Cache Timing Analysis of HC-256

Erik Zenner

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

Abstract. In this paper, we describe an abstract model of cache timing attacks that can be used for designing ciphers. We then analyse HC-256 under this model, demonstrating a cache timing attack under certain strong assumptions. From the observations made in our analysis, we derive a number of design principles for hardening ciphers against cache timing attacks.

1 Introduction

Cache timing attacks have been introduced in 2005 by Bernstein [1] and Osvik et al. [7] as a new class of side-channel attacks. The basic idea is that an adversary can under certain circumstances observe the cache accesses of a legitimate party by measuring cache timings. As it turns out, the Advanced Encryption Algorithm (AES) is particularly vulnerable to this kind of attack. In subsequent work, the initial results were verified and refined, and countermeasures were discussed [3, 6, 2]. Bertoni et al. described how in addition to timing attacks, cache misses can also be used for power analysis attacks [4].

However, the focus of the cryptanalysis 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 if appropriate countermeasures on implementation level are not present. We demonstrate our approach by analysing a different type of cryptographic primitive, namely the stream cipher HC-256.

1.1 Cache Timing Attacks

Cache timing attacks are a special class of timing attacks. They make use of the fact that loading data into a CPU register is faster when loading from the cache than when reading from RAM. By measuring cache timings, an observer can obtain information about the inner workings of a cipher. In the following, we give a simplified description of cache timing attacks. For a more precise description, see e.g. [7, 6].

Cache workings: The CPU cache of modern processors is organised into blocks of s bytes (for Pentium 4, $s = 64$). In the same way, RAM is considered to be (virtually) organised in s-byte blocks. If a piece of data has to be loaded from memory 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 can be done very fast (cache hit). If not, it is first loaded from RAM to cache, which takes longer time (cache miss). Mapping from RAM to cache is typically by a simple modulo operation, i.e. if the cache has size n blocks and if the data lies in RAM block a, then it is loaded into cache block $a \bmod n$. This means that neighboring data in RAM (e.g. tables) also lies close together in cache.

A simple attack: As a (simplified) example, consider the **prime-then-probe** method presented in [7]. The adversary starts by filling all the cache with his own data. After that, he gives the read/write token to the legitimate user A. User A loads the data required for his own computations into the cache, where it evicts the adversary's data. When the adversary receives the read/write token again, he tries to load his own data from cache again. For each cache block, if this takes long, it means that A has evicted the corresponding data.

From this analysis, the adversary obtains a profile of cache blocks that have been used by A. 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 A. But he learns something about the addresses of the data that was used. In the case of AES, the adversary learns something about the indexes of the S-box entries used for encryption, which in turn can be used for an attack.

Relevance: Cache timing attacks work only if the adversary is able to make cache timing measurements of sufficient precision[1], and if he can repeat the experiment sufficiently often. Obviously, these requirements are only rarely met. However, they seem to be relevant in a shared server setting, and the paper by Bertoni et al. [4] shows how to use cache misses in an environment where the adversary has physical access to a device, making the attack much more realistic.

In general, the existence of a cache timing attack against a cipher does not mean that this cipher is broken in a practical sense and has to be discarded. Otherwise, we would have to start replacing AES. However, cryptographic prudence dictates that we have to analyse ciphers with regards to all potential attacks. Until new hardware makes cache timing attacks impossible, implementers have to make sure that their system prevents cache timing attacks. However, if a cipher has no cache timing vulnerability in the first place, this would obviously be preferrable. Thus, in this paper, we try to derive some design guidelines that help avoid cache timing vulnerabilities.

1.2 Selection of cipher

Cache timing attacks are applicable against ciphers that use tables for efficient implementations. A number of such ciphers can be found in the eStream project

[1] Typically, this would be by sharing the same computer, even though Bernstein [1] uses the example of a server that gives the timings away as part of a timestamp.

[5], which is concerned with the security of stream ciphers. Thus, we decided to target one of those ciphers for our analysis. Table 1 lists the finalists in the eStream software category and indicates which ones require large tables for efficient implementation.

Cipher	Tables	Select
CryptMT	none	-
Dragon	Two 8x32-bit S-Boxes	†
HC-128 HC-256	Two 9x32-bit tables Two 10x32-bit tables	†
LEX-128	One 8x8-bit S-Box (ref. code) Eight 8x32-bit S-Boxes (opt. code)	†
NLS	One 8x32-bit S-Box	†
Rabbit	none	-
Salsa-20	none	-
Sosemanuk	One 8x32-bit, eight 4x4-bit tables (ref. code) One 8x32-bit table (opt. code)	†

Table 1. Candidate ciphers for cache timing analysis

The ciphers that require further analysis are marked with a '†'. As can be seen, the designs CryptMT, Rabbit, and Salsa-20 are naturally resistant to cache timing attacks[2]. Out of the five remaining ciphers, we selected HC-256 for analysis in this paper, since it is one of the fastest remaining designs, and since the large inner state tables make analysis particularly challenging.

2 Attack Model

2.1 Usable Routines

While different methods of conducting a cache timing attack have been proposed in the literature, they are all based on the effect that calling an encryption routine has on the cache. In the case of stream ciphers, there are three routines that could potentially be targeted: key setup, IV setup, and keystream generation[3]. In this paper, we will analyse for each of those routines whether it can be used for an attack.

2.2 Side-channel Information

It is very hard to predict what kind of information the adversary will have available in a real-world cache-timing attack. Thus, we take the view of a cipher

[2] In the particular case of Salsa-20, this is a deliberate design goal.

[3] These correspond to the functions `ECRYPT_keysetup`, `ECRYPT_ivsetup`, and `ECRYPT_keystream_bytes` defined in the eStream API.

designer: What happens in the worst case? We assume that the adversary gets the maximum amount of information that can reasonably be expected, and see what damage this would do to the cipher. Ciphers designed under such a model will most likely be secure in practice, too. On the other hand, a break in the model does not necessarily imply a break in practice. The relevance of the attack depends on whether or not the strong assumptions of the model are present in a concrete real-world system or not.

The idealised model: HC-256 uses tables with an entry size of 4 byte. Considering that the most wide-spread CPUs (such as Pentium 4 and Athlon) currently use an L1 cache block size of 64 byte, each cache block contains 16 table entries. In the following, we assume that the tables are aligned with cache blocks[4].

For our analysis, we use the *prime and probe* method proposed in [7] and as described in section 1.1. In the following, we assume that the adversary gets the maximum possible information out of the attack, namely a list of all cache blocks that have been used by the cryptographic function observed. Note that this is a strong simplification in comparison to the real world, where the adversary only obtains a list of all cache blocks that have **not** been used.

The list of cache blocks used gives the adversary information about the table entries used. Since each cache block contains 16 table entries, the adverary obtains the table index with the exception of the least significant 4 bits. Note that if the cache block size is smaller (larger) than 64 byte, he will obtain more (less) information about the table entries.

In addition, we assume that for calls to the IV setup function, the adversary can choose the initialisation vector (IV). For calls to the keystream generation function, the adversary learns the resulting keystream bits. Again, these assumptions are rather strong, but often considered standard in cryptanalytic analysis.

3 Description of HC-256

HC-256 was proposed by Wu in [8][5]. The cipher is based on the use of 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. Table entries are identified by $P[i]$ and $Q[i]$.

When dealing with HC-256, $\oplus$ denotes xor, $||$ concatenation (most significant bits first), and $\ggg$ a circular right shift. $\boxplus$ denotes addition modulo 32, and $\boxminus$ subtraction modulo 10.

[4] Otherwise, analysis becomes messier, but also more efficient.

[5] A reduced version of the cipher, HC-128, was introduced in [9]. It is not considered here, even though its working principles are very similar to those of HC-256.

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:

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

Key/IV Setup: For initialisation (i.e., key and IV setup), the key is split into 8 32-bit words $K[0], \dots, K[7]$, and the IV is split into 8 32-bit words $IV[0], \dots, IV[7]$. With the help of an auxiliary array $W[0], \dots, W[2559]$ and a global counter variable r, the algorithm can be described as in Figure 1.

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

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

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

Note that during the attack, we assume that we can access one single iteration of **Next()** at a time. This is the usual way of implementing HC-256 in software: The user can ask for individual words to be encrypted.

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

Fig. 2. Keystream generation for HC-256

4 Observing One Function Call

4.1 Attacking the Init() Function

The key and IV setup for HC-256 can not be separated in a meaningful way. Thus, we consider them as one function **Init()**. Let us now assume that this **Init()** function has been executed and that the adversary has learned exactly which cache blocks have been accessed. As it turns out, a full run of the key/IV setup invokes each entry of the tables W, Q and T at least once. Thus, in the attack model used for this paper, the adversary does not obtain any useful information, since all he learns is that all table entries have been used - something he knew beforehand anyway[6].

4.2 Measuring One Call to the Next() Function

Let the adversary invoke the **Next()** function exactly once. Since both tables P and Q have 1024 entries, i.e. table indices are 10 bits long, and since the adversary learns only the correct cache block, he obtains the 6 most significant bits of the table index from observing the cache block access.

Now consider a case where $(r \bmod 2048) \in \{0, \dots, 1023\}$, i.e. code lines 3 and 4 are executed. The indices of all accesses to table P are fixed and known, i.e. the adversary does not learn anything new from them. However, in the calls to functions g_1 and h_1, table Q is accessed by state-dependent indices. Here, we observe 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

[6] In certain extreme cases, cache timing attacks may allow the adversary to count how often a given entry has been invoked. Measurements are a lot more difficult, though, and the amount of noise due to the numerous table accesses will be significant. Thus, we do not expect those attacks to be successful.

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 $P[j \boxminus 12]$ from the cache accesses, i.e. bits $7..4, 15..12, 23..20$, and $31..28$.

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 halfbyte. 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: Note that exactly the same observations hold for table P for rounds r with $(r \bmod 2048) \in \{1024, \ldots, 2047\}$.

5 Description of the Full Attack

5.1 Notation

Before considering several calls to the **Next()** function, we have to define a unique notation for the table entries. Note that 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 $r = 3084$.

In Table 2, we describe the relationship between rounds and the sequence words that are needed for the purpose of this paper.

5.2 First Observations

Now let the adversary observe calls to the function **Next()** for the following rounds:

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

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

Table 2. Rounds of sequence word update

This way, he observes partial information about table entries as described in Section 4.2. Using our new notation, we have 2 candidate assignments for each of the following lines:

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

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

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

Conclusions: 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).

If 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 $Q_{1024}, \ldots, Q_{3071}$.

5.3 Reducing the number of candidates

In the following, we will further reduce the number of candidates for $Q_{1024}, \ldots, Q_{3059}$ and $P_{2048}, \ldots, P_{3071}$.

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 5.2, 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:

$$z_r \oplus Q[j] = P[\gamma_0] \boxplus P[\gamma_1] \boxplus P[\gamma_2] \boxplus P[\gamma_3] \qquad (1)$$

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 halfbytes 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. We end up with 4 verification equations, as follows:

$$\begin{aligned}
z_r^{(7..4)} \oplus Q[j]^{(7..4)} &= P[\gamma_0]^{(7..4)} \boxplus P[\gamma_1]^{(7..4)} \boxplus P[\gamma_2]^{(7..4)} \boxplus P[\gamma_3]^{(7..4)} \boxplus c_0 \\
z_r^{(15..12)} \oplus Q[j]^{(15..12)} &= P[\gamma_0]^{(15..12)} \boxplus P[\gamma_1]^{(15..12)} \boxplus P[\gamma_2]^{(15..12)} \boxplus P[\gamma_3]^{(15..12)} \boxplus c_1 \\
z_r^{(23..20)} \oplus Q[j]^{(23..20)} &= P[\gamma_0]^{(23..20)} \boxplus P[\gamma_1]^{(23..20)} \boxplus P[\gamma_2]^{(23..20)} \boxplus P[\gamma_3]^{(23..20)} \boxplus c_2 \\
z_r^{(31..28)} \oplus Q[j]^{(31..28)} &= P[\gamma_0]^{(31..28)} \boxplus P[\gamma_1]^{(31..28)} \boxplus P[\gamma_2]^{(31..28)} \boxplus P[\gamma_3]^{(31..28)} \boxplus c_3
\end{aligned}$$

Here, $c_0, \dots, c_3$ 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 $Q_{1024}, \dots, Q_{2035}$, we write these guesses into a table.

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

Sequence words $P_{2048}, \dots, P_{3071}$*:* The same technique can also be applied to reduce the number of candidates for the sequence words $P_{2048}, \dots, P_{3071}$. We do this by considering the rounds $r = 4108, \dots, 5119$, which use sequence words $P_{2048}, \dots, P_{3071}$ as well as $Q_{1024}, \dots, Q_{2047}$. From this, we can reduce the number of candidates for $P_{2048}, \dots, P_{3071}$ to 2^8. Afterwards, we consider rounds $6144, \dots, 6155$, which use sequence words $P_{3060}, \dots, P_{3083}$ as well as some of the table entries $Q_{2048}, \dots, Q_{3071}$.

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

5.4 A Backtracking Attack

In the next step, we will reduce the number of candidates for $Q_{1024}, \dots, Q_{2047}$ and $P_{2048}, \dots, P_{3071}$ 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 contains the sequence 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 only $1/2^{32}$ of them satisfy the equation, only 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[7].

[7] 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.

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 $r = 5120, \dots, 6143$, we have reconstructed the correct solution for table entries $Q_{1024}, \dots, Q_{2047}$.

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, such as:

- Running 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 $P_{3072}, \dots, P_{4095}$ to 2^8, each.
- Using code line 4 for rounds $r = 5008, \dots, 5119$. 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.

6 Consequences

6.1 Breaking the Cipher

Now, we have retrieved the full contents of tables P and Q at a fixed moment (namely 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, or
- run the generator backwards to retrieve the key (the state update function and the key/IV setup are invertible).

Thus, the attack achieves the goal of breaking the security of HC-256.

6.2 Cost of the Attack

The main computational step is the backtracking attack, which requires less than $5 \cdot 2^{64} < 2^{67}$ computational steps that consist in verifying the contents of 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 Megabytes for the candidate tables, plus some small amount of 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.

We point out that our attack is not optimised in any way. It is likely that it can be conducted with less cache measurements and with significantly less computational effort.

Also note that 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 attacker 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 described in Subsection 5.3. 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 difficult.

6.3 Security of the Cipher

However, the attack does not break HC-256 in the standard model. We have used an attack model that would break AES-128 with only one cache timing measurement and 2^{64} trial encryptions. Thus, as long as we are not concerned for the security of AES, we do not have to be concerned for the security of HC-256.

Nonetheless, cache timing attacks are currently a real possibility in certain scenarios. However, in order to obtain the cache access statistics as used by this attack, several thousand or even million measurements are necessary. In the case of the above attack on HC-256, these measurements have to be done *under the same IV*[8]. Thus, the attack would require the attacked system to repeatedly encrypt under the same nonce, which in itself is a breach of the security contract (unless the same plaintext is encrypted several times).

Thus, HC-256 can probably be considered secure for practical purposes. What is more, the amount of work that is required to break HC-256 in an attacker-friendly model as the one used in this paper can be used as an indication for how hard it is to break HC-256 in a scenario where no side-channel information is available.

6.4 Design Recommendations

While trying to break HC-256 (and doing initial analysis of other eStream candidates), we met a number of obstacles that might be possible defense mechanisms against cache timing attacks. Notably, the following design recommendations may help preventing cache timing attacks:

[8] It is not obvious how measurements under different IVs can be combined into one attack.

1. Prevent the use of tables or S-boxes. If no table-based information is read from RAM, then cache timing attacks become impossible.
2. If you have to use tables, make as many table accesses for one function call as possible. Since the adversary can not see in which order table accesses are made, this will make it more difficult for him to match the observed indices to the inner state. For HC-256, this matching (Subsection 4.2) was relatively easy, which made the attack possible in the first place.
3. Alternatively, 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.
4. Make use of the fact that the least significant bits of the cache block index remain unknown (in our analysis, those were the 4 least significant bits). This can be achieved by good 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.
5. As opposed to S-boxes, the variable tables used in HC-256 give the adversary insecurity both about the input and the output of the tables. Again, this makes life harder for the adversary than in the case of statical tables.

7 Conclusions

In this paper, we have described an abstract model of cache timing attacks that can be used for designing ciphers. We demonstrated a cache timing attack against HC-256 under this model, using certain strong assumptions that will most likely prevent the attack from being feasible in practice. From the observations made in our analysis, we derived a number of design principles for hardening ciphers against cache timing attacks.

Open Issues: The following questions have not been answered by the paper and could be starting points for further research:

– Optimisation of the attack. We believe that the cost for the attack – both in terms of running time and of timing measurements – can be reduced by a more careful analysis.
– Implementing the attack. All claims made in this paper are theoretical in nature. No implementation has been made to verify the attack, neither in the theoretical model nor using real cache timings.
– Fault induction attack. If we give the adversary the possibility of interrupting a computation or introducing a fault (as might be the case when analysing, e.g., smart card), much more efficient attacks (in particular against the key and IV setup) might become possible[9].

[9] This idea is due to Stefan Lucks.

– Analysis of HC-128. The small version of HC-256 might be susceptible to a similar attack. However, careful analysis is necessary to verify or disproof this proposition.
– Analysis of other eStream candidates. Obviously, the remaining eStream candidates should be analysed under the same conditions. While LEX will display at least some of the weaknesses of AES, since it is based on that cipher, applicability of cache timing attacks for Dragon, NLS, and Sosemanuk is unclear.

Acknowledgements

The author wishes to thank Elke Piel and Ursula Zenner. Without their entirely unacademical support, this paper could not have been written.

References

1. D. Bernstein. Cache Timing Attacks on AES. http://cr.yp.to/papers.html#cachetiming. 2005.
2. J. Blömer and V. Krummel. Analysis of Countermeasures Against Access Driven Cache Attacks on AES. In C. Adams, A. Miri, and M. Wiener, editors, *Proc. SAC 2007*, volume 4876 of *LNCS*, pages 96–109. Springer, 2007.
3. J. Bonneau and I. Mironov. Cache-Collision Timing Attacks Against AES. In L. Goubin and M. Matsui, editors, *Proc. CHES 2006*, volume 4249 of *LNCS*, pages 201–215. Springer, 2006.
4. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005)*, volume 1, pages 586–591. IEEE Computer Society, 2005.
5. eStream. ECRYPT stream cipher project. http://www.ecrypt.eu.org/stream.
6. M. Neve, J. Seifert, Z. Wang. Cache Time-Behavior Analysis on AES. http://www.cryptologie.be/document/Publications/AsiaCSS_full_06.pdf. 2006.
7. D. Osvik, A. Shamir and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In D. Pointcheval, editor, *Proc. CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
8. H. Wu. A New Stream Cipher HC-256. In B. Roy and W. Meier, editors, *Proc. FSE 2004*, volume 3017 of *LNCS*, pages 226–244. Springer, 2004.
9. H. Wu. The Stream Cipher HC-128. http://www.ecrypt.eu.org/stream/hcp3.html. 2006.