A side-channel attack against an automata theory based stream cipher

By

Pál Dömösi* and Géza Horváth**and Ferenc Tamás Molnár***and Szabolcs Kovács† and Adama Diene‡

Abstract

In this paper we consider a finite automaton based stream cipher given by P. Dömösi and G. Horváth and we show its immunity in side-channel timing attack.

§ 1. Introduction

Alternative approaches in the design of stream ciphers are very challenging for the cryptographic researches, because nowadays everybody is interested in provable secure ciphers/schemes w.r.t. some computationally hard problem. Moreover, alternative designs are sometimes helpful to obtain protection against side channel attacks or even quantum computers. This work is a further step in this direction.

Key Words: *side-channel attack, stream cipher, finite automata*

This work was supported by the Research Institute for Mathematical Sciences, an International Joint Usage/Research Center located in Kyoto University.

This work was supported by the European Union Grant No. GINOP-2.1.2-8-1-4-16-2018-00558.

This work was supported by United Arab Emirates Program for Advanced Research (UAEU UPAR) Grant # G00003431.

*Debrecen University, Faculty of Informatics, H-4002 Debrecen, POB 400, Hungary & Nyíregyháza University, Institute of Mathematics and Informatics, H-4400, Nyíregyháza, Sóstói út 31/B, Hungary.
e-mail: domosi@unideb.hu

**Debrecen University, Faculty of Informatics, H-4002 Debrecen, POB 400, Hungary.
e-mail: horvath.geza@inf.unideb.hu

***CCLAB Ltd. , H-1134 Budapest, Váci út 49, 6-th floor, DC offices, Hungary.
e-mail: ferenc.molnar@cclab.com

†CCLAB Ltd. , H-1134 Budapest, Váci út 49, 6-th floor, DC offices, Hungary.
e-mail: szabolcs.kovacs@cclab.com

‡United Arab Emirates University, Department of Mathematics, P.O.Box 15551, Al Ain, Abu Dhabi, United Arab Emirates.
e-mail: adiene@uaeu.ac.ae

Automata theory provides a natural basis for designing cryptosystems and several such systems have been designed.

In this paper we study a novel symmetric stream cipher of P. Dömösi and G. Horváth [3, 4] which overcomes all of the discussed disadvantages. Statistical properties of this cipher are discussed in [5]. Now we show its immunity for some side-channel attacks. In more details, we consider this symmetric stream cipher based on finite automata without outputs. The transition table of this finite automaton without outputs, called key automaton, forms a Latin square. The state and input sets of the key-automaton coincide with the plaintext and also the ciphertext alphabet. During the encryption the plaintext is red in sequentially character by character. After getting the next (initially the first) plaintext character, the system gets simultaneously the next (initially the first) pseudorandom string which is also an input string of the key-automaton. The corresponding ciphertext character will coincide with the state of the key-automaton into which this pseudorandom input string takes the automaton from the state which coincides with the corresponding plaintext character. The decryption becomes similarly, using a so-called inverse key-automaton instead of the key automaton such that the input strings will be the mirror images of the corresponding pseudorandom strings.

We start with some standard concepts and notations. All concepts not defined here can be found in [2] and [6]. By an *alphabet* we mean a finite nonempty set. The elements of an *alphabet* are called *letters*. A *word* over an alphabet Σ is a finite string consisting of letters of Σ. A word over a binary alphabet is called a *bit string*. The string consisting of zero letters is called the *empty word,* written by λ. The *length* of a word w, in symbols $|w|$, means the number of letters in w when each letter is counted as many times it occurs. By definition, $|\lambda| = 0$. At the same time, for any set H, $|H|$ denotes the cardinality of H. In addition, for every nonempty word w, denote by $\overrightarrow{w}$ the last letter of w. ($\overrightarrow{\lambda}$ is not defined.) If $u = x_1 \cdots x_k$ and $v = x_{k+1} \cdots x_\ell$ are words over an alphabet Σ (with $x_1, \ldots, x_k, x_{k+1}, \ldots, x_\ell \in \Sigma$), then their *catenation* $uv = x_1 \cdots x_k x_{k+1} \cdots x_\ell$ is also a word over Σ. In this case we also say that u is a *prefix* of uv and v is a *suffix* of uv. Catenation is an associative operation and, by definition, the empty word λ is the identity with respect to catenation: $w\lambda = \lambda w = w$ for any word w. For every word $w \in \Sigma^*$, put $w^0 = \lambda$, moreover, $w^n = ww^{n-1}, n \geq 1$. Let Σ^* be the set of all words over Σ, moreover, let $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. Σ^* and Σ^+ are the *free monoid* and the *free semigroup,* respectively, generated by Σ under catenation. In particular, we put $\Sigma^0 = \{\lambda\}, \Sigma^n = \{w : |w| = n\}, n \geq 1$, and $\Sigma^{(0)} = \Sigma^0, \Sigma^{(n)} = \{w : |w| \leq n\}, n \geq 1$. In addition, for every string $x_1 \cdots x_k$ with $x_1, \ldots, x_k \in \Sigma$, the string $x_k \cdots x_1$ is called a mirror image of $x_1 \cdots x_k$. We will use the notation p^R as the mirror image of p for every $p \in \Sigma^+$. Moreover, by definition, let $\lambda^R = \lambda$.

By an *automaton* we mean a deterministic finite automaton without outputs. In more details, an automaton is an algebraic structure $\mathcal{A} = (A, \Sigma, \delta)$ consisting of the nonempty and finite *state set* A, the nonempty and finite *input set* Σ, a *transition function* $\delta : A \times \Sigma \to A$. The elements of the state set are the *states,* and the elements of the input set are the *input signals.* An element of A^+ is called a *state word* [1] and an element of Σ^* is called an input word. State and input words are also called *state strings* and *input strings,* respectively. If a state string $a_1a_2 \cdots a_s$ $(a_1, \ldots, a_s \in A)$ has at least three elements, the states $a_2, a_3, \ldots, a_{s-1}$ are also called intermediate states. It is understood that δ is extended to $\delta^* : A \times X^* \to A^+$ with $\delta^*(a, \lambda) = a,\ \delta^*(a, xq) = \delta(a, x)\delta^*(\delta(a, x), q), a \in A, x \in \Sigma, q \in \Sigma^*$. In other words, $\delta^*(a, \lambda) = a$ and for every nonempty input word $x_1x_2 \cdots x_s \in \Sigma^+$ (where $x_1, x_2, \ldots, x_s \in \Sigma$) there are $a_1, \ldots, a_s \in A$ with $\delta(a, x_1) = a_1, \delta(a_1, x_2) = a_2, \ldots, \delta(a_{s-1}, x_s) = a_s$ such that $\delta^*(a, x_1 \cdots x_s) = a_1 \cdots a_s$.

In the sequel, we will consider the transition of an automaton in this extended form and thus we will denote it by the same Greek letter δ. If $\overrightarrow{\delta(a, w)} = b$ holds for some $a, b \in A, w \in \Sigma^*$ then we say that w *takes* the automaton from its state a into the state b, and we also say that the automaton *goes* from the state a into the state b under the effect of w.

The transition matrix of an automaton is a matrix with rows corresponding to each input and columns corresponding to each state; at the entry of any row indicated by an input $x \in \Sigma$ sign and any column indicated by a state $a \in A$ the state $\delta(a, x)$ is put. We say that an automaton $\mathcal{A} = (A, \Sigma, \delta)$ is a permutation automaton if all lines of the transition matrix form a permutation of the state set. The automaton $\mathcal{B} = (B, \Sigma', \delta')$ is called the inverse automaton of the permutation automaton $\mathcal{A} = (A, \Sigma, \delta)$ if $A = B, \Sigma = \Sigma'$, moreover, for any $a, b \in A, x \in \Sigma, \delta(a, x) = b$ if and only if $\delta'(b, x) = a$. In this case, we shall use the notation $\mathcal{A}^{-1} = (A, \Sigma, \delta^{-1})$ for $\mathcal{B} = (B, \Sigma', \delta')$. (We note that, in general, the transition matrix of a permutation automaton is not an inverse matrix of the transition matrix of its inverse automaton even if their tranistion matrices are quadrtic.)

A Latin square of order n is an $n \times n$ matrix (with n rows and n columns) in which the elements of an n-state set are entered so that each element occurs exactly once in each fixed (row, column) pair. In this paper we will consider a special type of automata having the transition matrix of the form Latin square. Evidently, these automata are permutation automata. By this property, we would like to avoid statistical attacks.

[1] The empty word is not considered as a state word.

§ 2. A finite automaton based stream cipher

The working of the considered system mainly differs from the most of the stream ciphers : it does not generate the ciphertexts in such a way that the plaintext bit stream is combined with a cipher bit stream by an exclusive-or operation (XOR).

The proposed cipher does not have this property.

Consider an automaton $\mathcal{A} = (A, \Sigma, \delta)$ with $A = \Sigma$, where for every $a, b \in A$ and $x, y \in \Sigma$, $\delta(a, x) \neq \delta(b, x)$ and $\delta(a, x) \neq \delta(a, y)$. Thus, $\mathcal{A}$ is a permutation automaton, i.e., each line of the transition table forms a permutation of the state set. This is an essential property to the unambigouty of the ciphertext for any plaintext.

For the security, we also assume that all columns of the transition table also forms a permutation of the state set.

Let $\mathcal{A}^{-1} = (A, \Sigma, \delta^{-1})$ be the automaton for which $\delta^{-1}(b, x) = a$ with $a, b \in A, x \in \Sigma$ if and only if $\delta(a, x) = b$.

In the furthers $\mathcal{A}$ will be called the *key-automaton* and $\mathcal{A}^{-1}$ will be called the *inverse key automaton.*

Next we give a short formal description of encryption and decrypton. There are detailed examples for these encryption and descryption procedures in [3].

§ 2.1. Encryption

Let $p_1, \ldots, p_k$ be a plaintext and let $r_1, \ldots, r_k \in \Sigma^+$ be random strings generated by the pseudorandom number generator starting by a seed r_0. We note that $|r_0|, \ldots, |r_k| = n$ holds for a fixed positive integer n.

The ciphertext will be $c_1 \cdots c_k$ with $c_1 = \overrightarrow{\delta(p_1, r_1)}, \ldots c_k = \overrightarrow{\delta(p_k, r_k)}$.

§ 2.2. Decryption

Let $c_1, \ldots, c_k$ be a ciphertext and let $r_1, \ldots, r_k \in \Sigma^+$ be the same random strings generated by the pseudorandom number generator starting by a seed r_0.

The decrypted plaintext will be $p_1 \cdots p_k$ with $p_1 = \overrightarrow{\delta^{-1}(c_1, (r_1)^R)}, \ldots p_k = \overrightarrow{\delta^{-1}(c_k, (r_k)^R)}$.

§ 3. Key automaton generation

We shall adopt the method elaborated in [3]. Our solution is an algorithm, which generates appropriate size Latin squares from 2 strings of length n for some positive integer n. We propose $n = 256$ to the technical realization. Each string contains the permutation of the numbers from 0 to $n - 1$, but the second string must start with 0. The steps of the algorithm are the following

(a) The first line of the matrix is the first vector itself.

(b) We receive the second line of the matrix if we shift the elements of the first vector to the right by the second element of the second vector. This is a circular shift – or so called rotate – operation.

(c) We receive the third line of the matrix if we shift the elements of the first vector to the right by the third element of the second vector. . . .

(d) We receive the $n-1$-th line of the matrix if we shift the elements of the first vector to the right by the $n-1$-th element of the second vector.

Now we have an $n \times n$ type matrix, which forms a Latin square.

We note that transmitting or storing this like Latin squares we need only (2n-1)-dimensional vectors. The first n components of these vectors determine the first line of the Latin square and the other n-1 components determine the length of the cyclic permutations of the first line determining the other lines. It is clear that all elements of the matrix can be computed easily by these two vectors.

§ 3.1. Example

The above algorithm generates the Latin square shown in the next table from the following two vectors: $V_1 = (1, 2, 0), V_2 = (0, 2, 1)$.

1	2	0
0	1	2
2	1	0

By this method the size of the key is reduced to $2n-1$ bytes, which can be easily handled by any present programmable device.(The first component of the second vector is always 0 and thus we need not store it.) The key space is still huge, because we have $n!(n-1)!$ possible keys where $n = 256$ is proposed leading to more than 2.87×10^1011 possibilities. In this way, having a huge key space, we can easily store the keys of the block cipher, and the key change is simple as well.

Searching in a matrix is really fast operation, because we have direct access to each element, so if the plaintext is huge, it looks more effective to calculate the transition matrix before encryption, and then search in the matrix during the encoding process. However, if the plaintext is short, it looks more effective to calculate just those elements of the matrix which are used during the encryption.

§ 4. Side-channel attacks (SCA)

A side-channel attack, unlike the other attacks, not analysis the statistical or mathematical weakness of a cryptographic algorithm, it targets the implementation instead,

with noninvasive methods. A side-channel is an indirect source of physical information leaking from the device, such as, the device electromagnetic radiation (EM), power consumption, or the runtime of the algorithm. [1]

§ 4.1. Timing attack

Timing analysis is an SCA that is used to extract critical information about the device under attack by analysing the execution time of each operation under different setups and input patterns. Every operation performed in a silicon-based device takes a certain amount of time to complete. This time can vary due to the type of operation, the input data, the technology used to build the device, and the properties of the environment, in which the device is operating.

An adversary often applies timing analysis on cryptographic systems to extract the secret key, where timing analysis can help the attacker determine which subsets of the key are correct, and which subsets are not. The way an adversary measures the delay of a signal is by applying a change in the input and recording the delay that occurs before the output is updated.

Timing attacks are usually applied along with other side-channel attacks since more information can be extracted when different analysis methods are employed. Power analysis is one example that works well with timing attacks; the power trace does not only show the pattern in which the operation performed is correlated to, but also how long it took before the operation is completed. The order of operation is also revealed when applying timing analysis to power signals; this order can help identify the type of process the device is running and may even allow the adversary to reverse engineer the device.

4.1.1. Timing attack countermeasures

One of the biggest things you can do to protect against timing attacks is to use proper cryptographic libraries and the helper functions they provide (for example, if you are using the bcrypt.js library, use the compare function that it provides, rather than doing your own string comparisons). Any time you try and implement your own crypto it is probably going to be vulnerable to timings attacks.

These kinds of attacks can be very hard to perform though - there are (normally) very small differences in the time to compare strings, for example, which would be harder to detect over the internet (although you can work around this with large samples).

The attacks that are easier to exploit are often where there's external interaction from your code. For example, a password reset feature might have to send an email (which can be slow) if the username is valid and may just return immediately if it is not. Asynchronous external calls can help here (and are probably better for the user anyway). Most timing attacks rely on an attacker being able to make many scripted

requests and analyse the response times, so anything you can do to make this harder (for example, rate limiting or a CAPTCHA) will also provide you a degree of protection. It does not necessarily solve them, but it would make them harder to pull off.

With all the different types of attacks and vulnerabilities out there, writing perfectly secure code is all but impossible (which is why we believe in Application Security Management), but by implementing best practices like secure code review and making smart choices, you can make a positive difference in your security posture. Understanding how different attacks work a big element of this are. Timing attacks are a great example of a vulnerability you can mitigate in code, but that can lead to account takeovers if not addressed. Of course, as with anything security-related, the devil is in the details.

§ 4.2. Protection against side-channel timing attack

Most of the vulnerabilities could be prevented during the design phase, for example with using well known mathematical methods or proven cryptographic algorithms. The novel stream cipher described by this paper is secure against the side-channel timing attack by design, because its running time depends only on the size of the input. Without any correlation between the sensitive information, like the Latin square, the AES keys, or the seed of the pseudorandom number generator, measuring the running time of the algorithm will not provide any result for the adversary.

To prove this theory, we tried to change keys and seed and measured the runtime of the cipher. The tests contain 100 runs with every change. First, we changed the seed of the pseudorandom number generator, but he runtime is closely the same all the time.

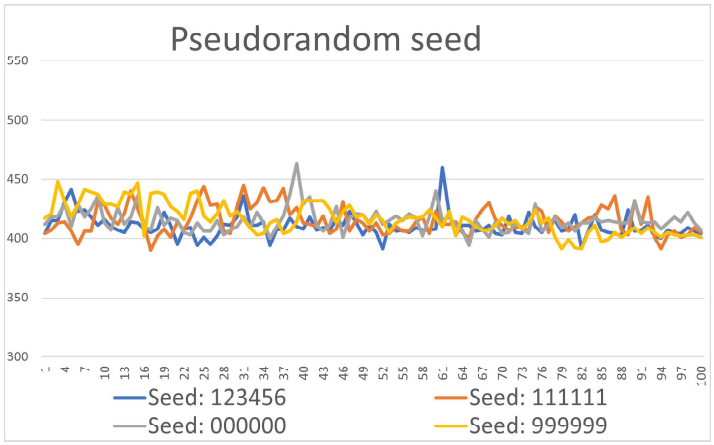

Figure 1. Pseudorandom generator seed

Changing the AES key of the OTP brought the same result, so we could not find any correlation between the runtime and the key.

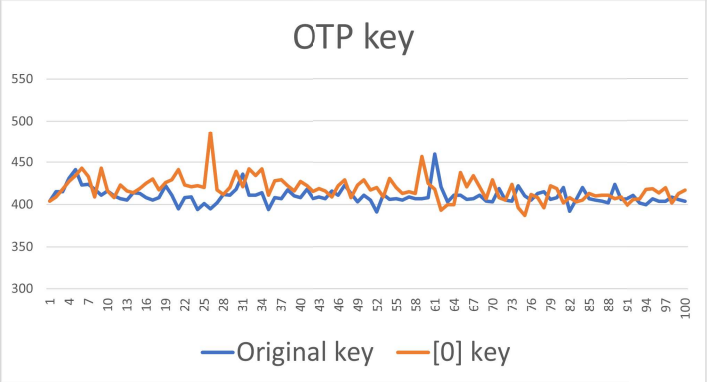

Figure 2. AES key of the OTP

Tests performed on the implementation have shown that the novel stream cipher, described by this paper, is secure against the side-channel timing attack.

§ 5. Performance

We applied key automata with 256 states and 256 inputs having the properties discussed before. The plaintext alphabet and also the set of the final states of the key automaton was the same consisting of 16 elements.

Testing software simulations of the proposed stream cipher were implemented using a computer program written in C^{++}. The implementation was tested on a conventional laptop with Intel Core i7-1065G7 processor, which has 4 cores and 8 threads clocked at 1.3 - 3.9 GHz with 8 MByte of Intel Smart Cache, under Windows 10 operation system. If the implemented system reaches the speed of 0.37 MByte/s as encryption and 0.36 MByte/s as decryption (in relation to the length of the plaintext). Because some stream ciphers reach the speed of more than 20 MByte/s as encryption or decryption on a standard PC (see, for example, [7]), the proposed stream cipher is relatively slow at least for the implemented software case.

Recall that the considered size of the key automaton takes 64 kByte which can be stored on 511 Byte in compressed form. (See Chapter 3.) Moreover, we note that the length of the encryption/decryption software is less than 1 MByte. Thus the system is appropriate to micro-size realization (microcontrollers, smart cards etc.). Choosing suitable hardware devices (microcontrollers, smart cards, true random number generators, etc.) in the technical realization, one can considerably improve the processing

speed.

§ 5.1. Conclusion

In this paper we studied a novel stream cipher based on finite automata without outputs. It was also shown that the system is secure against some attacks whenever some appropriate parameters are considered. A software implementation of the applied version is relatively slow and it is not time optimized yet.

There are a few major issues with the discussed stream cipher.

- There is no rigorous security analysis. Only some standard attack methods are considered. In fact, often these attacks are discussed by the means of an example instance of the cipher and not in general.

- The discussed stream cipher is not really efficient, at least for the software case. In comparison with other promising designs and even with the state of the art ciphers (see, e.g., [7]) the performance is poor. A rigorous machine-independent investigation should be necessary to explore the reasons of this drawback. In addition, it would be great to find minimization algorithms to reduce the size of the key automaton.

References

[1] S. Bhunia and M. Tehranipoor. 2018. Hardware Security: A Hands-on Learning Approach (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] J. Buchmann, Johannes. Introduction to Cryptography. Springer, 2002. DOI: 10.1007/978-3-642-11186-0.

[3] P. Dömösi, G. Horváth: A novel cryptosystem based on abstract automata and Latin cubes Studia Sci. Mathematicarum Hung. 52:(2), 2015, 221-232.

[4] Dömösi, P. and Horváth, G., Symmetric Key Sttream Cipher Cryptographic Method and Device, European Patent Register, EP3639464B1, Espacenet, 2021.

[5] P. Dömösi, J. Gáll, G. Horváth, N. Tihanyi: Statistical Analysis of DH1 Cryptosystem.Acta Cybernet., Szeged, 23 (2017), 371-378.

[6] J. Hopcroft, R. Motwani, J. Ullnan: Introduction to Automata Theory, Languages, and Computation. 3rd edition. Pearson Education, Addison Wesley, Boston, San Francisco, New York, etc,., 2008.

[7] Kashmar, A., Ismail, E. S., Blostream: A High Speed Stream Cipher. J. Eng. Sci. Techn., Vol. 12, No 4 (2017), 1111-1128.