Cache Timing Attacks in Symmetric Cryptography

Erik Zenner

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

JAIST, April 21, 2008

About this lecture

Thank you to

- Prof. Atsuko Miyaji for the kind invitation
- Mrs. Kumi Ito for the excellent organisation
- Everyone at the JAIST security labs for the kind welcome

About this lecture

Thank you to

- Prof. Atsuko Miyaji for the kind invitation
- Mrs. Kumi Ito for the excellent organisation
- Everyone at the JAIST security labs for the kind welcome

During this lecture:

Please feel free to interrupt and ask questions!

About the speaker

Name: Erik Zenner
Address: Just "Erik" is fine.

Short biography:

- 1991-1993: Education as a bank clerk (Germany)
- 1993-1999: BSc, MSc business and computer science (Germany + Scotland)
- 1999-2004: PhD in cryptography (Germany)
- 2004-2007: Chief cryptographer for Cryptico A/S (Denmark)
- Since 2007: Assistant professor at Technical University of Denmark

Research interest:

- Symmetric cryptography (in particular stream ciphers)
- Protocol design (in particular light-weight cryptography)
- Correct use of cryptography in practice

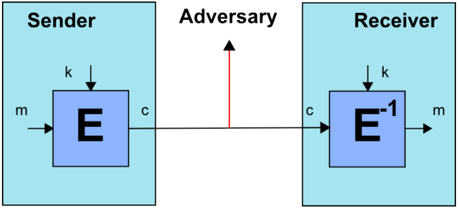
Outline

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations
2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations
3. Research Questions

Shannon's communication model

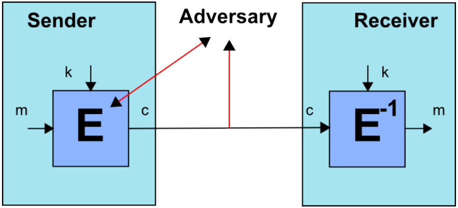
Traditional model (Shannon, 1949):



- Adversary knows the encryption algorithm E, but not the key k.
- Adversary observes ciphertext c.
- Adversary knows plaintext statistics, or maybe even part of plaintext m.

Side-channel attacks

Modified model:



In reality, additional information may leak to the adversary:

- Timing information
- Power consumption
- Fault induction

Relevance

Shannon model: Encryption happens in a secure box.

Side-channel model: Adversary has access to that box and can obtain certain real-world information.
Examples:

- Smart card
- Shared computer

Relevance: Not always, but sometimes!

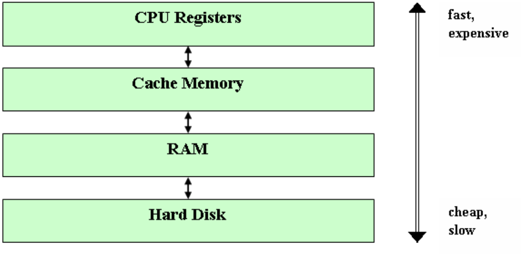
Responsibility: Cryptographers or engineers?

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations
2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations
3. Research Questions

Memory Hierarchy (Simplified)

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



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

Cache Workings

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

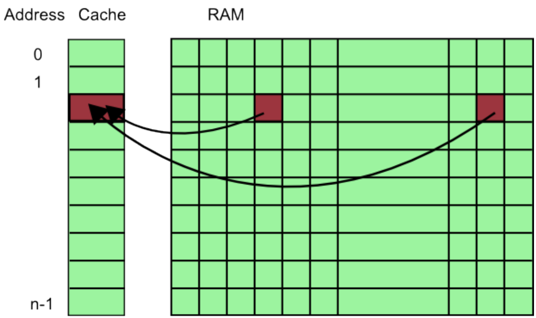
Similarly for writing data to RAM. See blackboard.

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 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.
See blackboard.

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

Outline

Advanced Encryption Standard (AES)

Officially: Encryption standard for (non-classified) U.S. government use.

De facto: World standard for encryption.

- 1997: NIST calls for candidate algorithms
- 1998: 15 algorithms submitted
- 1998-2000: *Demolition Derby*
- 2000: Selection of the winner (Rijndael)
- 2001: Publication of the standard

$\Rightarrow$ Widely evaluated, considered to be secure.

What need to know about AES...

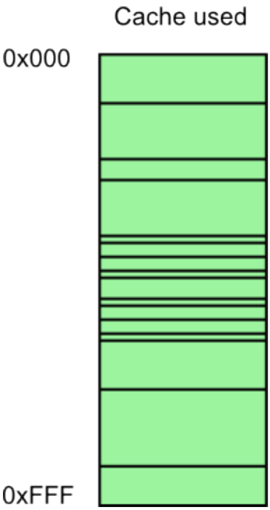
Notation: AES-128 transforms a 16-byte plaintext $m = (m_0, \ldots, m_{15})$ into a 16-byte ciphertext $c = (c_0, \ldots, c_{15})$, using a 16-byte key $k = (k_0, \ldots, k_{15})$.

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 ($i \in \{0, 1, 2, 3\}$).

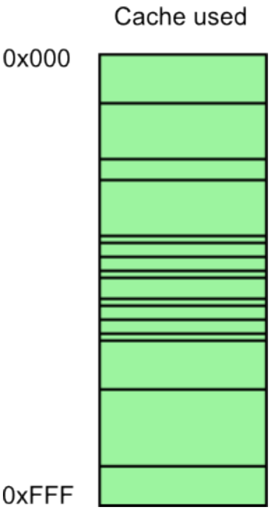
We ignore all the remaining steps here, we just point out that they also make use of the tables F_i.

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 S_j of candidates for $k_j = a \oplus m_j \ \ \forall a \notin \hat{L}$.
4. Re-run attack and intersect the resulting lists S_j.
5. Repeat until brute-force of the remaining key candidates becomes possible.

Consequences

Shannon's model: The best known attack against AES-128 is brute-force key search:
Can be achieved by 10^{22} Pentium 4 processors within 100 years.

Side-channel model: One possible attack is a cache timing attack:
Can be achieved by 1 Pentium 4 processor within a few microseconds.

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations
2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations
3. Research Questions

So is AES broken?

Cache Timing Attacks are only applicable under certain conditions:

- The adversary has (cache) access to the same machine as the user.
- The adversary knows the architecture and software of the machine very well.
- The adversary knows when the user will use encryption.
- Only few applications access the cache while the user is active (low noise).

But: Relevant in certain scenarios (e.g. servers, sandboxing of applications).

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.
  $\Rightarrow$ We do not learn the exact index, but only some index bits.
  See next slide.
- 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.

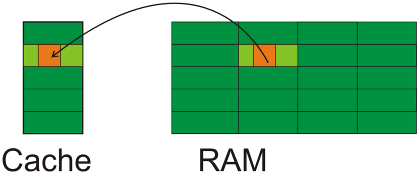
Nonetheless, as it turns out, these difficulties can be overcome in practice (Bernstein 2005, Osvik/Shamir/Tromer 2005, Bonneau/Mironov 2006).

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).
⇒ 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)

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.

Break

Any questions or comments?

Break

Any questions or comments?

Let's have a break!

After the break:
Applying cache timing attacks against stream ciphers.

Outline

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations
2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations
3. Research Questions

What is a stream cipher? (1)

Vernam cipher:

- Given an n-bit plaintext $m_1, \ldots, m_n$.
- Use an n-bit key $k_1, \ldots, k_n$.
- Produce an n-bit ciphertext $c_1, \ldots, c_n$ as follows:

$$\begin{array}{cccccc} & m_1 & m_2 & m_3 & \ldots & m_n \\ \oplus & k_1 & k_2 & k_3 & \ldots & k_n \\ \hline = & c_1 & c_2 & c_3 & \ldots & c_n. \end{array}$$

Security:

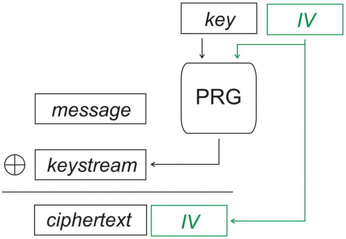
- If key is never re-used (and completely random)
  $\Rightarrow$ Vernam cipher is unbreakable (Shannon, 1949).
- If key is ever re-used
  $\Rightarrow$ Vernam cipher is totally insecure.

What is a stream cipher? (2)

Problem: n-bit keys very hard to manage in practice.

Solution:

- Use short key.
- Combine with initialisation vector (IV).
- Generate *keystream* using pseudo-random generator (PRG).
- Xor *keystream* to message for encryption.



Security: If *keystream* can not be distinguished from random bits, then stream cipher is as secure as Vernam cipher.

What is eStream?

Project: eStream is a subproject of the European ECRYPT project (2004-2008).
Purpose: Advance the understanding of stream ciphers and choose a portfolio of recommended algorithms.

Brief history:

- 2004 (Fall): Call for contributions.
- 2005 (Spring): Submission of 34 (!) stream ciphers for evaluation.
- 2006 (Spring): End of evaluation phase 1, reduction to 27 candidates.
- 2007 (Spring): End of evaluation phase 2, reduction to 16 finalists.
- 2008 (April 15): Announcement of the final portfolio of 8 ciphers.

Portfolio (Software): HC-128, Rabbit, Salsa20/12, Sosemanuk
Portfolio (Hardware): F-FCSR-H (v2), Grain, MICKEY (v2), Trivium

eStream Software Finalists

Cipher	Tables	Relevant
CryptMT	none	-
Dragon	Two 8 $\times$ 32-bit S-Boxes	†
HC-128	Two 512 $\times$ 32-bit tables	
HC-256	Two 1024 $\times$ 32-bit tables	†
LEX-128	One 8 $\times$ 8-bit S-Box (ref. code) Eight 8 $\times$ 32-bit S-Boxes (opt. code)	†
NLS	One 8 $\times$ 32-bit S-Box	†
Rabbit	none	-
Salsa-20	none	-
Sosemanuk	One 8 $\times$ 32-bit table, eight 4 $\times$ 4-bit S-Boxes (ref. code)	†

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 adversary'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 (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.

Assumption 3:

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

Formal model (1)

In order to formalise the adversary's ability, we use an oracle notation.

First, we consider the oracles available in traditional stream cipher analysis.

Traditional Oracles:

- $\textsc{KeySetup}()$: Sets up a new cipher instance. No output is returned.
- $\textsc{IVSetup}(N)$: Resets the cipher instance with initialisation vector N, as chosen by the adversary. No output is returned.
- $\textsc{Keystream}(i)$: Returns the keystream block i.

Formal model (2)

Next, we consider the oracles resulting from the new cache timing possibilities.

Cache Analysis Oracles:

In traditional stream cipher analysis, the adversary can use any of the following functions / oracles at will:

- $\mathrm{CA\_K{\scriptsize EY}S{\scriptsize ETUP}}()$: Returns a list of all cache accesses made by KeySetup().
- $\mathrm{CA\_IVS{\scriptsize ETUP}}(N)$: Returns a list of all cache accesses made by IVSetup(N).
- $\mathrm{CA\_K{\scriptsize EYSTREAM}}(i)$: a list of all cache accesses made by Keystream(i).

Discussion

Clarification:
This model is very generous towards the adversary. In the real world, he 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!

When using model for cryptanalysis

Voluntary Constraints:

In analysis, the following restrictions were made:

- We tried to find a **practical** attack, meaning that:
    - The adversary can only call the cache analysis oracles 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.
    - Distinguishing attack not sufficient.

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

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations

2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations

3. Research Questions

About HC-256

- Stream cipher (FSE 2004), eStream software finalist.
- **Key/IV:** 256 bit each.
- **Inner State:** Two tables, $1024 \cdot 32$ bit each.
  $\Rightarrow 65,536$ bits of inner state.
- **One Round:**
  - Update one of the tables.
  - Produce 32 bit of output.
  - See blackboard.
- **Performance:**
  - Designed for software.
  - Slow key/IV setup (due to table initialisation).
  - Fast keystream generation.

Sketch of the Attack

The adversary uses the following oracles:

- 6148 calls to $\text{CA\_K{\scriptsize EYSTREAM}}(i)$.
- 2048 calls to $\text{K{\scriptsize EYSTREAM}}(i)$.

Then he uses three layers of guess-and-verify to determine the inner state, followed by one step to recover the key:

1. Determine the block access ordering.
2. Guess-and-eliminate step.
3. Guess-and-determine step.
4. State inversion.

Step 1: Block Access Ordering (1)

Adversary makes 6148 calls to $\mathrm{CA\_K{\scriptsize EYSTREAM}}(i)$ and maps the resulting observations to inner state bits.

For each oracle call, we obtain

- 6 accesses to the moving register (Q on the blackboard).
  $\Rightarrow$ Useless, since the adversary knows their indices anyway.
- 5 accesses to the static register (P on the blackboard).
  $\Rightarrow$ New information about contents of the **moving** register.

Step 1: Block Access Ordering (2)

Problem: Mapping of cache accesses to state variables.

- Each oracle call: 5 cache accesses, e.g.:
  $001011xxxx$, $011100xxxx$, $010011xxxx$, $101101xxxx$, $111110xxxx$
- How to assign them to internal state variables? E.g.:
  $(00||Q_{13}^{(7..0)})$, $(01||Q_{13}^{(15..8)})$, $(10||Q_{13}^{(23..16)})$, $(11||Q_{13}^{(31..24)})$, $(Q_{22} \oplus Q_{-998})^{(9..0)}$

Solution: Simple internal consistency test works with high probability!

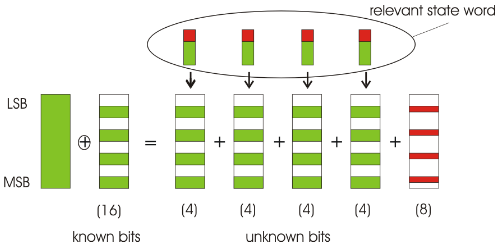
End of step 1:

For almost all inner state words, we know all upper half-bytes.
$\Rightarrow 2^{16}$ candidates for each inner state word.

Step 2: Guess-and-Eliminate Step (1)

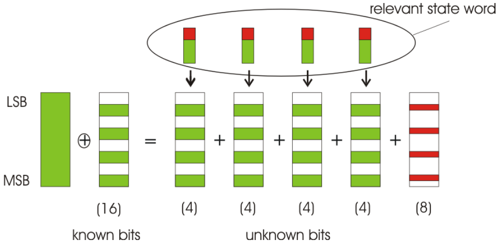
Adversary makes 2048 calls to $\textsc{Keystream}(i)$ and uses an internal equation to further reduce the number of candidates.

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



Problem: Carry bits complicate the equation.

Step 2: Guess-and-Eliminate Step (2)



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.

Solution: Guess the carry bits, too.

End of step 2:

2^8 remaining candidates for each inner state word.
$\Rightarrow$ Store in a table (size $\approx$ 3 MByte).

Step 3: Guess-and-Determine Step

Adversary uses guess-and-determine strategy with a different equation to determine the rest of the inner state.

Problem: Many bits have to be guessed before verification becomes possible.

- Guess 2^{48} assignments, obtain 32 verification bits. $\Rightarrow 2^{16}$ assignments remain.

Solution: Guesses start to overlap (See blackboard).

- Search tree grows less fast than in the beginning.
- After some steps, search tree starts to shrink.
- Maximum tree width: 2^{64} guesses.

End of step 3:

Full inner state for one point in time has been recovered.

Step 4: Inversion Step

Adversary runs the algorithm backwards to determine the initial state and the key.

- HC-256 can efficiently be run backwards.
- First invert keystream generation to determine initial state.
- Then invert key/IV setup to determine key.

End of step 4:

Full key has been reconstructed.
Adversary can encrypt and decrypt any message under this key.

The Attack in a Nutshell

Requirements:

- 6148 precise cache timing measurements.
- 2^{16} known plaintext bits.
- Computational effort corresponding to testing $\approx 2^{55}$ keys.
- ≈ 3 MByte of memory.

Practical Relevance

Question: So is HC-256 broken?

Answer: Not unless you already stopped using AES for security reasons.

- Attack uses very strong assumptions.
- AES would be completely broken under the same assumptions.

But: Relevance of cache timing attacks is currently an open issue.

- A distinguisher using 2^{60} known plaintexts is sufficient to discard a cipher.
- How about a key recovery attack using $\approx 6,000$ precise cache timings?

Outline

1. Introduction to Cache Timing Attacks
   - Side-channel Attacks
   - Cache-timing Attacks
   - The AES Case
   - Comments and Observations

2. Analysing Stream Ciphers
   - Stream Ciphers
   - Model for Analysis
   - Attacking HC-256
   - Design Recommendations

3. Research Questions

Other eStream Software Finalists

- **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, even though they were not designed to that purpose (exception: Salsa).
- Work on cryptanalysis is still in progress.
  - No one-size-fits-all attack
  - Different ciphers pose different problems
  - Individual analysis required
- **Guess:** Attacks are possible, but require some thought (exception: LEX).

A Trivial Design Recommendation

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.

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

We have already experienced the LSB problem for HC-256:

- LSB are not visible for the adversary by cache timing measurements.
- This creates problems for him if many unknown bits influence all parts of the computations (e.g., via carry).

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.

Outline

Developing Research Field

Research questions on cache-timing attacks:

- Vulnerability of various cryptographic primitives:
  - Can we attack stream ciphers, MACs, digital signatures, etc.?
- New attacks based on cache timings:
  - In what ways can an adversary make use of a cache timing weakness?
- Cryptographic counter-measures:
  - How can we design a cipher such that no cache-timing attack is possible?
- Engineering counter-measures:
  - How can we implement a cipher such that no cache-timing attack is possible?

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?

Questions? Comments?

Questions? Comments?

Thank you
for your
attention!