# Cache Storage Attacks

Billy Bob Brumley[(✉)]

Department of Pervasive Computing, Tampere University of Technology,
Tampere, Finland
`billy.brumley@tut.fi`

**Abstract.** Covert channels are a fundamental concept for cryptanalytic side-channel attacks. Covert timing channels use latency to carry data, and are the foundation for timing and cache-timing attacks. Covert storage channels instead utilize existing system bits to carry data, and are not historically used for cryptanalytic side-channel attacks. This paper introduces a new storage channel made available through cache debug facilities on some embedded microprocessors. This channel is then extended to a cryptanalytic side-channel attack on AES software.

**Keywords:** Side-channel attacks · Covert channels · Storage channels · Timing attacks · Cache-timing attacks

## 1 Introduction

In one of the seminal computer security works, Schaefer et al. [12] define a covert channel as follows.

> Covert channels are [data] paths not meant for communication but that can be used to transmit data indirectly.

They go on to define both storage and timing channels:

> Storage channels consist of variables that are set by a system process on behalf of the sender, e.g., interlocks, thresholds, or an ordering. In timing channels, the time variable is controlled: resource allocations are made to a receiver at intervals of time controlled by the sender. In both cases, the state of the variable ("on" or "off", "time interval is 2 seconds") is made to represent information, e.g., digits or characters.

Continuing this line of research, a team of researchers at DEC in the 1990s wrote a number of influential papers regarding covert channels [4–6,13], in particular those enabled by caching technologies.

Traditional cryptanalysis views cryptosystems as mathematical abstractions and develops attacks using theoretical models consisting of only inputs and outputs of the cryptosystem. In the case of e.g. a block cipher, the input would be the plaintext and output the ciphertext, and cryptanalysis tasked with recovering the key using sets of these inputs and outputs.

In contrast, side-channel cryptanalysis exploits implementation aspects to aid in key recovery. What constitutes a side-channel is technically ill-defined, but generally speaking it is an implementation-dependent signal procured during the execution of a cryptographic primitive. This is where the fields of covert channels and side-channel analysis intersect: identifying some microarchitecture or software feature within a cryptosystem implementation that can be used to transfer data between two legitimate parties, then developing it into a cryptanalytic side-channel attack when one party is legitimate and one illegitimate.

Cache-timing attacks exploit the varying latency of data load instructions to carry out cryptanalytic side-channel attacks. These attacks are recognized by both academia and industry as a serious threat to security-critical software: from Page's seminal work [9], to Bernstein's attack on AES [2], to Percival's attack on RSA [11], to Osvik et al.'s attack on AES [8], to Brumley and Hakala's attack on ECDSA [3], to Aciiçmez et al.'s attack on DSA [1]. Arguably the most recent example of cache-timing attacks affecting real-world systems and software is Yarom and Benger's work [14] that led to CVE-2014-0076 and induced changes[1] in OpenSSL's Montgomery ladder implementation.

Placing cache-timing attacks within the covert timing channel framework, it is fair to say that utilizing covert timing channels for cryptanalytic side-channel attacks is a popular, well-established paradigm. Covert storage channels, however, are essentially ignored due to lack of application.

This paper introduces a novel, practical covert storage channel. The basis for the channel is that many caches have hardware support for per-cache line privilege separation. The access control enforced by this separation creates a storage channel that can be used to violate the system security policy. As with most covert channels, it is then possible to extend this particular covert storage channel to a cryptanalytic side-channel attack.

The organization of this paper is as follows. Section 2 provides necessary background on popular AES software and existing cache-timing attacks against such software implementations. Then Sec. 3 describes the new covert storage channel, including the prerequisite hardware differences in the cache implementation compared to a traditional cache (Sec. 3.1), why such differences exist in modern caches (Sec. 3.2), how this feature leads to a covert storage channel (Sec. 3.3), how this channel extends to a cryptanalytic side-channel (Sec. 3.4), and what practical architectures this applies to (Sec. 3.5). Final thoughts and conclusions are drawn in Sec. 4.

## 2   Background

Applications of covert channels and subsequently side-channels are important aspects from the practicality perspective. To this end, Sec. 2.1 gives some background on typical high-performance AES software implementation, and Sec. 2.2 on cache-timing attacks on such software. While this background is important

---

[1] https://www.openssl.org/news/secadv_20140605.txt

to show the immediate applicability of the results in this paper, keep in mind the underlying main results of this paper is the covert channel itself, and not its application to any one cryptosystem in particular. That is, the covert channel described in this paper will absolutely have applications outside of AES, but at the same time AES serves as a good example of its application.

## 2.1   AES Software

Viewing the 16-byte AES state as a $4 \times 4$ matrix, the first nine AES rounds are identical and consist of steps SubBytes, ShiftRows, MixColumns, and AddRound-Key. The last round omits the MixColumns step. SubBytes $\gamma : \mathcal{M}_{4\times4}[\mathbb{F}_{2^8}] \rightarrow \mathcal{M}_{4\times4}[\mathbb{F}_{2^8}]$ is a fixed non-linear substitution $S : \mathbb{F}_{2^8} \rightarrow \mathbb{F}_{2^8}$ (S-box) using finite field inversion applied to all state bytes.

$$\gamma(a) = b \Leftrightarrow b_{ij} = S[a_{ij}], \ \ 0 \leq i,j < 4$$

MixColumns $\theta : \mathcal{M}_{4\times4}[\mathbb{F}_{2^8}] \rightarrow \mathcal{M}_{4\times4}[\mathbb{F}_{2^8}]$ is a fixed linear transformation.

$$\theta(a) = b \Leftrightarrow b = M \cdot a$$

Here $M$ is the following $4 \times 4$ matrix.

$$M = \begin{bmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{bmatrix}$$

Traditional AES software is heavily lookup table based. The reason for this is that many of the low-level finite field operations, such as multiplications in $\theta$ and inversions in $\gamma$, are simply not natively supported on mainstream microprocessors. To compensate for the understandable lack of Instruction Set Architecture (ISA) support for such operations, a 32-bit processor leverages the linearity property of the MixColumns step to improve performance. Consider the following four tables, each containing 256 4-byte words.

$$T_0[x] = (2 \cdot S[x], S[x], S[x], 3 \cdot S[x])$$
$$T_1[x] = (3 \cdot S[x], 2 \cdot S[x], S[x], S[x])$$
$$T_2[x] = (S[x], 3 \cdot S[x], 2 \cdot S[x], S[x])$$
$$T_3[x] = (S[x], S[x], 3 \cdot S[x], 2 \cdot S[x])$$

That is, each $T_i$ maps one byte for a particular component through the non-linear layer input to the linear layer output. With these tables in hand, one AES round amounts to 16 table lookups and 16 bitwise XORs, illustrated in Fig. 1.

Since the last round omits the MixColumns step, its implementation differs. One popular way to implement the last round is as follows. Consider the following table, containing 256 4-byte words.

$$T_4[x] = (S[x], S[x], S[x], S[x])$$

```
y0 = T0[x0 & 0xFF] ^ T1[(x1 >> 8) & 0xFF] ^ T2[(x2 >> 16) & 0xFF] ^ T3[x3 >> 24] ^ rk0;
y1 = T0[x1 & 0xFF] ^ T1[(x2 >> 8) & 0xFF] ^ T2[(x3 >> 16) & 0xFF] ^ T3[x0 >> 24] ^ rk1;
y2 = T0[x2 & 0xFF] ^ T1[(x3 >> 8) & 0xFF] ^ T2[(x0 >> 16) & 0xFF] ^ T3[x1 >> 24] ^ rk2;
y3 = T0[x3 & 0xFF] ^ T1[(x0 >> 8) & 0xFF] ^ T2[(x1 >> 16) & 0xFF] ^ T3[x2 >> 24] ^ rk3;
```

**Fig. 1.** One AES round with the $T$ tables approach. 32-bit unsigned integers $x_i$ hold state column $i$ and $rk_i$ are words of the particular round key.

Duplicating the S-box output across the word means that no shifting is necessary to place the S-box output in the proper component. Instead, the redundant bytes in the output get masked off with a bitwise AND operation after the lookup. Implementation of the last round otherwise follows the computation in Fig. 1, with the lookups into all $T_i$ replaced with lookups into $T_4$.

   As a final note, there are countless strategies for implementing AES software, but the preceding description is accurate for the popular C reference implementation `rijndael-alg-fst.c` by P. Barreto et al. used already in the AES competition.

### 2.2   Cache-Timing Attacks

This $T$ tables implementation approach potentially exposes the AES software to cache-timing attacks. Lookups into the memory-resident $T$ tables cause data cache lines to be populated and evicted. Consider a typical data cache line size of 64 bytes. Each $T$ table is 1kB, hence spans 16 lines in the cache. For one particular lookup of the $10 \times 16 = 160$ lookups in an AES encryption (or decryption), the latency of the lookup depends on the state of the cache and hence the state of the AES algorithm. Amongst the plentiful AES cache-timing results over the past decade that leverage this varying latency to carry out cryptanalytic side-channel attacks, two are particularly relevant to this paper and are discussed below.

**Prime and Probe.** Osvik et al. [8] devise a number of cache-timing attacks against $T$ table based implementations of AES. For the purposes of this paper, the most important part of their work is the strategy they devise to procure the timings called "Prime+Probe". In this strategy, the attacker first brings the cache to a known state by either filling the entire cache or relevant cache sets by performing loads and stores, inducing cache line population and eviction. The attacker then submits a plaintext block. After the encryption completes, the attacker, cache set-wise, measures the time required to re-read the data in the cache sets, obtaining a latency measurement for each set. High latency implies cache misses and that the victim accessed data mapping to the cache set, and low latency the opposite.

**Targeting the Last Round.** Considering the first 9 AES rounds, each $T$ table has 4 lookups into it per round for a total of 36 lookups. Assuming each

table spans 16 cache lines, the amount of state information that can be learned from these lookups is limited because the order of the lookups is not (necessarily) known w.r.t. the trace timing data. For example, after the probe step the attacker knows which lines were evicted, but not what exact lookup caused the eviction. Neve and Seifert [7] instead target the last round, specifically the $T_4$ table. The authors devise two attacks that seek to recover the last round key. The most important for the purposes of this paper is the "elimination method" summarized below.

The average number of cache sets accessed in the last round is 10.3 [7, Sec. 5] and not accessed is 5.7 [7, Sec. 7.2]. This method keeps a set of candidate bytes for each round key byte. An unreferenced set implies the corresponding upper four bits of state are not possible for *any* state byte. Use the corresponding ciphertext to compute the resulting impossible key bytes. This eliminates up to sixteen key byte candidates from each key byte, or 256 candidates total. The attack proceeds iteratively through the traces in this fashion, trimming the candidate sets. Naturally as more traces are processed less trims are made as collisions start occurring, i.e., eliminating bytes that have already been eliminated, but the authors show that roughly 20 queries suffices to recover the key using this method [7, Sec. 7.2].

## 3   Cache Storage Attacks

Consider the following hypothetical, simple data cache. There are 16 lines and each line is 64 bytes. Assume wlog the cache is direct mapped. Whether the cache is virtually/physically indexed/tagged is irrelevant to this paper. With respect to the cache, a 32 bit address breaks down as follows. The $\lg(64) = 6$ LSBs denote the offset within a line. The next $\lg(16) = 4$ bits denote the set index. The remaining $32 - 6 - 4 = 22$ bits denote the tag. The set index and tag combine to determine cache hits and misses, i.e. if the tag matches and the set index matches, a cache hit occurs. In practice, while there are often more lines and sets, this cache (or one extremely similar to it) is overwhelmingly what goes into modern commodity microprocessors.

### 3.1   Hardware Privilege Separation

Now consider the following hypothetical, simple data cache that is similar but supports privilege separation in hardware. What this means is the per-line meta-data for the previous cache consisting of the tag gets extended to also include the privilege level for that line's contents. For simplicity's sake this paper considers only 1-bit privilege levels but the results are more generally applied. Figure 2 compares these two cache structures. One example of this 1-bit privilege level could be identifying ring 0 or ring 3 in the cache for x86 protection mode: a 0 (or 1) could denote the physical memory corresponding to that particular cache line belongs to ring 0 (or 3).

```
idx   tag  data          idx priv   tag  data
--- ------ ----          --- ---- ------ ----
  0 de30ec ????            0    1 de30ec ????
  1 096324 ????            1    0 096324 ????
  .      .    .            .    .      .    .

  .      .    .            .    .      .    .
  F 61eff8 ????            F    1 61eff8 ????
```

**Fig. 2.** Left: Example traditional cache without hardware privilege separation. Right: Example cache augmented with hardware privilege separation.

This paper assumes the cache replacement policy is oblivious to the semantics of this privilege level bit, i.e., it is simply another bit of the tag that only determines cache hits and misses. This means that privilege level 0 can evict privilege level 1 data and vice versa. If this were not the case, resource starvation would occur unless employing a more sophisticated cache structure (see e.g [10] for a discussion). Also a common argument for this behavior is better cache utilization, causing improved software performance that is a leading driver in industry.

## 3.2   Motivation

There are potentially many reasons to store the per-line privilege level with the cache metadata. Arguably the most appropriate use case is in debug scenarios. To debug the cache itself or programs where cache performance is critical, some architectures expose low level instructions that allow invasive access to the cache data and metadata. For example, this could be used by software engineers:

–  To examine cache state and eliminate it as a potential source of bugs e.g. in hardware errata scenarios or coherency issues.
–  To better understand their software's cache impact, and subsequently improve performance through analysis of said impact.

However, the cache cannot simply allow unchecked access to the lines and metadata. For example, privilege separation fails if privilege level 1 directly reads a cache line belonging to privilege level 0. So the cache needs to know the privilege level of each line's data for security reasons to enforce a sane access control policy, and having that information stored directly alongside the tag is arguably the most logical solution for the hardware itself to enforce said policy. For attempted accesses that would violate the access control policy, a reasonable response would be to issue a processor exception. This is similar to how e.g. a Memory Management Unit (MMU) handles accesses to unmapped virtual addresses, i.e. invalid page faults that usually result in segmentation faults.

### 3.3    A Covert Channel

Alice (privilege level 0) and Bob (privilege level 1) construct a storage covert channel out of the cache with privilege separation as follows. Assume wlog the cache structure in Fig. 2 and that Alice wants to send $\lg(16) = 4$ bits to Bob, denoted nibble $b$.

1. Bob loads from 16 memory locations that all have different index bits. This is the "prime" step and completely pollutes the cache, as well as populates all privilege level bits in the cache to 1, corresponding to Bob's privilege level.
2. Alice loads from a single memory location with index bits $b$. She gets a cache miss and evicts Bob's line from index $b$. Note that, after this step, Alice leaves the cache in the same state as Bob left it, other than index $b$: all lines have privilege level bit set to 1 except line with index $b$ now set to 0.
3. Bob tries to read *directly* from all 16 lines in the cache: this is the "probe" step. When he reaches index $b$ he triggers a processor exception because he is attempting to violate privilege separation, but nonetheless receives $b$ from Alice, evidenced by the exception.

From the dishonest users' perspective, the main disadvantage of this covert channel is its detectability. Timing covert channels are difficult to detect since the only evidence of their presence is performance degradation. In this case, every time this particular processor exception occurs the system gets informed so there is an audit trail.

The main advantage of this covert channel is its signal-to-noise ratio. By nature, timing channels are heuristic – they are noisy and require tuning to a particular system and cache performance. This cache storage channel, however, goes unaffected by these variables that affect cache hit and miss latencies. The only thing the recipient needs to observe is the presence of the processor exception. This exception is deterministic, not heuristic.

### 3.4    A Side-Channel Attack

An access-driven cache-timing trace, as used in e.g. the attacks described in Sec. 2.2, is interpreted as a sequence of cache hits (H) and misses (M) on a per cache set (or line) basis. Note that the hits and misses are based on the memory access timings being above or below some threshold, so they are quite sensitive to a particular processor, cache, operating system, and system load – in practice they are rarely error-free but instead require some statistical analysis. Nevertheless, assume this timing trace is error-free. Attackers can "reconstruct" access-driven cache-timing traces with the cache storage channel described above with the following steps.

1. Read directly from a cache line. A processor exception indicates M, otherwise H.
2. If M go back to the first step. This requires another query because the processor exception most like wipes the cache state and/or triggers a reset.
3. If H continue with the next line.

For example, Consider the following timing trace.

HMHHHMHHHHHHMHHH

The read from line 0 does not cause an exception, so the attacker logs H and continues. Line 1 causes an exception. The attacker logs M, queries the same plaintext again, reads from line 2, logs H and continues in this manner. Line 5 causes an exception. The attacker logs M, queries the same plaintext again, and continues in this manner. It takes the attacker four queries to reconstruct the trace: one for the initial query, and one for each processor exception ("cache miss").

Given the above analysis, cache storage attacks should exhibit the following characteristics when compared to cache-timing attacks.

– The number of queries theoretically increases because, compared to cache-timing attacks, each "cache miss" costs an additional query due to the processor exception.
– The traces themselves, however, are overwhelmingly more accurate because they are not heuristics based on timings.

This ease of reconstructing error-free cache-timing traces from error-free cache storage traces allows leveraging previous cache-timing results directly. For example, consider the attack by Neve and Seifert [7] summarized in Sec. 2.2. The key recovery algorithm is essentially the same, but the number of queries will increase. Figure 3 illustrates the implementation of the Neve and Seifert cache-timing attack using the cache storage attack techniques in this paper. In the cache-timing case, they state roughly 20 queries are needed to recover the last round key. Given that their analysis shows the average number of cache sets accessed is 10.3 and cache storage attacks need an initial query plus one query for each cache set accessed ("cache miss"), the expectation is $20 \cdot (10.3 + 1) = 226$ queries on average for the cache storage attack to succeed. The simulation results in Fig. 3 are consistent with this estimate.

### 3.5    Relevant Architectures

The running example in this paper has been privilege levels 0 and 1 corresponding to e.g. ring 0 and ring 3. To make these results more concrete, arguably the most relevant architecture for cache storage attacks is ARM with TrustZone extensions.

TrustZone technology provides hardware-assisted security mechanisms to software, in particular Trusted Execution Environments (TEE). TEEs are ubiquitous in the embedded space, e.g. mobile phones. In these cases, the mobile operating system such as Android runs in the normal world or untrusted world or insecure world or rich execution environment while the security-critical code runs in the secure world or trusted world or trusted execution environment.

At any given moment, ARM microprocessors that support TrustZone extensions operate in either secure or insecure mode. Insecure mode uses system calls
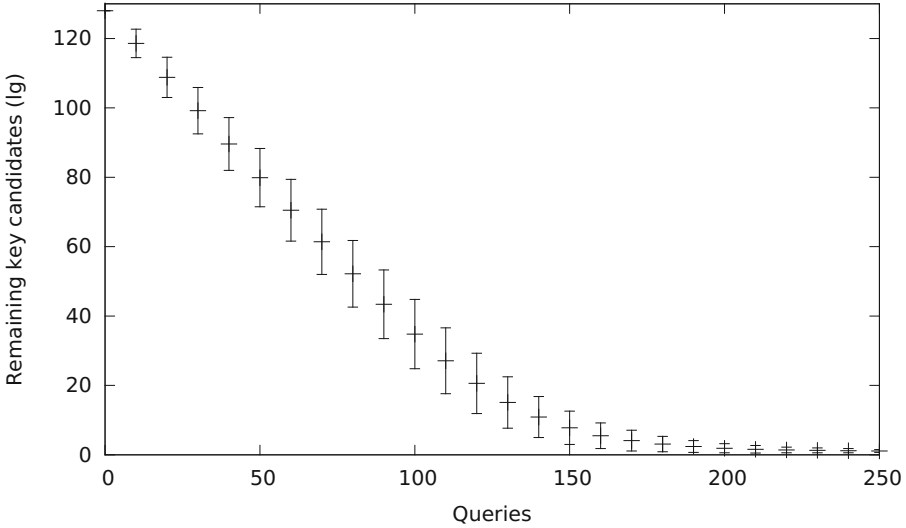
**Fig. 3.** AES cache storage attack simulation results. Average number of remaining key candidates (base-2 logarithm) as the number of encryption queries increases. Error bars are one standard deviation on each side.

(dedicated instructions in the ISA) to switch to secure mode, the transition handled by a piece of software ARM calls the monitor. From the system perspective, bus transactions originating from either the secure or non-secure world are tagged using the `AxPROT` bus attribute, essentially a binary value that tracks the privilege level of the transaction. Figure 4 illustrates this concept. Quoting the ARM documentation:

> In the caches, instruction and data, each line is tagged as Secure or Non-secure, so that Secure and Non-secure data can coexist in the cache. Each time a cache line fill is performed, the NS tag is updated appropriately.

Mapping this architecture to the previously described cache storage covert channel is simple: privilege level 0 corresponds to NS=0 and privilege level 1 to NS=1. This statement directly from ARM validates the previous assumptions in this paper with respect to the cache replacement policy – the data at different privilege levels coexists in the cache yet the replacement policy is oblivious to this distinction. Secure data can evict non-secure data and vice versa.

Further illustrating the applicability of cache storage attacks to ARM architecture with TrustZone extensions, the documentation continues, illustrated in Fig. 5:

> It is a desirable feature of any high performance design to support data of both security states in the caches. This removes the need for a cache flush when switching between worlds, and enables high performance software to communicate over the world boundary. To enable this the L1,

and where applicable level two and beyond, processor caches have been extended with an additional tag bit which records the security state of the transaction that accessed the memory.

The content of the caches, with regard to the security state, is dynamic. Any non-locked down cache line can be evicted to make space for new data, regardless of its security state. It is possible for a Secure line load to evict a Non-secure line, and for a Non-secure line load to evict a Secure line.

The cache attempts the [sic] match the PA and the NS-bit from the TLB with the tag of an existing cache line. If this succeeds it will return the data from that cache line, otherwise it will load the cache line from the external memory system.
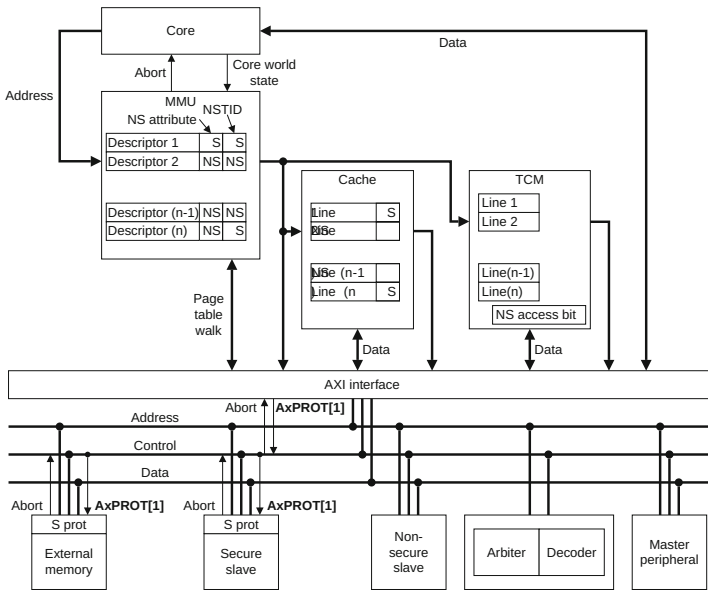


**Fig. 4.** ARM architecture with TrustZone extensions: propagation of the normal (NS=1) and secure (NS=0) signal (`AxPROT`) system-wide via bus transactions. Source: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc. ddi0333h/Chdfjdgi.html

The last ingredient missing for realizing the cache storage covert channel is invasive access for direct cache line reads. These particular instructions will generally depends on the chip manufacturer, exposed through instruction-level CP15 ("coprocessor 15") commands. Such commands are generally used for e.g. performance monitoring, but these cache commands are encoded in an
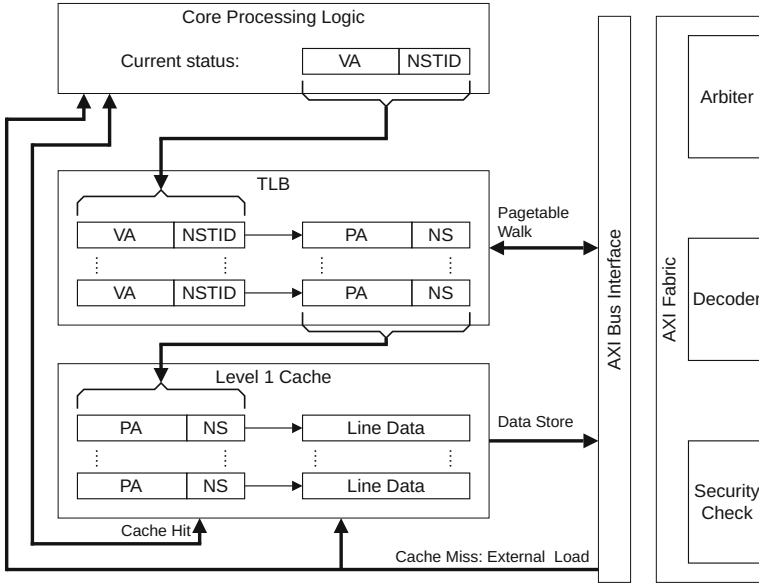
**Fig. 5.** ARM architecture with TrustZone extensions: cache logic with respect to the normal (NS=1) and secure (NS=0) worlds. Source: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch03s03s02.html

*implementation defined* space (C15) so they are manufacturer dependent. Nevertheless, some examples follow, e.g. through CP15 and C15[2]:

> The purpose of the data cache Tag RAM operation is to:
> - read the data cache Tag RAM contents and write into the Data Cache Debug Register.
> - write into the Data Cache Debug Register and write into the Data Tag RAM.
>
> To read the Data Tag RAM, write CP15 with:
>
> ```
> MCR p15, 3, <Rd>, c15, c2, 0  ;Data Tag RAM read operation
> ```
>
> Transfer data to the Data Cache Debug Register to the core:
>
> ```
> MRC p15, 3, <Rd>, c15, c0, 0 ;Read Data Cache Debug Register
> ```

While these particular commands are for reading tag data associated with a particular line, the effect for the purposes of this paper is the same. The documentation goes on to describe the register format to specify the set and way combination for the desired cache operation. Not specific to cache storage attacks but more generically for issuing instructions to CP15, MCR is for coprocessor to ARM transfers and MRC for ARM to coprocessor transfers.

---

[2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0338g/Bgbedgaa.html

## 4    Conclusion

This paper introduces a new covert channel enabled by the data cache populating per-line privilege level bits and subsequently enforcing privilege separation on the lines. In contrast to previous covert timing channels that are inherently noisy, this covert storage channel is much easier to utilize because it does not rely on heuristic timings. While the use of this channel is easier to detect than covert timing channels since the attacker will trigger processor exceptions, it clearly fits the covert channel definition of Schaefer et al. [12] since the value of privilege level bits is certainly not intended to carry data.

Cache storage attacks are related to cache-timing attacks in the sense that the former can be used to construct an error-free side-channel trace for the latter, shown in Sec. 3.4. The resulting cache storage attack given on AES is otherwise a direct analog of the cache-timing attack, but requires more queries as shown by the experiment results. The outcome is the leaking of an AES key across privilege levels, clearly a violation of a system security policy.

Section 3.5 shows how cache storage attacks map nicely to ARM's TrustZone technology. It is worth noting that the instructions and commands needed to carry out the cache storage attack are almost certainly not available in NS=1 user space, so the attack would be from NS=1 kernel space (e.g. Android) to the NS=0 secure space (e.g. a TEE).

While ARM dictates the format used for cache line operations in the specification, the actual operations used for e.g. data cache line reads and writes are in the CP15 implementation defined C15 instruction space, left to the manufacturer. As such, the most logical countermeasure to cache storage attacks lies in these implementation defined instructions. Chip manufacturers should disallow these instructions while in NS=1 mode, or at a minimum default to disallow yet have the ability to issue these instructions from NS=1 be software configurable from NS=0.

## References

1. Acıçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010)
2. Bernstein, D.J.: Cache-timing attacks on AES (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
3. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 667–684. Springer, Heidelberg (2009)
4. Hu, W.: Reducing timing channels with fuzzy time. In: IEEE Symposium on Security and Privacy, pp. 8–20 (1991)
5. Hu, W.: Lattice scheduling and covert channels. In: IEEE Symposium on Security and Privacy, pp. 52–61 (1992)
6. Karger, P.A., Wray, J.C.: Storage channels in disk arm optimization. In: IEEE Symposium on Security and Privacy, pp. 52–63 (1991)

7. Neve, M., Seifert, J.-P.: Advances on access-driven cache attacks on AES. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 147–162. Springer, Heidelberg (2007)
8. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
9. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169 (2002). https://eprint.iacr.org/2002/169
10. Page, D.: Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280 (2005). https://eprint.iacr.org/2005/280
11. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005). http://www.daemonology.net/papers/cachemissing.pdf
12. Schaefer, M., Gold, B., Linde, R., Scheid, J.: Program confinement in KVM/370. In: Proceedings of the 1977 Annual Conference, pp. 404–410. ACM (1977)
13. Wray, J.C.: An analysis of covert timing channels. In: IEEE Symposium on Security and Privacy, pp. 2–7 (1991)
14. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140 (2014). https://eprint.iacr.org/2014/140