Open access • Book Chapter • DOI:10.1007/978-3-540-68351-3_16

The Self-synchronizing Stream Cipher Moustique — Source link

Joan Daemen, Paris Kitsos

Institutions: STMicroelectronics, University of Peloponnese

Published on: 01 Apr 2008

Topics: Stream cipher, Stream cipher attack, Cipher, Block cipher and Transposition cipher

The self-synchronizing stream cipher MOUSTIQUE

Joan Daemen, STMicroelectronics Belgium, joan.daemen@st.com
Paris Kitsos, Hellenic Open University, Patras, Greece, pkitsos@eap.gr

June 30, 2006

Abstract

In this note we specify the hardware-oriented self-synchronizing stream cipher MOUSTIQUE, a tweaked version of MOSQUITO that was submitted to the eSTREAM project and broken in [2]. We motivate the modifications with respect to MOSQUITO[1].

1 Introduction

Recently, the self-synchronizing stream cipher MOSQUITO was broken with a chosen-ciphertext attack in [2]. In this note we specify MOUSTIQUE, a tweaked version of MOSQUITO, or MOSQUITOV2 if you want. In this note we only specify the cipher, motivate the tweak with respect to MOSQUITO and give some incremental implementation results. For an explanation of the general design approach and more implementations results, we refer to [1].

2 Self-synchronizing stream encryption

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 self-synchronizing stream encryption with symbol size $n_\mathrm{s} = 1$, 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*. 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, taking different IV values may result in the same ciphertext values.

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

More particularly, 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 $10-\ell$ bits, the ℓ leading bits of the ciphertext will be the same with certainty.

3 The MOUSTIQUE cipher function

MOUSTIQUE is a self-synchronizing stream cipher with:

- : Symbol size n_{s}: 1
- : Key size n_{k}: 96
- : Input memory n_{m}: 105
- : cipher function delay b_{s}: 9

3.1 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. This topology combined with the updating function results in an input memory of 96 bits.

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 1 shows the expansion of the CCSR at the high input memory end and the two ways of indexing. 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: $k_0 \ldots k_{95}$.

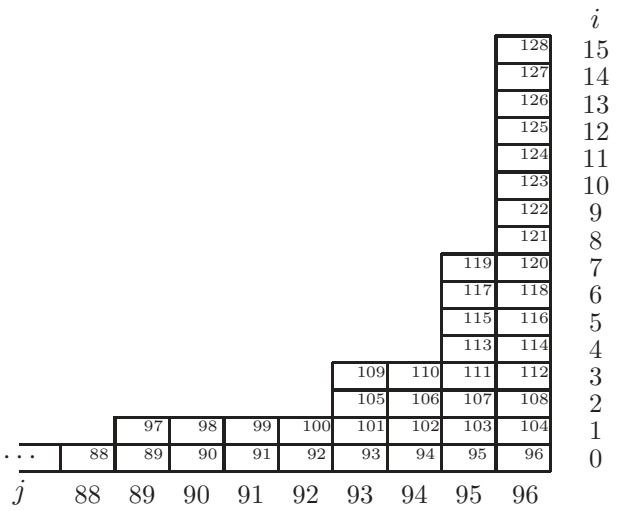

Figure 1: Expansion of the CCSR in the high memory region. q indexing at the right and bottom, a^0 indexing inside the boxes.

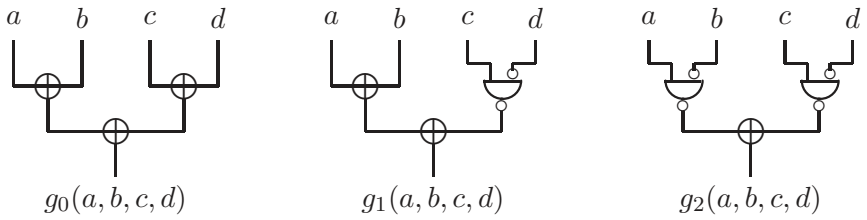

Figure 2: The three functions used in the state-updating transformation

3.2 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):

$$g_0(a,b,c,d) = a+b+c+d \quad (5)$$
$$g_1(a,b,c,d) = a+b+c(d+1)+1 \quad (6)$$
$$g_2(a,b,c,d) = a(b+1)+c(d+1)\ . \quad (7)$$

Figure 2 gives combinatorial circuits of these functions.
For the bits of the CCSR we have:

$$q_i^j \Leftarrow g_x(q_{i \bmod n_{j-1}}^{j-1}, k_{j-1}, q_{i \bmod n_v}^v, q_{i \bmod n_w}^w)\ , \quad (8)$$

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:

$$q_i^{96} \Leftarrow g_2(q_{i \bmod 8}^{95}, q_0^{95-i}, q_{i \bmod 4}^{94}, q_{1 \bmod n_{94-i}}^{94-i})\ . \quad (9)$$

Table 2: Function and v and w values for equation 8

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 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., $a_{53}^3 = 0$.

The keystream bit is given by

$$z = a_0^7 + a_1^7 + a_2^7 \ . \tag{10}$$

This yields:

$$p \Leftarrow g_0(c, a_0^7, a_1^7, a_2^7) \ . \tag{11}$$

and

$$c \Leftarrow g_0(p, a_0^7, a_1^7, a_2^7) \ . \tag{12}$$

3.3 Putting it together

Figure 3 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 3 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.

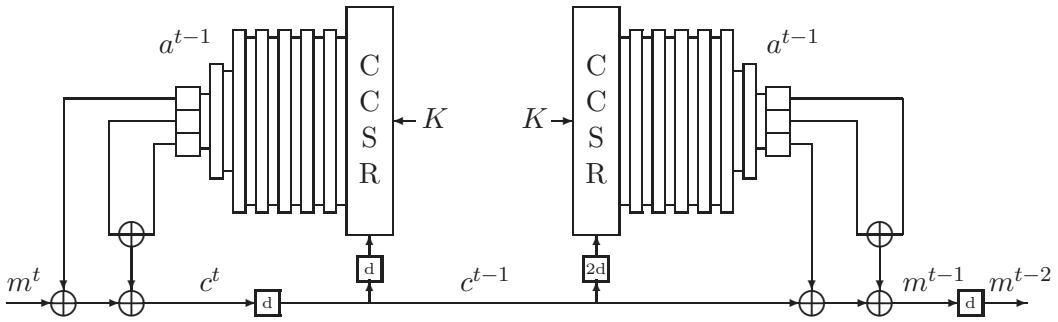

Figure 3: Encryption and decryption with MOUSTIQUE.

4 Security claims

The claimed security properties of a self-synchronizing stream cipher may be expressed in terms of its cipher function.

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.*

We do not claim resistance against attackers that may manipulate the key and that in our attack model, the attacker has no knowledge about the key whatsoever.

5 Motivation for the tweak

We first give a short and simplified description of the attack that broke MOSQUITO, for a more detailed treatment we refer to the original paper [2].

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 in the CCSR is minimal, 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 [2] 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 16 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 lack of 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 leading to much better worst-case diffusion. 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 bits of the CCSR to a small number of bits during a significant number of iterations infeasible. Therefore, we believe chosen-ciphertext key-guessing attacks as in [2] cannot be mounted for MOUSTIQUE. We are aware that 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. For the moment we do not see a method for exploiting this decrease in nonlinearity in an attack. Still, we invite the cryptographic community to try to break MOUSTIQUE.

6 VLSI implementation synthesis results

We have implemented a MOUSTIQUE encryption/decryption circuit using Field Programmable Gate Array (FPGA). This circuit has the same architecture as our circuit for MOSQUITO and is described in [1]. 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 [3] and ALTERA [4] FPGAs and used the same FPGA families as for MOSQUITO: from XILINX FPGAs the Virtex, Virtex-E and Virtex-II family and from ALTERA FPGAs the Apex, Flex and Max family.

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.

The main difference between MOSQUITO and MOUSTIQUE is in the connections between the Flip-Flops in the CCSR. The ciphers have the same critical path delay: $2 * t_{XOR}$. As can be seen when comparing Table 4 and the corresponding table in [1], we achieved some improvements in speed and the usage of logic cells with respect to our MOSQUITO implemen-

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

tations, thanks to some optimizations. The only negative point is the slight decrease of speed for the Xilinx Virtex-II FPGA. Probably our proposed VHDL implementation of MOUSTIQUE matched less with the hardware resources of this kind of FPGA than that of MOSQUITO.

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 MOSQUITO and its predecessor KNOT. Without these attacks MOUSTIQUE would not exist.

References

[1] Joan Daemen and Paris Kitsos, Submission to ECRYPT call for stream ciphers: the self-synchronizing stream cipher Mosquito (2005), available from `http://www.ecrypt.eu.org/stream/`

[2] A. Joux and F. Muller, "Chosen-Ciphertext Attacks against MOSQUITO," *Fast Software Encryption 2006, LNCS*, M. Robshaw, ed., Springer-Verlag, 2006, to appear.

[3] Xilinx Virtex FPGA Data Sheets (2005), URL: `http://www.xilinx.com`

[4] Altera FPGA Data Sheets (2005), URL: `http://www.altera.com`