Susceptibility of eSTREAM Candidates towards Side Channel Analysis[⋆]

Benedikt Gierlichs[1], Lejla Batina[1], Christophe Clavier[2], Thomas Eisenbarth[3], Aline Gouget[4], Helena Handschuh[5], Timo Kasper[3], Kerstin Lemke-Rust[6], Stefan Mangard[7], Amir Moradi[8⋆⋆], and Elisabeth Oswald[9]

[1] K.U. Leuven, ESAT/SCD-COSIC, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium, `firstname.lastname@esat.kuleuven.be`

[2] Gemalto, Security Labs, Avenue du Jujubier, ZI Athélia IV, 13705 La Ciotat Cedex, France, `christophe.clavier@gemalto.com`

[3] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, 44780 Bochum, Germany, `eisenbarth@crypto.rub.de`, `tkasper@crypto.rub.de`

[4] Gemalto, Security Labs, 6 rue de la Verrerie, 92190 Meudon, France, `aline.gouget@gemalto.com`

[5] Spansion,105 Rue Anatole France, 92684 Levallois-Perret Cedex, France, `helena.handschuh@spansion.com`

[6] T-Systems GEI GmbH, Rabinstr. 8, 53111 Bonn, Germany `kerstin.lemke@t-systems.com`

[7] Infineon Technologies AG, Am Campeon 1-12, 85579 Neubiberg, Germany, `stefan.mangard@infineon.com`

[8] Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, `amir.moradi@crypto.rub.de`

[9] University Bristol, Department of Computer Science, Bristol, United Kingdom, `elisabeth.oswald@bristol.ac.uk`

Abstract. We analyze the relevant candidates in phase 3 of the eSTREAM project with respect to side channel analysis in a theoretical approach.

Keywords: Side Channel Analysis, eSTREAM

1 Introduction

The eSTREAM project [1] is an open multi-year effort to identify new stream ciphers that might become suitable for widespread adoption. Stream ciphers are

[⋆] The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT, the European Network of Excellence in Cryptology.

[⋆⋆] Amir Moradi performed most of the work described in this contribution as a visiting researcher at Ruhr University of Bochum.

evaluated in two categories of applications. Profile 1 includes stream ciphers for software applications with high throughput requirements and Profile 2 includes stream ciphers for hardware applications with restricted resources such as limited storage, gate count, or power consumption. Table 1 summarizes the candidates in phase 3 of the eSTREAM project.

Table 1. Stream cipher candidates in phase 3 of the eSTREAM project.

Profile 1 (SW)	**Profile 2 (HW)**
CryptMT (CryptMT Version 3)	DECIM (DECIM v2, DECIM-128)
Dragon	Edon80
HC (HC-128, HC-256)	F-FCSR (F-FCSR-H v2, F-FCSR-16)
LEX (LEX-128, LEX-192, LEX-256)	Grain (Grain v1, Grain-128)
NLS (NLSv2, encryption only)	MICKEY (MICKEY 2.0, MICKEY-128 2.0)
Rabbit	Moustique
Salsa20	Pomaranch (Pomaranch Version 3)
SOSEMANUK	Trivium

It is worth noting that side channel analysis is an implementation attack, i.e., in practice the susceptibility depends on the design of the cipher, the design of the implementation, and the target platform. However, this work follows a more general theoretical approach and focuses on the susceptibility of implementation properties for a given cipher design. The susceptibility towards side channel analysis is assessed by considering established implementation techniques, side channel leakage models, and side channel attacks. Previously, similar approaches were carried out by Biham and Shamir [3] as well as Daemen and Rijmen [5] for the AES candidates and by Oswald and Preneel for the NESSIE candidates [9].

This paper is organized as follows. Section 2 introduces the framework and evaluation criteria for assessing the theoretical susceptibility of stream ciphers towards side channel analysis. In Sections 3 and 4 these evaluation criteria are applied to all relevant stream ciphers in phase 3 of the eSTREAM project. Section 5 summarizes our results. For the specifications of the eSTREAM ciphers, the reader is referred to [1].

2 Evaluation Criteria

A stream cipher typically includes the operations key setup, IV setup and key stream generation (encryption). Note that the mandatory interface for the reference implementations specifies four interface functions: ECRYPT_init (initialization), ECRYPT_keysetup (key setup), ECRYPT_ivsetup (IV setup) and ECRYPT_process_bytes (encryption or decryption). Depending on the construction of the cipher initialization, key setup, and IV setup may comprise a single function.

2.1 The Adversary

For the subsequent evaluation of eSTREAM candidates we model the adversary as follows:

- she knows the IV and the keystream, optionally she can choose IVs
- she can reset the stream cipher (i.e., re-invoke IV and/or key setup)
- she aims at key recovery.

2.2 Timing Analysis (TA)

Timing Analysis [7] exploits a dependency between the execution time of an algorithm and the course of its (secret) internal state. If such dependencies exist, it is possible to recover the secret by carefully analyzing the algorithm's execution time. With respect to Timing Analysis we consider key setup, IV setup and encryption. We look into the following issues:

- Does the stream cipher include conditional branches depending on the internal state?
- Does the stream cipher include table look-ups (which might allow cache timing attacks [2] against software implementations on certain platforms)?
- Can a timing analysis attack be mounted (in key setup, IV setup, and encryption)? If so
  - How many key bits (or internal state bits) are compromised? For software implementations we consider a 32-bit device.
  - Which countermeasures can be implemented to counteract timing analysis?
  - What are the efficiency costs of countermeasures?

2.3 Power Analysis

Leakage Model The adversary observes a perfect Hamming weight (or Hamming distance) leakage. Algorithmic noise (i.e., noise by parallel operation of the cipher) should be considered, especially in hardware implementations due to wide data paths. Non-algorithmic noise (measurement, external and further intrinsic noise) is neglected.

Simple Power Analysis Simple Power Analysis (SPA) [8] exploits dependencies between the instantaneous power consumption and instructions and/or (secret) data being processed. The attacker infers information from one or a few power measurements. In our analysis SPA applies to key setup, IV setup and encryption. We look into the following issues:

- Does the stream cipher include conditional branches depending on the internal state?
- Can a SPA attack be mounted (in key setup, IV setup, and encryption)? If so,

    - how many key bits are compromised as result of the attack? For software implementations we consider a 32-bit device.
    - Is the SPA attack considered to be straight-forward or complex?
    - Can an efficient masking scheme be applied to prevent SPA?

In order to quantify the information leakage we use the Shannon Entropy as defined in [4]. The entropy of the Hamming weight of an 8-bit, 16-bit, and 32-bit random variable is 2.54 bits, 3.05 bits, and 3.55 bits, respectively. For a more elaborate discussion, see Appendix.

Differential Power Analysis Differential Power Analysis (DPA) [8] is based on the functional dependency of the power consumption and the data processed. In particular, it analyzes the *differences* in the power consumption due to the processing of *different* data values. In this work (first-order) DPA applies to IV setup and encryption. The following questions are considered:

- What are the fundamental operations of the stream cipher?
- Can a Differential Power Analysis attack be mounted (in IV setup and encryption)? If so
    - how many key bits are compromised as result of the attack? For software implementations we consider a 32-bit device.
    - How many (different) operations of the cipher are needed to be attacked?
    - What is the number of key hypotheses? For software implementations we consider a 32-bit device.
    - Is the DPA attack considered to be straight-forward or complex?
    - Can an efficient masking scheme be applied to prevent DPA?

Note that, although we focus on 32-bit devices for software implementations, an adversary will likely have the choice to apply the divide-and-conquer principle in order to attack smaller parts of the key. The increased algorithmic noise can usually be compensated by an increased number of measurements. Basically, one trades computational complexity for measurements. The same observations hold for hardware implementations.

2.4 Countermeasures

We provide an intuition on how costly possible power analysis countermeasures for software candidates are in terms of efficiency based on the following assumptions:

- boolean masking is easy,
- masking of small tables (up to 256 bytes) is medium,
- masking of larger tables and protection of algorithms that contain boolean and arithmetic operations is costly.

For hardware candidates the topic is more difficult. A general approach to protect a given circuit is the use of secure logic. However, this implies an area increase by a factor of about three to five and an increase of power dissipation by a factor of about two to three, depending on the logic style.

2.5 Summary of the evaluation

We summarize our findings about each eSTREAM candidate in a table as follows. Note that “maybe” means that a vulnerability exists but we are not sure whether it can be exploited efficiently.

Table 2. Summary of (theoretical) side channel susceptibility for `** cipher name **`

Exploitable (cache) timing vulnerability:	yes/ no/ maybe
Exploitable conditional branches vulnerability:	yes/ no/ maybe
Exploitable HW leakage of data, SPA vulnerability:	yes/ no/ maybe
Exploitable DPA vulnerability:	yes/ no/ maybe
DPA Attack complexity:	low/ medium/ high
Software candidates only:	
Cost of countermeasures:	low / medium / high

3 Phase 3 candidates profile 1 (SW)

In this section we focus on the profile 1 candidates mentioned in Table 1.

3.1 CryptMT

The stream cipher CryptMT consists of a huge state linear generator (called the mother generator) and a non-linear filter with memory. In the most recent version (version 3) of CryptMT, the so-called Simple Fast Mersenne Twister (SFMT) is used as mother generator. The SFMT has an internal state of 156 128-bit integers and a period that is a multiple of the Mersenne prime $2^{19937}-1$. The output of the SFMT is a 128-bit integer that is fed into a non-linear filter with a memory of 128 bit. The key stream of CryptMT is generated based on the output of this filter.

The internal state of the SFMT is very large and therefore a so-called booter is used for initialization. This booter expands the key and the IV to a sequence of 156 128-bit integers that are used to fill the state of the SFMT. However, during the initialization the integers are not only used for the SFMT. The output of the booter is also sent to the non-linear filter to already generate a pseudorandom sequence during initialization. After the internal state of the SFMT has been filled, the input of the filter is switched from the booter to the SFMT.

The cipher is in particular suited for 32-bit platforms as it uses addition and multiplication modulo 2^{32}. Furthermore, CryptMT also uses permutations, shift operations as well as bitwise AND, OR, and XOR operations.

There are no conditional jumps in the algorithm and no table-lookups. Furthermore, the permutations and shift operations are fixed and do not depend on key-related material. Hence, standard timing attacks should not be an issue for typical implementations.

In case of power analysis attacks, the initialization is the most interesting point of attack. In this phase, the booter expands the key and the IV to a sequence of 128-bit integers. This expansion can be exploited in a DPA attack, if the IV is known. There are intermediate 32-bit values that depend on the IV and the key. The attacker can in general reveal the entire key by formulating 2^{32} hypotheses for each 32-bit subkey. However, there are also instances where the attacker can perform DPA attacks with less key hypotheses. For example, at the very beginning of the initialization of the booter there are subtraction/addition operations modulo 2^{32} of the key and the IV. Such operations can be attacked by only predicting the LSB of the key first and then by successively predicting more and more bits of the key. The fact which parts of the IV are combined with which parts of the key depends on the key size. Hence, the exact attack strategy for a DPA attack on CryptMT depends on the key size.

Besides DPA attacks, it is also possible to mount SPA and template attacks. There are several instances in the algorithm, where the attacker can learn bits about the key or the internal state. However, as all operations of the cipher are 32-bit operations, these attacks require that the attacker can distinguish the Hamming weights $0 \ldots 32$ in the power traces. In the case of template attacks therefore 33 templates are required.

An attacker who observes the Hamming weight of a 32-bit values that occur during the operation of a cipher, on average learns 3.55 bit of information about each value. This general leakage occurs for every operation in the cipher. In addition to this, shift operations are a potential source of information. If these shifts are done bitwise, the attacker can learn one bit during each shift. Shift operations are performed at several instances in the algorithm. Another source of additional information are logical operations with constants. There are instances in the cipher, where bitwise AND and OR operations of fixed public values and secret information is performed. By comparing the Hamming weight of the output of the operation and the Hamming weight of the secret input, the attacker can learn some bits of information. There are several instances where the algorithm leaks information. However, for a concrete statement on how the leakage can be combined to a powerful SPA or template attack, it would be necessary to analyze a concrete implementation. Counteracting power analysis attacks on CrypMT by masking seems expensive because the algorithm uses boolean as well as arithmetic operations.

Table 3. Summary of (theoretical) side channel susceptibility for `CryptMT`

Exploitable cache timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium
Cost of Countermeasures:	high

3.2 Dragon

We focus here on the 128-bit key version of the stream cipher DRAGON. All security comments exposed in the sequel are more or less applicable on the 256-bit key version as well.

The stream cipher DRAGON takes on input a 128-bit secret key k and a known and controllable 128-bit initialization vector iv.

The stream cipher DRAGON does not contain any conditional branching and is so immune to classical forms of timing analysis. Nevertheless, the F function of the algorithm makes use of two 8-bit to 32-bit substitution boxes S_1 and S_2 (1 KB each) which make the algorithm possibly vulnerable to some kind of cache-based timing analysis.

As the cipher does not contain any conditional branching, it is not subject to Simple Power Analysis at the instruction level.

It is conceivable to apply SPA at the data level if the adversary is able to infer from the side channel signal the Hamming weight of some intermediate data. As a simple example, the initialization function (which generates the starting value of the internal state from both k and iv) begins with the manipulation of all four 32-bit chunks of the 128-bit sensible data $(a, b, c, d) = W_0 \oplus W_6 \oplus W_7$. Denoting $k = (k_0, k_1, k_2, k_3)$ and $iv = (iv_0, iv_1, iv_2, iv_3)$, we have $a = d = k_0 \oplus k_2 \oplus iv_0 \oplus iv_2$ and $b = c = k_1 \oplus k_3 \oplus iv_1 \oplus iv_3$. The observation of the Hamming weights of a and b leaks $2 \cdot 3.55$ bits on k. By invoking the cipher with different initialization vectors, the adversary can learn these Hamming weight for different values of a and b. This results in the recovery of 64 bits of the key corresponding to the precise determination of $k_0 \oplus k_2$ and $k_1 \oplus k_3$. We believe that the observation of some other intermediate data further in the initialization process will likely help the adversary to recover the whole key value.

While not impossible, a data masking scheme that could prevent this attack will probably result in a large increase of memory (due to the need of masking both S-Boxes) and execution time (due to the mix of arithmetic and boolean operations).

Differential Power Analysis (DPA) is also possible on this algorithm. As an example, one of the very first operations in the initialization phase computes $G_1(a + f)$ where $a = k_0 \oplus k_2 \oplus iv_0 \oplus iv_2$ and f is a known constant. The non-linear function G_1 applied to the 32-bit data $x = (x_0, x_1, x_2, x_3)$ is defined using S-Boxes S_1 and S_2 by $G_1(x) = S_1(x_0) \oplus S_1(x_1) \oplus S_1(x_2) \oplus S_2(x_3)$. For each S-Box computation, a guess on only 8 bits of a key-related value allows to predict the output of this S-Box. Making varying the relevant part of the initialization vector, the adversary is thus able to mount a DPA based upon this prediction. For each attacked S-Box, 8 key bits are recovered by considering 256 key hypotheses. In this example the secret value $k_0 \oplus k_2$ is retrieved. Applying DCSA on other (possibly deeper) parts of the initialization phase will likely allow to reveal the whole key.

Table 4. Summary of (theoretical) side channel susceptibility for `DRAGON`

Exploitable cache timing vulnerability:	yes
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	low
Cost of Countermeasures:	high

3.3 HC-128 and HC-256

HC-128 and HC-256 respectively use a 128-bit and 256-bit key. For simplicity, we will describe our results on HC-128, but a straightforward extension of these findings can be made to HC-256. HC-128 uses two secret tables P and Q of 2^9 elements each; each element in the tables is a 32-bit value. 6 different update functions are used within HC-128. f_1 and f_2 are similar to the message expansion functions of SHA-256. They compute the eXOR sum of rotated and shifted versions of a unique variable. g_1 and g_2 have three variables each and compute the eXOR sum of rotated versions of their three variables. h_1 and h_2 compute the sum of the outputs of two terms of the secret tables P and Q respectively; one byte of the variable is used as an index to the lower half of the table and another byte is used as an index to the upper half of the table for each one of the functions. The key and IV setup phase repeats the secret key twice in the first 8 state words $W_0 \dots W_7$, then repeats the IV twice in the next 8 state words $W_8 \dots W_{15}$ and then generates the next 1264 state words by applying a recursive addition of five terms, namely $W_i = f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i$ for $i \geq 16$. After this, the secret tables P and Q are formed with the last 1024 such generated words (i.e. the first 256 ones are thrown away, the next 512 words are allocated to P and the last 512 words are allocated to Q).

Once the setup phase is done, the key stream generation phase takes place by updating each element of the tables P and Q in turn using a recursive addition of plain table elements and images of table elements by g_i, and by outputting a 32-bit word which depends on the last updated term of the table and an h_i function of another term of the table. For more precise details, see the reference submission.

We note that the f_i functions use both rotation operations and shift operations. In an implementation which shifts or rotates the 32-bit operands bit by bit, those values are revealed bit by bit. This can be precluded by careful hardware and software implementations in which rotations and shifts are done in a single clock cycle on 32-bit data operands.

We further note that the ciphers use secret S-Boxes in the form of the P and Q tables. This could indicate that cache timing attacks could be possible on high-end processors. However this is not the case since there are no secret indices used for the table look-ups, at least in the setup phase. In other words, if cache lines get evicted, the attacker learns nothing since the indexes to the tables are already known, and no exploitable collisions occur in those indexes. Later on, in the keystream generation phase, the indexes do become secret, but by then the tables are not known to the attacker anymore, so straightforward cache attacks with unknown table elements do not seem to lead to promising results.

If we consider SPA, for example using a simple Hamming weight model, we can exploit the fact that the f_i functions use rotations and shifts of the 32-bit key words. Thus the Hamming weight of every 32-bit key word is revealed during that computation. This means we have an entropy loss of about 3.55 bits per 32-bit key element, which means 14 key bits are lost on the overall 128-bit key (or equivalently 28 key bits are lost for the 256-bit key version in HC-256). Furthermore, the f_i functions use shift operations, which shift out 3 bits of the key word. Therefore the difference between the total Hamming weight and the Hamming weight of the remaining 29-bit word reveals the Hamming weight of the 3 shifted bits. On these three bits the entropy loss is 1.81 bits, and on the 29-bit word, there is a further entropy loss of 3.47 bits. In total, the overall remaining key size is thus $4 \cdot 29 - 4 \cdot 3.47 = 102$ key bits plus $4 \cdot 1.19 = 4.76$ bits, or in total 107 key bits.

For DPA attacks, during the key setup phase, one of the most straightforward ideas is to measure the Hamming weight HW_1 of $f_1(K_1)$ and then to guess the value of the secret key word K_0*. Next, we let $\Phi(IV) = f_2(W_{i-2}) + W_{i-7} + i$ for $i = 16$. We solve for an IV such that $\Phi(IV) = -K_0*$ and we measure the Hamming weight of $\Phi(IV) + K_0 + f_1(K_1)$. If both measured Hamming weights are equal our guess for K_0* was correct with high probability, if not we try the next value. Overall we need to guess and test 2^{32} key values to recover a set of possible values for that 32 bit chunk of the key. Of course equal Hamming weights do not imply equality in the operands, but a few additional queries with different IVs can rule out the majority of wrong guesses easily. Therefore the full key can be recovered in close to four times 2^{32} complexity and power curves. This is of course not very efficient.

A better attack scenario is the following: we choose IVs such that $\Phi(IV) = j$, where $i = 16$ and starting from $j = 0$, $j = 1$, $j = 2$, $j = 4$ up to $j = 2^{31}$ for every possible power of two. The first value gives us the Hamming weight of $\alpha = K_0 + f_1(K_1)$. The next value gives us the Hamming weight of $\alpha + 1$. If this weight is incremented by 1, we conclude the last bit of α was zero. If the Hamming weight stays identical, the last two bits of α were $0x01$. If the Hamming weight decreases by n, we conclude the last $n+1$ bits of α were equal to $0x1 \dots 1$. Once we have found one, two or n bits of α, we take the next relevant power of two for j and measure the Hamming weight changes anew. This will reveal the whole value of α after a maximum of 32 chosen IVs. In order to recover the full secret key, we need to construct 32 chosen IV's for every relation of the type $K_l + f_1(K_{l+1})$ for l from 0 to 3. For the first two relations, obtained from $i = 16$ and $i = 17$ this is straightforward, however, for $i = 18$ and $i = 19$, the f_2 function is applied to words W_{16} and W_{17} which are functions of $K_0 + f_1(K_1)$ and $K_1 + f_1(K_2)$ respectively. Therefore we first need to solve the first two relations, and then to adaptively query for IVs such that we can again isolate the terms $K_2 + f_1(K_3)$ and $K_3 + f_1(K_0)$. We end up with four equations involving four unknown key words, which can either be solved by guessing one key word and deriving the three others, in complexity 2^{32}, or by trying to solve the four equations in a more efficient direct way.

In HC-256, we need the double number of adaptively chosen IVs, namely 8 times 32 IVs instead of 4 times 32 adaptively chosen IVs, but the overall complexity stays the same since we only need to guess one 32-bit key word to solve the other seven words.

The key stream generation algorithm seems to be more complex to attack since by then all the table elements are basically unknown and the equivalent key size has become huge. So the most promising attacks are the ones mentioned above. In order to protect against these attacks, we should consider a masking scheme for the cipher. However, the fact that the cipher heavily uses 32-bit operands, and that it constantly mixes boolean and arithmetic operations implies that the masking countermeasures will be relatively inefficient and slow down the algorithm by a non-negligible factor. It seems particularly difficult to protect the implementation of the random tables P and Q; however, this may not be required if no efficient side-channel attacks are found on the keystream generation phase.

Table 5. Summary of (theoretical) side channel susceptibility for `HC-128 and HC-256`

Exploitable cache timing vulnerability:	maybe
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
SPA/DPA Attack complexity:	low
Cost of Countermeasures:	high

3.4 LEX

The design of LEX is a variant of the AES, i.e., a cipher which follows the substitution-permutation structures. LEX uses the AES S-Box that is defined over GF(2^8). The LEX key schedule is identical to the AES key schedule. According to AES, LEX is defined for 128 bit key length (standard LEX) and for 192 and 256 bit key length (LEX-192 and LEX-256, respectively). IV setup is done by encrypting an 128-bit IV with a single AES invocation: $S = AES_K(IV)$. The 128-bit result S and the key K constitute the secret state of the cipher. The result S is used as plaintext for the first LEX operation in key stream generation. In keystream operation, LEX outputs four bytes of the evolving internal state in each AES round and the resulting ciphertext is fed in as the plaintext for the next LEX operation.

The building blocks of LEX are the substitution box and XOR. LEX is suited for both 8-bit and 32-bit software implementations. For 32-bit implementations, the AES operations SubBytes(), ShiftRows() and MixColumns() can be combined in four tables of 2^8 32-bit entries as done in the reference implementation.

The conditional dependent operation xtime() of the AES standard is avoided this way. Note that in the reference implementation, bit shifts are applied to isolate an input byte to the pre-computed tables. Bit shifts are extremely vulnerable to SPA attacks and compromise the operand bit-by-bit if applied successively. Due to the use of pre-computed tables in efficient software implementations, LEX is susceptible to cache based timing attacks.

In key schedule, the Hamming weight of portions of the key can be determined by a side channel adversary. In 32-bit operations, the entropy loss of the secret key is about 3.55 bit per 32-bit block. Note that one 32-bit LEX key word need to be fed into the AES S-Box for the round key derivation. Because of this we assume that each 8-bit input to the pre-computed table can be intercepted in portions of 8-bit leading to an entropy loss of about $4 \cdot 2.54 = 10.16$ for this 32-bit block. In total, entropy loss sums up to $3 \cdot 3.55 + 10.16 = 20.81$. This leakage can be avoided by applying a standard AES masking scheme for the key derivation.

In IV setup, the situation is identical to a DPA attack on the AES encryption provided that several IVs are known. DPA requires 2^8 key hypotheses per 8-bit subkey and is assumed to be successful to expose the entire key after applying DPA to all sixteen subkeys of the first AES round on any unmasked implementation. The DPA attack is assessed to be simple and straight-forward. Accordingly, mounting a template attack should succeed with 9 templates on the Hamming weight at the 8-bit S-Box output. Further, it is assumed that these templates are recyclable to recover all sixteen subkeys. A standard AES masking scheme for the IV setup should help to prevent DPA and templates attacks.

In keystream generation, four bytes of each AES round constitute the output stream. A side channel adversary can collect the corresponding output bytes of the same location in subsequent AES invocations that belong to the same byte of a roundkey. The four bytes of the last AES round can be used to mount a DPA attack after the first AddRoundKey() of the subsequent AES. For all output positions, a selection function can target the internal state before the last AddRoundKey() operation using an XOR selection function. This DPA attack may result in an entropy loss of 32 for each AES roundkey.

Table 6. Summary of (theoretical) side channel susceptibility for `LEX`.

Exploitable cache timing vulnerability:	yes
Exploitable conditional branches vulnerability:	maybe (operation xtime())
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	low
Cost of Countermeasures:	medium

3.5 NLSv2

NLS stands for Non-Linear SOBER and originates from the SOBER family of stream ciphers. NLSv2 also provides message authentication functionality, however, the authentication part of NLSv2 is not considered in phase 3 of eSTREAM. NLSv2 is constructed from a non-linear feedback shift register (NLFSR) and a non-linear filter (NLF). The internal state σ consists of 544 bits and is stored in seventeen 32-bit registers $r[0], \dots, r[16]$ of the NLFSR. Basic operations of NLSv2 are addition modulo 2^{32}, XOR of 32-bit words, rotations, and an S-Box mapping eight bit input to 32 bit output.

NLSv2 does not contain any conditional branches. The use of a table look-up is a potential vulnerability towards cache timing attacks. Other operations which may cause a data dependent execution time do not exist. The use of rotations is known to be susceptible to SPA attacks in the Hamming distance model.

Key setup repeats the following sequence until all key words are loaded: (i) modular addition of the next key word to register $r[15]$, (ii) clocking of the NLFSR, and (iii) XOR of the NLF output to register $r[4]$. Afterwards, step (ii) and (iii) are repeated seventeen times. Power analysis might recover the Hamming weight of 32-bit intermediate results as well as the Hamming weight of the 8-bit input and the 32-bit output of the S-Box that is part of NLF.

IV setup works in exactly the same way as key setup when using IV words instead of key words and offers the starting point for a DPA attack to recover the internal state after key set-up. The DPA selection functions are mounted on the 32-bit operations modular addition and XOR. Note that the number of key hypotheses is usually much smaller than 2^{32}, i.e., the 32-bit intermediate results are analyzed in parts of one or a few unknown bits. Note further that DPA on boolean and arithmetic operations usually requires that the known data part is randomly distributed. The use of a counter for the IV would significantly increase DPA efforts in a known-IV attack.

In more detail, the first DPA selection function is mounted on the modular addition of the first IV word and $r[15]$ aiming at recovering $r[15]$ of the internal state after key set-up. Clocking the NLFSR includes the computation of the non-linear feedback function which consists of a modular addition of $r[0]$ and $r[15]$, the S-Box lookup, and the XOR of the S-Box output and $r[4]$. DPA selection functions in the non-linear feedback function are mounted on the modular addition of $r[0]$ and the XOR operation with $r[4]$. Let denote the contents of the registers after clocking as $r_1[0] = r[1]$, $r_1[1] = r[2]$, $\dots$, $r_1[15] = r[16]$, $r_1[16] = t$ where t is the output of the non-linear feedback function. The NLF uses several registers, namely $\mathrm{NLF}(\sigma_1) = (r_1[0]+r_1[16]) \oplus (r_1[1] + r_1[13]) \oplus (r_1[6] + Konst)$. Except for $r_1[16]$, all other registers are unknown. It might be possible that DPA first recovers $r_1[0]$, then the sum of the other involved registers, and finally the value of $r_1[4]$ that is XOR-ed with the NLF output. The NLF is the crucial step for DPA. If it succeeds a DPA on the overall internal state after key set-up seems to be possible.

Implementing a masking scheme for NLSv2 is assessed to be expensive because the algorithm uses boolean operations, arithmetic operations, and an S-Box table.

Table 7. Summary of (theoretical) side channel susceptibility for `NLS`

Exploitable cache timing vulnerability:	yes
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	high
Cost of Countermeasures:	high

3.6 Rabbit

Rabbit is a synchronous stream cipher that uses a 128-bit key and a 64-bit initialization vector. The size of the internal state is 513 bits divided between eight 32-bit state variables $x_0, \ldots, x_7$, eight 32-bit counters $c_1, \ldots, c_7$ and one counter carry bit ϕ. We call an *iteration*, the update of the counters done using a `counter system` followed by the update of the state variables done using the core function of Rabbit named `Next-State` function. The design of Rabbit does not include table look-ups and there are no conditional branches depending on the internal state, therefore classical forms of cache-timing attacks and timing attacks are infeasible.

The *key setup algorithm* expands the secret key into the 513-bit internal state. The last internal state of this algorithm is denoted by $S = (x_{0,4}, \ldots, x_{7,4}, c_{0,4}, \ldots, c_{7,4}, \phi_{7,4})$. Next, the *IV Setup algorithm* starts by XORing the 64-bit IV on all the 256 bits of the counter state, and next it runs four iterations. Then, the generation of the keystream can start.

During the key setup, the Hamming weight of portions of the key can be determined by side channel. The key is divided into eight subkeys of 16 bits and each subkey is loaded into the internal state. The entropy loss is about 3.05 per 16-bit block. Thus, the entropy loss is at least $8 \times 3.05 = 24.4$. Note that the subkeys are next manipulated under various forms, e.g. the next operation after the state initialization is an addition of a 32-bit key with a known constant a_0 which also leaks additional information.

The aim of the proposed DPA with chosen IVs is to recover the initial value of the full state at the beginning of the IV setup algorithm by recovering first $(c_{0,4}, \ldots, c_{7,4})$ and second $(x_{0,4}, \ldots, x_{7,4})$. Note that this initial state does not change between two encryptions. The first step of the IV setup consists in modifying the counter state by XORing the 64-bit IV on all the 256 bits of the counter state. The XOR operation has two inputs of 32 bits $c_{i,4}$ and $\widetilde{IV}_i$ where $c_{i,4}$ is secret and constant, and $\widetilde{IV}_i$ is a subpart of the public initialization value IV and the attacker can control it. For every i, the attacker will guess bit by bit the value of $c_{i,4}$. Note that once the attacker observes sufficient number of the power consumptions, he does not have to re-observe them for another target bit. Due to the last operation of the key setup scheme, it is not possible to recover the key by inversion of the counter system. However, the adversary can compute all the next values of the counter state. For the second part, the attacker performs a DPA on the addition used in the function $g_{j,i}$. The modular addition has two inputs of 32 bits $x_{i,4}$ and $c_{i,5}$ where $x_{i,4}$ is secret and constant and $c_{i,5}$ is known from the attacker. For every i, the attacker will guess, bit by bit from the least significant bit to the most significant bit, the value of $x_{i,4}$.

A possible countermeasure to protect Rabbit could be to use a data masking scheme that will probably result in a large increase execution time due to the mix of arithmetic and boolean operations.

Table 8. Summary of (theoretical) side channel susceptibility for `Rabbit`

Exploitable cache timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium
Cost of Countermeasures:	high

3.7 Salsa20

The Salsa20 encryption function is an application of the Salsa20 hash function in counter mode. Salsa20 contains neither an explicit key schedule nor an IV setup. It is defined for 128 bit and 256 bit key length. The keystream is generated by hashing a 64-byte sequence that is a concatenation of 16 constant bytes, $2 \cdot 16$ or 32 key bytes, an 8 byte nonce, and an 8 byte block counter. The resulting hash value (keystream) is 64 byte in length and XORed with the corresponding 64 byte plaintext block to encrypt.

The Salsa20 hash function is a long chain of three simple operations: 32-bit addition, 32-bit XOR, and constant-distance 32-bit rotation. It is suited for both 8-bit and 32-bit software implementations as well as hardware implementations. Due to the simplicity of the core operations there is no incentive for software authors to use table lookups. Therefore software implementations can be considered secure against cache timing attacks. The elementary operations should execute in data-independent time on usual platforms and should resist timing attacks.

The Salsa20 hash function makes heavy use of 32-bit rotations in its core quaterround function. Depending on the architecture and in particular on small, say 8-bit, devices, a rotation might be performed as a bit-shift with additional buffer. Bit-shifts are vulnerable to Simple Power Analysis and may compromise the operand bit-by-bit. During key setup the key material is copied into arrays. In the given power model each 32-bit part of the key leaks 3.55 bits, so $4 \cdot 3.55 = 14.2$ bits for 128 bit keys and $8 \cdot 3.55 = 28.4$ bits for 256 bit keys.

In the following we assume a key length of 128 bit and focus on a 32-bit software implementation. Further we assume that the adversary knows the varying nonces (IVs) and note that control over the nonces very likely eases the attack. The entire key can be recovered in four portions of 32 bit by a differential side channel attack. In the first iteration of the round function, the state consists of 128 constant bits (known), 64 counter bits (known and varying), 64 nonce bits (known and varying), and 128 constant secret key bits. The first call of doubleround eventually leads to the first call of columnround, which separately processes four 128 bit chunks of the state with the quaterround function. During the last execution of the quaterround function, an adversary can recover 64 bits of the secret key. First the computation of z_2 is attacked to recover the value of z_1. Now the adversary knows the constant z_1 and can compute the varying z_2. Second the computation of z_3 is attacked to recover the 32 key bits stored in y_3. Given z_1, y_3, and the constant y_0 the adversary computes $y_1 = z_1 \oplus ((y_0 + y_3) <<< 7)$. This attack requires at most two times 2^{32} hypotheses, but can be rendered more practical using the divide-and-conquer principle. A slightly modified attack against the penultimate execution of the quaterround function reveals the other half of the key. Note that, since the keystream is assumed to be known, similar attacks can be mounted against the last round of Salsa20. Overall, the DPA attack is assessed to be non-trivial but completely feasible. A template attack should succeed with 33 templates on the Hamming weight of a value transferred to a register (or memory). Software implementations of Salsa20 can be protected against first-order DPA and basic template attacks by the masking countermeasure, but the effort is assessed to be high due to the mix of boolean and arithmetic operations.

Table 9. Summary of (theoretical) side channel susceptibility for `Salsa20`

Exploitable cache timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	low
Cost of Countermeasures:	high

3.8 SOSEMANUK

SOSEMANUK is a software-oriented stream cipher, which builds upon the block cipher SERPENT and uses design principles from the stream cipher SNOW. More precisely, the block cipher SERPENT is used for the key schedule and the IV injection. The stream cipher SNOW has inspired the design of the LFSR and the FSM. SOSEMANUK operates as follows. First, the initialization procedure, which involves the key schedule and the IV injection is computed. Second, the values from the IV injection are used to initialize the internal state of SOSEMANUK, and based on this the actual key stream can be generated.

The key schedule is independent of the IV and produces 25 128-bit subkeys using the SERPENT key schedule. The IV injection uses the IV as input to the SERPENT block cipher. In both, the key schedule and the IV injection, SERPENT is run with 24 rounds only. The IV injection delivers the outputs of the reduced version of SERPENT of the 12th, 18th, and 24th round as final output.

In the scenario of DPA attacks, the IV injection is the major step of interest, as DPA attacks make the assumption that the attacker can execute an algorithm with a fixed key using variable input data. Hence, in order to attack SOSEMANUK using DPA, we assume that the attacker knows the IVs and can run the IV injection a number of times (known plaintext attack). In case the attacker knows the output of the the IV injection also known ciphertext attacks would be possible. It is however unclear if such an assumption is realistic. It is possible that DPA attacks can be mounted also during the key stream generation. However, DPA attacks work best when applied to results of highly non-linear operations, and hence the S-Box operations in SERPENT are probably the easiest to attack.

To the best of our knowledge, there is no in-depth investigation of the susceptibility of SERPENT against DPA attacks available. Hence, we can only make general statements (that would apply to any modern block cipher) about DPA attacks. It is well known that software implementations of block ciphers, unless specifically protected, are vulnerable to DPA attacks. In particular, attacks after a non-linear operation (such as the SERPENT S-Boxes) provide optimal conditions for a DPA attack to work. Consequently, unless the IV injection of SOSEMANUK is protected, this part is likely to succumb to standard DPA attacks. Using standard DPA attacks, the 25 SERPENT sub-keys can be revealed and hence the cipher key can be reconstructed. The complexity of DPA attacks can be characterised in different ways. For instance, one way of comparing DPA attacks would be the number of key bits that must be guessed at once. In the case of attacking intermediate values after the SERPENT S-Boxes, which are 4-bit permutations, this means that an attacker must guess at least 4 key bits simultaneously.

Under different assumptions, and depending on the actual implementation, other parts of SOSEMANUK could be vulnerable to SPA and template attacks.

For timing and cache attacks, no general statements can be made either. Too much depends on the actual implementation. However, SERPENT has been designed such that it is particularly suitable for bit-slice implementations. This type of implementation has shown to be more resistant to timing and cache attacks than other types of implementations. Hence, unless another timing leakage is introduced by careless programming, the SERPENT part should be resistant against different types of timing attacks.

Masking is a standard countermeasure to secure software implementations against first-order DPA attacks. It can certainly be applied to SERPENT, and to the key stream generation. Previous work on masking software implementations of block ciphers has shown that masking increases the code size and the memory requirements of standard implementations of block ciphers in software. It is however unclear how expensive it is to mask bit-sliced implementations.

Table 10. Summary of (theoretical) side channel susceptibility for `SOSEMANUK`

Exploitable cache timing vulnerability:	maybe
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	low
Cost of Countermeasures:	high

4 Phase 3 candidates profile 2 (HW)

In this section we focus on the profile 2 candidates mentioned in Table 1.

4.1 DECIM^{v2}

DECIM^{v2} is a hardware-oriented stream cipher which takes as an input a 80-bit secret key and a 64-bit IV. The building blocks of DECIM^{v2} are an LFSR of length 192 over $\mathbf{F}_2$, a 14-variable Boolean filter function f, a decimation process called ABSG algorithm, and a 32-bit output buffer. In key and IV setup, the LFSR is initially filled with the 80 key bits K_i, the 64 bits $K_i \oplus IV_i$, 16 bits $K_{i+64} \oplus IV_i \oplus IV_{i+16} \oplus IV_{i+32} \oplus IV_{i+48}$ and 32 bits $IV_i \oplus IV_{i+32} \oplus 1$. Afterwards the LFSR is clocked 768 times by using a nonlinear feedback function, i.e., the feedback bit is $lv_t \oplus y_t$ wherein lv_t is the linear feedback value and y_t denotes the output of f at time t. In key stream operation, the output of the function f is switched to the ABSG mechanism and the LFSR is updated with the linear feedback value lv_t. As the rate of the ABSG mechanism is irregular, a buffer of length 32 is introduced to guarantee a constant throughput of one bit for every four clock cycles. The output of the keystream generation starts when the buffer is full.

The number of clock cycles until the buffer is initially filled depends on the internal state of the cipher and may offer a starting point for a timing attack. A further timing delay occurs with probability less than 2^{-89} if the buffer becomes empty in keystream generation. However, for side channel analysis such a small probability is not of practical relevance.

In key setup, 80 key bits are loaded into the LFSR. In a standard bit-serialized implementation, SPA is assumed to succeed in recovering all key bits by observing the first 80 clock cycles of the key set-up. Thereby, the adversary measures the Hamming distance of subsequent key bits. Similarly, a DPA attack on the loading of $K_i \oplus IV_i$ and $K_{i+64} \oplus IV_i \oplus IV_{i+16} \oplus IV_{i+32} \oplus IV_{i+48}$ is assumed to succeed in recovering the key bits in IV setup. These attacks succeed easily if all flip-flops of the LFSR are reset to a definite state before starting key setup. Avoiding a reset of the flip-flops, i.e., leaving the LFSR in an unknown state before key and IV setup may be one approach to prevent these kinds of attacks.

As the IV is assumed to be known, 32 bits of the LFSR are known after initial filling. An expand-and-prune DPA attack may be feasible on the start of the update of the LFSR state. The selection function is the feedback bit. The observed leakage is the change of the Hamming distance in the LFSR. For the first DPA iteration it requires a 20-bit key hypothesis (nine bit input to filter function f and eleven bit input to the taps of the LFSR are unknown). On the other hand, five input bits to the filter function and three bits at the feedback taps are known. As the number of key hypotheses is beyond the number of known bits, we assume that DPA withdraws only a certain fraction of key hypotheses in the first iteration. In the second iteration, the unknown key space is increased by 13 bits, in the third iteration by 11 bits, in the fourth iteration by nine bits. From the fifth iteration on, the increase of the key space becomes less than the number of known and predictable bits. Assuming that on average the key space divides by 2^8 per DPA iteration, the maximum number of key hypotheses would be around 2^{29} in the fourth iteration which is by far more expensive than common DPA attacks on block ciphers. As this attack has not even been simulated yet, it is hard to say whether these considerations can lead to an overall key exposure in practice. Choosing of IVs may facilitate an alternative approach that consists of building a set of equations for the unknown key bits by measuring the change of the Hamming distance in the LFSR depending on known IV bits.

In key stream generation, a further point of SPA attack is in the ABSG algorithm whereat the adversary may observe whether or not the ABSG algorithm outputs one bit. Alternatively, templates may be built on this decision process. The number of clock cycles is directly related to the runs of a bit sequence in y_t, i.e., one observes sequences of the form $(\bar{b}, b^i, \bar{b})$ with $i \geq 0$, $b \in \{0, 1\}$ and $\bar{b}$ is the complement of b. The linear complexity of the stream y_t is at most 18528 as f is a quadratic Boolean function. Provided that SPA has recovered 37056 successive bits of y_t, the Berlekamp-Massey algorithm determines an LFSR that generates the key stream. The number of decision processes to be observed is high. This typically requires the ability to reset the stream cipher with the same IV multiple times to reduce the error rate.

Table 11. Summary of (theoretical) side channel susceptibility for `DECIM`.

Exploitable timing vulnerability:	maybe
Exploitable conditional branches vulnerability:	yes (ABSG algorithm)
Exploitable HW leakage of data, SPA vulnerability:	yes (key setup)
Exploitable DPA vulnerability:	yes (IV setup)
DPA Attack complexity:	high

4.2 F-FCSR

A Feedback with Carry Shift Register (FCSR) can be considered as a cyclic right shift register where serial full adders are placed between the feedback signal and selected cells. The position of the full adders depends on the automaton main parameter. In order to extract one word from the FCSR cells in each clock cycle, a filter is selected to construct Filtered FCSR. The filter is identified by a fixed value (with the same length as the FCSR), and the output word bits are obtained by computing the weight parity of parts of the bitwise AND of the FCSR cells and the fixed value. Two different sets of parameters are proposed:

– F-FCSR-H that uses 160 cells, keys of length 80, and an IV of a bitsize between 32 and 80. It generates 1-byte words in each clock cycle.
– F-FCSR-16 that uses 256 cells, keys of length 128, and an IV of bitsize v with $64 \leq v \leq 128$. The output of the filter is 2-byte words in each clock cycle.

The key setup and the IV setup of F-FCSR can comprise a single function. Next to clearing the carry register of the full adders and filling the FCSR cells with the key and the zero padded IV, a certain number of rounds (20 times for F-FCSR-H and 16 times for F-FCSR-16) is iterated to obtain new values for the FCSR cells. Afterwards the FCSR cells are reloaded with the new values, and the carry registers are cleared again. The FCSR is then clocked 162 times for F-FCSR-H and 258 times for F-FCSR-16 ignoring the output.

Whereas 162 and 258 rounds are performed for the IV setup of F-FCSR-H and F-FCSR-16 profiles respectively, extracting the internal states of the encryption process by studying power consumption values does not lead to reveal the secret key bits straightforwardly. Therefore, the initial loops of the IV setup (20 rounds for F-FCSR-H and 16 rounds for F-FCSR-16) are taken into account to evaluate the vulnerability of F-FCSR implementations to SPA and DPA attacks.

There are three fundamental operations which can be considered by a side channel adversary:

1. Updating the FCSR cells (storage the new state of FCSR in its register cells).
2. Updating the carry registers.
3. Computing the weight parity of the FCSR cells selected by the filter.

All of these operations are simultaneously performed in every clock cycle. We propose a correlation DPA attack in two phases (F-FCSR-H is taken into account):

In the first phase, Hamming distance of the 80 most significant bits of FCSR cells is selected as the power consumption model. The 20 least significant bits of the secret key, $K_0, K_1, \ldots, K_{19}$, are revealed using the power traces of the 20-round initial loop of the IV setup and different IVs. One bit of the secret key (K_i at the ith round) is guessed (and is recovered) in each round of attack.

In the second phase, the attack consists of 60 rounds, and each round uses the power values of all 20 clock cycles. The Hamming distance of the $(80 + i + j)$ most significant cells plus the Hamming distance of the $(20 - j)$ least significant cells of the FCSR constructs the hypothetical power model of the jth clock cycle at round i. Note that the 20 first feedback bits revealed at the first phase are used in this phase to compute the Hamming distances. The secret information which is guessed and is recovered at the ith round is K_{80-i}. In fact, in the both phases there are two key hypotheses in each round. Note that the attack described above can be easily adopted for F-FCSR-16 in 16 and 112 rounds at the first and the second phase respectively. Moreover, the accuracy of the power models can be improved by consideration of Hamming distance of the carry registers that are known by the adversary.

To the best of our knowledge there is no masking scheme for this automaton to counteract DPA attacks. In order to design a masking scheme, a boolean-masked full adder should be designed, and a strategy to use and update mask bits must be defined.

Table 12. Summary of (theoretical) side channel susceptibility for `F-FCSR`

Exploitable timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes (key setup)
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium

4.3 Grain^{v1}

Grain^{v1} is a hardware-oriented stream cipher which is based on two shift registers (one LFSR and one NLFSR) and a nonlinear output function. It is a bit oriented synchronous stream cipher, where both shift registers are 80 bits. The key size is 80 bits and the IV size is 64 bits. The cipher consists of three main building blocks *i.e.* an LFSR and NLFSR of lengths 80 and a non-linear function $h : \mathbb{F}_2^5 \rightarrow \mathbb{F}_2$. The contents of two shift registers represents the state of the cipher and from this state, 5 variables are taken as input to the function h. Before a key stream is generated, the NLFSR is initially filled with the 80 key bits k_i and the first 64 bits of the LFSR are loaded with the IV and the remaining bits of the LFSR are filled with ones. Then the cipher is clocked 160 times by using the output function that is actually fed back and XOR-ed with the inputs to the LFSR and NLFSR without producing a key.

A straightforward hardware implementation [6] consists of 160 Flip-Flops (80 for each, LFSR and NLFSR), some combinatorial logic for realization of the three blocks and some additional XORs. The cipher does not include conditional branches and any reasonable implementation should be constant in time so timing attacks would not be possible.

As already stated above, in key setup, 80 key bits are loaded into the NLFSR. In a standard bit-serialized implementation, SPA could be possible *i.e.* an adversary should succeed in recovering all key bits by observing the first 80 clock cycles of the key set-up. Thereby, similar to DECIM, the adversary measures the Hamming distance of subsequent key bits. As mentioned above the attacks could be done easily if all flip-flops of the LFSR are reset before starting key setup.

A successful DPA attack on Grain was published by Fischer et al. [6]. It consists of three steps, where in the first two 34 and 16 bits are recovered respectively. The third step is an exhaustive search on the remaining 30 bits. The authors applied a chosen IV attack, which helped in eliminating the algorithmic noise. The idea is to attack the key setup and learn key bits iteratively as in each round the results of the previous ones are used. The first step takes 17 rounds and the second one only 16 but it is using the bits derived in the first round.

The same attack is applicable to known IV but the influence of noise might be substantial which could require a large number of samples. Similarly, DPA can be mounted on the update of the state of the cipher. In that case a hypothesis could be made on 6 key bits that are output of NLFSR. Another points of attacks could be before or after the output function. The observed leakage is based on the Hamming distance model. However, all these attacks would be less efficient comparing to the published attack. There might be a possibility to further optimize the known DPA attack by building templates.

Table 13. Summary of (theoretical) side channel susceptibility for `Grain`

Exploitable timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium

4.4 MICKEY 2.0

MICKEY 2.0 (which stands for Mutual Irregular Clocking KEYstream generator) is a hardware-oriented stream cipher which is aimed at resource-constrained platforms. It has two versions, MICKEY 2.0 uses an 80-bit secret key and MICKEY-128 2.0 is using a 128-bit key. We focus here on the version with an 80-bit key but the same analysis is applicable to the other one as well. It takes two input parameters: an 80-bit secret key and an IV, which can be between 0 and 80 bits is length, but it is not possible to use two IVs of different lengths with the same key. The building blocks of MICKEY 2.0 are two registers R and S, each of which is 100 stages long (each stage contains 1 bit). The register R is envisioned as “the linear register”, and S as “the non-linear” one. There exist also a version with a 128-bit key but the structure is similar, so we discuss the shorter key version here.

The clocking of R is done in two ways, depending on the control bit of R. When the bit is 0 the clocking of R is a standard LFSR clocking operation with a primitive polynomial of degree 100. When the bit is 1, each bit in the register is shifted to the right as well as XOR-ed into the current state.

The clocking of S is done by use of given four sequences of 100 bits each. Two of the sequences are used to derive an intermediate state and the other two are then multiplied with the feedback bit, one when the control bit of S is 0 and the other one otherwise. The generator is clocked on depending on both of the registers R and S.

The registers are initialized with all zeros. After that IV and key are loaded. Keystream bits are generated by XOR-ing the registers R and S.

The clocking of the overall generator includes some conditional branching that is based on one bit (so called $MIXING$). Therefore, there is a possibility for timing and SPA attacks. In key setup, 80 key bits are loaded into the LFSR. In a standard bit-serialized implementation, SPA is assumed to succeed in recovering all key bits by observing the first 80 clock cycles of the key set-up. This type of implementation is a common way to obtain a compact solution. In this case the adversary measures the Hamming distance of subsequent key bits. These attacks succeed easily because all flip-flops of the LFSR are reset with zeros before starting key setup.

Assuming known IV and the mixing bit, and the the following observation holds. One bit information from IV is used to clock the registers depending on the mixing bit and the bit of IV. The same holds for loading key bits. Both operations are done by use of operation $CLOCK_{KG}$. Therefore, it is possible to perform a DPA attacks that is targeting one bit at a time.

However, it seems to be quite easy to complicate a DPAT attack. The reason is in a possibility to parallelize a hardware implementation of the cipher, which can be considered as a countermeasure.

Table 14. Summary of (theoretical) side channel susceptibility for `MICKEY 2.0`

Exploitable timing vulnerability:	yes
Exploitable conditional branches vulnerability:	yes
Exploitable HW leakage of data, SPA vulnerability:	yes (key setup)
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium

4.5 Moustique

The cipher MOUSTIQUE is a self-synchronizing stream cipher using a key size of 96 bits and an IV size of 0 to 104 bits. The cipher consists of a so-called Conditional Complementing Shift Register (CCSR), followed by 7 pipelined stages. Since it is a self-synchronizing cipher, the key as well as one bit of ciphertext per round are fed into the CCSR during en- and decryption. During IV setup, the IV is fed into the CCSR instead of the ciphertext, to initialize the cipher state.

Output is suppressed during IV setup. One bit of key stream per clock cycle is produced during encryption phase.

The whole state of Moustique comprises 408 registers. Moustique uses three different bit updating functions with four bit inputs and a single bit output, making use of NAND and XOR gates. Due to the pipelined design and the efficient bit updating functions, the critical path delay is kept as low as 2 XOR gates. All registers are clocked each cycle. No timing analysis is possible, since no conditional branches depending on the internal states are used.

During key setup, the key is simply loaded into the registers. The loading can be done parallel or bit- or word-serial. Alternatively the key can be stored in some permanent registers, since it is not altered after being loaded once. Depending on the key loading method, a SPA attack might be possible, but this is no specific property of the cipher.

The IV is bitwise loaded into the CCSR during IV setup. At this time the key is fully loaded and the cipher is fully operational. The output is suppressed until the key is fully loaded.

An attack during the IV setup does not lead to an advantage compared to an attack during en- or decryption, because the attacker knows the input to the CCSR at all times, since it is the ciphertext. Furthermore, there is no obvious advantage by choosing certain IVs for an attack. Hence an attack during IV setup is the same as during encryption/decryption time.

The cipher has two properties that are very relevant for side channel analysis. The first one is the 408 register state, which is updated each round. Most of these state registers are unknown to an attacker and must be considered as noise. This will prevent SPA in all cases, except if the registers are all set to zero just after key setup. In this case, the key would leak its hamming weight during the first clock cycle of the IV setup. Yet, one could simply not initialize the registers to prevent this leakage. Hence we consider a SPA impossible. On the other hand, the cipher uses the full key in each round and never alters the key. Since the key is never altered, one can gather a huge amount of measurement data to recover the key, both during IV setup and encryption phase. The single bit input to the CCSR, q_0, is, as stated earlier, known to the attacker at all times.

A promising approach for performing a DPA is a divide and conquer attack beginning at the lowest key bit. For the CCSR bit updating functions, we have a known input (q_0, the feedback input for each clock cycle). The attacker can choose a number n of key bits k_0 to k_{n-1} he wants to predict and, together with the known input of the last n rounds predict n registers of the CCSR. A correlation attack can now be used to determine the correct key bits. The whole attack is repeated to recover the subsequent key bits $k_{(m-1)\cdot n}$ to $k_{m\cdot n-1}$, until the whole key is recovered. During the mth iteration, the attacker is able to predict $m \cdot n$ bits of the CCSR. Yet only a small portion of the registers of the cipher can be predicted by the attacker, especially for recovering the first few bits of the key. This leads to a high number of needed measurements to reduce the noise of the unpredicted registers.

Table 15. Summary of (theoretical) side channel susceptibility for `Moustique`

Exploitable timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes (key setup)
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium

4.6 Trivium

The design of Trivium is kept simple to allow for an efficient implementation in hardware. Three shift registers forming a 288-bit internal state $(s_1, s_2, \ldots, s_{288})$ are interconnected in a cyclic manner, so that the output of the last one, s_{288}, contributes to the input of the first one, s_1. In addition to a linear feedback into its own input by means of XOR gates, each shift register influences the following one in a non-linear fashion. This nonlinearity originates from ANDing the output of two adjacent flip-flops and feeding it via an XOR into the input of the successive shift register. 15 bits of the internal state are used to update three other bits of the state in each clock cycle. The key stream is produced by means of a three-input XOR gate that is connected to intermediate values from between the shift registers.

Prior to the initialization phase of the cipher, the shift registers are preloaded as follows. The first shift register, consisting of 93 flip-flops s_1 to s_{93}, is filled with the 80-bit key $(K_1, K_2, \ldots, K_{80})$ appended with thirteen zeros. The next shift register, holding 84 bits s_{94} to s_{177}, is similarly filled with the 80-bit IV in ascending order and four zeros. The third shift register is composed of 111 flip-flops s_{178} to s_{288} and contains only zeros except for the last three bits which are set to one. For the IV and key setup, the cipher is executed for $4 \cdot 288 = 1152$ clock cycles with no output. Afterwards, the output is activated for generating the key stream.

Trivium does not need any conditional branches depending on the internal state, nor does it employ S-Boxes or look-up tables. Therefore it is generally not vulnerable to timing attacks.

For describing the power consumption of this hardware-oriented cipher a discrete Hamming distance model is appropriate. Typically, it is assumed that each of the flip-flops of the shift registers behaves in the same way and toggling of its value produces a significantly higher power consumption than if it remains unchanged. The power consumed by the XOR and AND gates can usually be neglected.

In case of a bit-serial loading of the key and the IV into the shift registers, a straightforward SPA is very likely to be successful in recovering all key bits by taking into account at least 80 clock cycles, depending on the implementation. This standard SPA could be improved using templates in case of a very strong adversary that may pre-profile the cipher with known keys.

Moreover, a DPA can be carried out based on the input of the second shift register, s_{94}, that is influenced by key bits from the beginning of the initialization phase. Being connected to the output of an XOR that combines known bits with bits of the secret key, the content of this flip-flop after the ith clock cycle is

$$s_{94}(i+1) = s_{66}(i) \oplus s_{91}(i) \cdot s_{92}(i) \oplus s_{93}(i) \oplus s_{171}(i).$$

Initially, all of the above values are known except for s_{66} that is equal to the 66th bit of the key, K_{66}. For a generalized description of the attack, we will rewrite the equation with new variables, because s_i is not defined for negative values. Hence, for every clock cycle $1 \leq i \leq 93$, the adversary obtains one equation of the form

$$q_i = r_{67-i} \oplus r_{92-i} \cdot r_{93-i} \oplus r_{94-i},$$

where q_i is known due to the initialization as stated above and the r_j are derived from the following:

$$\begin{array}{ll} r_j = 0 & ; \quad j > 80 \\ r_j = K_j & ; 1 \leq j \leq 80 \\ r_j = s_{243}(i) \oplus s_{286}(i) \cdot s_{287}(i) \oplus s_{288}(i) \oplus K_{69+j} & ; \quad j \leq 0 \end{array}$$

From this approach, the adversary obtains an overdetermined system of 93 equations. Solving it will allow for recovering the full 80-bit secret key. Note that the cipher has to be executed 93 times, unless the attacker wants to brute force some values.

The signal-to-noise ratio can be decreased, if we assume a strong adversary that is allowed to select specific values for the IVs aiming at a minimized algorithmic noise of the cipher. This results in less traces being needed for obtaining the secret key [6].

As there is no obvious way to recover the internal state of the cipher after the initialization phase when the key is spread amongst the 288 bits of internal state, applying any kind of side channel attack at a later stage is not promising.

A straightforward hardware implementation of Trivium, as analyzed above, generates one bit of key stream per clock cycle. Trivium is designed such that no value of its internal state is re-used after its modification for at least 64 clock cycles. Hence, the cipher can be parallelized so that 64 iterations are carried out in one clock cycle and a 64-bit word is output. The increased algorithmic noise of such an implementation will make power analysis more difficult, i.e., more traces will be needed for recovering the secret key.

Table 16. Summary of (theoretical) side channel susceptibility for `Trivium`

Exploitable timing vulnerability:	no
Exploitable conditional branches vulnerability:	no
Exploitable HW leakage of data, SPA vulnerability:	yes (key setup)
Exploitable DPA vulnerability:	yes
DPA Attack complexity:	medium

5 Conclusions

In this section we summarize and assess the observations made in the previous sections in order to provide an overview of the phase 3 candidates with respect to their vulnerability to side channel attacks. Note, however, that the analysis of each cipher is done on an abstract level and is in particular not based on a concrete implementation. Anyway, our evaluation points out potential vulnerabilities and can be useful at the time of implementation. Further, we provide a first intuition about the cost of protecting an implementation.

Table 17 summarizes the analysis results for the software candidates. It seems that the criterion "exploitable (cache) timing vulnerability" is best suited to categorize the candidates. CryptMT, Rabbit, and Salsa20 appear to be immune to timing attacks, whereas Dragon, LEX, HC, NLS, and SOSEMANUK should be considered vulnerable due to the lookup tables.

Table 17. Summary of (theoretical) side channel susceptibility, profile 1 candidates

Cipher	Exploitable (cache) timing vulnerability	Exploitable conditional branches	Exploitable HW leakage of data	Exploitable DPA vulnerability	DPA attack complexity	Masking effort
CryptMT:	no	no	yes	yes	medium	high
Dragon:	yes	no	yes	yes	low	high
HC:	maybe	no	yes	yes	low	high
LEX:	yes	maybe	yes	yes	low	medium
NLS:	yes	no	yes	yes	high	high
Rabbit:	no	no	yes	yes	medium	high
Salsa20:	no	no	yes	yes	low	high
SOSEMANUK:	maybe	no	yes	yes	low	high

It is worth mentioning that LEX, because it is based on the AES, is the only candidate for which masking solutions exist. Another remark is, that for at least some of the software candidates, the use of a counter as the IV can significantly increase the number of measurements required for DPA attacks.

Table 18 summarizes the analysis results for the hardware candidates. Again, the ciphers are assessed to behave quite similar which makes a ranking difficult. However, it is worth noting that Decim and Mickey 2.0 are the only candidates with a potential timing issue and a vulnerability due to conditional branches.

Table 18. Summary of (theoretical) side channel susceptibility, profile 2 candidates

Cipher	Exploitable timing vulnerability	Exploitable conditional branches	Exploitable HW leakage of data	Exploitable DPA vulnerability	DPA attack complexity
Decim:	maybe	yes	yes	yes	high
F-FCSR:	no	no	yes	yes	medium
Grain:	no	no	yes	yes	medium
MICKEY:	yes	yes	yes	yes	medium
Moustique:	no	no	yes	yes	medium
Trivium:	no	no	yes	yes	medium

References

1. eSTREAM Phase 3 Candidates. eSTREAM, ECRYPT Stream Cipher Project. http://www.ecrypt.eu.org/stream/phase3list.html.
2. Daniel Bernstein. Cache-timing attacks against AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
3. Eli Biham and Adi Shamir. Power Analysis of the Key Scheduling of the AES Candidates. The Second Advanced Encryption Standard (AES) Candidate Conference, 1999. http://csrc.nist.gov/archive/aes/round1/conf2/papers/biham3.pdf.
4. T.M. Cover and J.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, 2006.
5. J. Daemen and V. Rijmen. Resistance against implementation attacks: A comparative study of the AES proposals. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, March 1999.
6. Wieland Fischer, Berndt M. Gammel, O. Kniffler, and J. Velten. Differential Power Analysis of Stream Ciphers. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 2007.
7. P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology: Proceedings of CRYPTO'96*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer-Verlag, 1996.
8. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology: Proceedings of CRYPTO'99*, number 1666 in Lecture Notes in Computer Science, pages 388–397. Springer-Verlag, 1999.
9. Elisabeth Oswald and Bart Preneel. A Theoretical Evaluation of some NESSIE Candidates regarding their Susceptibility towards Power Analysis Attacks. Public reports of the NESSIE project, Report No. NES/DOC/KUL/WP5/022/1, 2002. https://www.cosic.esat.kuleuven.be/nessie/reports/phase2/kulwp5-022-1.pdf.

Appendix

Let X be a random variable on a discrete space $\mathcal{X} = \{0, 1, \ldots, 255\}$ with probability distribution $\mathbb{P}_X = \left\{\frac{1}{256}, \ldots, \frac{1}{256}\right\}$, hence uniformly distributed. The entropy of X is

$$\mathsf{H}(X) = -\sum_{x \in \mathcal{X}} \mathbb{P}_X[X = x] \log_2 \mathbb{P}_X[X = x] = 8 \text{ [bits]}. \tag{1}$$

The Hamming Weight function, denoted by HW($\cdot$), maps a bit string x to the number of bits in the string $x = x_1 x_2 \cdots x_8$ which are set to '1' and obviously the resulting Hamming weight of a byte value is $\mathrm{HW}(x) = \sum_{i=1}^{8} x_i$ which implies $\mathrm{HW}(x) \in \{0, ..., 8\}$.

We let HW(X) be the random variable denoting the Hamming weight of the values x of X. If x is an 8-bit string, HW(X) takes values in $\mathcal{W} = \{0, 1, \ldots, 8\}$ with probability distribution

$$\mathbb{P}_{\mathrm{HW}(X)} = \left\{\tfrac{1}{256}, \tfrac{8}{256}, \tfrac{28}{256}, \tfrac{56}{256}, \tfrac{70}{256}, \tfrac{56}{256}, \tfrac{28}{256}, \tfrac{8}{256}, \tfrac{1}{256}\right\} \tag{2}$$

which can be found by evaluating the binomial coefficients $\binom{8}{w}$ for $w \in \mathcal{W}$. The entropy of $\mathrm{HW}(X)$ is

$$\begin{aligned} \mathsf{H}(\mathrm{HW}(X)) &= -\sum_{w \in \mathcal{W}} \mathbb{P}_{\mathrm{HW}(X)}[\mathrm{HW}(X) = w] \log_2 \mathbb{P}_{\mathrm{HW}(X)}[\mathrm{HW}(X) = w] \\ &= 2.5442 \text{ [bits]}. \end{aligned} \tag{3}$$

Similar computation can be performed for 16 and 32-bit variables.