Consecutive S-box Lookups: A Timing Attack on SNOW 3G*

Billy Bob Brumley[1], Risto M. Hakala[1], Kaisa Nyberg[1,2], and Sampo Sovio[2]

[1] Aalto University School of Science and Technology, Finland
{billy.brumley,risto.m.hakala,kaisa.nyberg}@tkk.fi
[2] Nokia Research Center, Finland
{kaisa.nyberg,sampo.sovio}@nokia.com

Abstract. We present a cache-timing attack on the SNOW 3G stream cipher. The attack has extremely low complexity and we show it is capable of recovering the full cipher state from empirical timing data in a matter of seconds, requiring no known keystream and only observation of a small number of cipher clocks. The attack exploits the cipher using the output from an S-box as input to another S-box: we show that the corresponding cache-timing data almost uniquely determines said S-box input. We mention other ciphers with similar structure where this attack applies, such as the K2 cipher currently under standardization consideration by ISO. Our results yield new insights into the secure design and implementation of ciphers with respect to side-channels. We also give results of a bit-slice implementation as a countermeasure.

Keywords: side-channel attacks, cache-timing attacks, stream ciphers.

1 Introduction

Cache-timing attacks are a type of software side-channel attack relying on the fact that the latency of retrieving data from memory is essentially governed by the availability of said data in the processor's cache. Attackers capable of measuring the overall latency of an operation (time-driven attacks) or a more granular series of cache hits and cache misses (trace-driven attacks) use this information to determine portions of the cryptosystem state. Implementations making use of memory-resident table lookups are particularly vulnerable. For example, it is not uncommon for a block or stream cipher to implement an S-box by using some portion of the state as an index into a lookup table: this potentially leaks a portion of the index and hence state to an attacker.

Zenner established a model for cache-timing attacks against stream ciphers [14]. It facilitates the theoretical analysis of stream cipher cache-timing properties. This analysis helps identify potential cache-timing vulnerabilities in ciphers. Using this model, Leander, Zenner, and Hawkes gave an attack framework for

* Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

M. Soriano, S. Qing, and J. López (Eds.): ICICS 2010, LNCS 6476, pp. 171–185, 2010.

a number of LFSR-based ciphers [10]. This included a theoretical attack on SOSEMANUK, SNOW 2.0, SOBER-128, and Turing. The attack framework only considers side-channel data related to the LFSR and is mainly concerned with LFSR state recovery. As such, it leaves any associated FSM state recovery as the most computationally complex part of the attack, and furthermore requires synchronous keystream to do so.

The global standardization body 3GPP specifies two standard sets of encryption and integrity algorithms for use in the 3rd generation mobile communication system. The first set, UEA1 and UIA1, is based on the KASUMI block cipher. The second set, UEA2 and UIA2, consists of algorithms built around the SNOW 3G stream cipher [6] as the main cryptographic primitive. The algorithms of the second set have also been adapted for use in the emerging Long Term Evolution (LTE) system under the acronyms EEA1 and EIA1.

In mobile devices, SNOW 3G is implemented in hardware. Hence, discussing timing attacks in 3G environment is meaningful only for the cipher implementation in the network element. In addition to side-channel data, the previous attack [10] requires also certain amount of known keystream in synchronization with the side-channel data. In practice, realizations of these two data sources may be essentially different and their synchronization poses an additional obstacle. The purpose of this paper is to show that SNOW 3G has an internal structure that can be exploited to recover the internal state based solely on the side-channel data from the timings, and hence allows us to remove the need for synchronous keystream. While the most efficient version of the new attack presented in this paper uses information from all cache timings, within both LFSR and FSM, the attack also works, with a small penalty in performance, if only the timing data from the FSM lookups is used. That is, even if countermeasures to the attack of [10] are implemented, our attack still succeeds.

The main observation is that timings from two consecutive table lookups determine the inputs almost uniquely. In addition to SNOW 3G such a structure of the state update function is used in K2 stream cipher [9] currently under standardization consideration by ISO. We show practical attacks obtained by analyzing the effect of consecutive table lookups and related feedback equations. This result gives new insight into the secure design and implementation of ciphers with respect to side-channels.

Lastly, as a countermeasure to such timing attacks we present a bit-slice implementation of SNOW 3G components. Compared to table-based implementations, for batch keystream generation our results indicate bit-slicing offers timing attack resistance without penalizing performance.

2 Attack Model

Zenner [14] proposed a model for theoretically analyzing the cache-timing properties of stream ciphers. An attacker makes use of the following two synchronous oracles:

– KEYSTREAM(i) returns the ith keystream block.

– SCA_KEYSTREAM(i) returns an error-free unordered list of cache accesses made during the creation of the ith keystream block.

See [10] for a discussion on the features of this model from both the theoretical and practical perspectives. While our attack on SNOW 3G keystream generator conforms to the above model, it relaxes it significantly by removing the assumption of the KEYSTREAM oracle; that is, we assume *no* known keystream.

Assumptions. We restrict to the common case of 64-byte cache lines. Furthermore, we restrict to the case where tables do not overlap with respect to cache sets—that is, cache accesses can be mapped to distinct tables. These assumptions fit well within the above model. From a practical perspective they depend on the underlying cache size and structure. They will not always hold, but do hold in our environment described later in Sect. 5.3.

3 SNOW 3G

SNOW 3G [6] is a stream cipher used to preserve confidentiality and integrity of communication in 3GPP networks. It was developed during the ETSI SAGE evaluation by modifying the design of SNOW 2.0 [5] to increase its resistance against algebraic attacks. In the following sections, we give a short description of the keystream generator of SNOW 3G and note a few implementation details that are relevant to our analysis.

A finite field with q elements is denoted by $\mathbb{F}_q$. The elements in $\mathbb{F}_{2^m}$ are also identified with the integers in $\mathbb{Z}_{2^m}$. We use $\oplus$ to denote the bitwise XOR and $\boxplus$ to denote the addition in $\mathbb{Z}_{2^m}$. Given an integer $x \in \mathbb{Z}_{2^m}$, we use $x \ll a$ and $x \gg a$ to denote the left and right shifts of x by a bits, respectively.

3.1 Description of SNOW 3G

The SNOW 3G keystream generator uses a combination of a linear feedback shift register (LFSR) and a finite state machine (FSM) to produce the output keystream. This structure is depicted in Fig. 1. The state of the LFSR at time $t \geq 0$ consists of sixteen 32-bit values $s_{t+i} \in \mathbb{F}_{2^{32}}$, $i = 0, \ldots, 15$, and it is updated according to the relation

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

where $\alpha \in \mathbb{F}_{2^{32}}$ is a 32-bit constant, and the field arithmetic is as specified for SNOW 3G in [6]. The LFSR feeds s_{t+5} and s_{t+15} into the FSM, which has three registers, $R1$, $R2$, and $R3$. The contents of these registers at time $t \geq 0$ are denoted by $R1_t$, $R2_t$, and $R3_t$, respectively. The output F_t of the FSM is calculated as

$$F_t = (s_{t+15} \boxplus R1_t) \oplus R2_t \tag{2}$$

for all $t \geq 0$. The output z_t of the keystream generator is given as

$$z_t = F_t \oplus s_t. \tag{3}$$

The registers in the FSM are updated according to

$$R1_{t+1} = R2_t \boxplus (s_{t+5} \oplus R3_t), \tag{4}$$
$$R2_{t+1} = S1(R1_t), \quad \text{and} \tag{5}$$
$$R3_{t+1} = S2(R2_t), \tag{6}$$

where $S1$ and $S2$ are permutations of $\mathbb{F}_{2^{32}}$. The $S1$ permutation is composed of four parallel AES S-boxes followed by the AES MixColumn transformation. The second permutation, $S2$, is otherwise identical to $S1$ but the AES S-box is replaced by a bijective mapping derived from a Dickson polynomial. In SNOW 2.0, the FSM contains only two 32-bit registers, $R1$ and $R2$. These registers are updated as $R1_{t+1} = R2_t \boxplus s_{t+5}$ and $R2_{t+1} = S1(R1_t)$. The output F_t is calculated as in SNOW 3G. For a complete description of SNOW 3G, we refer to [6].

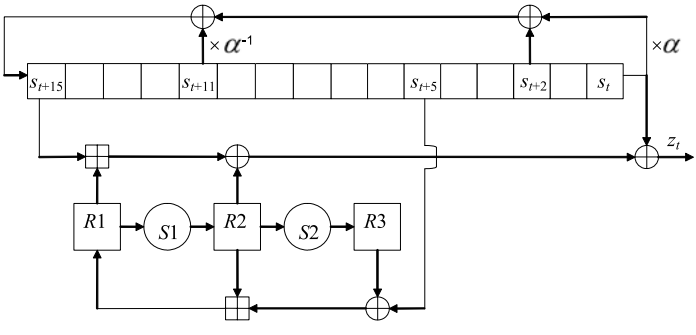

Fig. 1. SNOW 3G keystream generator

3.2 Implementation

Our analysis is based on certain operations in SNOW 3G being implemented by lookup tables. In the specification of SNOW 3G [6], the implementation for both $S1$ and $S2$ has been specified by using four lookup tables with 256 elements each. Multiplications by α and α^{-1} have been both specified by one lookup table with 256 elements. Formally, these operations are implemented as follows. Let $x = x_3x_2x_1x_0 \in \mathbb{F}_{2^{32}}$ be an arbitrary 32-bit value, where each x_i denotes an 8-bit block in x. Multiplications by α and α^{-1} in $\mathbb{F}_{2^{32}}$ use 8×32-bit lookup tables T_1 and T_2, and they are implemented as

$$\alpha x = (x \ll 8) \oplus T_1[x_3] \quad \text{and}$$
$$\alpha^{-1}x = (x \gg 8) \oplus T_2[x_0].$$

Mappings $S1$ and $S2$ use 8×32-bit tables $T_{10}, \ldots, T_{13}$ and $T_{20}, \ldots, T_{23}$, and they are implemented as

$$S1(x) = T_{10}[x_0] \oplus T_{11}[x_1] \oplus T_{12}[x_2] \oplus T_{13}[x_3] \quad \text{and}$$
$$S2(x) = T_{20}[x_0] \oplus T_{21}[x_1] \oplus T_{22}[x_2] \oplus T_{23}[x_3].$$

4 Previous Work

We give an overview of the cache-timing attack framework by Leander et al. [10] and describe how it is used to attack SNOW 2.0, which is one of the ciphers analyzed in their paper. Although their attack is targeted on SNOW 2.0, it can also be applied on SNOW 3G with small adjustments. The attack is done under the model proposed by Zenner [14] (see Sect. 2).

4.1 Attack Framework

The attack framework is built upon the assumption that clocking the LFSR involves table lookups. Let $(s_t, \ldots, s_{t+n-1}) \in \mathbb{F}_{2^m}^n$ denote the state of the LFSR at time $t \geq 0$. When the LFSR is clocked at time t, each table lookup uses some bits of s_{t+i}, $i = 0, \ldots, n-1$. Depending on the cache line size, the cache timing measurements will reveal some of these bits. Since clocking the LFSR is an $\mathbb{F}_2$-linear operation, each observed bit can be expressed as a linear combination of the bits in the initial state. Thus, once sufficiently many linear equations involving initial state bits have been collected, the initial state can be retrieved by solving the equation system. If b linear equations can be derived in each clock, about mn/b clocks are needed to recover the initial LFSR state with high probability.

The framework makes use of information obtained only from LFSR lookups; cache timings obtained from other lookups, such as S-boxes, are not utilized in the framework. For this reason, the framework is mostly concerned with recovery of the initial LFSR state. Recovery of other unknown state values, such as registers in the FSM, are left to be studied separately.

4.2 Application to SNOW 2.0 and SNOW 3G

The LFSR update function in SNOW 2.0 is the same as in SNOW 3G. Multiplications by α and α^{-1} are implemented with two 8×32-bit tables as explained in Sect. 3. Hence, each table lookup involves eight state bits. In our environment, we are able to observe four of these bits from cache measurements. Clocking the LFSR involves two table lookups, so we get eight linear equations in the initial state bits from each clock. Recovery of the initial LFSR state thus requires $16 \cdot 32/8 = 64$ clocks. To recover the full initial cipher state, the FSM registers still need to be solved at time $t = 0$. Two keystream words, z_0 and z_1, are needed for this. The initial FSM state can be recovered as follows:

1. Guess $R2_0$.
2. Compute $R1_0$ using (2) and (3) at $t = 0$.
3. Compute the output at $t = 1$, and compare it with z_1: if they match, output $R1_0$ and $R2_0$; otherwise, return to Step 1.

This process takes at most 2^{32} guesses.

In SNOW 3G, recovery of the initial LFSR state can be done as in SNOW 2.0. To recover the initial FSM state, three keystream words, z_0, z_1, and z_2, are needed. As in SNOW 2.0, the process takes at most 2^{32} guesses. It can be done as follows:

1. Guess $R3_1$.
2. Compute $R2_0$ using (6) at $t = 0$.
3. Compute $R1_0$ using (2) and (3) at $t = 0$.
4. Compute $R2_1$ using (5) at $t = 0$.
5. Compute $R1_1$ using (2) and (3) at $t = 1$.
6. Compute $R3_0$ using (4) at $t = 0$.
7. Compute the output at $t = 2$, and compare it with z_2: if they match, output $R1_0$, $R2_0$, and $R3_0$; otherwise, return to Step 1.

We conclude this section with a number of noteworthy observations.

- The previous steps would not be possible without obtaining certain keystream blocks from the KEYSTREAM oracle.
- For SNOW 2.0 and SNOW 3G the obvious albeit often costly countermeasure to the Leander et al. attacks is to remove the table lookups for the multiplications and divisions by α, for example by computing their outputs each time. Although in the description of our attack below we utilize the side-channel data from these lookups, an interesting feature of our attack is it can easily be modified to work in the absence of this data by using information from the FSM over only a few more clock cycles.

5 Our Attack

We mount a state recovery attack on SNOW 3G under the model explained in Sect. 2. In the following sections, we describe what information can be obtained from cache-timing measurements and how this information can be used to recover the full cipher state. We also present results obtained by running the attack using empirical cache-timing data.

5.1 Cache Measurements

We can assume cache-timings reveal four out of eight bits involved in each table lookup; the tables are 1kB each and span 16 cache lines. A 64-byte line fits 16 4-byte values from a table thus there are $\lg(16) = 4$ unknown bits: the timings reveal what line was accessed, but not the offset. At time $t \geq 0$, we are able

to obtain information about s_t, s_{t+11}, $R1_t$, and $R2_t$ when the cipher is clocked. Table 1 summarizes the information obtained in each clock: the left column denotes the table lookup, the center column the value involved in the lookup, and the right column the bits that are revealed. Revealed bits are given using bit masks in hexadecimal. For example, the four most significant bits of s_t are revealed.

Table 1. Revealed bits from each operation

Operation	Value	Revealed bits
α	s_t	`0xF0000000`
α^{-1}	s_{t+11}	`0x000000F0`
$S1$	$R1_t$	`0xF0F0F0F0`
$S2$	$R2_t$	`0xF0F0F0F0`

5.2 State Recovery

We describe how the cipher state can be recovered using obtained information. Suppose that the cipher is clocked for $t = 0, \ldots, c-1$, where $c \geq 20$, and that each clock reveals the bits given in Table 1. Let A_t and B_t denote the set of candidate values for $R1_t$ and s_t, respectively. In the algorithm, we first initialize A_t and B_t using the information obtained from the cache-timings. Then we trim these sets using the following relations to determine which candidates cannot be correct:

$$R1_{t+3} = S1(R1_{t+1}) \boxplus (s_{t+7} \oplus S2(S1(R1_t))) \quad \text{and} \tag{7}$$

$$s_{t+16} = \alpha s_t \oplus s_{t+2} \oplus \alpha^{-1} s_{t+11}. \tag{8}$$

We have obtained (7) by combining (5) and (6) with (4). Relation (8) is just the LFSR update function (1).

The state recovery algorithm aims at recovering the cipher state at time $t = 7$, that is, $(s_7, \ldots, s_{22})$ and $(R1_7, R2_2, R3_7)$. This is the earliest time instance after which both (7) and (8) can be used efficiently to eliminate incorrect candidates. We mostly use (7) in testing the candidates, since covering $(s_7, \ldots, s_{22})$ and $(R1_7, R2_2, R3_7)$ with (7) involves state variables from a smaller time window than covering them with (8). Thus, less clock cycles are needed to obtain sufficient amount of information about the state values. The state can be recovered in the following five steps. The actual implementation differs slightly from the description; it is discussed in the next section.

1. We first initialize candidate sets A_t, $t = 0, \ldots, c-2$. The candidates for $R1_t$ are determined based on the information about $R1_t$ and $R2_{t+1} = S1(R1_t)$ given in Table 1. For $t = 0, \ldots, c-2$, we set

$$A_t = \{x \in \mathbb{F}_{2^{32}} \mid v \wedge x = v \wedge R1_t \text{ and } v \wedge S1(x) = v \wedge R2_{t+1}\},$$

where v = 0xF0F0F0F0 and $\wedge$ denotes the bitwise AND. This set can be created by going through the 2^{16} values of $x \in \mathbb{F}_{2^{32}}$ for which $v \wedge x = v \wedge R1_t$ and checking if $v \wedge S1(x) = v \wedge R2_{t+1}$ holds.

2. We then initialize candidate sets B_t, $t = 0, \ldots, c+10$, based on the information given in Table 1. For $t = 0, \ldots, c+10$, we set

$$B_t = \{x \in \mathbb{F}_{2^{32}} \mid v \wedge s_t = v \wedge x\},$$

where

$$v = \begin{cases} \texttt{0xF0000000}, & t = 0, \ldots, 10, \\ \texttt{0xF00000F0}, & t = 11, \ldots, c-1, \\ \texttt{0x000000F0}, & t = c, \ldots, c+10. \end{cases} \tag{9}$$

3. Next, we try to eliminate as many incorrect candidates in the candidate sets using (7). For $t = 0, \ldots, c-5$ and for all $(x_0, x_1, x_2, x_3) \in A_{t+3} \times A_{t+1} \times B_{t+7} \times A_t$, we check whether

$$x_0 = S1(x_1) \boxplus (x_2 \oplus S2(S1(x_3))) \tag{10}$$

holds: if it does, we mark the corresponding candidate values as possibly correct. When all combinations have been checked, we remove the candidate values that have not been marked as possibly correct. Thus, the candidates that cannot be correct are removed.

4. We then eliminate more incorrect candidates using (8). For $t = 7, \ldots, c-9$ and for all $(x_0, x_1, x_2, x_3) \in B_{t+16} \times B_t \times B_{t+2} \times B_{t+11}$, we check whether

$$x_0 = \alpha x_1 \oplus x_2 \oplus \alpha^{-1} x_3 \tag{11}$$

holds: if it does, we mark the corresponding candidate values as possibly correct. When all combinations have been checked, the candidate values that have not passed the test are removed. The remaining values form the candidates for $(s_7, \ldots, s_{22})$.

5. To recover the full cipher state at time $t = 7$, we need to recover $(R1_7, R2_7, R3_7)$ in addition to $(s_7, \ldots, s_{22})$. For this we need $R1_7$, $R1_6$, and $R1_5$ since $R2_7 = S1(R1_6)$ and $R3_7 = S2(S1(R1_5))$. We use (10) in checking at $t = 3, 4$ because those checks involve $R1_t$ at $t = 5, 6, 7$. The values that pass the test form the candidates for $(R1_7, R2_7, R3_7)$.

The time windows in step 3 and step 4 are determined according to the indices of the candidate sets involved in these steps. Candidate sets $A_t, A_{t+1}, A_{t+3}, B_{t+7}$ are pruned in step 3. Since A_t and B_t have been initialized for $t = 0, \ldots, c-2$ and $t = 0, \ldots, c+10$, respectively, we perform the check in step 3 for $t = 0, \ldots, c-5$. The situation in step 4 is slightly more complicated. Candidate sets $B_t, B_{t+2}, B_{t+11}, B_{t+16}$ are pruned in step 4, and we take into account which of these sets have been checked in the previous step. The check in step 4 is performed from $t = 7$ because it is the first time instance when B_t is pruned in step 3. The check is performed until $t = c-9$ since it is the last time instance

when B_{t+11} is pruned in step 3. Experimentation shows that the check in step 4 is useful even if one of the candidate sets has not been pruned before.

We can easily modify the state recovery algorithm to work without the side-channel data from the α and α^{-1} lookups. This is achieved by setting $B_t = \mathbb{F}_{2^{32}}$ for $t = 0, \ldots, c + 10$ in step 2.

5.3 Attack Performance

The crux of the state recovery algorithm complexity is the size of the sets A_t in step 1, where candidates for $R1_t$ are determined based on observed bits of $R1_t$ and $R2_{t+1} = S1(R1_t)$. These bits are enough to determine a very small set of candidates for $R1_t$. Assuming uniformly distributed $R1_t$ values, we calculated the set size for all $x \in \mathbb{F}_{2^{32}}$. The average is a surprisingly low 4.26. There is an algorithm requiring 2^{32} steps and 2^{16} storage to verify this average. Step 1 requires $c \cdot 2^{16}$ steps and experiment results show this is easily the most costly step of the algorithm. Hence we omit any formal complexity analysis of the remaining steps.

The state recovery algorithm can be implemented as a backtracking algorithm by combining steps 3 and 4. To make the algorithm more efficient, incorrect candidates can be eliminated as soon as they are known to be invalid. It is not necessary to store the whole B_t set into memory in step 2; storing only $v \wedge s_t$ is sufficient. A more detailed description about the implementation is given in Appx. A.

Increasing the number of clock cycles c increases the probability of a unique solution because check (11) can be applied more times. It can be run $c - 15$ times: in the ith run, it eliminates incorrect s_{t+i-1} for $t = 7, 9, 18, 23$. Check (11) can be used $c - 20$ times such that every s_t candidate in the check has been tested with (10) before. For example, this happens only in the first run if the cipher is clocked $c = 21$ times.

Simulated Side-Channel. One model present in the literature for obtaining and/or simulating cache-timing data is to verify theoretical cache-timing attack results while at the same time abstracting away all details of the underlying cache structure. This involves modifying the implementation of the cipher to manually store the top bits of indices used in table lookups. This inherently simulates an error-free side-channel. For example, Acıiçmez and Koç used this approach to verify results on an AES trace-driven cache attack [1, Sect. 5].

Table 2 gives the attack results under this model. The last column in the table gives the average number of operations after step 1.

Empirical Side-Channel. Considering the side-channel model, another approach is to view the cipher as a black-box where essentially the attacker can control the rate at which the cipher is clocked. This allows the attacker a certain level of granularity between successive calls to obtain the needed trace data. Unlike the previous method, here the cache-timing data is empirical. For example,

Table 2. Remaining candidate states and their frequencies

Cycles	1 state	2 states	≥ 3 states	Avg. ops
20	0.110	0.247	0.643	3100
21	0.351	0.379	0.270	3157
22	0.991	0.009	0	3105
23	0.995	0.005	0	3121

Osvik, Shamir, and Tromer used this approach to run trace-driven cache attacks on AES running on an AMD Athlon 64 [12, Sect. 3.7]. They applied their results to two black-boxes: 1) AES through OpenSSL function calls and 2) AES through the `dm-crypt` disk encryption subsystem for Linux.

To implement the attack under this model, we used a C implementation of SNOW 3G making use of the table lookups described in Sect. 3 with tables provided by the specification [6]. We considered the AMD Athlon 64 3200+ Venice that has a 64KB 2-way associative L1 data cache and 64-byte cache lines running 32-bit Ubuntu Linux 8.04. See [8, Chap. 18] for background on trace-driven cache attacks.

We ran one thousand iterations of the attack, carrying out each iteration as follows. Here "each set" is perhaps better interpreted as being from the subset of cache sets where the considered tables map to.

1. Initialize a SNOW 3G instance; it takes the 128-bit key and IV from `/dev/urandom`.
2. For each cache set, read from two distinct areas of memory that map to that cache set. This completely pollutes a two-way associative cache. Osvik et al. call this the "Prime" step [12, Sect. 3.5].
3. Clock the cipher instance.
4. Measure the time required to read back the previously read data from each set. Osvik et al. call this the "Probe" step.
5. Repeat these steps for 23 clocks.
6. Using the obtained cache-timing data, run the attack outlined in this section.

In each iteration of the "Probe" step, for each table the result is sixteen latency measurements, each for a distinct cache set. Hence the cache set with the highest latency is the best guess for which cache set the cipher accessed and the inferred input index bits (state bits) follow accordingly.

The attack runs in a matter of seconds. Of these one thousand attack iterations, 647 succeeded in recovering the full LFSR and FSM state of the SNOW 3G instance. In three cases we were left with two candidate states; we arrived at unique solutions for the remaining cases. As far as the probability of a unique solution, this agrees with the simulated data in Table 2. Note that values listed in the table and the observed 64.7% success rate are not the same metric. The former uses error-free simulated traces and measures the probability of a unique solution, and the latter empirical traces and the probability of obtaining at least

one candidate solution. For our spy process on this architecture, we experienced extremes that either a trace was completely error-free or contained largely errors. Perhaps it is possible to modify the state recovery algorithm to resolve conflicting state information and compensate for errors, but with such a high success rate this would have little to no impact in the end.

To summarize, in this environment the expected number of attack iterations to succeed is only two.

6 Countermeasures

High-speed software implementations of SNOW 3G are largely table-based. In environments where cache-timing attacks pose a threat, countermeasures are needed. Straightforward approaches such as aligning the tables in memory at the same boundary provide little assurance.

Bit-slicing is one approach to eliminate state-dependent table lookups from memory. In the case of SNOW 3G and a processor with word size w, we represent w streams of the cipher running in parallel under w distinct keys and IVs, each stream at a single fixed bit index within the word. This allows a quasi-hardware design approach to components where instructions implement gates. Table lookups are replaced by their computational counterpart: computing S-box outputs instead of looking them up. Bit-slicing is a popular, established paradigm for high-speed and side-channel secure implementations of cryptographic primitives [3,11,7]. Our main focus here is on processors with Streaming SIMD Extensions 2 (SSE2) where $w = 128$. SNOW 2.0 runs on a subset of this machinery, hence this countermeasure can be directly applied there as well.

We reviewed SNOW 3G components in Sect. 3; here we discuss their implementation.

The LFSR. On the software side, to avoid excessively relocating data when clocking the LFSR we employ a standard sliding window; this changes the memory address of the LFSR instead of moving the data in the LFSR. Briefly, the trick is to allocate twice (or more) the memory required for the LFSR and keep a pointer to the current offset; the window representing the current LFSR state starts at this pointer and covers the length of the original LFSR. Then to clock, only the pointer increments (the window slides) and the new value from the feedback polynomial is stored at the end of the window. Thus the rest of the values can remain where they are, and only when the window runs out of space all LFSR values are moved back to the beginning and the pointer reset. For the LFSR feedback function, multiplications and divisions by α each involve a distinct linear map $\mathbb{F}_2^8 \to \mathbb{F}_2^{32}$. We implement these using 117 and 103 gates, respectively.

The FSM. We implement modulo 2^{32} additions using a textbook ripple-carry adder of 154 gates. The $S1$ component includes four evaluations of the AES S-box followed by a MixColumns operation. We use the 115 gate design by Boyar and Peralta [4], requiring 119 gates in the absence of an XNOR instruction.

With respect to gate count, the bottleneck of clocking the cipher is the $S2$ component. This involves four computations of a bijective S-box $\mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ defined by evaluating the Dickson polynomial

$$g_{49}(t) = t + t^9 + t^{13} + t^{15} + t^{33} + t^{41} + t^{45} + t^{47} + t^{49}$$

followed by a translation; all of the outputs then go through a MixColumns operation. A Dickson polynomial g_i permutes $\mathbb{F}_q$ when $\gcd(q^2 - 1, i) = 1$. They have a recursive definition, and in this setting it holds that

$$\begin{aligned} g_{49}(t) &= g_7(g_7(t)), \quad \text{where} \\ g_7(t) &= t + t^5 + t^7, \end{aligned}$$

which suggests implementing this S-box with four multiplications (two for each successive g_7 evaluation) in $\mathbb{F}_{2^8}$. We use a composite field isomorphism $\mathbb{F}_{2^8} \to \mathbb{F}_{2^4}^2$ to employ Paar's multiplier that requires 110 gates [13, Chap. 6]. We briefly examined the effect of 64 different isomorphisms on the overall gate count of the S-box, found it nominal, and settled on the mapping $(x^5+1)^i \mapsto ((x^2+1)y+x^3)^i$. With the isomorphism (and inverse), multiplications, squarings, and assorted additions, we realize this S-box design using 498 gates. Although this is the smallest public design we are aware of, future work on more compact designs for this S-box is needed: our implementation spends roughly 46.6% of its time in this S-box, compared to 10.8% in the AES S-box.

Results. We summarize the design in Table 3. Median timings for our implementation producing long sequences of keystream running on an Intel Core 2 Quad Q6600 are just over 12K cycles per 512 keystream words, or 5.9 cycles per stream byte inclusive of keystream conversion from bit-slice representation (Matsui-Nakajima method [11, Sect. 4.2]). For a terse comparison, a serial table-based sliding window implementation running on the same machine weighs in at roughly 23.9 cycles per keystream word, or 6.0 cycles per byte. This suggests that in environments where batch keystream generation can be applied, bit-slicing affords SNOW 3G cache-timing attack resistance without degrading performance.

Table 3. Gate counts for various components in the design

Component	Count	Gates
α	1	117
α^{-1}	1	103
AES S-box	4	119
Dickson S-box	4	498
MixColumns	2	152
32-bit addition	2	154
32-bit XOR	5	32

Furthermore, with Advanced Vector Extensions (AVX) on the horizon, bit-slice implementations scale nicely with the widened data path ($w = 256$).

Since the cipher splits a 32-bit word into bytes for the S-box evaluations, it is tempting to use a bit-slice representation which aligns the bits of these individual bytes and runs a quarter of the streams instead. However, there are two components which then become awkward to implement: the modulo 2^{32} addition and the linear maps involving α. We chose the former representation due to these factors.

7 Conclusion

In this paper, we developed a cache-timing attack on SNOW 3G, the standard keystream generator for mobile communications, that adheres to an existing attack model. Our attack is an improvement in several aspects over previous attacks:

- We presented a new, efficient attack exploiting the special structure of the FSM used in SNOW 3G, applicable also to K2 currently under standardization consideration by ISO. When countermeasures to the Leander et al. [10] attacks are deployed, our attack still succeeds.
- On the theoretical side, unlike the attack presented by Leander et al. ours uses all information gained from the SCA_KEYSTREAM side-channel oracle. This allows us to mount a substantially more efficient attack and recover the FSM state without known keystream, and furthermore requires notably fewer cipher clocks to be observed.
- On the practical side, we ran our attack using empirical cache-timing data and were able to recover the full cipher state with all probability and no computational effort to speak of.

Finally, we presented a bit-slice implementation of SNOW 3G, applicable to SNOW 2.0 as well, to defend against timing attacks in general. For batch keystream generation, we were able to accomplish this without a performance hit compared to high-speed table-based implementations.

We close with an important note on further applications of this work. One of the main reasons we are able to make our attack more efficient is the structure of the FSM: it has two consecutive S-boxes, $S1$ and $S2$, so we get information about the input *and* output of $S1$. This information makes it possible to determine a small set of candidates for the inputs of $S1$, i.e., $R1_t$. Similar structures are present in other ciphers as well:

- The FSM in the K2 cipher [9] also contains consecutive S-boxes.
- SNOW 2.0 [5] and SOSEMANUK [2] do not have consecutive S-boxes, but their structure still allows us to use a similar idea to recover the FSM state without the keystream given that the LFSR state is recovered first. For example, $R1$ in SNOW 2.0 is updated as

$$R1_{t+1} = R2_t \boxplus s_{t+5} = S1(R1_{t-1}) \boxplus s_{t+5}.$$

The $S1$ lookups reveal bits of $R1_{t+1}$ and $R1_{t-1}$ as in SNOW 3G. These bits (and the knowledge of s_{t+5}) allow us to determine $R1_{t+1}$ almost uniquely. The same idea can be used for SOSEMANUK.

These results should be taken into account when designing stream ciphers resilient against side-channel attacks.

References

1. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 112–121. Springer, Heidelberg (2006)
2. Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., Sibert, H.: Sosemanuk, a fast software-oriented stream cipher. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 98–118. Springer, Heidelberg (2008)
3. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (1997)
4. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 178–189. Springer, Heidelberg (2010)
5. Ekdahl, P., Johansson, T.: A new version of the stream cipher SNOW. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 47–61. Springer, Heidelberg (2003)
6. ETSI/SAGE: Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 2: SNOW 3G specification. Version 1.1. Tech. rep. (2006), `http://gsmworld.com/documents/snow_3g_spec.pdf`
7. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009)
8. Koç, Ç.K. (ed.): Cryptographic Engineering. Springer, Heidelberg (2009)
9. Kiyomoto, S., Tanaka, T., Sakurai, K.: K2: A stream cipher algorithm using dynamic feedback control. In: Hernando, J., Fernández-Medina, E., Malek, M. (eds.) SECRYPT, pp. 204–213. INSTICC Press (2007)
10. Leander, G., Zenner, E., Hawkes, P.: Cache timing analysis of LFSR-based stream ciphers. In: Parker, M.G. (ed.) Cryptography and Coding. LNCS, vol. 5921, pp. 433–445. Springer, Heidelberg (2009)
11. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel core2 processor. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 121–134. Springer, Heidelberg (2007)
12. 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)
13. Paar, C.: Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. Ph.D. thesis, Institute for Experimental Mathematics, Universität Essen, Germany (1994)
14. Zenner, E.: A cache timing analysis of HC-256. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 199–213. Springer, Heidelberg (2009)

A Implementation Details

We implemented the state recovery algorithm as a backtracking search. The algorithm enumerates partial candidate solutions and eliminates incorrect candidates as soon as they are known to be invalid.

The first step in the algorithm is the same as described in Sect. 5: candidate sets A_t, $t = 0, \ldots, c-2$, are initialized. In the second step, initial candidates for s_t, $t = 0, \ldots, c+10$, are determined. Instead of storing the candidates as described in Sect. 5, we simply set $B_t = v \wedge s_t$, where v is chosen as in (9). The algorithm enumerates possible candidate combinations creating a tree structure, where a node at depth $t+1$ represents a candidate from A_t. If the tree cannot be further extended, a complete valid solution has been found. When a node at depth t is extended by picking a new candidate from A_t, the algorithm checks if the new candidate can be correct. Depending on the depth $t+1$, it performs up to three checks based on the following relations:

$$s_{t+16} = \alpha s_t \oplus s_{t+2} \oplus \alpha^{-1} s_{t+11}, \tag{12}$$

$$v \wedge s_{t+16} = v \wedge (\alpha s_t \oplus s_{t+2} \oplus \alpha^{-1} s_{t+11}), \quad \text{and} \tag{13}$$

$$v \wedge s_{t+7} = v \wedge ((R1_{t+3} \boxminus S1(R1_{t+1})) \oplus S2(S1(R1_t))), \tag{14}$$

where $\boxminus$ denotes subtraction in $\mathbb{Z}_{2^{32}}$ and v is chosen as in (9). These relations have been derived using the relations mentioned in Sect. 5. A path from the root node to the most recently expanded node in the search tree forms a sequence consisting of candidates for $R1_t$ at different times. Given enough candidates, we can determine candidates for different LFSR values, which can be used in the checks. Each time a node is expanded, the algorithm performs three checks using the above relations to determine whether the newest node can be valid. In each check, the time t is adjusted such that the candidate represented by the newest node is involved in the check (if possible).

1. If the depth allows, we use (12) to check the newest node. Using the candidates for $R1_t$ at different times and (7), we determine the candidates corresponding the LFSR values in (12). We then check if the relation holds with the candidate values. We need at least 20 candidates for $R1_t$ at consecutive time instances to perform this check.
2. If the previous check did not fail and if the depth allows, we then use (13) to check the newest node. We first determine the candidates corresponding the LFSR values on the right-hand side in (13). Using these candidates and the known value for $v \wedge s_{t+16}$, we check if the relation holds. At least 15 candidates for $R1_t$ at consecutive time instances are needed in this check.
3. If the previous checks have not failed and if the depth allows, we then use (14) to check the newest node. Using the known value of $v \wedge s_{t+7}$ and the candidates for $R1_t$ at different times we check if the relation holds. At least 4 candidates for $R1_t$ at consecutive time instances are needed in this check.

The algorithm can be modified to use only the side-channel data from the FSM lookups by omitting the two latter checks which also utilize data from the LFSR lookups.