Cache Timing Analysis of LFSR-based Stream Ciphers

Gregor Leander, *Erik Zenner* and Philip Hawkes

Technical University Denmark (DTU)
Department of Mathematics
e.zenner@mat.dtu.dk

Cirencester, Dec. 17, 2009

Outline

Cache Motivation

What is a CPU cache?

- Intermediate memory between CPU and RAM
- Stores data that was recently fetched from RAM

What is is good for?

- Loading data from cache is much faster than loading data from RAM (e.g. RAM access $\approx$ 50 cycles, cache access $\approx$ 3 cycles).
- Data that is often used several times.
- $\Rightarrow$ Keeping copies in cache reduces the average loading time.

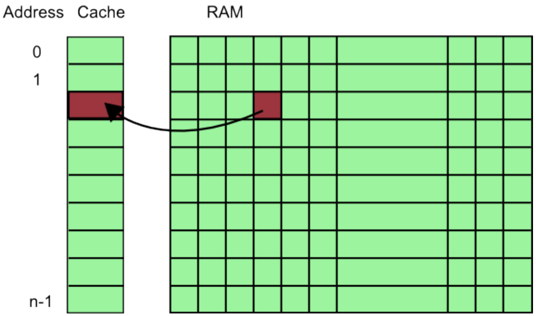
Why is this a problem?

- As opposed to RAM, cache is shared between users.
- $\Rightarrow$ Cryptographic side-channel attack becomes possible.

Cache Workings

Working principle (simplified): Let n be the cache size.
When we read from (or write to) RAM address a, proceed as follows:

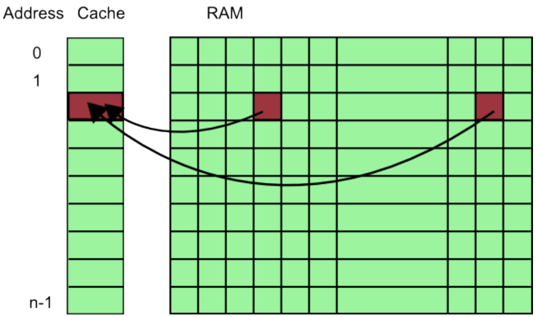
- Check whether requested data is at cache address ($a \bmod n$).
- If not, load data into cache address ($a \bmod n$).
- Load data item directly from cache.



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

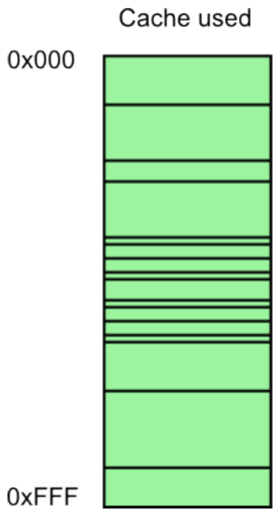
Starting point: Reading data is faster if it is in cache (cache hit), and slower if it has to be loaded (cache miss).

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.

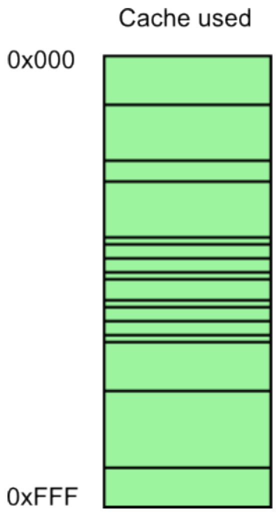
This way, A learns which cache addresses have been used by B.

Example



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

Example



1. Running a cache timing attack gives the adversary a table with this structure.
2. We can clearly see that B used a table (e.g. S-Box, lookup-table etc.).
3. We can also see which table entries have been used.

Note: Adversary learns only the table **indices** used by B, but not the table **contents**!

Practical Difficulties

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

Real-world complexities include:

- Cache data is not organised in bytes, but in blocks.
  $\Rightarrow$ *see next slides*.
- 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.

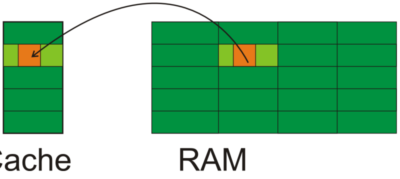
Nonetheless, these difficulties can be overcome in practice [Bernstein 2005, Osvik/Shamir/Tromer 2005, Bonneau/Mironov 2006].

Improved Cache Model (1)

Extension of cache model: 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.

Real cache design:

- 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.



Improved Cache Model (2)

⇒ We can only observe **cache blocks** that have been accessed, which is not the same as **table indices**.

Example:

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

⇒ 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.

Outline

Attacking Algorithms vs. Implementations

Basically, side-channel attacks target the **implementation**, not the **algorithm**.

Who is responsible - cryptographers or implementers?

Attacking Algorithms vs. Implementations

Basically, side-channel attacks target the **implementation**, not the **algorithm**.

Who is responsible - cryptographers or implementers? $\Rightarrow$ **Both!**

- Ideal: Cryptographers design algorithms that are not vulnerable to side-channel attacks.
- This saves **all** implementers the trouble of introducing protection measures.
- However: Cryptographers have to make assumptions (model) about the target system.

Assumptions for our Analysis

Available oracles:

- Adversary can trigger key/IV setup with IV of his choice. (standard)
- Adversary can step through the stream cipher, one round at a time. (standard)
- Adversary can obtain any keystream block of his choice. (standard)
- Adversary can obtain any *precise* cache measurement of his choice. (new!)

Limitations:

- Adversary is limited to “realistic” number of keystream blocks.
- Adversary is limited to small number of cache measurements.
- Adversary is limited to “realistic” computational resources.

Outline

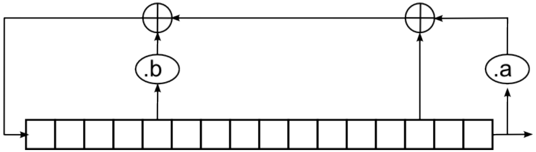
A New Target

Known cache-timing attacks:

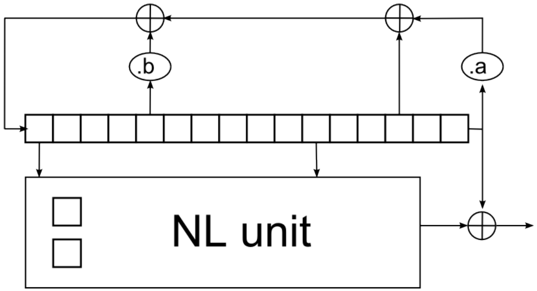
- ... against S-boxes (e.g. AES and many others)
- ... against rolling arrays (e.g. RC4, HC-256)

New target:

- ... LFSR lookup tables (e.g. Snow, Sosemanuk)



Sample Cipher: Snow 2.0



Three components (1 word = 32 bits):

- LFSR: 16 words
- NL state: 2 words
- Output: 1 word / round

Step 1: Cache-timing phase

Target the multiplications in the LFSR update:

- Multiplications are implemented using 8×32-bit lookup tables T_1 and T_2:
  - $x \cdot a = ((x \ll 8) \oplus T_1[x^{(24..31)}])$
  - $y \cdot b = ((y \gg 8) \oplus T_2[y^{(0..7)}])$
- Ideally, we observe one access each to T_1 and T_2, yielding some information about x and y.
- Repeat until we have slightly more than $16 \cdot 32 = 512$ inner state bits.

Effort:

If each table access gives the b uppermost bits of the table index:
We need $512/2b$ rounds of precise cache timing measurements.

Step 2: Reconstructing the LFSR state

Fact 1: An LFSR consisting of w elements in $\mathbb{F}_{2^m}$ can equivalently be written as an LFSR consisting of wm elements in $\mathbb{F}_2$.

Fact 2: Given an L-bit LFSR and $L + \delta$ arbitrary inner state bits, the initial state can be reconstructed efficiently by solving a system of linear equations.

Combining fact 1 and 2:

- Observing ≈ 512 arbitrary state bits allows reconstruction of LFSR initial state.
- Knowing initial state allows reconstruction of any LFSR state bit.

Effort:

1. Representing 512 state bits as lin. comb. of the initial state bits.
2. Solving an equation system in $\mathbb{F}_2$ with 512 variables.

Step 3: Reconstructing the NL state

Status:

- Attacker knows full LFSR sequence.
- Attacker also knows keystream sequence.
- Unknown: 2 words of NL state (64 bits in total).

Attack:

- Attacker guesses first NL word (32 bit).
- Uses knowledge about LFSR and output sequence.

$\Rightarrow$ Easy to determine second NL word arithmetically.

Effort:

2^{32} guess-and-determine steps.

Sosemanuk: Additional Problems

Other ciphers:

- Sober, Turing are even easier.
- Sosemanuk produces one 128-bit output block from 4 NL words.
  $\Rightarrow$ more difficult

This gives additional problems:

- *Problem 1:* Every measurement shows 4 table accesses.
  $\Rightarrow$ Unknown ordering!
  - Instead of using individual bits, use sum of 4 bits.
- *Problem 2:* With $\Pr \approx 1/3$, a cache line is used twice.
  $\Rightarrow$ We don't know which!
  - Possible: Guess which access occurs twice.
  - Better: Discard measurement.

Attack Overview

Attack parameters against target stream ciphers:

	LFSR size	Guess Steps	# Cache Measurements		Known output
			General	Pentium 4	
Sosemanuk	320	2^{32}	$160/b$ clks	40 clks	16 bytes
Snow 2.0	512	2^{32}	$256/b$ clks	64 clks	8 bytes
Sober-128	544	-	$544/b$ clks	136 clks	4 bytes
Turing	544	-	$544/b$ clks	136 clks	-

$\Rightarrow$ Given precise measurements, the attacks work within seconds on a PC.

Practical Relevance

However: Attacks require precise cache timing measurements.

What does that mean?

- We assume measurements to be noise-free, identifying exactly the correct table index.
- In practice:
  - We usually obtain a **set** of candidates for the table index.
  - Repeat experiment (same key/IV pair) to narrow down candidate set.
  - Try above attack for all remaining candidate combinations.
- Whether this is feasible or not depends on the target platform.
- Rule of thumb:
  The “cleaner” the target platform, the more likely the attack.

Thank you for your attention!

Questions? Comments?