The Self-Synchronizing Stream Cipher Moustique

Joan Daemen[1] and Paris Kitsos[2]

[1] STMicroelectronics Belgium, joan.daemen@st.com
[2] Hellenic Open University, Patras, Greece and Dept. of Computer Science and Technology, University of the Peloponnese, Tripoli, Greece, pkitsos@eap.gr

Abstract. We present a design approach for hardware-oriented self-synchronizing stream ciphers and illustrate it with a concrete design called Moustique. The latter is intended as a research cipher: it proves that the design approach can lead to concrete results and will serve as a target for cryptanalysis where new attacks may lead to improvements in the design approach such as new criteria for the cipher building blocks.

1 Introduction

This chapter is an abridged version of the two documents [6] and [8], both submitted to eSTREAM as documentation material for the ciphers Mosquito and its tweaked version Moustique. Most of the ideas were already presented in [3] and some of them even earlier in [2] as an alternative to the design approach as proposed by Ueli Maurer in [1]. We refer to [6] for a discussion on the latter design approach and alternative modes of operation of Mosquito (and similarly Moustique), such as using it as a MAC function or for authenticated encryption and synchronous stream encryption.

Single-bit self-synchronizing stream encryption has a unique advantage: in providing an existing communication system with encryption, it can be applied without the need for additional synchronization or segmentation. Actually, single-bit self-synchronizing stream encryption can be performed by using a block cipher in (single-bit) CFB mode. Still, we see two reasons for designing dedicated single-bit self-synchronizing stream ciphers.

First, the attainable encryption speed is a factor n_b slower than the encryption speed of the underlying block cipher implementation, with n_b the block length. For high-speed applications they may not be fast enough and a dedicated self-synchronizing stream cipher is required.

Second, a dedicated self-synchronizing stream cipher is a primitive different from both synchronous stream ciphers and block ciphers and is therefore theoretically interesting. Up to date, only a handful of dedicated self-synchronizing stream ciphers have been published and all except one (being the recent proposal Moustique) have been broken. In our opinion, only the availability of concrete targets for cryptanalysis may lead to a better insight in the design of self-synchronizing stream ciphers.

The following of this document is structured as follows. After introducing self-synchronizing stream encryption and its security properties in Section 2, we present the architecture underlying the design of MOUSTIQUE in Section 3. In Section 4 we specify MOUSTIQUE and we motivate the design choices in Section 5. Finally, Section 6 discusses the performance and resource usage of field programmable gate array implementations of MOUSTIQUE.

2 Self-synchronizing stream encryption

In this section we define self-synchronizing stream encryption, propose a pair of security claims and deduce from that some criteria for the cipher function that stem from differential and linear cryptanalysis.

2.1 Definition

In stream encryption operating at the bit level, each plaintext bit m^t is encrypted by adding a keystream bit z^t modulo two resulting in a ciphertext symbol c^t:

$$c^t = m^t \oplus z^t \ . \tag{1}$$

Decryption is:

$$m^t = c^t \oplus z^t \ . \tag{2}$$

In single-bit self-synchronizing stream encryption, the keystream symbol z^t is the result of applying a *cipher function* $\mathrm{f_c}$ to a window of the ciphertext stream with index range $[t - n_\mathrm{m}, t - (b_\mathrm{s} + 1)]$ and a *cipher key* K of n_k bits:

$$z^t = \mathrm{f_c}[K](c^{t-n_\mathrm{m}} \dots c^{t-(b_\mathrm{s}+1)}) \ . \tag{3}$$

n_m is called the *input memory* and we call b_s the *cipher function delay*. A block diagram of self-synchronizing stream encryption is given in Fig. 1.

For the encryption of the first n_m bits of the plaintext, there are no ciphertext bits available. The place of these bits are taken by an initialization vector that must be shared between sender and receiver and that may be public:

$$c^{-n_\mathrm{m}} \dots c^0 = \text{initialization vector (IV)} \ . \tag{4}$$

In general, encrypting a plaintext with a key using different IV values results in different ciphertexts. However, one should be careful. If the IV values only differ in the first ℓ bits, the probability that the two ciphertexts are equal is $2^{-\ell}$. Additionally, if the IV values only differ in the last $b_\mathrm{s} - \ell$ bits, the ℓ first bits of the ciphertext will be the same with certainty.

Despite their name, self-synchronizing stream ciphers are more similar to block ciphers than to synchronous stream ciphers, where the keyed cipher function takes the place of the keyed permutation in a block cipher. An attacker can query the output of the cipher function (keystream symbols) for chosen values of its input: a series of n_m ciphertext symbols. We call the latter an *input vector*.

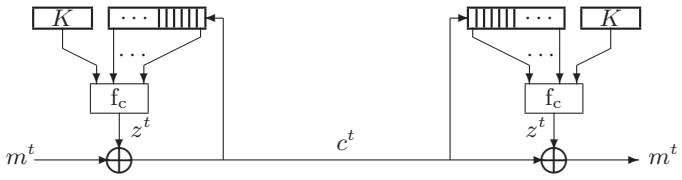

Fig. 1. Self-synchronizing stream encryption.

2.2 Security claims

The claimed security properties of a self-synchronizing stream cipher may be expressed in terms of its cipher function. In our opinion, the following two security claims are reasonable.

Claim 1 *The probability of success of an attack not involving key recovery, that guesses the output of the cipher function corresponding to ℓ input vectors C_i while given the cipher function output corresponding to any set of (adaptively) chosen input vectors not containing any of the C_i, is $2^{-\ell}$.*

Claim 2 *There are no key recovery attacks faster than exhaustive key search, i.e. with an expected complexity less than 2^{n_k} cipher function executions.*

Note that these claims do not include resistance against so-called related-key attacks. One may extend the claims to include related-key attacks. In our attack model, the attacker has no knowledge about the key whatsoever. It is the responsibility of the application developer to employ key management functions ensuring the adversary has no knowledge about the key. If the same key is used for encrypting different sequences and if one fears ciphertext collisions leaking information on the plaintext, one should use unique IV values to diversify the ciphertexts.

Additionally, these claims do not cover resistance against attackers that have access to (part of) the internal state or that can disrupt the proper operation of an implementation of the cipher function. While such attack scenarios may be realistic in the context of side-channel attacks, we do not consider that these problems should be tackled in the cipher design but rather in its implementation. For a discussion on how a hardware implementation of MOSQUITO (and likewise MOUSTIQUE) can be made with a high resistance against side channel attacks, we refer to [6].

2.3 Differential cryptanalysis

A class of attacks that can be very powerful when applied to self-synchronizing stream ciphers is differential cryptanalysis.

For every pair of n_m-bit (ciphertext) input vectors with a specific difference a', $\mathrm{f_c}$ returns a pair of keystream bits. The probability that the keystream bits

are different is denoted by $\mathrm{DP}(a', 1)$. The usability in differential cryptanalysis of $\mathrm{DP}(a', 1)$ is determined by its bias from 1/2. If this probability is $(1 \pm \ell^{-1})/2$, the number of input pairs needed to detect this bias is approximately ℓ^2.

Consequently, a cipher function should not have differentials with probabilities that deviate significantly more than $2^{-(n_\mathrm{m}-b_\mathrm{s})/2}$ from 1/2. The input differences a' with the highest biases should depend in a complex way on the cipher key.

Differential attacks can be generalized in several ways. One generalization that proved to be powerful in the cryptanalysis of some weak proprietary designs can be labeled as *second order* differential cryptanalysis. Here the inputs to the cipher function are applied in 4-tuples. The 4 inputs denoted by a_0, a_1, a_2 and a_3 have differences $a' = a_0 + a_1 = a_2 + a_3$ and $a'' = a_0 + a_2 = a_1 + a_3$. By examining the 4 corresponding output bits it can be observed whether complementing certain input bits (a'') affects the propagation of a difference (a'). This can be used to determine useful internal state bits or even key bits. Typically these attacks exploit properties very specific to the design under analysis. This can be generalized to even higher order DC in a straightforward way.

2.4 Linear cryptanalysis

The number of inputs needed to detect a correlation C of the keystream bit with a linear combination of input bits is C^{-2}. It follows that a cipher function should not have input-output correlations significantly larger than $2^{-(n_\mathrm{m}-b_\mathrm{s})/2}$. The selection vectors v_a with the highest correlations should depend in a complex way on the cipher key. By imposing a number ℓ of affine relations on the input bits, the cipher function is effectively converted to a Boolean function in $n_\mathrm{m} - b_\mathrm{s} - \ell$ variables. These functions should have no correlations significantly larger than $2^{-(n_\mathrm{m}-b_\mathrm{s}-\ell)/2}$ for any set of affine relations.

A special case of a selection vector is the zero vector. An output function that is correlated to the constant function is unbalanced. A correlation of C to the constant function gives rise to an information leakage of approximately $C^2/\ln 2$ bits per encrypted bit for $C < 2^{-2}$.

3 Cipher function architecture

We address the problem of realizing a cipher function providing high resistance against cryptanalysis and high speed in dedicated hardware by combining two structures: *pipelining* and *conditional complementing shift registers*.

3.1 Pipelining

We can realize a cipher function as a number of b_s *stages* G_i. In hardware, every stage can be implemented by a combinatorial circuit and a register storing the intermediate result. This pipelined approach is illustrated in Figure 2. As the encryption speed is limited by the critical path (largest occurring gate delay), the

stages should have small gate delay and hence be relatively simple. This approach impacts the general dependency relations of the self-synchronizing stream cipher: the implementation of the cipher function in b_s stages causes the keystream bit z^t to depend on the contents of the shift register b_s time steps ago. The pipelining increases the input memory n_m of the cipher by b_s symbols. However, the number of input symbols in the cipher function remains the same, as a keystream symbol z^t is independent of the ciphertext symbols c^{t-b_s} to c^{t-1}. Therefore we call the quantity b_s the *cipher function delay*.

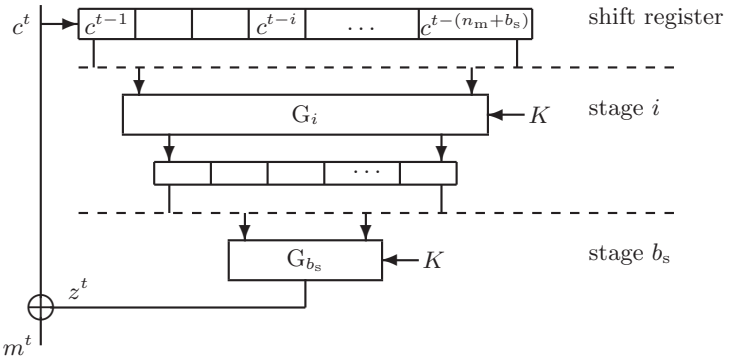

Fig. 2. Self-synchronizing stream cipher with a cipher function consisting of stages.

3.2 Machines with finite input memory

The input to the first stage of the pipelined structure consists of the last $n_\mathrm{m} - b_\mathrm{s}$ ciphertext bits, contained in a shift register. This construction guarantees that the keystream bit z^t only depends on the cipher key K and ciphertext bits c^{t-n_m} to $c^{t-(b_\mathrm{s}+1)}$.

Replacing the shift register by a finite state machine with finite input memory n_m *can* improve the propagation properties without violating this dependence restriction. If the gate delay of this finite state machine is not larger than the critical path, the maximum encryption speed of a hardware implementation is not impacted.

A finite state machine with finite input memory has specific propagation properties. Let q be the internal state and G the state-updating transformation. Then

$$q^{t+1} = \mathrm{G}(q^t, c^t) \ , \tag{5}$$

with c^t the ciphertext bit at time t.

One can associate with every component of the internal state q an input memory, i.e., the number of past ciphertext bits that it depends on. The internal

state, confined to the components with input memory j is denoted by q^j, with q_i^j its ith component. While not a part of the internal state, c can be considered as the component with input memory zero: q^0. The input memory of the finite state machine is equal to the largest occurring component input memory.

Clearly, q_i^j at time $t+1$ must be independent of all q^ℓ with $\ell \geq j$ at time t and *must* depend on q^{j-1} at time t. From this, it follows that the input memory partitions the components of the internal state into non-empty subsets with input memory 1 to $n_\mathrm{m} - b_\mathrm{s}$. The components of the state-updating transformation are of the form:

$$q_i^{j\,t+1} = \mathrm{G}[K]_i^j(c^t, q^{1\,t}, \ldots, q^{j-1\,t}) \ , \tag{6}$$

for $0 < j \leq n_\mathrm{m} - b_\mathrm{s}$.

3.3 Conditional complementing shift registers

An important potential problem in a finite state machine with finite memory is the existence of high-probability extinguishing differentials. An extinguishing differential is a difference in the (ciphertext) input vector leading to a zero difference in the internal state. This may lead to exploitable differentials in the cipher function. Assume that for the function corresponding with the stages $\mathrm{DP}(q', 0) = 1/2$ for all nonzero difference patterns q'. In that case if the CCSR has an extinguishing differential $(a, 0)$ with high probability $\mathrm{DP}(a, 0) = p$, the differential $(a, 0)$ in the cipher function will have a high bias from 1/2: $\mathrm{DP}(a, 0) \approx (1 + p)/2$. The existence of extinguishing differentials can be prevented by imposing (partial) linearity on the components of the state-updating transformation. For simplicity we impose the preliminary restriction that all q^j have only one component, i.e., that there is only one bit for every input memory value. The components of the state-updating transformations are of the form

$$q^{j\,t+1} = q^{j-1\,t} + \mathrm{E}[K]^j(q^{j-2^\mathrm{t}}, \ldots, q^{1\,t}, c^t) \ . \tag{7}$$

Since the new value of q^j is equal to the bitwise sum of the old value of q^{j-1} and some Boolean function, we call this type of finite state machine a *conditional complementing shift register* (CCSR).

A finite state machine with finite input memory ℓ realizes a mapping from a length-ℓ sequence of ciphertext bits $c^{t-\ell}, \ldots, c^{t-1}$ to an internal state q^t. For a CCSR we have the following result.

Proposition 1. *The mapping from $c^{t-\ell}, \ldots, c^{t-1}$ to the internal state q^t of a CCSR is an injection.*

Proof : We show how to reconstruct $c^t q^{1\,t} \ldots q^{j-1\,t}$ from $q^{1\,t+1} \ldots q^{j\,t+1}$. The components are reconstructed starting from c and finishing with q^{j-1}. For q^1 Equation (7) becomes

$$q^{1\,t+1} = q^{0\,t} + \mathrm{E}[K]^1 = c^t + \mathrm{E}[K]^1 \ ,$$

since $\mathrm{E}[K]^1()$ depends only on K. From this we can calculate c^t. The values of ${q^{k-1}}^t$ for k from 2 to j can be calculated iteratively from the previously found values by

$$ {q^{k-1}}^t = {q^k}^{t+1} + \mathrm{E}[K]^j({q^{k-2}}^t, \ldots, {q^1}^t, c^t) \ . $$

$c^{t-\ell} \ldots c^{t-1}$ can be calculated uniquely from ${q^1}^t \ldots {q^\ell}^t$ by iteratively applying the described algorithm. □

It follows that a nonzero difference in $c^{t-\ell} \ldots c^{t-1}$ must give rise to a nonzero difference in q^t. Therefore in a CCSR there are no extinguishing differentials between the input vector and its state.

The CCSR has the undesired property that a difference in $c^{-\ell-t}$ propagates to ${q^\ell}^t$ with a probability of 1. This can be avoided by "expanding" the high input memory end of the CCSR, i.e., taking more than a single state bit per input memory value near memory value ℓ.

3.4 The pipelined stages revisited

In our architecture, the cipher function consists of a CCSR followed by a number of pipelined stages. The stages are similar to the rounds in a block cipher but are less restricted.

A round of an iterated block cipher must be a permutation, and its inverse must be easily implementable. The stages do not have this restriction and the length of their outputs can be different from that of their inputs. The output of the last stage is a Boolean function of the components of the state q some cycles ago. An imbalance in this function leads to an imbalance in the cipher function. This Boolean function can be forced to be balanced by imposing that all the stage functions are *semi-invertible*. We call an n-bit to m-bit mapping $b = f(a)$ semi-invertible if there exists an n-bit to $(n-m)$-bit mapping $b' = f'(a)$ so that a is uniquely determined by the couple (b, b'). In that case the output bit may have figured as a component of the output of an invertible function of the state q.

The last round of an iterated block cipher must be followed by a key application or include a key dependence. This is necessary for preventing the cryptanalyst calculating an intermediate encryption state thereby making the last round useless. For the cipher function the calculation of intermediate values is impossible since only a single output bit z^t is given per input. Therefore, key dependence is not a strict requirement for the stage functions.

4 The Moustique cipher function

Moustique is a single-bit self-synchronizing stream cipher with:

- : Key size n_k: 96
- : Input memory n_m: 105
- : cipher function delay b_s: 9

4.1 The MOUSTIQUE internal state

MOUSTIQUE consists of a conditional complementing shift register (CCSR) and a number of pipelined stages. The MOUSTIQUE CCSR has 128 bits that are partitioned in 96 *cells* denoted by q^j. The index j ranges from 1 to 96. The number of bit per cells depends on the value of j and is denoted by n_j. The values of n_j are specified in Table 1.

Table 1. Number of bits per cell

Range of j	n_j
$1-88$	1
$89-92$	2
$93-94$	4
95	8
96	16

The bits within a cell q^j are denoted by q_i^j with $0 \leq i < n_j$. We index the bits of the CCSR in two ways: we use q_i^j in the specification of the updating function of the CCSR itself, and a_i^0 in the specification of of the updating function of the first stage. Figure 3 shows the expansion of the CCSR at the high input memory end and the two ways of indexing.

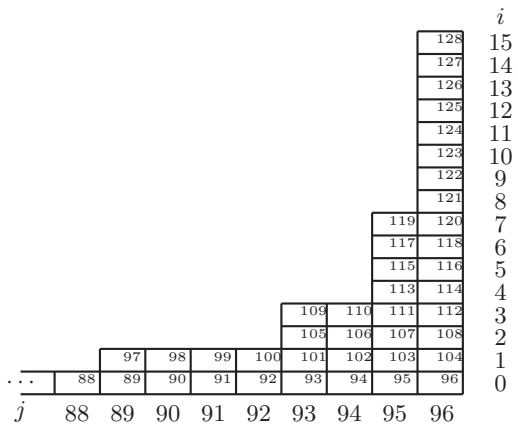

Fig. 3. Expansion of the CCSR in the high memory region. q indexing at the right and bottom, a^0 indexing inside the boxes.

The MOUSTIQUE internal state has 8 stage registers denoted by a^i, including the CCSR:

- a^0 is the CCSR and has a length of 128.

– a^1 to a^5 have length 53.
– a^6 has length 12.
– a^7 has length 3.

The bits of the registers a^1 to a^7 are indexed starting from 0, those of a^0 start from 1. The cipher key k consists of 96 bits: $k_0 \dots k_{95}$.

4.2 The MOUSTIQUE state updating function

For all bits in the internal state, the value of a bit at time t is a simple function of bits of the internal state, possibly a key bit and possibly the ciphertext bit at time $t-1$. We distinguish three Boolean functions, defined in terms of addition and multiplication in the field GF(2):

$$g_0(a,b,c,d) = a + b + c + d \tag{8}$$
$$g_1(a,b,c,d) = a + b + c(d+1) + 1 \tag{9}$$
$$g_2(a,b,c,d) = a(b+1) + c(d+1)\ . \tag{10}$$

Figure 4 gives combinatorial circuits of these functions.

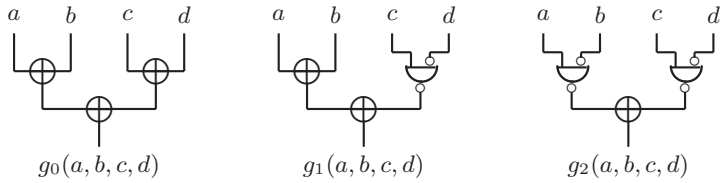

Fig. 4. The three functions used in the state-updating transformation

For the bits of the CCSR we have:

$$q_i^j \Leftarrow g_x(q_{i \bmod n_{j-1}}^{j-1}, k_{j-1}, q_{i \bmod n_v}^v, q_{i \bmod n_w}^w)\ , \tag{11}$$

with $0 \le v, w < j-1$. The values of x, v and w for all combinations (i,j) are specified in Table 2, except those for $j \le 2$ and those with $j = 96$ and $i > 1$. In this table a 0 in columns v or w denotes the bit at the input to the CCSR.

For $j \le 2$, the q^v and q^w entries are taken to be 0. The 15 bits q_i^{96} with $i > 0$ are specified by:

$$q_i^{96} \Leftarrow g_2(q_{i \bmod 8}^{95}, q_0^{95-i}, q_{i \bmod 4}^{94}, q_{1 \bmod n_{94-i}}^{94-i})\ . \tag{12}$$

The bit updating functions for the stages are specified in Table 3. In this table, if a lower index in the right-hand side of the equations is out of the specified range, the corresponding bit is taken to be 0, e.g., $a_{53}^3 = 0$.

Table 2. Function and v and w values for equation 11

Index	Function	v	w
$(j-i) \bmod 3 = 1$	g_0	$2(j-i-1)/3$	$j-2$
$(j-i) \bmod 3 = 2$	g_1	$j-4$	$j-2$
$(j-i) \bmod 6 = 3$	g_1	0	$j-2$
$(j-i) \bmod 6 = 0$	g_1	$j-5$	0

Table 3. Bit updating function for the stages

Output	Equation	Input
$a_i^1, 0 \le i < 53$	$a_{4i \bmod 53} \Leftarrow g_1(a_{128-i}, a_{i+18}, a_{113-i}, a_{i+1})$	$a_i^0, 1 \le i < 128$
$a_i^2, 0 \le i < 53$	$a_{4i \bmod 53} \Leftarrow g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^1, 0 \le i < 53$
$a_i^3, 0 \le i < 53$	$a_{4i \bmod 53} \Leftarrow g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^2, 0 \le i < 53$
$a_i^4, 0 \le i < 53$	$a_{4i \bmod 53} \Leftarrow g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^3, 0 \le i < 53$
$a_i^5, 0 \le i < 53$	$a_{4i \bmod 53} \Leftarrow g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^4, 0 \le i < 53$
$a_i^6, 0 \le i < 12$	$a_i \Leftarrow g_1(a_{4i}, a_{4i+3}, a_{4i+1}, a_{4i+2})$	$a_i^5, 0 \le i < 53$
$a_i^7, 0 \le i < 3$	$a_i \Leftarrow g_0(a_{4i}, a_{4i+1}, a_{4i+2}, a_{4i+3})$	$a_i^6, 0 \le i < 12$

The keystream bit is given by

$$z = a_0^7 + a_1^7 + a_2^7 \ . \tag{13}$$

This yields:

$$p \Leftarrow g_0(c, a_0^7, a_1^7, a_2^7) \ . \tag{14}$$

and

$$c \Leftarrow g_0(p, a_0^7, a_1^7, a_2^7) \ . \tag{15}$$

4.3 Putting it together

Figure 5 shows the MOUSTIQUE self-synchronizing stream cipher. Its critical path delay is 2 XOR gates, equal to the gate delay of the state-updating transformation. Building a circuit that can perform both encryption and decryption while maintaining this path delay necessitates the introduction of extra intermediate storage cells, denoted in Figure 5 by boxes containing a **d**. In the encryptor this cell is located between the encryption and the input of the CCSR. For correct decryption this necessitates a double delay at the input of the CCSR.

5 Design rationale

In this section we discuss the structure of the components in MOUSTIQUE. Actually, the design of MOUSTIQUE goes back to KNOT [2], that was improved to become $\Upsilon\Gamma$ [3]. We submitted $\Upsilon\Gamma$ to eSTREAM [5] under the name MOSQUITO[6].

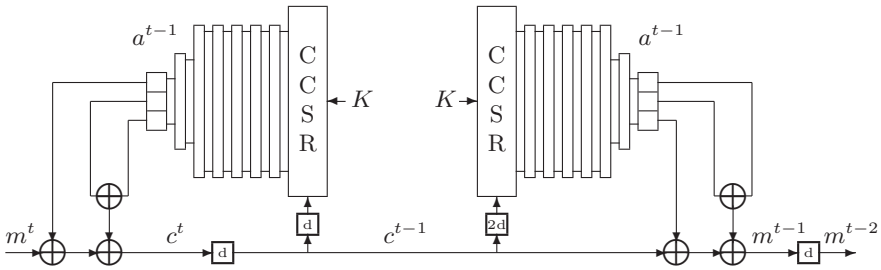

Fig. 5. Encryption and decryption with MOUSTIQUE.

$\Upsilon\Gamma$ and MOSQUITO have the same cipher function but the cipher function delay b_s has increased from 8 in $\Upsilon\Gamma$ to 9 in MOSQUITO. After MOSQUITO was broken in [7], we tweaked it and called the new version MOUSTIQUE.

5.1 The CCSR

The CCSR of MOUSTIQUE is a tweaked version of the one in MOSQUITO to address the attack in [7] that in turn is a tweaked version of the one in KNOT due to our discovery of extinguishing differentials.

The CCSR is designed to prevent differentials from $c^{t-96} \dots c^{t-1}$ to Q^t with a probability larger than 2^{-15}, while keeping the gate delay very small and the description simple. Observe that the 15 components G_i^{96} with $i > 0$ are unbalanced functions resulting in a bias in the corresponding components q_i^{96}.

In KNOT, the component G_0^{96} was also an imbalanced function, resulting in extinguishing differentials from the input vector to the CCSR state. For this reason, we replaced this component function from KNOT to $\Upsilon\Gamma$ by a balanced function. The extinguishing differentials in the CCSR of KNOT were later exploited to break it in [4].

In all versions of the CCSR, an input difference diffuses immediately to components all over q. This is a consequence of the fact that c^t is not only injected in q^1, but in many components at once. These are represented by the zero v and w entries in Table 2 (keep in mind that $q^0 = c$). For $j - i$ a multiple of 3, depending on the value of q_0^{j-2}, a difference in c propagates to either q_0^j or q_0^{j+3}. Since there are more than 15 of these "double injections", the probabilities are below 2^{-15}. In subsequent iterations this pattern is subject to the nonlinearity of the CCSR state-updating transformation.

In the CCSR of MOUSTIQUE, the components q_i^j with $j-i-1$ a multiple of 3 are updated according to the linear function g_0 while for the CCSR of MOSQUITO and KNOT, all components q_i^j with $j < 96$ used the nonlinear function g_1. This modification was due to the attack on MOSQUITO in [7] shortly described hereafter.

If the first ℓ bits of the key are (assumed to be) known, the propagation of a difference applied at the input can be controlled up to q^ℓ. The attacker can

apply a difference in the ciphertext that leads at time $t = 1$ to a difference equal to 1 in cell q^1 and 0 in cells q^2 to q^ℓ. He then iterates the CCSR ℓ times, while ensuring that the difference in q^1 at time $t = 1$ propagates to a difference in only q^i at time $t = i$, and nowhere else in the complete CCSR. Due to the fact that the worst-case diffusion inside the CCSR is very small, the attacker can easily enforce this by choosing the appropriate ciphertext bits. At time $t = \ell$, the difference in the cells q^1 to q^ℓ is a single 1 in q^ℓ and zero elsewhere.

Consider now the cells $q^{\ell+1}$ and higher. At time $t = 1$, the attacker has no knowledge of the difference in this section. However, at time $t = i$, the difference in cells $q^{\ell+1}$ up to $q^{\ell+i-1}$ is zero. If the input memory of the SSSC is 2ℓ or smaller, at time $t = \ell$ the difference in the CCSR is 1 in cell q^ℓ and zero elsewhere. The stages realise some confusion in the mapping from the CCSR state to the output bit, but clearly not sufficient to such a powerful differential. They have been designed assuming that an attacker cannot construct high probability differentials in the CCSR. The authors of [7] proved this assumption to be wrong and showed that guessing about half of the key and decrypting some chosen ciphertext pairs suffices to find the remaining part of the key, thereby breaking the cipher.

In the design of the CCSR of Mosquito care was taken to have high diffusion from its input bit to the cells by injecting the input bit in at least 15 positions. However, the attack exploits the low worst-case diffusion *within* the CCSR. Actually, the attack exploits this low diffusion in combination with two other properties of Mosquito: the insufficient confusion realised by the stages and the fact that guessing part of the cipher key gives access to the first part of the CCSR. A tweak should therefore address at least one of these three properties. In our choice of the tweak, we also considered that the efficiency of the cipher in dedicated hardware should not degrade too much: the area and the critical path delay should not change significantly with respect to Mosquito. This rules out the introduction of a key schedule, the augmentation of the number of stages or their width or an increase of the width of cells of the CCSR. Only the CCSR updating function remains.

The worst-case diffusion in the CCSR is dramatically improved by using for about one third of the bits the function g_0 instead of g_1. The indexing ensures that differences in the low-end part of the CCSR propagate much faster to differences in the high-end part of the CCSR. This makes containment of single-bit differences in the first cells of the CCSR to a small number of cells during a significant number of iterations infeasible. Therefore, we believe chosen-ciphertext key-guessing attacks as in [7] cannot be mounted for Moustique. Clearly, replacing the nonlinear function g_1 by the linear function g_0 for one third of the bits of the CCSR may introduce new weaknesses and possibly lead to new attacks. It remains to be seen whether there will appear attacks that manage to exploit this.

5.2 The pipelined stages

The input to the first stage consists of the state bits of the CCSR. Special care has been taken with respect to difference patterns restricted to the high-memory region and those resulting from a difference in the most recent cipher bit. The purpose of stages $\langle 1 \rangle$ to $\langle 6 \rangle$ is the elimination of low-weight linear and differential trails. The components of these stage functions combine diffusion, nonlinearity and dispersion respectively in the linear term, in the quadratic term and in the arrangement of inputs and outputs. Their effectiveness is reinforced by the diffusion in stage $\langle 7 \rangle$ and the output function that computes the keystream bit as the bitwise addition of all 12 bits of $a^{\langle 6 \rangle}$ to the output. During the writing of [3], we discovered that the output function of KNOT had a detectable imbalance. This problem was solved in $\Upsilon\Gamma$ by modifying the stages to be semi-invertible.

6 Hardware performance and implementation aspects

MOUSTIQUE has been designed with dedicated hardware implementations in mind and does not lend itself to software implementations at all. Therefore we only give performance results for dedicated hardware implementations.

We have implemented a MOUSTIQUE encryption/decryption circuit with gate delay of 2 XOR gates as described in [6] using Field Programmable Gate Array (FPGA). We designed and coded the hardware implementation in VHSIC Hardware Description Language (VHDL) with structural description logic and verified the resulting implementation using the Mentor Graphics ModelSim simulation environment, with test vectors returned by the software implementation. We synthesized the circuit using Mentor Graphics LeonardoSpectrum tool in both XILINX [9] and ALTERA [10] FPGAs.

The synthesis results and performance analysis are shown in Table 4 indicating the number of D Flip-Flops (DFFs), Configurable Logic Blocks (CLBs) and Function Generators (FGs) for XILINX FPGAs and the number of D Flip-Flops (DFFs) and Logic Cells (LCs) in cases of ALTERA FPGAs. The indicated throughput is that for encryption/decryption, after the initialization phase.

Table 4. MOUSTIQUE synthesis results and performance numbers

FPGA Device	# DFF		# FG/LC		# CLB		Speed
	total	used	total	used	total	used	Mb/sec
XILINX VIRTEX (V50BG256)	1536	503	1536	405	768	252	228
XILINX VIRTEX-E (V50EPQ240)	2010	503	1536	405	768	252	263
XILINX VIRTEX-II (2V80FG256)	1384	503	1024	405	512	252	369
ALTERA APEX (EP20K200RC208)	-	-	8320	503	-	-	336
ALTERA FLEX (EPF10K70RC240)	4096	503	3744	503	-	-	146
ALTERA MAX (EPM3512AQC208)	512	503	512	503	-	-	167

Almost in all the cases, both for XILINX and ALTERA, we used the smallest FPGA devices with low hardware resources utilization for each FPGA family. A circuit with fully parallel key loading has 103 I/Os, one with single-bit serial key loading has only 8 I/Os.

The experimental delay measurements (critical path delay, 1/Freq.) are very close to the expected values produced by the theoretical expression (critical path delay $= 2 * t_{XOR}$). The slight differences between the experimental and the theoretical values are due to the fact that in the theoretical values the FPGA internal interconnection wires delays, D flip flop or buffer transfer delays are not calculated. All in all the cipher achieves a low level of FPGA utilization and is suitable for hardware implementation. In [6] we have compared our implementations of MOSQUITO with that of block ciphers operating in single-bit CFB mode and show that they are an order of magnitude faster and more efficient.

7 Acknowledgements

We would like to thank Joe Lano for stimulating us to submit MOSQUITO to e-STREAM and Sanand Sule, Ralf-Philipp Weinmann and Sean O'Neal for reporting problems with the reference implementation in MOSQUITO and draft versions of MOUSTIQUE. Finally we would like to thank Frédéric Muller and Antoine Joux for doing the effort to cryptanalyze KNOT and MOSQUITO, which have led to MOUSTIQUE.

References

1. U.M. Maurer, "New Approaches to the Design of Self-Synchronizing Stream Ciphers," *Advances in Cryptology, Proc. Eurocrypt '91, LNCS 547*, D. Davies, Ed., Springer-Verlag 1991, pp. 458–471.
2. J. Daemen, R. Govaerts and J. Vandewalle, "On the Design of High Speed Self-Synchronizing Stream Ciphers," *Singapore ICCS/ISITA '92 Conference Proceedings* P.Y. Kam and O. Hirota, Eds., IEEE 1992, pp. 279–283.
3. J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," *Doctoral Dissertation*, March 1995, K.U.Leuven.
4. A. Joux and F. Muller, "Loosening the KNOT," *Fast Software Encryption 2003, LNCS 2887*, T. Johansson, ed., Springer-Verlag, 2003, pp. 87–99.
5. `http://www.ecrypt.eu.org/stream/`
6. J. Daemen and P. Kitsos, Submission to ECRYPT call for stream ciphers: the self-synchronizing stream cipher Mosquito: eSTREAM documentation, version 2, December 8, 2005. Available from `http://www.ecrypt.eu.org/stream/`
7. A. Joux and F. Muller, "Chosen-Ciphertext Attacks against MOSQUITO," *Fast Software Encryption 2006, LNCS 4047*, M. Robshaw, ed., Springer-Verlag, 2006, pp.390–404.
8. J. Daemen and P. Kitsos, Submission to ECRYPT call for stream ciphers: the self-synchronizing stream cipher Moustique, June 30, 2006. Available from `http://www.ecrypt.eu.org/stream/`
9. Xilinx Virtex FPGA Data Sheets (2005), URL: `http://www.xilinx.com`
10. Altera FPGA Data Sheets (2005), URL: `http://www.altera.com`