Cache Timing Attacks on eStream Finalists

Erik Zenner

Technical University Denmark (DTU)
Institute for Mathematics
e.zenner@mat.dtu.dk

Echternach, Jan. 9, 2008

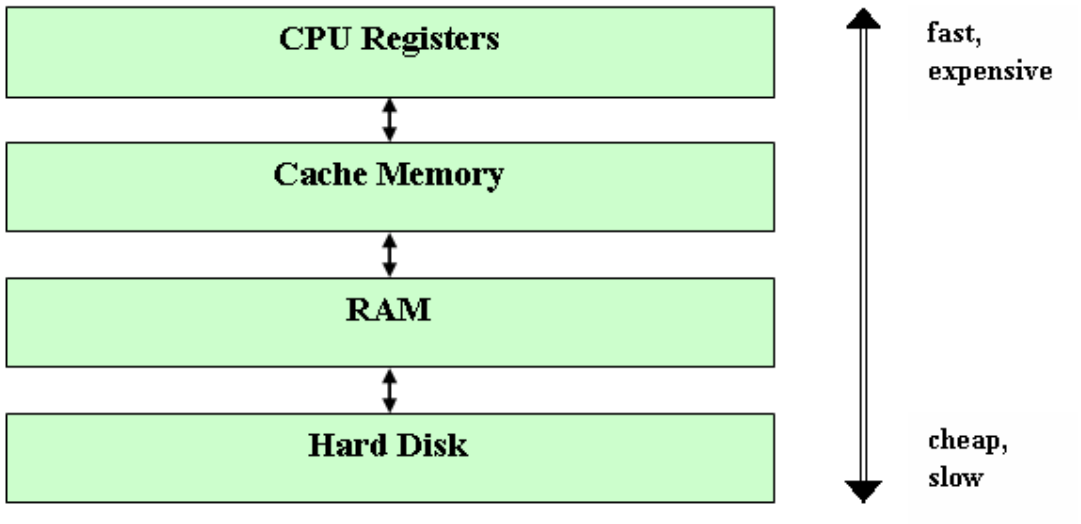
1. Cache Timing Attacks
   - Basics
   - The AES Case
   - Comments

2. Analysing eStream Candidates
   - Model for Analysis
   - Candidates
   - Obstacles to Analysis
   - Some Questions

Outline

Memory Hierarchy

In a modern computer, different types of memory are used (simplified):



While CPU, RAM, and hard disk are typically protected against use by another user on the same machine, the cache is not.

Cache Workings (1)

Motivation: Loading data from cache is much faster than loading data from RAM (by a factor of ≈ 10).

Working principle (simplified): Let n be the cache size.
When data from RAM address a is requested by the CPU, proceed as follows (simplified):

- Check whether requested data is at cache address (a mod n).
- If not, load data into cache address (a mod n).
- Load data item directly from cache.

Similarly for writing data to RAM.

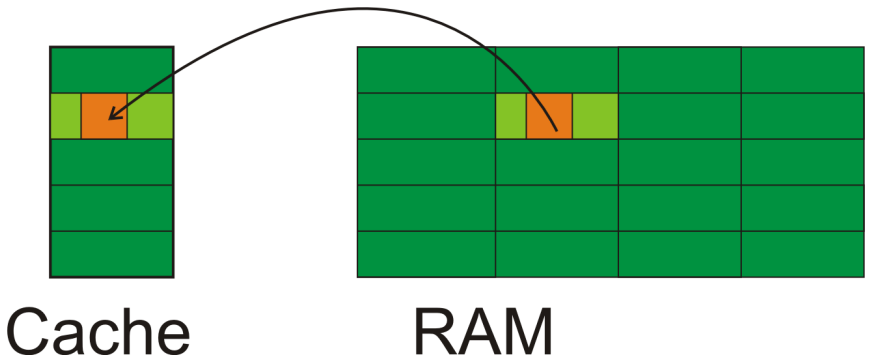
Idea: Data that is used now will more likely be used again in the future (temporal proximity).
$\Rightarrow$ Keeping copies in cache reduces the average loading time.

Cache Workings (2)

Extension: In addition, data that is physically close to currently used data will also more likely be used in the future (spatial proximity).
$\Rightarrow$ Keeping copies of physically close data in cache also reduces the average loading time.

Consequence:

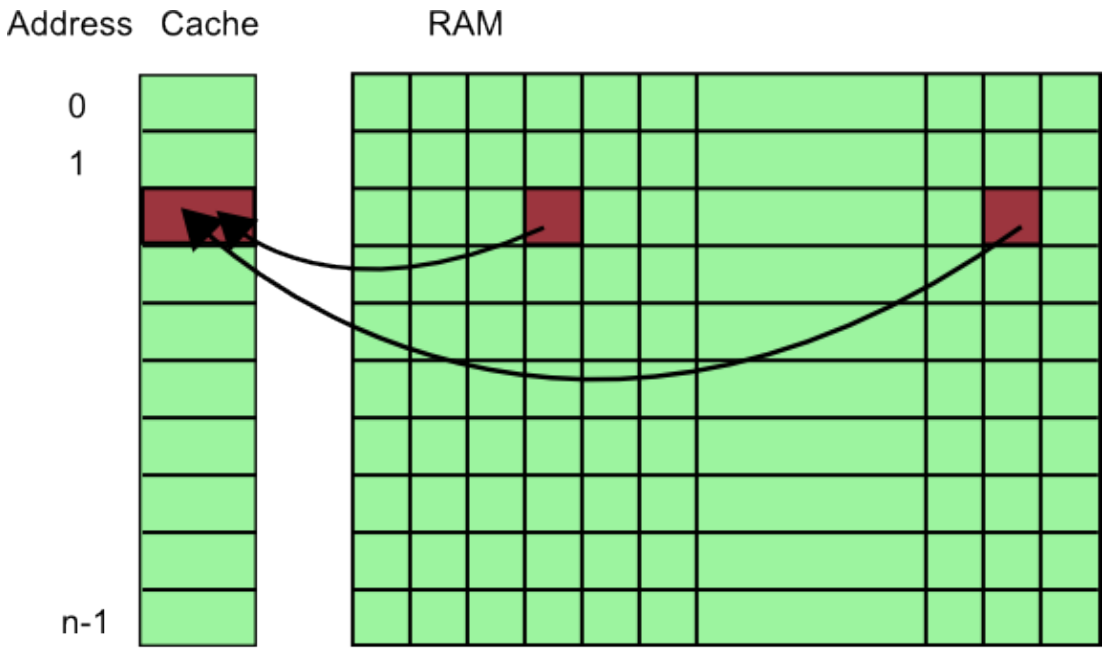
- Organise both cache and RAM into blocks of size s.
- When loading a piece of data to cache, load the whole block that surrounds it.



Cache Eviction (Simplified)

Problem: Cache is much smaller than RAM.

Consequence: Many RAM entries compete for the same place in cache.



Handling: New data overwrites old data (First in, first out).

Sample Attack Setting

Starting point: Reading data is faster if it is in cache (cache hit), and slower if it has to be loaded (cache miss).

Sample attack (prime-then-probe): Imagine two users A and B sharing a CPU. If user A knows that user B is about to encrypt, he can proceed as follows:

1. A fills all of the cache with his own data, then he stops working.
2. B does his encryption.
3. A measures loading times to find out which of his data have been pushed out of the cache.

This way, A learns which cache addresses have been used by B.

What need to know about AES...

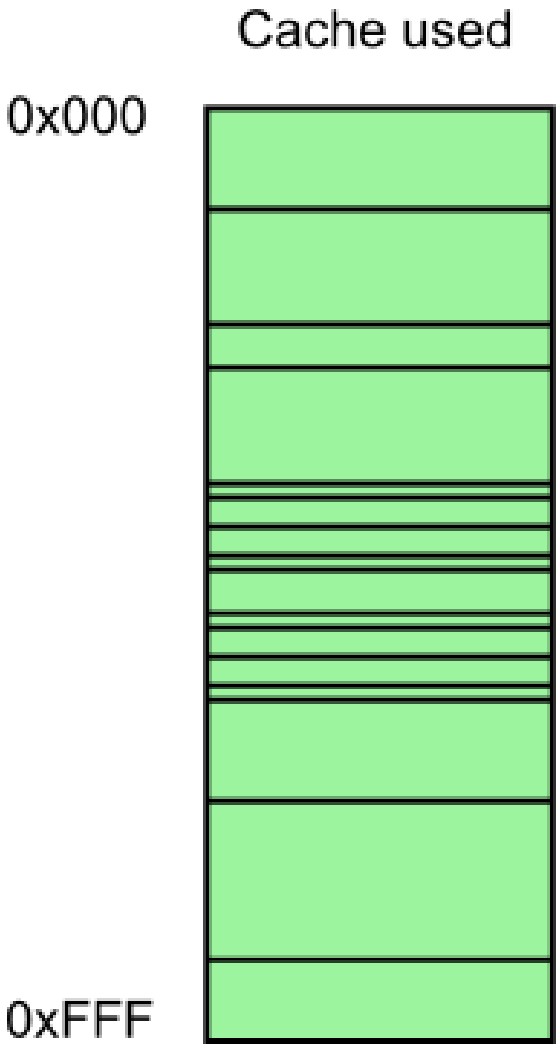
Notation: AES-128 transforms a 16-byte plaintext $m = (m_0, \ldots, m_{15})$ into a 16-byte ciphertext $c = (c_0, \ldots, c_{15})$, using a 16-byte key $k = (k_0, \ldots, k_{15})$.

Description: We give no full description of AES here.
All we need to know is step 1 of a typical AES-128 implementation (optimised; not identical to the textbook description):

- For all $j = 0, \ldots, 15$:
  Look up $F_{j \bmod 4}[m_j \oplus k_j]$ in an 8×32 table F_i $(i \in \{0, 1, 2, 3\})$.

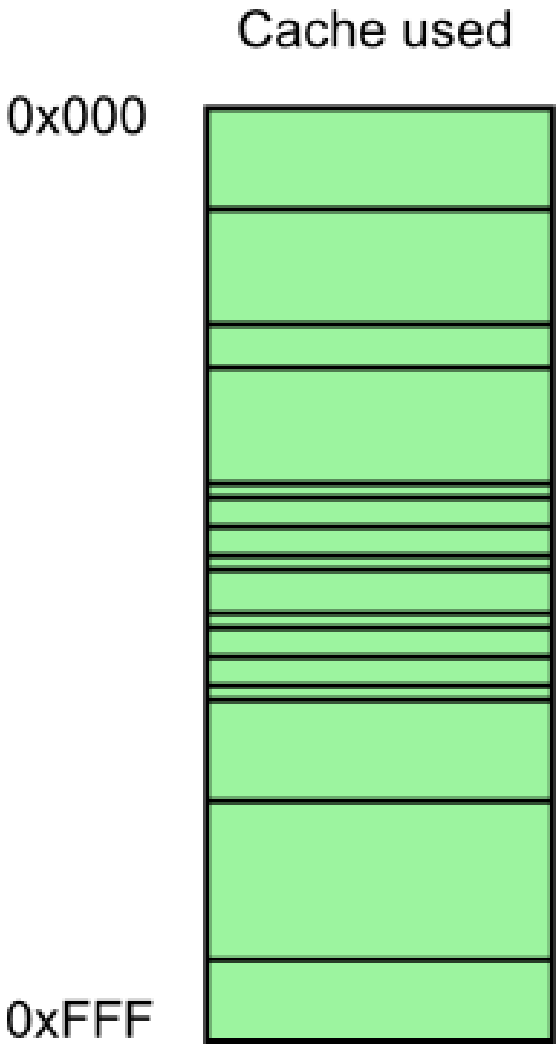
We ignore all the remaining steps here, we just point out that they also make use of the tables F_i.

Cache Timing Attack against AES (1)



1. Running a cache timing attack gives the adversary a table with this structure.

Cache Timing Attack against AES (1)



1. Running a cache timing attack gives the adversary a table with this structure.
2. We can clearly see where the tables F_i lie in cache.
3. We can also see which blocks in the tables F_i have not been accessed.

Cache Timing Attack against AES (2)

1. This gives us a list $\hat{L}$ of candidate indices a for which $F_i[a]$ has not been used.
2. In step 1, AES accessed the table for $F_{j \bmod 4}[m_j \oplus k_j]$.
   $\Rightarrow m_j \oplus k_j$ can not be in $\hat{L}$!
3. Make list of candidates for $k_j = a \oplus m_j \ \ \forall a \notin \hat{L}$.
4. Re-run attack and intersect the resulting lists.
5. Repeat until brute-force of the remaining key candidates becomes possible.

Practical Difficulties

For didactical reasons, we worked with a simplified cache model.

Real-world complexities include:

- Other processes (e.g. system processes) use the cache, too.
  $\Rightarrow$ We can not tell "encryption" cache accesses apart from others.
- Timing noise disturbs the measurement.
  $\Rightarrow$ Not all slow timings are due to cache misses.
- Cache hierarchy is more complex.
  $\Rightarrow$ Several layers of cache, several cache blocks for each memory block.

Nonetheless, as it turns out, these difficulties can be overcome in practice (Bernstein 2005, Osvik/Shamir/Tromer 2005, Bonneau/Mironov 2006).

Outline

Motivation

When setting up the attack model, our motivation was as follows:

- Abstract away technical details of the cache timing attacks.
- Have a model that is suitable for **designing** cryptographic algorithms.

$\Rightarrow$ Model has to be rather generous w.r.t. the attacker's options.

Attack Model (1)

Assumption 1:

The adversary can trigger the execution of any of the following functions at will:

- Key setup
- IV setup (with chosen IV)
- Keystream generation (with chosen index)

Attack Model (1)

Assumption 1:

The adversary can trigger the execution of any of the following functions at will:

- Key setup
- IV setup (with chosen IV)
- Keystream generation (with chosen index)

Assumption 2:

The adversary can choose the IV, and he can observe the keystream as usual.

Attack Model (2)

Assumption 3:

For each function call, the adversary obtains a correct and noise-free list of the cache blocks accessed by this function call.

Attack Model (2)

Assumption 3:

For each function call, the adversary obtains a correct and noise-free list of the cache blocks accessed by this function call.

Voluntary Constraints:

In the first phase of our analysis, we made the following restrictions:

- We tried to find a **practical** attack, meaning that:
  - The adversary can only call the above functions for a limited number of times (say, $< 1,000,000$).
  - The attack should be executable on non-agency equipment (running time, memory etc.).
- The adversary is only successful if he can reconstruct the key or at least the inner state.

Obviously, these restrictions should be dropped when the model is used for cipher design.

eStream Software Finalists

Cipher	Tables	Relevant
CryptMT	none	-
Dragon	Two 8×32-bit S-Boxes	†
HC-128	Two 9×32-bit tables	
HC-256	Two 10×32-bit tables	†
LEX-128	One 8×8-bit S-Box (ref. code)	
	Eight 8×32-bit S-Boxes (opt. code)	†
NLS	One 8×32-bit S-Box	†
Rabbit	none	-
Salsa-20	none	-
Sosemanuk	One 8×32-bit table,	
	eight 4×4-bit S-Boxes (ref. code)	
	additional tables for fast Serpent (opt. code)	†

A Trivial Design Recommendation

About the eStream finalists:

- **Salsa-20** is designed to be resistant to Cache Timing Attacks.
- **CryptMT** and **Rabbit** are resistant, probably by accident.
- **LEX** falls to the same attacks as AES, since it uses AES for key/IV setup.
- **Dragon**, **HC-256/128**, **NLS**, and **Sosemanuk** have to be analysed.

Design Technique 1:

Do not use table lookups in a cryptographic design at all.

In the following: Design techniques where technique 1 is not applicable.

Work in Progress...

- **Expectation:** When starting analysis in the above (generous) model, we expected most eStream candidates to break down completely.
- **Surprise:** Most candidates seem to withstand analysis even in the generous model surprisingly well (not unbreakable, but complicated).
- Work on cryptanalyis is still in progress.
- **Here:** Discuss some of the obstacles encountered, and possible consequences for algorithm design.

From Cache Block Access to Inner State (1)

Example: Dragon

- 2 S-Boxes (8×32 bit), each of which fills 16 cache blocks (Pentium 4).
- In each call to the keystream generation function, each S-box is called 12 times.

Problems:

- For each S-Box, up to 12 out of 16 cache blocks are accessed (on average: 8.6).
  $\Rightarrow$ Less information than we hoped for.
- It is unclear in which order those cache blocks were accessed. If a full 12 different blocks were accessed for both S-boxes, there would be $2^{57.7}$ possible ways of ordering them.
  $\Rightarrow$ Without algebraic tools, a lot of guessing + verifying is necessary.

From Cache Block Access to Inner State (2)

Observation:
Similar problems occurred for other stream ciphers, too.

Design Technique 2:

For each function call, call many different table entries, in order

- to reduce the amount of information obtained and
- to make ordering of the cache accesses difficult.

Note that if all table entries are called at least once, no cache timing information can be obtained.

Inner State Size (1)

For protection against Time-Memory-Data tradeoff attacks, inner state size has to be at least twice the key size (i.e., 512 bit for 256-bit keys).

Cipher	Key Size	Inner State (bit)
Dragon	256	1,088
HC-128	128	32,768
HC-256	256	65,536
LEX-128	128	256
NLS	128	576
Sosemanuk	128	384

Inner State Size (2)

Example: HC-256

- The inner state size is $65,536$ bit.
- Each call to the keystream generation function gives
  - 5 table accesses, which ultimately give us 52 bit of information, and
  - 1 output word, giving 32 bit of information.
- In order to obtain sufficient information to even theoretically solve for the inner state, we need $65,536/84 \approx 780$ precise cache access measurements (or many more noisy ones).

Design Technique 3:

Make the inner state large compared to the information that can be obtained from one cache access measurement. In addition, make the connection between key / IV and inner state as complex as possible, to avoid easy relations between key and cache access measurements.

The LSB Problem (1)

Remember that we only can observe cache blocks that have been accessed, which is not the same as table indices.

Example:

- Pentium 4 L1-Cache holds 64 byte per cache block.
- Often, tables have entry sizes of 32 bit (4 byte).
- Each cache block holds $64/4 = 16$ table entries.

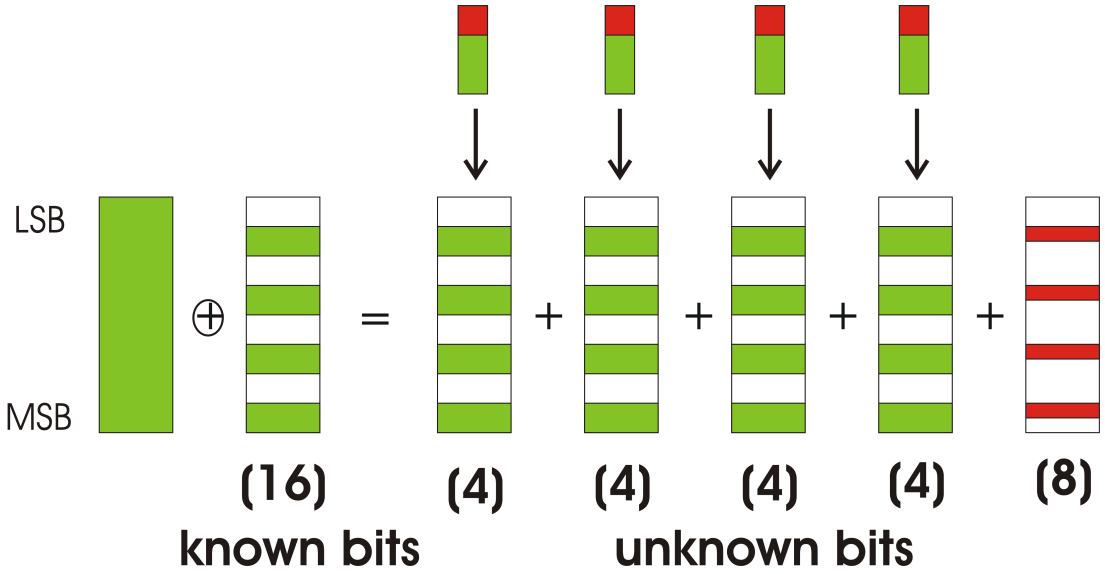
$\Rightarrow$ If table entries are aligned with cache blocks, we can not say anything about the 4 least significant bits of the table index!

This typically gives us a number of bits for some inner state words, but not the lowest bits.

The LSB Problem (2)

Example: HC-256

For one internal equation, known bits are marked green, while unknown information is marked red.



Without carry, we could verify guesses for $\gamma_1, \ldots, \gamma_4$ (guess 16 bit, verify 16 bit). But the carry introduces another 8 unknown bits, which complicate the equation.

The LSB Problem (3)

Observation:
Similar problems occurred in other places and for other stream ciphers, too.

Design Technique 4:

Introduce diffusion when combining inner state words, e.g. by using operations like addition and multiplication.
Do not rely solely on S-boxes for the diffusion.

Practical Issues

Reminder:
Remember that our model is a simplification. In the real world, the adversary may not be able to

- observe every encryption operation,
- get a precise list of cache block accesses,
- choose the IV, or
- observe the keystream.

This means that cryptanalytic results obtained in this model are not necessarily attacks in the real world.

But: As with all other design criteria in cryptography, the designer should not rely on things that the adversary *might* not be able to do!

A Puzzling Question

- With the exception of Salsa, the eStream finalists were not designed to resist cache timing attacks.
- In addition, the attack model is very generous to the adversary.
- Nonetheless, they seem to withstand an attack where the adversary learns a lot about the inner state surprisingly well.

Why?

Explanation Attempts

- Is it really just the protection measures against bit guessing that save us here?
- Could it be that the stream ciphers are overdesigned ($\Leftrightarrow$ AES)?
  In this case, what efficiency gains would be possible?
- Or could it be that our cryptanalytical toolbox is rather empty when we do not have huge amounts of (known or chosen) data available?
  - Are there really no tools for analysing elementary combinations of xor, addition, and shift?
  - Could we have developed better tools if we were not content with distinguishing attacks requiring 2^{70} known plaintext words?
  - How would we proceed if we really needed to break a cipher in a practical sense? In other words: How do agencies work?

Thank you
for your
attention!