-D.

It should be noted though, that this attack only achieves a rough granularity, as cache-sets are targeted instead of single cache-lines, as in the Flush+Reload attack explained in section IV-C. Still, Evict+Time attacks have been used to successfully extract the secret key when attacking OpenSSL in about half a minute [14] and have been used on the Android to attack disaligned AES T-Tables [3]. Even though the attack on OpenSSL was much slower than when using Prime+Probe, it still presents a viable way of gaining information about a victim program.

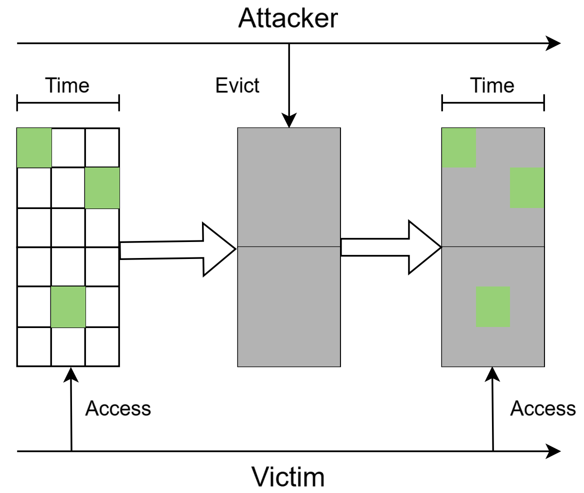

Fig. 2. Flow of the Evict+Time attack

.

B. Prime+Probe

Prime+Probe can be imagined as the inverse of Evict+Time. In this type of attack, the attacker occupies specific cache-sets and checks which cache sets are still occupied by their own data, after the victim program is executed. The important difference to Evict+Time is, that for Prime+Probe, the attacker measures the execution time of their own process, rather than the execution time of a potentially complex victim program. This helps to loosen the restrictions the timing problem places on the accuracy of timers, discussed in section V-F [14]. Still, only cache-sets are occupied, again resulting in a rough granularity, although allowing for a faster execution than with Evict+Time [14]. However, for cross-core attacks using the LLC, priming and probing the whole LLC is infeasible due to its large size, and special preparations are required to identify relevant cache sets and only monitor those [15].

C. Flush+Reload

The Flush+Reload attack is an advanced variant of the Prime+Probe attack, that focuses on specific cache-lines and achieves a much finer granularity than the other types of attacks discussed in section IV-A and IV-B. To achieve this, Flush+Reload uses the page-sharing mechanism [4]. Many operating systems use page-sharing as a way to save memory by letting multiple processes use the same page (e.g a shared library). However, to not influence all the other processes using the same page, if an edit occurs the page is copied and the copy is edited. The action of copying introduces a delay that can be measured and is exploited in the Flush+Reload attack.

If the respective processor supports a flush instruction, the attacker first flushes a memory line and then waits for a pre-defined amount of time [5]. If the attacker waits longer, the granularity of the attack is reduced, if the attacker continues with phase three faster, they could miss the access of the victim program. Next, the attacker reloads the cache-line and

measures the time it takes to load [4]. A shorter loading time implies that the victim has accessed the memory contained in the cache-line. A sample attack-flow for this attack can be found in Figure 3. Workarounds for cases where the processor doesn't support flush instructions can be found in section V-C.

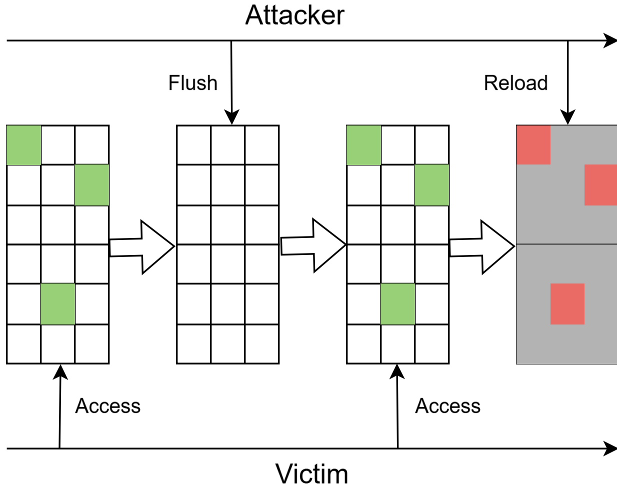

Fig. 3. Flow of the Flush+Reload attack, adapted from Su and Zeng [9]
.

D. Evict+Reload

Evict+Reload is similar to Flush+Reload, although this attack is typically used if spy and victim do not share memory [9]. For Evict+Reload the spy loads data into the cache to evict the whole cache-set by finding data that would map to the respective cache-lines. After the victim accesses the memory, the attacker reloads the data and checks which accesses take longer [9]. Similarly to Evict+Time however, we also require an efficient eviction strategy.

E. Flush+Flush

Flush+Flush relies on timing the flush instruction, which experiences a slight timing variance when accessing cached versus not cached data [9]. Of course, this attack is only applicable to ARM CPUs offering flush instructions in their instruction set, such as the Cortex-A53 and Cortex-A57 in the Samsung Galaxy S6 [5]. However, as it relies on the execution time of instructions, it can be quite imprecise and does not necessarily offer an advantage in comparison to similar attacks discussed above [9].

F. Invalidate+Transfer

This attack type does not rely on an inclusive LLC or any other feature of a cache hierarchy and achieves a similar granularity to Flush+Reload. As shown in section V-A, this can be especially relevant for some ARM processors. Instead of inclusive LLC's, the Invalidate+Transfer attack relies on directory-based coherence protocols between different CPUs [2]. Specifically, if data is accessed in the cache of one CPU that was invalidated by another CPU, the data is fetched from the invalidating CPU. This results in a timing difference between fetching invalidated data and fetching not invalidated data. To exploit this difference, the Invalidate+Transfer attack first invalidates the data present in their own CPU's cache. If the data is present in any other CPU's cache, it is also invalidated there using the AMBA Cache Coherent Interconnect protocol. After a period of waiting, the attacker tries to access the invalidated memory again [2]. If the memory block was used by the victim during the wait period, the data can be retrieved measurably faster from the cache on the CPU of the victim rather than if it has to be fetched from the main memory.

In their original paper on the Invalidate+Transfer attack *Cross Processor Cache attacks*, Irazoqui et al. only focus on AMD's HyperTransport and on Intel's QuickPath Interconnect technology in specific, but claim that this attack can be applied to ARM's AMBA technology. This is due to AMBA largely working in the same way as AMD's HyperTransport and also using a directory-based coherence protocol [2]. In fact, although not under the same name, Lipp et al. exploit this attack in circumventing non-inclusive LLC's in their paper *ARMageddon: Cache Attacks on Mobile Devices* [5].

V. Potential ARM Design Variations

Several problems are outlined in the existing literature regarding cache attacks, some of which specifically apply to ARM processor architectures which are used in mobile devices. Due to the ARM architectures not being as homogenous as Intel's architectures [5], not all processors may be affected by some of the following problems. However, some problems are inherent to cache attacks across all devices, such as the timing problem [5] discussed in section V-F. Figure 1 shows an example of a cache hierarchy on the ARM architecture. In this example, the CPU has two cores, with their own L1 and L2 cache, but a shared Last-Level-Cache (LLC). Additionally, the LLC is inclusive, meaning it includes the data from the other cache levels, but uses slower memory. However, the LLC does not have to be inclusive, which can present a serious problem for the Flush+Reload attack [5].

A. LLC not being inclusive

While inclusive caches can be attacked with Flush+Reload and it's variants, the other partially non-inclusive architectures require the Invalidate+Transfer method for cross-core attacks. This variation in cache architecture compared to Intel CPUs can therefore be solved by using a different method of attack [9], which has been experimentally proven to work on smartphones containing ARM CPUs [5].

B. Cross-Core Architecture

Multi-core architecture is inherent to almost all of today's smartphones and can thus require cross-core attacks, if attacker and victim don't reside on the same core. Same core attacks just require the attacker to monitor memory in a section that they already have access to. Same core attacks were already

quite well known by 2006, when attacks focused on CPUs such as the single-core *Intel Pentium 4* [16].

The challenge with cross-core attacks is that the caches of the other core have to be monitored and therefore influenced somehow. In the case of inclusive LLCs this is relatively straight forward by removing the desired data from the LLC. For non-inclusive LLCs the method shown above can be used. The first instance of such a cross-core attack on mobile devices was shown by Lipp, et al. [5]. The authors showed that Evict+Time, Prime+Probe, Flush+Reload (and, although not named Invalidate+Transfer) could be successfully executed on cross-core smartphone SoCs.

C. Unsupported Flush Instructions

Only some ARM processors feature flush instructions (such as the Coretex-A57 on the Exynos 7 Octa 7420 SoC [5]), while others (such as the Krait 400 on the Snapdragon 801) don't. Therefore, flush-based attacks Flush+Reload and Flush+Flush can not be executed on these devices and alternative attacks have to be considered. However, for these eviction-based attacks to work, evictions have to be executed efficiently. How evictions can be performed depends in large parts on the replacement policy used.

D. Replacement Policy

The replacement policy of modern processors can also present a novel challenge for attackers trying to evict data from caches. This replacement policy can be pseudo-random, but was shown to be defeatable. For it to be defeatable, the attacker has to access physical address mappings to compute eviction sets that are able to evict the desired data with a relatively high chance [17]. However, while this was still possible in 2016 when the paper by Lipp, et al. was released, this is not possible any more on recent Android versions, without root privileges [18].

Still, defeating the replacement policy is only a necessity in the case of unsupported flush instructions.

E. AutoLock

For inclusive ARM caches, cache-lines throughout the cache hierarchy can be evicted by simply evicting them from the LLC. Due to cache coherence protocols, the cache line will in turn be evicted from the higher cache levels as well. This is a mechanism exploited in the eviction strategy for cross-core cache attacks. However, AutoLock prevents this by protecting cache lines in the LLC that are also present in higher-level caches. However, AutoLock does not seem to have an effect on flush based attacks and same-core attacks and can thus be circumvented by creating a situation where such attacks are applicable [10]. Whether AutoLock is implemented in a specific chip is not publicly documented and has to be tested for.

F. The Timing Problem

All cache attacks listed above require some sort of timer to determine whether data was fetched from the main memory or from the cache. As ARM CPUs do not offer exact timers without root permission, this presents a challenge that can be solved through various means. Sample solutions that work adequately well on the ARM CPUs tested by Lipp, et al. are unprivileged syscalls, the POSIX function `clock_gettime()` and a dedicated thread timer [5]. While the availability of the unprivileged syscall `perf_event_open()` depends on the kernel configuration of the Android device, the POSIX function and thread timer work regardless. The experimental results from Lipp, et al. show the counter finding a time difference between cache hits and misses to be about 20 times smaller, the POSIX function about seven times smaller, and the syscall about four times smaller than the timing difference between hits and misses suggested by the cycle count register. Still, the same resolution is maintained according to the authors, meaning the same amounts of hits and misses could be differentiated [5].

As this challenge is practical, and also depends on if CPUs allow unprivileged programs to count cycles, this challenge is not discussed in the survey paper we found [9].

VI. Discussion

In this paper, we tried to summarize what cache attacks exist and which cache organizations they threaten. As is the case for a survey paper, this was solely based on existing work, and we did not try to establish new or additional hypothesis. However, an implication of this summary would be, that it can hopefully serve as a valuable introduction to cache attacks with a focus on mobile devices. To get an overview of the work primarily used in the survey, we briefly summarized it in the beginning. The related work was also contextualized with the work from other authors, and slight discrepancies were discovered. One of these was Lipp, et al. not mentioning the Invalidate+Transfer attack by name, but simply referring to it as a modified version of the Flush+Reload attack [5].

However, this survey has several limitations and only scratched the surface of the material available on cache attacks. This was due to both the time restraints and the limited scale of seminar papers. Although some parts of the structure of the survey were adapted from the paper by Lipp, et al., it presents attacks not mentioned at all in the base paper, novel challenges (e.g the AutoLock feature), and offers a broader scope. It is also written to promote understanding of the topic at the undergraduate level and features several graphics which present concepts visually.

As attacks on ARM only present a small part of the research on cache attacks, it may be valuable to analyse existing research of attacks on Intel or AMD processors. This paper mostly focused on ARM, and we did not try to draw new conclusions about attacks on ARM from attacks on other architectures.

There may also be new design variations or security features introduced in new ARM processors that quickly make, or have already made this survey, redundant.

VII. CONCLUSION

In conclusion, cache attacks present a viable way of executing side-channel attacks with little privileges on many devices. We have discussed the six types of cache attacks and have shown what prerequisites are required for them to work. As ARM processors are not as homogenous as processors by Intel, design deviations that set them apart from Intel's standard x86 processors can make attacks more complicated. These were discussed at length and while many of them could be circumvented, the combination of no flush instructions being available to attackers and a pseudo-random replacement policy could pose a problem. Therefore, future work could focus on if or what cache attacks can be executed on modern devices, and whether new ways of defeating a pseudo-random replacement policy exist.

REFERENCES

[1] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1406–1418, Association for Computing Machinery, Oct. 2015.

[2] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross Processor Cache Attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, (New York, NY, USA), pp. 353–364, Association for Computing Machinery, May 2016.

[3] R. Spreitzer and T. Plos, "Cache-Access Pattern Attack on Disaligned AES T-Tables," in *Constructive Side-Channel Analysis and Secure Design* (E. Prouff, ed.), (Berlin, Heidelberg), pp. 200–214, Springer, 2013.

[4] Y. Yarom and K. Falkner, "{FLUSH+RELOAD}: A High Resolution, Low Noise, L3 Cache {Side-Channel} Attack," pp. 719–732, 2014.

[5] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," pp. 549–564, 2016.

[6] "Mobile Fact Sheet," Nov. 2024.

[7] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive {Last-Level} Caches," pp. 897–912, 2015.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: reading kernel memory from user space," *Communications of the ACM*, vol. 63, pp. 46–56, May 2020.

[9] C. Su and Q. Zeng, "Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures," *Security and Communication Networks*, vol. 2021, no. 1, p. 5559552, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2021/5559552.

[10] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "{AutoLock}: Why Cache Attacks on {ARM} Are Harder Than You Think," pp. 1075–1091, 2017.

[11] "Cortex-A720."

[12] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-Level Cache Hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 96–109, June 2018. ISSN: 2575-713X.

[13] "ARM Cortex-A Series Programmer's Guide for ARMv7-A."

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006* (D. Pointcheval, ed.), (Berlin, Heidelberg), pp. 1–20, Springer, 2006.

[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*, pp. 605–622, May 2015. ISSN: 2375-1207.

[16] A. Canteaut, C. Lauradoux, and A. Seznec, *Understanding cache attacks*. report, INRIA, 2006.

[17] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (J. Caballero, U. Zurutuza, and R. J. Rodríguez, eds.), (Cham), pp. 300–321, Springer International Publishing, 2016.

[18] "Documentation/vm/pagemap.txt - kernel/common - Git at Google."