A Cache Timing Analysis of HC-256

Erik Zenner

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

Abstract. In this paper, we describe a cache-timing attack against the stream cipher HC-256, which is the strong version of eStream winner HC-128. The attack is based on an abstract model of cache timing attacks that can also be used for designing stream ciphers. From the observations made in our analysis, we derive a number of design principles for hardening ciphers against cache timing attacks.

Keywords: Cryptanalysis, side-channel attack, cache timing attack, stream cipher, HC-256.

1 Introduction

Cache timing attacks are a new class of side-channel attacks. They received significant attention after being applied to the Advanced Encryption Standard (AES) independently by Bernstein [1] and Osvik, Shamir, and Tromer [12,13] in 2005[1]. The idea is that in some settings, the adversary can obtain information about the cache accesses of a legitimate party by measuring timings. Optimised software implementations of the AES turned out to be particularly vulnerable to this kind of attack.

The discovery was met with great interest. Subsequent research verified the correctness of the findings [11,10,9,15], improved the attack technically [14,3,8] or algorithmically [5], and devised and analysed countermeasures [6,4,16].

However, the focus of the attacks was on the AES, and the countermeasures mainly targeted the *implementation* of cryptographic designs. In this paper, we take a different approach: We discuss how cipher *designers* can make such attacks more difficult. In order to demonstrate our approach, instead of considering a block cipher like AES, we analyse the stream cipher HC-256.

1.1 Cache Timing Attacks

Cache timing attacks exploit that loading data into a CPU register is faster when done from cache than from RAM. By measuring cache timings, an observer can obtain information about the inner state of a cipher. In the following, we give a simplified description of cache timing attacks; for a more complete description, see e.g. [13,9].

[1] For prior work on cache timing attacks, see the references contained in [1] and [12].

R. Avanzi, L. Keliher, and F. Sica (Eds.): SAC 2008, LNCS 5381, pp. 199–213, 2009.

Cache workings: The CPU cache of modern processors is organised into blocks of s bytes. Correspondingly, RAM is considered to be (virtually) divided into s-byte blocks. When loading data from RAM into a CPU register, the system first checks whether the corresponding RAM block already lies in cache. If yes, it is loaded directly from cache, which is very fast. If not, it is first loaded from RAM to cache, which takes longer. Mapping from RAM to cache is typically by a simple modulo operation, i.e. if the cache can hold n blocks and if the data lies in RAM block a, then it is loaded into cache block $a \bmod n$. This means that neighbouring data in RAM (e.g. tables) stays clustered in cache.

A simple attack: As an example, consider the **prime-then-probe** method presented in [13]. The adversary starts by filling all the cache with his own data. Then the legitimate user U gets the read/write token. U loads the data required for his own computations into cache, where it evicts the adversary's data. When the adversary reobtains the read/write token, he tries to reload his own data from cache. For each cache block, if this takes long, it means that U has evicted the corresponding data.

From this analysis, the adversary obtains a profile of cache blocks that have been used by U. This profile is a noisy version of the cache blocks that have been used for the encryption. By repeating the experiment a number of times, a good approximation of the real cache access profile can be obtained.

Note that the adversary does not learn the *data* that was written in the cache by U – he learns something about the *addresses* of the data that was used. In the case of the AES, this corresponds to the indices of the S-box entries used for encryption, which in turn can be used for an attack.

Practicality: Cache timing attacks require cache timing measurements of sufficient precision. In addition, the experiment has to be repeated sufficiently often. Obviously, these requirements are not always met. However, they are relevant in shared server and in sandbox scenarios, and Bertoni et al. [3] show how to use cache timings if the adversary has physical access to a device, making the attack much more realistic.

Responsibility: Some researchers claim that defending against side-channel attacks should be the responsibility of the implementer, not the cipher designer. However, this view is not shared by everyone. As an example, the AES was chosen partially due to its inherent resistance against side-channel attacks (see e.g. Section 7 of [1]). The reason is that algorithms are designed only once, but implemented many times on many platforms. Thus, if side-channel attacks can be avoided in the design phase, implementation becomes easier, which seems to be preferable. In order to emphasise the designer's responsibility, we use a simplified terminology in this paper: We say that a cipher can be "broken" in a cache timing model if an unprotected implementation is vulnerable to a cache timing attack.

2 The Stream Cipher HC-256

HC-256 was proposed by Wu in [17], and its reduced version HC-128 is part of the eStream portfolio [7]. The cipher is based on two large, key-based tables (i.e., no fixed S-boxes) that change content over time. With each call to the keystream generation function, the cipher updates one table entry and outputs one 32-bit keystream word.

Notation: HC-256 requires a 256-bit key K and a 256-bit IV IV. It uses two tables P and Q, which contain 1024 32-bit words each. Table entries are identified by $P[i]$ and $Q[i]$.

In the following, $\oplus$ denotes xor, $||$ concatenation (most significant bits first), $\ggg$ a circular right shift, $\boxplus$ addition modulo 2^{32}, and $\boxminus$ subtraction modulo 2^{10}.

If X is a word, we denote by $X^{(b..a)}$ the bits $b..a$, where $b > a$. For all notations, the most significant bits are written to the left, while the least significant bits are written to the right. Thus, we can write $X = X^{(31..0)}$.

Auxiliary Functions: The following auxiliary functions on 32-bit variables are used:

$$
\begin{aligned}
f_1(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3) \\
f_2(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10) \\
g_1(x,y) &= ((x \ggg 10) \oplus (y \ggg 23)) \boxplus Q[(x \oplus y)^{(9..0)}] \\
g_2(x,y) &= ((x \ggg 10) \oplus (y \ggg 23)) \boxplus P[(x \oplus y)^{(9..0)}] \\
h_1(x) &= Q[00||x^{(7..0)}] \boxplus Q[01||x^{(15..8)}] \boxplus Q[10||x^{(23..16)}] \boxplus Q[11||x^{(31..24)}] \\
h_2(x) &= P[00||x^{(7..0)}] \boxplus P[01||x^{(15..8)}] \boxplus P[10||x^{(23..16)}] \boxplus P[11||x^{(31..24)}]
\end{aligned}
$$

Key/IV Setup: For initialisation, the key is split into 32-bit words $K[0], \ldots, K[7]$, and the IV is split into 32-bit words $IV[0], \ldots, IV[7]$. With the help of an auxiliary array $W[0], \ldots, W[2559]$ and a global counter variable r, the algorithm can be described as in Figure 1.

Init(K, IV)
1. For $i = 0, \ldots, 7$:
2. $\quad W[i] = K[i]$
3. For $i = 8, \ldots, 15$:
4. $\quad W[i] = IV[i-8]$
5. For $i = 16, \ldots, 2559$:
6. $\quad W[i] = f_2(W[i-2]) \boxplus W[i-7] \boxplus f_1(W[i-15]) \boxplus W[i-16] \boxplus i$
7. For $j = 0, \ldots, 1023$:
8. $\quad P[j] = W[j+512]$
9. $\quad Q[j] = W[j+1536]$
10. Set $r = -4096$
10. Repeat 4096 times:
10. $\quad$ **Next()** (* Ignore the output *)

Fig. 1. Key/IV setup for HC-256

Next()
1. Set $j = r \bmod 1024$
2. If $((r \bmod 2048) \in \{0, \ldots, 1023\})$:
3. $\quad P[j] = P[j \boxminus 1024] \boxplus P[j \boxminus 10] \boxplus g_1(P[j \boxminus 3], P[j \boxminus 1023])$
4. $\quad z_r = h_1(P[j \boxminus 12]) \oplus P[j]$
5. Else:
6. $\quad Q[j] = Q[j \boxminus 1024] \boxplus Q[j \boxminus 10] \boxplus g_2(Q[j \boxminus 3], Q[j \boxminus 1023])$
7. $\quad z_r = h_2(Q[j \boxminus 12]) \oplus Q[j]$
8. $r = r + 1$

Fig. 2. Keystream generation for HC-256

Keystream Generation: The r-th call to the **Next()** function updates one table entry and produces one 32-bit output word z_r. The function is described in Figure 2. Note that $r = 0$ for the first output word.

3 Mapping Measurements to Inner State

3.1 Preliminaries

State-dependent table lookups: From a cache timing attack, the adversary learns (part of) the table indices that were accessed by the encryption algorithm. However, most table lookups made by HC-256 depend on a public counter which is known to the adversary anyway. The only exceptions are the state-dependent table lookups within the functions g_1 and h_1 (leaking information about table P) as well as g_2 and h_2 (table Q).

Observable index bits: Ideally, the adversary would learn the full table index from each cache observation. In practice, however, the cache is organised in blocks that store several RAM table entries. Thus, all the adversary can learn from his measurements is the cache block containing the table entry.

In the following, we assume that the tables P and Q are perfectly aligned with the cache blocks[2]. Thus, the tables themselves can be considered as being split into blocks that have the same size as the cache blocks.

The cache block size is processor dependent and varies typically between 16 and 128 byte. In the following, we use a cache block size of 64 byte, since it is currently particularly wide-spread (e.g. in Pentium 4 and Athlon). Since tables P and Q have 1024 entries with an entry size of 4 byte each, each table block contains 16 table entries, and each table consists of 64 blocks. Thus, by measuring cache timings, the adversary learns index bits 4..9, but not 0..3.

Note that if the cache block size is smaller (larger) than 64 byte, he will obtain more (less) information about the table entries.

[2] If this is not the case, our attack becomes easier, since unaligned table entries leak additional information about the inner state.

3.2 Keystream Generation vs. Key/IV Setup

The functions g_1, g_2, h_1, and h_2 are accessed both during keystream generation and key/IV setup. However, during key/IV setup, *all* entries in the tables W, Q and P are accessed at least once. Thus, the adversary will obtain no additional information compared to the standard model.

Instead, we concentrate on the keystream generation phase, where we make repeated measurements for each output block. This is modelled by giving the adversary access to two oracles:

- KEYSTREAM(i): The adversary requests the cipher to return the i-th keystream block to him. The block length depends on the cipher design.
- SCT_KEYSTREAM(i): The adversary obtains an unordered list of all cache accesses made by KEYSTREAM(i).

A justification and generalisation of this model is given in Section 6.

3.3 Initial State Candidates

If the adversary calls the SCT_KEYSTREAM(R) oracle, this corresponds to a call to the **Next()** function. Consider such a call for a round r with $(r \bmod 2048) \in \{0, \ldots, 1023\}$, i.e. code lines 3 and 4 are executed. From functions g_1 and h_1, he observes either 4 or 5 accesses to table Q, as follows.

Function h_1: In function h_1, table Q is accessed at indices $(00||P[j \boxminus 12]^{(7..0)})$, $(01||P[j \boxminus 12]^{(15..8)})$, $(10||P[j \boxminus 12]^{(23..16)})$, and $(11||P[j \boxminus 12]^{(31..24)})$. While in general, the adversary does not know which table access belongs to which variable, things are more obvious here. Each of the four 10-bit indices starts with a unique 2-bit prefix and can thus be clearly assigned to one of the four variables. Thus, if it were not for code line 3, the adversary could immediately determine the upper half-bytes for $P[j \boxminus 12]$.

Function g_1: However, in the same function call, g_1 accesses table Q at index $(P[j \boxminus 3] \oplus P[j \boxminus 1023])^{(9..0)}$. This index can have any of the prefixes $00, 01, 10$, or 11. Thus, we can not distinguish it from one of the accesses by h_1 which has the same prefix (unless it accidentially uses the same cache block, which happens with probability $1/16$).

Concluding, for three of the four bytes in $P[j \boxminus 12]$, we know precisely their upper half-byte. For the fourth one, we normally have two candidates, which we can not distinguish without additional information. In addition, for $(P[j \boxminus 3] \oplus P[j \boxminus 1023])$, we know exactly what the bits 9 and 8 are, and we have two candidate assignments for bits 7..4.

Functions h_2 and g_2: Exactly the same observations hold for table P for rounds r with $(r \bmod 2048) \in \{1024, \ldots, 2047\}$.

4 Reconstructing the Full Inner State

4.1 Notation

Before considering several calls to the **Next()** function, we have to define a unique notation for the table entries. Since the table is constantly updated, we have to make it clear which of a succession of values in e.g. table cell $P[12]$ we mean.

To this end, for table P, we write P_u when we mean the u-th value that was updated for this table, where P_0 is updated in round $r = 0$. As an example, table cell $P[12]$ has the value P_{-1012} after initialisation, obtains value P_{12} in round $r = 12$ and value P_{1036} in round $r = 2060$.

Similarly, for table Q, we write Q_u when we mean the u-th value that was updated for this table, where Q_0 is updated in round $r = 1024$. As an example, table cell $Q[12]$ has the value Q_{-1012} after initialisation, obtains value Q_{12} in round $r = 1036$ and value P_{1036} in round $r = 3084$.

The following table describes the the relationship between rounds and sequence words that are used for the attack.

Round	Table P	Table Q
$0, \ldots, 1023$	$P_0, \ldots, P_{1023}$	-
$1024, \ldots, 2047$	-	$Q_0, \ldots, Q_{1023}$
$2048, \ldots, 3071$	$P_{1024}, \ldots, P_{2047}$	-
$3072, \ldots, 4095$	-	$Q_{1024}, \ldots, Q_{2047}$
$4096, \ldots, 5119$	$P_{2048}, \ldots, P_{3071}$	-
$5120, \ldots, 6143$	-	$Q_{2048}, \ldots, Q_{3071}$
$6144, \ldots, 7167$	$P_{3072}, \ldots, P_{4095}$	-
$7168, \ldots, 8191$	-	$Q_{3072}, \ldots, Q_{4095}$

4.2 Step 1: Determining the Half-Bytes

The purpose of the first step is to uniquely identify the correct assignments to the upper half-bytes of $P_{1024}, \ldots, P_{3083}$ and $Q_{1024}, \ldots, Q_{3071}$.

Measurement: By using the SCT_KEYSTREAM() oracle for rounds

$$r = 25, \ldots, 1023,\ r = 2048, \ldots, 3071,\ r = 4096, \ldots, 5119,\ r = 6144, \ldots, 6176,$$

the adversary observes partial information about table entries as described in Subsection 3.3. This gives him 2 candidate assignments for each of the following lines:

From h_1				From g_1
$P_{13}^{(7..4)}$	$P_{13}^{(15..12)}$	$P_{13}^{(23..20)}$	$P_{13}^{(31..28)}$	$P_{22}^{(9..4)} \oplus P_{-998}^{(9..4)}$
$P_{14}^{(7..4)}$	$P_{14}^{(15..12)}$	$P_{14}^{(23..20)}$	$P_{14}^{(31..28)}$	$P_{23}^{(9..4)} \oplus P_{-997}^{(9..4)}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$P_{3092}^{(7..4)}$	$P_{3092}^{(15..12)}$	$P_{3092}^{(23..20)}$	$P_{3092}^{(31..28)}$	$P_{3101}^{(9..4)} \oplus P_{2081}^{(9..4)}$

In particular, for the equations $P_{1033} \oplus P_{13}, \ldots, P_{3092} \oplus P_{2072}$, we have 2 candidates for the bits 7..4 from g_1. At the same time, from h_1, we have 1 candidate (with probability $\approx 3/4$) or 2 candidates (with probability $\approx 1/4$) for bits 7..4 of the corresponding values $P_{13}, \ldots, P_{3092}$.

A simple consistency check: We will now try to figure out which of the two candidates for each g_1 equation is the correct one. First note that with probability $1/16$ there is really only one candidate for this equation, namely if bits 7..4 are the same as for h_1. If this is not the case, there are three subcases:

1. For the corresponding h_1 values, there is only 1 candidate each. In this case (which happens with prob. $\approx 9/16$), checking by xoring those h_1 values will always identify the correct candidate for the g_1 value.
2. One of the h_1 values has 1 candidate and one has 2 candidates. In this case (which happens with prob. $\approx 6/16$), there is only one wrong combination of h_1 candidates, and it is identical to the wrong g_1 candidate with probability $1/16$. Thus, the test identifies the wrong g_1 candidate with probability $15/16$.
3. Both h_1 values have 2 candidates. In this case (which happens with prob. $\approx 1/16$), there are 3 wrong combinations of h_1 candidates. They identify the wrong g_1 candidate with probability $\frac{15 \cdot 15 \cdot 14}{16^3}$.

Concluding, the probability of identifying a wrong g_1 candidate by a simple test is

$$\frac{1}{16} + \frac{15}{16} \cdot \left(\frac{9}{16} \cdot 1 + \frac{6}{16} \cdot \left(\frac{15}{16} \right) + \frac{1}{16} \cdot \left(\frac{15 \cdot 15 \cdot 14}{16^3} \right) \right) \approx 0.9646.$$

Consequence: In the following, we will thus assume that the correct candidates for equations $P_{1033} \oplus P_{13}, \ldots, P_{3092} \oplus P_{2072}$ have been identified. In reality, there will be a small number of such equations that have two candidates, but the percentage is small enough not to significantly influence the analysis in the following sections (it will only make an implementation of the attack slightly messier).

Once the correct candidates for equations $P_{1033} \oplus P_{13}, \ldots, P_{3092} \oplus P_{2072}$ are known, we can also identify the correct candidates for the h_1 values of the same lines. Thus, in the following, we can assume that the upper half-bytes are known for the h_1 values under consideration, i.e. the sequence words $P_{1024}, \ldots, P_{3083}$.

By repeating the same procedure for rounds

$$r = 1049, \ldots, 2047,\ r = 3072, \ldots, 4095,\ r = 5120, \ldots, 6143,\ r = 7168, \ldots, 7188,$$

the same bits can be determined for sequence words $Q_{1024}, \ldots, Q_{3071}$.

4.3 Step 2: Reducing the Number of Candidates

In the second step, we will reduce the number of candidates for $Q_{1024}, \ldots, Q_{3059}$ and $P_{2048}, \ldots, P_{3071}$ from 2^{16} to 2^8.

Sequence words $Q_{1024}, \ldots, Q_{2035}$: Let us consider the calls to the function

$$z_r = h_2(Q[j \boxminus 12]) \oplus Q[j]$$

that occur in rounds $r = 3084, \ldots, 4095$. They access the sequence words Q_{1024}, $\ldots$, Q_{2047} and $P_{1024}, \ldots, P_{2047}$. According to Subsection 4.2, we know all upper half-bytes for these entries. Now we have to try and learn as much as possible about the remaining inner state from this information.

Let $\gamma_0, \ldots, \gamma_3 = (00||Q[j \boxminus 12]^{(7..0)}), \ldots, (11||Q[j \boxminus 12]^{(31..24)})$. Then we can re-write the above equation as follows:

$$z_r \oplus Q[j] = P[\gamma_0] \boxplus P[\gamma_1] \boxplus P[\gamma_2] \boxplus P[\gamma_3] \tag{1}$$

Remember that the adversary knows the keystream word z_r. Also note that for $Q[j]$, $Q[j \boxminus 12]$ and for all $P[\gamma_i]$ involved, we know the upper half-bytes. We will now proceed by guessing the remaining 16 bits of $Q[j \boxminus 12]$ and then verifying the result by using eq. (1).

If the equation would use $\oplus$ instead of $\boxplus$, verification would be straightforward. We would use the upper half-bytes to obtain 16 linear equations in GF(2). Since we also have to guess 16 bit for $Q[j \boxminus 12]$, only one false guess would pass this test on average.

However, for addition, we have to take carries into account. If we write $A^I, \ldots, A^{IV}$ instead of $A^{(7..4)}, \ldots, A^{(31..28)}$ for the four upper half-bytes of a word A, then we can write 4 verification equations as follows:

$$\begin{array}{llllll}
z_r^I & \oplus Q[j]^I & = P[\gamma_0]^I & \boxplus P[\gamma_1]^I & \boxplus P[\gamma_2]^I & \boxplus P[\gamma_3]^I \boxplus c_0 \\
z_r^{II} & \oplus Q[j]^{II} & = P[\gamma_0]^{II} & \boxplus P[\gamma_1]^{II} & \boxplus P[\gamma_2]^{II} & \boxplus P[\gamma_3]^{II} \boxplus c_1 \\
z_r^{III} & \oplus Q[j]^{III} & = P[\gamma_0]^{III} & \boxplus P[\gamma_1]^{III} & \boxplus P[\gamma_2]^{III} & \boxplus P[\gamma_3]^{III} \boxplus c_2 \\
z_r^{IV} & \oplus Q[j]^{IV} & = P[\gamma_0]^{IV} & \boxplus P[\gamma_1]^{IV} & \boxplus P[\gamma_2]^{IV} & \boxplus P[\gamma_3]^{IV} \boxplus c_3
\end{array}$$

Here, $c_0, \ldots, c_3$ are the carry values, taken from $\{0, 1, 2, 3\}$.

Thus, if we want to use the above equations to verify our guess for $Q[j \boxminus 12]$, we have to guess the carry values, too. In total, this gives us $2^{16} \cdot 2^8 = 2^{24}$ possible guesses. On the other hand, we have 16 verification bits. This means that on average, 2^8 guesses for $Q[j \boxminus 12]$ will survive the test. For the table entries $Q_{1024}, \ldots, Q_{2035}$, we write these guesses into a table.

Sequence words $Q_{2036}, \ldots, Q_{3059}$: It remains to reconstruct the remaining words $Q_{2036}, \ldots, Q_{3059}$, which can be done in a similar manner by considering rounds $r = 5120, \ldots, 6143$. These rounds use the sequence words $Q_{2036}, \ldots, Q_{3071}$, as well as some of the sequence words $P_{2048}, \ldots, P_{3071}$. Using the same technique as above, we can reduce the number of candidates for $Q_{2036}, \ldots, Q_{3059}$ to approximately 2^8 candidates each.

Sequence words $P_{2048}, \ldots, P_{3071}$: The same technique can also be applied to reduce the number of candidates for the sequence words $P_{2048}, \ldots, P_{3071}$. We do this by considering the rounds $r = 4108, \ldots, 5119$, which use sequence words

$P_{2048}, \ldots, P_{3071}$ as well as $Q_{1024}, \ldots, Q_{2047}$. From this, we can reduce the number of candidates for $P_{2048}, \ldots, P_{3071}$ to 2^8. Afterwards, we consider rounds $6144, \ldots, 6155$, which use sequence words $P_{3060}, \ldots, P_{3083}$ as well as some of the table entries $Q_{2048}, \ldots, Q_{3071}$.

Resulting table: For $Q_{1024}, \ldots, Q_{3059}$ and $P_{2048}, \ldots, P_{3071}$, the surviving candidate words are written in a table. The total size of this table is $3060 \cdot 2^8 \cdot 4 \approx 3 \cdot 2^{20}$ byte, i.e. 3 MByte.

4.4 Step 3: Backtracking Attack

In the final step, we reduce the number of candidates for $Q_{1024}, \ldots, Q_{2047}$ and $P_{2048}, \ldots, P_{3071}$ to one.

Reconstructing table Q: Consider code line 6 as it is called in round $r = 5120$. It has the following form:

$$Q_{2048} = Q_{1024} \boxplus Q_{2038} \boxplus g_2(Q_{2045}, Q_{1025}).$$

This means that the equation uses the variables $Q_{1024}, Q_{1025}, Q_{2038}, Q_{2045}, Q_{2048}$ and an entry of table P with unknown index. For each of these 6 variables, we have an average of 2^8 possible assignments. If we guess all of these assignments, we obtain 2^{48} possible candidates. Since wrong guesses for the 32-bit values satisfy a linear equation with probability $1/2^{32}$, only $\approx 2^{16}$ of them remain as valid states.

We proceed in the same way for round $r = 5121$, which requires variables $Q_{1025}, Q_{1026}, Q_{2039}, Q_{2046}, Q_{2049}$ and an entry of table P. Note that Q_{1025} is already known from last round, meaning that we only have to guess 5 variables[3]. Our search space increases to $2^{16} \cdot 2^{40} = 2^{56}$, then it collapses to 2^{24} when filtering out the assignments that don't fulfil the equation.

Repeating the same step for round $r = 5122$ increases our search tree to 2^{64}, then collapsing it to 2^{32}. For round $r = 5123$, however, *two* of the required variables are already known. This means that only 4 variables have to be guessed, and the search tree expands to 2^{64} and reduces itself to 2^{32} after verification.

It continues to behave that way until round $r = 5127$. In this round, we need *three* variables that have already been guessed before. This means that the tree only expands to 2^{56} candidates and then collapses back to 2^{24}. From now on, the tree size will reduce itself with every round, until round $r = 5130$ when it has size ≈ 1 after verification, i.e. only valid guesses remain. From now on, every candidate guess can be verified right away.

Concluding, after running through rounds $r = 5120, \ldots, 6143$, we have reconstructed the correct solution for table entries $Q_{1024}, \ldots, Q_{2047}$.

[3] Of course, there is also a possibility that the table entry for table P repeats itself, but this probability is not very high in the first rounds. Should this happen by chance, the attack becomes even more efficient.

Reconstructing table P: Note that from the guesses above, a significant number of entries for table P have already been reconstructed. There are numerous possiblities for determining the remaining entries. A very simple one would be to run the same attack as above, using code line 3 instead of line 6. Note that this requires extra cache timings to reduce the number of candidates for $P_{3072}, \ldots, P_{4095}$ to 2^8, each.

A more intelligent approach uses code line 4 for rounds $r = 5008, \ldots, 5119$. This code line requires only two guesses from table P (with high probability at least one of them is known anyway) and allows verification against the full 32-bit keystream word (16 bits of which have not yet been taken into account). This technique should rapidly identify the missing entries for table P.

5 Consequences

5.1 Cost of the Attack

The above attack retrieves the full contents of tables P and Q at the beginning of round $r = 6144$. Given such a snapshot of both tables, we can run the generator forwards to generate previously unknown keystream bits. We can also run it backwards to retrieve the key (the state update function and the key/IV setup are invertible). This shows that an attack is even possible for a synchronous cache adversary (not only for an asynchronous adversary, as suspected by Bernstein [2])[4].

The main computational step is the backtracking attack, which requires less than $5 \cdot 2^{64} < 2^{67}$ computational steps that consist in verifying one equation. Since the key/IV setup of HC-256 has to compute the same equation $4096 = 2^{12}$ times (plus does a number of other computations), the effort is less than trying 2^{55} keys in a brute-force setting. The memory requirements are around 3 MB for the candidate tables, plus a little memory for the search tree (implementing it in a depth-first search fashion keeps the memory consumption low). In addition, we have assumed the availability of precise cache measurements for 6148 chosen rounds, and of 2048 known keystream words. We point out that our attack is not optimised in any way. It is likely that a better attack can be found using less cache measurements and computational effort. Nonetheless, the huge number of necessary cache timing measurements required for this attack indicates how difficult it would be to apply a similar attack in the standard model.

If the attack is run on a processor with a different cache block size, efficiency is influenced. For example, if the cache block size is only 32 byte instead of 64 byte, the adversary learns 7 bit for each table lookup. In this case, no backtracking phase is required at all – the solution can already be determined by the reduction step in Subsection 4.3. On the other hand, if the cache block size is e.g. 128 byte, then only 5 bits for each table lookup are recovered, and the backtracking gets a lot more expensive.

[4] For a definition of synchronous and asynchronous cache adversaries, see Section 6.

5.2 Design Recommendations

While trying to break HC-256 (and doing initial analysis of other stream ciphers), we met a number of obstacles that might be possible defense mechanisms against cache timing attacks. Some of them may be known to cipher designers already, but to the best of our knowledge, they have not been documented. Thus, we provide a short list of design recommendations that make cache timing attacks more difficult if use of tables can not be avoided altogether:

1. Make as many table accesses for one function call as possible. This makes things harder for a synchronous adversary, who has to match the observed indices to the inner state. For HC-256, this matching was relatively easy, which made the attack possible in the first place.
2. Make the inner state size large compared to the information obtained from one cache measurement. In the case of HC-256, one call to **Next()** yields 32 bit of keystream information and 52 bit of side-channel information. Because of the large inner state, this means that at least $65,536/84 \approx 780$ precise cache access measurements (or many more noisy ones) have to be made to retrieve the inner state.
3. Exploit that the least significant bits of the table index remain unknown (in our analysis, bits 3..0). This can be achieved by using state update and output generation functions that generate a lot of diffusion without the use of S-boxes. As an example, functions using carry (like addition and multiplication) are suitable for this purpose.
4. Use variable tables instead of (fixed content) S-boxes. This gives the adversary insecurity both about the input and the output of the tables.

6 Attack Model

In the following, we justify and generalise the abstract attack model that was used for our attack, such that it also can be used to analyse other stream ciphers.

6.1 Motivation

Whether or how cache timing attacks can be used against a cipher depends on the details of the system deploying it. This is not helpful for cipher designers who are not allowed to make assumptions about the deployment environment. While it is possible that certain attack options are not available in a fielded system, the cipher designer must not rely on this unavailability.

Thus, he works under worst-case assumptions. As an example, while most practical systems will not give the adversary 2^{40} plaintext/ciphertext pairs, ciphers are nonetheless designed to withstand an attack that has this amount of information available. Unless we want to make very detailed restrictions on how the cipher is to be used, overestimating the adversary's abilities is a key strategy.

A cipher designer who is concerned about cache timing attacks has to proceed in the same way. He has to assume that the adversary gets the maximum amount of information, and then see what damage this would do to the cipher. Ciphers secure under such a model will most likely be secure in practice, too.

6.2 Standard Adversary

In traditional (non-side-channel) stream cipher design, the adversary is assumed to have the following oracles available:

- KeySetup(): The adversary requests the cipher to be re-initialised with a new key. No output is returned.
- IVSetup(N): The adversary requests the cipher to be re-initialised with the initialisation vector N that has not been used before. No output is returned.
- Keystream(i): The adversary requests the cipher to return the i-th keystream block to him. The block length depends on the cipher design.

An adversary is considered successful if he can distinguish an instance of the stream cipher from a random function producing appropriately formatted (but random) answers to his oracle queries. A cipher is considered secure if for any adversary, the success probability is less than that of the generic adversary using the same computational resources on brute-force key testing.

6.3 Synchronous Cache Adversary

The notion of synchronous cache attacks was introduced by Osvik et al. in [13]. In such an attack, the adversary interacts with the encryption code through some type of interface, and he obtains additional information by making cache measurements before or after execution of this code.

In our model, such an adversary can use the same oracles as the standard adversary. In addition, he also has access to the following cache timing oracles:

- SCT_KeySetup(): The adversary obtains a list of all cache accesses made by KeySetup().
- SCT_IVSetup(N): The adversary obtains a list of all cache accesses made by IVSetup(N).
- SCT_Keystream(i): The adversary obtains a list of all cache accesses made by Keystream(i).

In particular, this reflects accurate measurements in a prime-then-probe attack, which seems to be the strongest SCT technique to date; making weaker assumptions would not cover this attack method adequatly. Note that the attack described in Sections 3 and 4 use the synchronous attack model.

6.4 Asynchronous Cache Adversary

While a synchronous adversary has to wait until user U has finished the execution of a certain operation, asynchronous adversaries run *in parallel* to U. This is possible e.g. on processors with hyperthreading. In this setting, the adversary can constantly monitor the cache state, which gives him an *ordered* list of all cache accesses made during the observation.

Osvik et al. [13] assume that the adversary obtains no additional information beyond the cache accesses. However, from a designer's point of view, we can

not restrict ourselves in this way. It is easy to imagine an adversary who both controls some of the input/output data and observes cache behaviour. Thus, in our model, an asynchronous cache adversary has access to the standard oracles as well as the following side-channel oracles:

- ACT_KeySetup(): The adversary obtains a list of all cache accesses made by KeySetup() in *chronological order*.
- ACT_IVSetup(N): The adversary obtains a list of all cache accesses made by IVSetup(N) in *chronological order*.
- ACT_Keystream(i): The adversary obtains a list of all cache accesses made by Keystream(i) in *chronological order*.

6.5 Discussion

Our attack model abstracts away a number of practical difficulties the adversary might encounter:

- The encryption process is not the only one using the cache. Cache accesses made by other processes generate false positives. Thus, instead of a list of encryption cache accesses, a real-world adversary only obtains a list of cache blocks that have *not* been used by the encryption process.
- Cache timing measurements are subject to timing noise. Thus, the list obtained by the adversary may contain false information that has to be filtered out by statistical or analytical methods.
- The granularity of the measurements may not correspond to the above oracle calls. This depends on how time sharing on the processor is organised.
- The adversary may be unable to choose the IV, or to observe the keystream.

Thus, the model has to be considered as being generous towards the adversary. However, while doing one measurement only creates a noisy version of the cache access list, repeating the measurement and using statistical methods will often eliminate most of the noise.

In order to do this, the function calls have to be repeated under the same key and IV. While at the first glance, this seems to be IV re-use and thus a breach of the security contract, a second look shows that this is not the case at all. All the security contract disallows is re-using the IV for a *different* plaintext, i.e. IV re-use for the *same* plaintext is allowed. In particular, it is easy to imagine scenarios where the rightful user *decrypts* the same ciphertext several times (e.g. an entry in an encrypted database that is accessed repeatedly). Thus, in certain settings, obtaining the necessary measurements might actually be possible.

Note that in addition to analysing cipher resistance against cache timing attacks, the model can also be used to derive security margins for the standard model. If the best cache timing attack against a given cipher requires a large number of cache measurements, then the cipher may be considered as being more robust than one that can be broken by only a few calls to the side-channel oracles. Thus, analysing a cipher in our model achieves a similar effect as analysing modified (e.g. reduced-round) versions of a design: Even though an attack may not constitute a break in the standard model, it indicates how far we are from attacking the full cipher according to specification.

7 Conclusions

In this paper, we have described a cache-timing attack against the stream cipher HC-256, which is the strong version of eStream winner HC-128. The attack was based on a abstract model of cache timing attacks that can also be used for designing stream ciphers. From the observations made in our analysis, we have derived a number of design principles for hardening ciphers against cache timing attacks.

Acknowledgements

The author wishes to thank D.A. Osvik, D.J. Bernstein, G. Bertoni, C. de Cannière, L.R. Knudsen, and S. Lucks for discussions that helped improving this paper.

References

1. Bernstein, D.: Cache timing attacks on AES (2005), `http://cr.yp.to/papers.html#cachetiming`
2. Bernstein, D.: Leaks (February 2007), `http://cr.yp.to/streamciphers/leaks.html`
3. Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., Palermo, G.: AES power attack based on induced cache miss and countermeasure. In: International Symposium on Information Technology: Coding and Computing (ITCC 2005), vol. 1, pp. 586–591. IEEE Computer Society, Los Alamitos (2005)
4. Blömer, J., Krummel, V.: Analysis of countermeasures against access driven cache attacks on AES. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 96–109. Springer, Heidelberg (2007)
5. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006)
6. Brickell, E., Graunke, G., Neve, M., Seifert, S.: Software mitigations to hedge AES against cache-based software side-channel vulnerabilities (2006), `http://eprint.iacr.org/2006/052.pdf`
7. The eStream Portfolio, `http://www.ecrypt.eu.org/stream/portfolio.pdf`
8. 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)
9. Neve, M., Seifert, J., Wang, Z.: Cache time-behavior analysis on AES (2006), `http://www.cryptologie.be/document/Publications/AsiaCSS_full_06.pdf`
10. Neve, M., Seifert, J., Wang, Z.: A refined look at bernstein's AES side-channel analysis. In: Proc. AsiaCSS 2006, p. 369. ACM, New York (2006)
11. O'Hanlon, M., Tonge, A.: Investigation of cache-timing attacks on AES (2005), `http://www.computing.dcu.ie/research/papers/2005/0105.pdf`
12. Osvik, D., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES (2005), `http://eprint.iacr.org/2005/271.pdf`

13. 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)
14. Percival, C.: Cache missing for fun and profit. Paper accompanying a talk at BSDCan 2005 (2005), http://www.daemonology.net/papers/htt.pdf
15. Salembier, R.: Analysis of cache timing attacks against AES. Scholarly Paper, ECE Department, George Mason University, Virginia (May 2006), http://ece.gmu.edu/courses/ECE746/project/F06_Project_resources/Salembier_Cache_Timing_Attack.pdf
16. Wang, Z., Lee, R.: New cache designs for thwarting software cache-based side channel attacks. In: Proc. ISCA 2007, pp. 494–505. ACM, New York (2007)
17. Wu, H.: A new stream cipher HC-256. In: Roy, B., Meier, W. (eds.) FSE 2004, vol. 3017, pp. 226–244. Springer, Heidelberg (2004)