Cache Timing Analysis of LFSR-based Stream Ciphers

Gregor Leander[1], Erik Zenner[1], and Philip Hawkes[2]

[1] Technical University of Denmark
Department of Mathematics
{g.leander,e.zenner}@mat.dtu.dk
[2] Qualcomm Incorporated
phawkes@qualcomm.com

Abstract. Cache timing attacks are a class of side-channel attacks that is applicable against certain software implementations. They have generated significant interest when demonstrated against the Advanced Encryption Standard (AES), but have more recently also been applied against other cryptographic primitives.
In this paper, we give a cache timing cryptanalysis of stream ciphers using word-based linear feedback shift registers (LFSRs), such as Snow, Sober, Turing, or Sosemanuk. Fast implementations of such ciphers use tables that can be the target for a cache timing attack. Assuming that a small number of noise-free cache timing measurements are possible, we describe a general framework showing how the LFSR state for any such cipher can be recovered using very little computational effort. For the ciphers mentioned above, we show how this knowledge can be turned into efficient cache-timing attacks against the full ciphers.

1 Introduction

Cache Timing Attacks [24, 19, 23] are a class of side-channel attacks. They assume that the adversary can use timing measurements to learn something about the cache accesses of a legitimate party, which turns out to be the case in some practical applications. In 2005, Bernstein [2] and Osvik, Shamir, and Tromer [17, 18] showed in independent work that the Advanced Encryption Standard (AES) is particularly vulnerable to this type of side-channel attack, generating a lot of attention for the field. Subsequent work dealt with verifying the findings [16, 15, 14, 22, 7], improving the attack [20, 3, 13, 5], and devising and analyzing countermeasures [6, 4, 25].

However, the cryptanalytic attention was mainly focussed on AES, while other ciphers were treated only handwavingly. For example, the eStream report on side-channel attacks [10] simply categorizes all stream ciphers that use tables in their implementations as vulnerable, independent of whether or not a cache timing attack was actually feasible. The cache timing analysis of the HC-256 stream cipher [26] presented at SAC 2008 was the first paper to actually analyze the cache timing resistance of a stream cipher. It also provided a model for the design and analysis of stream ciphers with regards to cache timing attacks.

In this paper, we discuss a different class of stream ciphers, namely those using tables to speed up software implementations of word-based linear feedback shift registers (LFSRs). The technique was introduced around the year 2000 and is used by ciphers such as Snow [8, 9], Sober-128 [11], Turing [21], or Sosemanuk [1].

1.1 Organisation of the Paper

The paper is organized as follows. In Section 2, we review the cache timing model that is used for the analysis and discuss its goal and practical relevance. Section 3 gives the general framework for the attack and describes its applicability to a wide range of stream cipher, including but not limited to the concrete examples being discussed in Section 4. Finally, in Section 5, we summarize our findings.

1.2 Notation

All ciphers discussed below are word-based, where one word consists of 32 bits. On a 32-bit value, we denote by $\oplus$ the bitwise addition in $\mathbb{F}_{2^{32}}$, by $\boxplus$ the addition modulo 2^{32}. The notations $\ll n$ and $\gg n$ define left and right shifts by n bits (modulo 2^{32}), and $\lll n$ the left rotation by n bits. For any vector $x = (x_0, \ldots, x_{n-1})$ in $\mathbb{F}_2^n$ and integers $0 \leq a < b < n$ we denote by $x^{(a,\ldots,b)}$ the vector $(x_a, \ldots, x_b)$.

2 Cache Timing Model

As with all side-channel attacks, cache timing attacks are not inherently attacks against the algorithm, but against its implementation[3]. Thus, there are basically two ways of analyzing the cache timing resistance of a cipher. One can either consider a concrete implementation of the cipher, or do a general analysis in the framework of a model that gives the adversary certain rights, which can be modeled as oracle accesses. In

[3] For readers not familiar with cache timing attacks, Appendix A gives a short introduction.

the latter case, a “break” within the model does not necessarily imply a break of all practical implementations, but it can indicate that extra care has to be taken when implementing the cipher.

The model that we want to use for our analysis is the one proposed in [26]. It models a synchronous cache attacker, i.e. an adversary who can only access (and thus perform measurements on) the cache after certain elementary operations by the legitimate users have finished. In particular, a synchronous cache adversary can do cache measurements before and after a full update of the stream cipher’s inner state, but not while the update is in progress.

Formally, the adversary uses two oracles:

– KEYSTREAM(i): He requests the cipher to return the i-th keystream block to him.
– SCA_KEYSTREAM(i): He obtains a noise-free list of all cache accesses made by KEYSTREAM(i), but does not learn anything about their order. These cache accesses give him information about the actual table entries as described in the paragraph on “Handling Cache Line Sizes” below.

Model vs. Real World: Note that the KEYSTREAM(i) oracle is considered “standard” for stream cipher analysis, i.e. it is also available to an adversary in a non-side-channel setting. The SCA_KEYSTREAM(i) oracle, on the other hand, gives the adversary more information than will typically be available in a real-world side-channel setting, since it assumes that his measurements are undisturbed by noise. Thus, the results obtained in this paper are for an idealized cache measurement setting. Whether attacks under this model also constitute attacks in the real world depends on the implementation details and has to be verified on a case-by-case basis.

In the real world, the adversary usually needs the ability to repeat his measurements several times in order to remove the noise from the list of cache accesses. This corresponds to running the stream cipher under the same key and initialization vector (IV) for a given number of times, each time measuring the cache lines accessed and intersecting the resulting tables. Note that while encrypting different plaintexts under the same IV should not be possible for most implementations, one can imagine applications where the adversary can ask the system to decrypt the *same* ciphertext several times under the same IV, thus forcing the stream cipher to execute an identical sequence of operations.

Also note that there may be “wrong” cache accesses that occur very frequently, i.e. that do not disappear when repeating the measurement and intersecting several cache access lists. These accesses may originate with external processes such as applications or the operating system. In some cases, it may be possible to identify these wrong accesses by repeatedly running the stream cipher for a number *different* IVs – the cache accesses that stay constant are the ones that are independent of the cipher and can thus be ignored.

In all cases, if the noise can not be eliminated completely, discarding the set of measurements is always an option. As will be described in Section 3, our attack technique already works if noise-free measurements are possible only once in a while.

Handling Cache Line Sizes: In theory, addresses of cache lines translate into information about the table indices used. However, in real-world cache timing attacks, a cache line can hold several table entries, i.e. a cache line represents several table indices. Thus, even if the adversary could do noise-free measurements, he would still not learn the *exact* table indices used, but only obtain b bits of *partial* information about the table index.

Consider the case of the popular Pentium 4 processors as an example. All LFSR lookup tables used by the ciphers in this paper contain 256 32-bit entries, i.e. each table entry fills 4 bytes. A cache line in a Pentium 4 processor is 64 bytes broad, meaning that it can contain 16 table entries. Thus, given the correct cache line, the adversary learns only a subset of 16 table entries which contains the right one. Since these table entries are typically aligned[4], this corresponds to learning the $b = 4$ most significant bits of the table index, while the adversary obtains no information whatsoever about the 4 least significant ones.

More generally, if a table contains 2^c entries of d bytes each, and if the processor has a cache line size of λ bytes, then each cache line will hold λ/d table entries. Thus, even in a completely noise-free scenario, the attacker can not reconstruct more than the uppermost $b = c - \log_2(\lambda/d)$ bits of the table index (i.e. c bits in the best and 0 bits in the worst case).

3 General Framework of the Attack

In this section we present the general framework of our cache timing attack. All ciphers that make use of a LFSR for which clocking the LFSR involves table lookups are covered by this framework. This includes the

[4] If they aren't, the attack gets easier, since certain lines leak more information than assumed here.

stream ciphers Sosemanuk, Snow, Sober and Turing, which we will analyze separately below.

3.1 Basic Idea

Observing inner state bits: Given a stream cipher with an LFSR of length n defined over $\mathbb{F}_{2^m}$ we denote by

$$s = (s_0, \ldots, s_{n-1}) \in \mathbb{F}_{2^m}^n$$

the initial state of the LFSR after initialization and by

$$(s_t, \ldots, s_{t+n-1}) \in \mathbb{F}_{2^m}^n$$

its internal state at time t. Our model assumes that for clocking the LFSR the implementation makes use of table lookups. This is the case for almost all stream ciphers using an LFSR defined over $\mathbb{F}_{2^m}$, the reason being that this usually is the fastest manner to implement multiplication by one fixed element in $\mathbb{F}_{2^m}$. At time t this table lookup uses some bits of one of the elements s_{t+i} for $0 \leq i < n$ (in the case of Sosemanuk, Snow, Sober and Turing it involves 8 bits). Depending on the cache line size (cf. the discussion above) the cache timing measurements will reveal b of those bits. The trivial, but important, observation is that all those bits can be expressed as linear combinations of bits of the initial state s. Moreover, computing the actual linear combination can easily be done as follows.

Transforming into initial state bits: Elements in $\mathbb{F}_{2^m}$ can be identified with m-bit words (i.e. elements of $\mathbb{F}_2^m$) via a vectorspace isomorphism

$$\psi : \mathbb{F}_{2^m} \to \mathbb{F}_2^m.$$

Using the isomorphism ψ we can consider the state of the LFSR as an element in $\mathbb{F}_2^{nm}$ instead of an element of $\mathbb{F}_{2^m}^n$ via

$$(s_t, \ldots, s_{t+n-1}) \to (\psi(s_t), \ldots, \psi(s_{t+n-1})).$$

Then, clocking the LFSR can be described by applying an invertible $nm \times nm$ matrix over $\mathbb{F}_2$ to the current state. This is true simply because updating the LFSR is certainly a $\mathbb{F}_{2^m}$-linear operation and therefore in particular $\mathbb{F}_2$-linear. We denote the matrix that corresponds to updating the LFSR by M. Moreover, the matrix M can be easily computed given the feedback polynomial of the LFSR.

Each table lookup reveals some bits of s_{t+i}, i.e. $\psi(s_{t+i})^{(a..a+b-1)}$ for some $a \in \{0, \ldots, m-b-1\}$. Writing $\psi(s_{t+i})$ as $M^t\psi(s)$ we see that

$$\psi(s_{t+i})^{(a..a+b-1)} = [M^t\psi(s)]^{(a'..a'+b-1)}$$

for some $a' \in \{0, \ldots, nm-b-1\}$ and thus linear in the bits of the initial state as claimed. In this way, each bit observed in a cache timing measurement yields a linear equation in the initial state bits, and once sufficiently many equations have been collected, the initial state can be retrieved by solving the equation system.

3.2 Number of Required Noise-Free Measurements

Practical problems: It remains to discuss the number of measurements that are required to obtain a solvable equation systems. We start by pointing out that in practice, we can not expect to obtain all cache measurements that seem possible in theory. The following problems can occur, but as it turns out, they can be overcome using the linearity of the above equations:

1. If the cipher clocks the LFSR several (say c) times for each call to KEYSTREAM(i), or if the table is accessed several times for each clock, then a synchronous adversary can not measure each single table access separately. Instead, he learns c indices at a time (see e.g. Sosemanuk, where $c = 4$). This implies that the attacker gets to know the values

$$\psi(s_{t+i})^{(a..a+b-1)}, \ldots \psi(s_{t+i+c-1})^{(a..a+b-1)}$$

   but not the order of those.
   This problem can be dealt with by forming a linear equation using the sum of all the observed values. This way, the adversary obtains only 1 observation (instead of c) for each round, but otherwise, there is no effect on the overall effort of the attack.
2. Also in the case where a table is accessed several times for each call to KEYSTREAM(i), there is the problem of collisions. If two or more table accesses use the same cache line, then above trick no longer works. For example, if he measures accesses to cache lines L_1, L_2, L_3 for Sosemanuk (4 accesses per call), then he does not know which cache line information he has to use twice. Guessing is usually not a good strategy, since it rapidly increases the overall work effort for the attack. Instead, if such collisions are not too frequent, simply discarding the measurements and trying again next round gives much better results.

3. As described in Section 2, the number of bits available for analysis depends on the cache line size. In particular, for processors with large cache lines, the information available decreases. Nonetheless, as long as at least one bit of information leaks, the attack still works as described above.

Number of Noise-Free Measurements: Assume we are attacking an LFSR with an internal state of n elements of $\mathbb{F}_{2^m}$ and the cache measurement reveals only a fixed linear combination of internal state bits at each k-th iteration. In this case, nmk iterations suffice to completely reveal the initial state of the LFSR. The easiest way to see that is to interpret the initial state s of the LFSR as an element of $\mathbb{F}_{2^{nm}}$ (see for example [12, Chapter 8] for details). Clocking the LFSR corresponds to multiplying the internal state by a fixed element $\alpha \in \mathbb{F}_{2^{nm}}$ and the information leaking at time $T = tk$ can be written as $u_T = \mathrm{Tr}(\theta(\alpha^k)^t s)$ for an fixed element $\theta \in \mathbb{F}_{2^{nm}}$ depending on the linear combination of bits that are leaked. Here Tr denotes the trace function

$$\begin{aligned} \mathrm{Tr} : \mathbb{F}_{2^{nm}} &\rightarrow \mathbb{F}_2 \\ \mathrm{Tr}(x) &= \sum_{i=0}^{nm-1} x^{2^i}. \end{aligned}$$

The usual requirement that the LFSR should have maximal period corresponds to α being a primitive element in $\mathbb{F}_{2^{nm}}$. Thus, for reasonably small k, the element α^k is not in a proper subfield of $\mathbb{F}_{2^{nm}}$ and

$$(\alpha^k)^0, (\alpha^k)^1, \ldots, (\alpha^k)^{nm-1}$$

form an $\mathbb{F}_2$ basis of $\mathbb{F}_{2^{nm}}$. Therefore, all the linear equations $u_T = \mathrm{Tr}(\theta(\alpha^k)^t s)$ for $T = tk$, $t \leq nm - 1$ are linearly independent and the initial state can be uniquely recovered by solving the corresponding system of linear equations.

Note that this result only holds if all known keystream bits are exactly equidistant. For the ciphers discussed in this paper, this effect can be achieved by measuring only 1 bit for each clocking of the LFSR (where in principle, we could measure b or even $2b$ bits). If, however, some measurements have to be discarded due to collisions or noise, then the result can not be applied. Nonetheless, it is still very likely that we need only a very small overhead of noise-free measurements to get a uniquely solvable system. Namely, under the assumption that it behaves like a random system of linear equations, the probability that after $nm + \delta$ noise-free measurements the resulting system has full rank is given by

$$p = \prod_{j=0}^{nm-1} \frac{2^{nm+\delta} - 2^j}{2^{nm+\delta}} \approx 1 - 2^{-\delta}.$$

For example, when using the Sosemanuk parameters $n = 10$ and $m = 32$, the equation system will have rank nm with probability > 0.969 after only $\delta = 5$ additional noise-free measurements.

4 Analyzing Specific Ciphers

In this section we apply our general framework to the ciphers Sosemanuk, Snow, Sober and Turing. All ciphers are described briefly, giving only the details required to understand the attack.

For all those ciphers it turns out that once the internal state of the LFSR is recovered, the remaining bits of the internal state (if any) can be determined very efficiently. We explain the attack on Sosemanuk in detail; the other ciphers are discussed much shorter as the general approach is always the same.

In order to simplify notation in this section, we do not explicitly write the isomorphism operator ψ but assume that the reader is aware of whether a given vector is in $\mathbb{F}_{2^m}$ or in $\mathbb{F}_2^m$.

4.1 Analysis of Sosemanuk

The Sosemanuk cipher was proposed by Berbain et al. in 2005 as an eStream candidate [1]. Sosemanuk consists of a 10-word LFSR over $\mathbb{F}_{2^{32}}$, a finite-state machine (FSM) with two 32-bit words, and an output function combining LFSR and FSM output into the keystream. No valid attacks against the cipher have been proposed so far.

LFSR: The linear recursion of the LFSR is defined by

$$s_{t+10} = s_{t+9} \oplus \alpha^{-1} s_{t+3} \oplus \alpha s_t,$$

where α is a fixed element in $\mathbb{F}_{2^{32}}$. An optimized implementation of the multiplications by α and α^{-1} uses 8x32-bit lookup tables T_1 and T_2. Ignoring the isomorphism operator ψ, the multiplications can be implemented as follows:

$$x \cdot \alpha = \left((x \ll 8) \oplus T_1[x^{(24..31)}]\right)$$
$$x \cdot \alpha^{-1} = \left((x \gg 8) \oplus T_2[x^{(0..7)}]\right).$$

FSM: In addition, we need a simplified description of the FSM. If we denote the two 32-bit state words by $R1$ and $R2$ and the inner state words of the LFSR by s_t as above, we can describe the production of an intermediate value f_t as follows:

$$\begin{aligned}R1_t &= \mathit{Update1}(R1_{t-1}, R2_{t-1}, s_{t+1}, s_{t+8})\\ R2_t &= \mathit{Update2}(R1_{t-1})\\ f_t &= (s_{t+9} \boxplus R1_t) \oplus R2_t\end{aligned}$$

We don't need the internals of the functions *Update1* and *Update2* for the analysis.

Output: Before producing output, Sosemanuk will clock the LFSR 4 times and generate four intermediate values. Then the keystream output is generated as

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = \mathit{Serpent1}(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t),$$

where we don't have to know more about the *Serpent1* function than that it is a permutation that is easily invertible.

Cache Timing Attack

Cache Measurements: Assume for simplicity that $b = 8$, i.e. that the adversary observes all 8 bits of the table indices for T_1 and T_2 after any output block of his choice. This means that he obtains 4 measurements of accesses to T_1 and T_2 each. In the case of T_1, he knows that they correspond to inner state bytes

$$s_t^{(24..31)}, s_{t+1}^{(24..31)}, s_{t+2}^{(24..31)}, s_{t+3}^{(24..31)},$$

but he does not know the correct order. The same also holds for the accesses to T_2, which give information about

$$s_t^{(0..7)}, s_{t+1}^{(0..7)}, s_{t+2}^{(0..7)}, s_{t+3}^{(0..7)}$$

without revealing the proper ordering.

Reconstructing the LFSR State: The attacker knows at each time $T = 4t$ the values

$$u_T = s_t^{(24..31)} \oplus s_{t+1}^{(24..31)} \oplus s_{t+2}^{(24..31)} \oplus s_{t+3}^{(24..31)} \qquad (1)$$

and

$$v_T = s_t^{(0..7)} \oplus s_{t+1}^{(0..7)} \oplus s_{t+2}^{(0..7)} \oplus s_{t+3}^{(0..7)}, \tag{2}$$

as those sums do not depend on the ordering anymore. As both Eq. 1 and 2 yield $b = 8$ linear equations over $\mathbb{F}_2$, we get a total of 16 linear equations for the initial state s at each time T. It turns out that after 20 LFSR clockings the resulting 320 equations already have full rank and therefore the initial state can easily be computed given $u_T, \dots, u_{T+19}$ and $v_T, \dots, v_{T+19}$. Note that if $b < 8$, then the number of LFSR clockings whose timings have to be observed is $320/2b = 160/b$.

Reconstructing the FSM State: Given the correct inner state of the LFSR, the inner state of the FSM is easily reconstructed. To this end, we proceed as follows:

1. Given output words $z_t, \dots, z_{t+3}$, the adversary can subtract $s_t, \dots, s_{t+3}$ and obtains the output of the *Serpent1* S-box.
2. The S-box is invertible, yielding the values $f_t, \dots, f_{t+3}$.
3. The adversary guesses the state of $R1_t$ (32 bits), which allows him to compute the state of $R2_t$ from the equation $f_t = (s_{t+9} \boxplus R1_t) \oplus R2_t$.
4. The adversary updates the state of the FSM once to obtain $R1_{t+1}$ and $R2_{t+1}$. He checks whether the output matches the observed f_{t+1}.

Normally, after this step, only the correct guess for $R1_t$ should have survived. If more guesses survive, one simply continues updating the inner state and checking against the output, until the state $(R1, R2)$ is uniquely determined. In total, this step requires not more than 2^{32} simple guess-and-determine steps. Note that algorithmically more elegant ways of reconstructing the FSM state might exist, but since 2^{32} guess-and-determine steps are easily computable on e.g. a standard PC, we did not search for such attacks.

Resources: The overall effort for the attack is dominated by reconstructing the FSM state and therefore has an overall complexity of 2^{32} guess-and-determine steps, and the memory consumption is dominated by the space for storing the 320×320-bit equation system. Thus, assuming the availability of noise-free timing observations for $\approx 160/b$ LFSR clockings, we have an efficient cache timing attack against Sosemanuk.

4.2 Analysis of Snow 2.0

Snow 2.0 was proposed by Ekdahl and Johansson in 2002 [9]. It uses an LFSR over $\mathbb{F}_{2^{32}}$ of length 16, an FSM with two 32-bit words, and an output function combining LFSR and FSM output into the keystream. No valid attacks against the cipher have been proposed so far.

LFSR: The linear recursion of the LFSR is defined by

$$s_{t+16} = \alpha^{-1} s_{t+11} \oplus s_{t+2} \oplus \alpha s_t,$$

where α is a fixed element in $\mathbb{F}_{2^{32}}$. Just as for Sosemanuk, two 8x32-tables T_1 and T_2 are involved for the multiplication by α and α^{-1}. Each of them is called exactly once for each state update.

FSM: Snow uses a FSM consisting of two 32-bit words $R1_t$ and $R2_t$. The exact specification is not important for our attack. It suffices to know that given the internal state, the output stream and one of the two words of the FSM the remaining word is uniquely determined.

Cache Timing Attack

Cache Measurements: Assuming precise measurements, the adversary obtains the uppermost b bits of the table indices for T_1 and T_2. In each round, he obtains one measurement for T_1 and T_2 each. In the case of T_1, he knows that they match to inner state bytes $s_t^{(31-b+1..31)}$ and in the case of T_2 the attacker learns $s_t^{(7-b+1..7)}$. As both observations yield b linear equations over $\mathbb{F}_2$, we get a total of $2b$ linear equations for the initial state s for each LFSR clocking. Thus, after approximately $512/2b = 256/b$ LFSR clockings, the equation system can be solved.

Reconstructing the FSM State: As mentioned above, the inner state of the FSM consists of two words. As for Sosemanuk, by guessing one word (e.g $R1_t$), the adversary can derive the other. He can then update the inner state and verify whether his guess was correct. Again, the expected workload is not more than 2^{32} simple guess-and-determine steps.

4.3 Analysis of Sober-128

Sober-128 was proposed by Hawkes and Rose in 2003 [11]. It makes use of an LFSR over $\mathbb{F}_{2^{32}}$ of length 17, a key dependent 32-bit constant K, and a nonlinear output function combining the LFSR and the constant K into the keystream.

	Size of	Guessing	# Cache Measurements		Known
	eq. system	Steps	General	Pentium 4	Keystream
Sosemanuk	320	2^{32}	$160/b$ clks	40 clks	16 bytes
Snow 2.0	512	2^{32}	$256/b$ clks	64 clks	8 bytes
Sober-128	544	-	$544/b$ clks	136 clks	4 bytes
Turing	544	-	$544/b$ clks	136 clks	-

Table 1. Attack parameters against various ciphers

LFSR: The linear recursion of the LFSR is defined by

$$s_{t+17} = s_{t+15} \oplus s_{t+4} \oplus \alpha s_t,$$

where α is a fixed element in $\mathbb{F}_{2^{32}}$. Just as for Sosemanuk and Snow 2.0, multiplication by α is done using a lookup table.

Cache Timing Attack Assuming precise measurements, the adversary observes b bits for each round, i.e. $s_t^{(7-b+1..7)}$. Thus, after $\approx 544/b$ rounds, we expect to be able to reconstruct the LFSR state. Given this state, the constant K can trivially be computed given one keystream word.

4.4 Analysis of Turing

Turing was introduced by Rose and Hawkes in 2002 [21]. It is based on the same LFSR as Sober-128 and uses a fixed non-linear filter function on the internal state of the LFSR to generate the keystream. In particular the cache timing part is exactly the same as for Sober-128. As the internal state of Turing consists only of the LFSR state, no additional bits have to be recovered and in particular no keystream bits are needed to perform the attack.

5 Conclusions

We have shown how to mount cache timing attacks against all word-based LFSR implementations that use lookup tables to speed up multiplications. As described above such ciphers are especially vulnerable to cache timing attacks. All of them we are aware of can be broken very efficiently within our theoretical model.

What is more, our attack is tolerant with respect to noisy measurements: the information delivered by the cache timings may be few (1 bit once in a while is enough) and far between (the distances between the bits can be arbitrary). This is due to the fact that a noise-free measurement always reveals a linear equation for the internal state. Thus, as long as we can detect errors, we can simply discard noisy measurements without significantly increasing the complexity of the equation system to be solved.

Clearly, implementing cache timing attacks on real life systems is a difficult and cumbersome task that requires dedicated skills. However, due to the reasons outlined above, we anticipate that it is significantly easier to implement our attack on the above mentioned stream ciphers than most other cache timing attacks.

Countermeasures: A possible countermeasure is to split the lookup table into several, smaller tables such that each table fits into one cache line. Here is a short example using C notation. Suppose `LinTab[256]` is a lookup table with 32-bit entries and our processor has cache lines of 64 bytes, so each cache line can contain at most 16 table entries. The linearity of the table can be exploited to compute `LinTab[x]` using two smaller tables `LinTabUpper[16]` and `LinTabLower[16]` with entries defined as:

```
for(y=0; y<16; y++) LinTabUpper[ y ] = LinTab[ y << 4 ];
for(y=0; y<16; y++) LinTabLower[ y ] = LinTab[ y ];
```

The linearity of `LinTab[]` means that we can generate the value of `LinTab[x]` at compute time using five operations:

```
LinTabUpper[ x >> 4 ] ^ LinTabLower[ x & 0xF ];
```

This way, no information about the cache entries can be obtained by timing measurements because `LinTabUpper[]` and `LinTabLower[]` each fit within one cache line. It should be noted that there is a performance penalty since the one table-lookup operation is replaced by five operations, and this performance penalty might be prohibitively large for the speed-sensitive stream ciphers.

References

1. C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. SOSEMANUK, a fast software-oriented stream cipher. eStream submission, http://www.ecrypt.eu.org/stream/sosemanuk.html, 2005.
2. D. Bernstein. Cache timing attacks on AES. http://cr.yp.to/papers.html#cachetiming, 2005.

3. 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.
4. 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.
5. 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.
6. E. Brickell, G. Graunke, M. Neve, and S. Seifert. Software mitigations to hedge AES against cache-based software side-channel vulnerabilities. http://eprint.iacr.org/2006/052.pdf, 2006.
7. A. Canteaut, C. Lauradoux, and A. Seznec. Understanding cache attacks. Technical Report 5881, INRIA, 2006.
8. P. Ekdahl and T. Johansson. SNOW - a new stream cipher. `http://www.it.lth.se/cryptology/snow/`. NESSIE project submission.
9. P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In H. Heys and K. Nyberg, editors, *Proc. SAC 2002*, volume 2595 of *LNCS*, pages 47–61. Springer, 2002.
10. B. Gierlichs, L. Batina, C. Clavier, T. Eisenbarth, A. Gouget, H. Handschuh, T. Kasper, K. Lemke-Rust, S. Mangard, A. Moradi, and E. Oswald. Susceptibility of eSTREAM candidates towards side channel analysis. In C. de Cannière and O. Dunkelmann, editors, *SASC '08 Workshop Record*, pages 123–150, 2008.
11. P. Hawkes and G. Rose. Primitive specification for Sober-128. `http://www.qualcomm.com.au/Sober128.html`.
12. R. Lidl and H. Niederreiter. *Finite Fields.* Cambridge University Press, 1997.
13. M. Neve and J. Seifert. Advances on access-driven cache attacks on AES. In E. Biham and A. Youssef, editors, *Proc. SAC 2006*, volume 4356 of *LNCS*, pages 147–162. Springer, 2006.
14. M. Neve, J. Seifert, and Z. Wang. Cache time-behavior analysis on AES. http://www.cryptologie.be/document/Publications/AsiaCSS_full_06.pdf, 2006.
15. M. Neve, J. Seifert, and Z. Wang. A refined look at bernstein's AES side-channel analysis. In *Proc. AsiaCSS 2006*, page 369. ACM, 2006.
16. M. O'Hanlon and A. Tonge. Investigation of cache-timing attacks on AES. http://www.computing.dcu.ie/research/papers/2005/0105.pdf, 2005.
17. D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. http://eprint.iacr.org/2005/271.pdf, 2005.
18. 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.
19. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, University of Bristol, June 2002. http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625.
20. C. Percival. Cache missing for fun and profit. Paper accompanying a talk at BSDCan 2005; available at http://www.daemonology.net/papers/htt.pdf, 2005.
21. G. Rose and P. Hawkes. Turing: A fast stream cipher. In T. Johansson, editor, *Proc. Fast Software Encryption 2003*, volume 2887 of *LNCS*, pages 290–306. Springer, 2003.

22. R. Salembier. Analysis of cache timing attacks against AES. Scholarly Paper, ECE Department, George Mason University, Virginia; available from: http://ece.gmu.edu/courses/ECE746/project/F06_Project_resources/ Salembier_Cache_Timing_Attack.pdf, May 2006.
23. Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miayuchi. Cryptanalysis of DES implemented on computers with cache. In C. Walter, Ç. Koç, and C. Paar, editors, *Proc. CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, 2003.
24. Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miayuchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proc. ISITA 2002*, 2002.
25. Z. Wang and R. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. ISCA 2007*, pages 494–505. ACM, June 2007.
26. E. Zenner. A cache timing analysis of HC-256. In R. Avanzi, L. Keliher, and F. Sica, editors, *Proc. SAC '08*, volume 5381 of *LNCS*, pages 199–213. Springer, 2009.

A Cache Timing Attack Basics

In the following, we give a short introduction to cache timing attacks. For more detailed information, the reader is referred to the introductory papers by Bernstein [2] and Osvik, Shamir, and Tromer [17, 18].

Cache motivation: Modern processors store data in different types of storage. Data that is currently being processed is stored in the so-called registers; however, only few of these are available. Instead, reasonably large amounts of data (such as look-up tables or S-boxes) are stored in RAM. Since access to RAM is relatively slow compared to executing an arithmetic operation, frequently used data is also stored in an intermediate type of memory, the so-called cache.

Cache workings: The CPU cache of modern processors is organised into blocks – so-called *lines* – of λ bytes. Correspondingly, RAM is considered to be (virtually) divided into λ-byte lines. When loading data from RAM into a CPU register, the system first checks whether the corresponding RAM line 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 lines and if the data lies in RAM line 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 [18]. 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 line, 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 lines 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 an LFSR, this corresponds to the indices of the LFSR cells that were accessed.