HARRAN ÜNİVERSİTESİ MÜHENDİSLİK FAKÜLTESİ
AKADEMİK BİLİM DERGİSİ
HARRAN ÜNİVERSİTESİ MÜHENDİSLİK DERGİSİ
HARRAN UNIVERSITY JOURNAL OF ENGINEERING
e-ISSN: 2528-8733

A New Software Implementation of the TRIVIUM Stream Cipher

TRIVIUM Akış Şifreleyici için Yeni bir Yazılım Uygulaması

Yazarlar (Authors): *Mehmet Hadi Suzer[1], Nurettin BEŞLİ[2]*

[1] ORCID ID: 0000-0002-0083-8757
[2] ORCID ID: 0000-0003-3657-1393

HRU Muh Der, 5(1): 01-08 (2020)

Harran Üniversitesi

Mühendislik Dergisi

http://dergipark.gov.tr/humder

A New Software Implementation of the TRIVIUM Stream Cipher

Mehmet Hadi SUZER[1,*], Nurettin BEŞLİ[2]

[1]*Harran Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, 63100, Haliliye/Şanlıurfa*

[2]*Harran Üniversitesi, Mühendislik Fakültesi, Elektrik-Elektronik Mühendisliği Bölümü, 63100, Haliliye/Şanlıurfa*

Makale Bilgisi

Başvuru: 21/01/2020
Düzeltme: 29/01/2020
Kabul: 04/02/2020

Anahtar Kelimeler

Gizli Anahtarlı Şifreleme
Akış Şifreleyicileri
TRIVIUM

Keywords

Secret Key Cryptography
Stream Ciphers
TRIVIUM

Abstract

Security, reliability and robustness against reverse engineering attacks are crucial for a high-quality cryptographic tool. Additionally, speed, efficiency and portability are also key components of such a tool. TRIVIUM has been specified as an International Standard since it is a lightweight yet highly secure stream cipher. However, TRIVIUM is designed to be hardware-oriented and its software implementation still lacks portability for high-level programming languages. In this study, we propose a software implementation of TRIVIUM, which enables us to achieve higher bandwidth and lower machine and programming language dependency, comparing to its original software implementation. Our implementation outperforms the original software implementation of TRIVIUM for widely used programming languages in terms of speed and applicability, which makes it possible to reach broader usage.

TRIVIUM Akış Şifreleyici için Yeni bir Yazılım Uygulaması

Öz

Güvenlik, güvenirlik ve tersine mühendislik saldırılarına karşı gürbüzlük yüksek kaliteli bir şifreleme aracı için vazgeçilmezdir. Ek olarak, hız, verimlilik ve taşınabilirlik de bu tür bir aracın temel bileşenleridir. TRIVIUM hafif fakat yüksek güvenlikli bir akış şifreleyici olduğu için Uluslararası Standard olarak belirlenmiştir. Buna karşın, TRIVIUM donanım temelli olarak tasarlanmıştır ve yazılım temelli uygulaması halen yüksek seviyeli dillere taşınırlıktan yoksundur. Bu çalışmada, TRIVIUM için orijinal yazılım uygulamasına kıyasla daha yüksek bant genişliği ve daha düşük makine ve programlama dili bağımlılığına erişmemizi sağlayacak yazılım temelli bir uygulama önermekteyiz. Uygulamamız geniş kullanıma sahip programlama dillerinde hız ve uygulanabilirlik açısından TRIVIUM'un orijinal uygulamasını aşmakta, bu da onun daha geniş kullanımına olanak sağlamaktadır.

1. INTRODUCTION

Transmitting information in a medium which potentially has uncountable number of threats rises an absolute need for security measures. Encrypting data at the source with a key which the destination peer also possesses seems to be an effective way to evade threats residing in the middle. The data is encrypted at the source, transmitted through insecure medium and it gets decrypted into plain text at the destination. Many algorithms have been developed throughout the computer era for the purpose of encryption and decryption. In modern days, cryptographic methods are grouped into Public Key Cryptography (PKC) and Secret Key Cryptography (SKC) based on their use of the key [1]. PKC methods use a pair of public/private key to encrypt/decrypt data respectively. The sender has receiver's public key which can only be used to encrypt data, while the receiver has a private key used to decrypt the data. The encryption and decryption stages are asymmetric and do not rely on key pre-sharing. PKC methods are slow to operate on the bulk data; therefore, they are generally used for exchanging symmetric keys needed by SKC methods. Secret Key Cryptography methods (also known as symmetric key cryptography), on the other hand, use the same pre-shared key at both sides. They are quite faster at encryption or decryption of large volume data, comparing to their PKC counterparts. But sharing a secret key through insecure communication medium

*İletişim yazarı, e-mail: mhsuzer@harran.edu.tr

poses a big threat on security of the methods. At the end, PKC methods are utilized for the purpose of exchanging secret keys of SKC methods, while SKC methods are being utilized to do the real encryption/decryption on large volume data.

A stream cipher is an SKC type of cryptographic tool which works on streaming data with either unknown or infinite size such as live audio/video or sensor data. A plain text data is encrypted using a stream cipher initialized with a seed, travels in encrypted form, and gets decrypted at destination which has the same stream cipher initialized with the same seed of the sender. Figure 1 illustrates the lifecycle of a plain text BMP image which gets encrypted and decrypted by the same stream cipher.

The Stream Cipher Project eSTREAM [2] run by the European Network of Excellence in Cryptology (ECRYPT) brought us three hardware-oriented stream ciphers as finalists, namely TRIVIUM [3], Grain v1 [4] and Mickey v2 [5]. Included in the ECRYPT final annual report [6], these algorithms are then compared in [7] and the TRIVIUM is shown to outperform both Grain v1 and Mickey v2 in terms of throughput and power consumption. Possible weaknesses of the TRIVIUM and Grain are investigated by a couple of studies such as [8 – 15]. These studies show that when robustness of internal parameters, such as state size, the size and truthiness of random numbers used to initialize the ciphers etc., are loosened, these cipher algorithms becomes vulnerable to attacks including full state uncover. However, to date, no known attacks faster than brute force could succeed breaking the TRIVIUM.

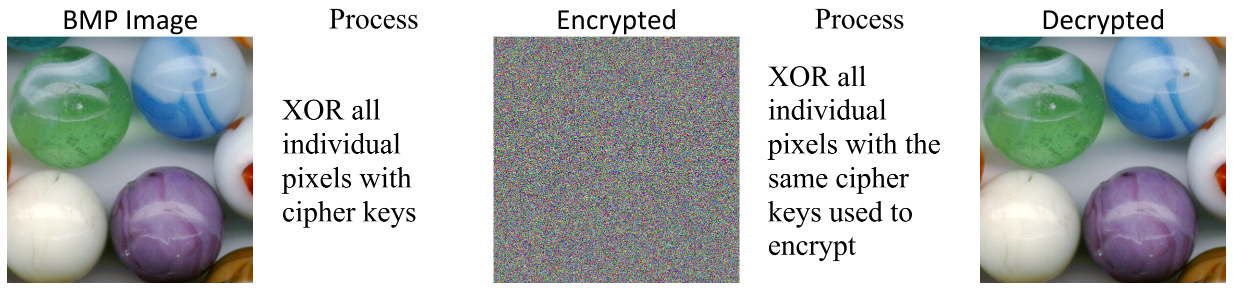
BMP Image
Process
XOR all individual pixels with cipher keys
Encrypted
Process
XOR all individual pixels with the same cipher keys used to encrypt
Decrypted

Figure 1. Bitmap Image Encryption and Decryption with TRIVIUM Symmetric Stream Cipher

There are hardware (FPGA) and software implementations of TRIVIUM [16] coded and published by its designers. The hardware implementation focuses on resource efficiency, i.e. consuming the lowest gate count on an FPGA, while the software implementation aims to achieve the highest bandwidth possible. Keeping these constraints in mind, TRIVIUM designers coded its software version in C language because the C language, among several other well-known languages, is known to be the closest to the assembly language and the machine binary as well. Their original C code heavily depends on low-level bitwise shifts and rotations. At the end, they have a moderately fast running code with the expense of machine and programming language dependency.

We re-designed the original software implementation not only to increase its bandwidth, but we also developed a highly portable code to a broad spectrum of languages. Anyone can now easily port our code to languages such as Java, C# or MATLAB with less effort and still be able to reach the highest possible speeds for that specific language. We already coded TRIVIUM in C, C#, Java, MATLAB and PHP and made them publicly available [17]. Our test results show that, we achieve 33% - 98% bandwidth increases against the original software implementation of TRIVIUM selected as our baseline.

2. MATERIAL and METHOD

The TRIVIUM design is centered around an internal state consisting of 288-cell circular array. Each cell can be a single bit or multi bit and the cipher key is defined as a logical combination of these state cells. Every time the array is rotated, the state cells gets shuffled in a non-linear way and a new cipher is generated at the output. The cipher output is irreversible, i.e. it cannot be used to go back to the previous state. The whole TRIVIUM structure is illustrated in Figure 2.

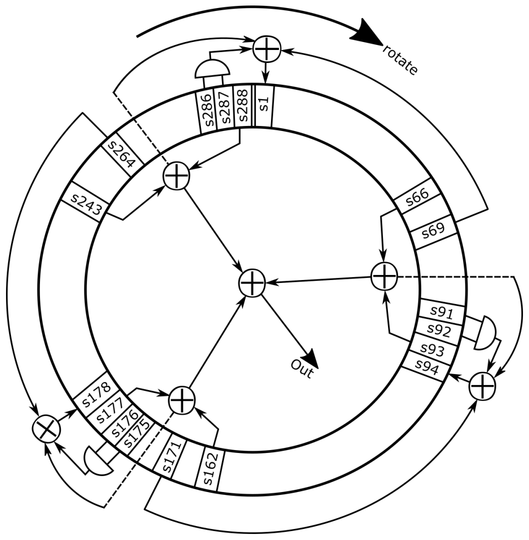

Figure 2 TRIVIUM Structure, ⊕ is XOR, D is AND operator

Initialization of the state array starts by loading a Seed and an Initialization Vector (IV) into the State as shown in lines 1st – 3rd in the Listing 1. Following setup of the state array with Seed and IV vectors, the array is obfuscated by 4 full cycle rotations (4x288 = 1152 moves) which completes the initialization phase. Initialization phase is performed implicitly, i.e. no cipher key is generated at the output during the phase, to hide state transitions taking place inside. After initialization, the state is rotated (lines 4th to 13th) once per iteration and a cipher key is generated at the output (lines 4th – 7th).

```
1.  (State[1], State[2], ..., State[93]) ← (Seed[1], Seed[2], ..., Seed[80], 0, ..., 0)
2.  (State[94], State[95], ..., State[173]) ← (IV[1], IV[2], ..., IV[80])
3.  (State[174], State[175], ..., State[288]) ← (0, 0, ..., 0, ~0, ~0, ~0)

4.  k1 ← State[66] ⊕ State[93]
5.  k2 ← State[162] ⊕ State[177]
6.  k3 ← State[243] ⊕ State[288]

7.  Zi ← k1 ⊕ k2 ⊕ k3

8.  t1 ← k1 ⊕ (State[91] & State[92]) ⊕ State[171]
9.  t2 ← k2 ⊕ (State[175] & State[176]) ⊕ State[264]
10. t3 ← k3 ⊕ (State[286] & State[287]) ⊕ State[69]

11. (State[1], State[2], ..., State[93]) ← (t3, State[1], ..., State[92])
12. (State[94], State[95], ..., State[177]) ← (t1, State[94], ..., State[176])
13. (State[178], State[179], ..., State[288]) ← (t2, State[178], ..., State[287])
```

Listing 1. Pseudo Code for TRIVIUM

Implementation of the pseudo code given in Listing 1 thoroughly depends on the type of platform which is intended to be the host. As TRIVIUM is designed to be hardware-oriented, implementation on a hardware platform such as FPGA, is more straight forward comparing to implementation on a software platform. The internal state which consists of 288 cells, can easily be defined as a 288-bit shift register in FPGA. Bitwise shifting the state (11th – 13th lines in Listing 1) is then just a matter of a single clock, allowing a super-fast update and rotate. However, in software, things get complicated: There is no 288-bit wide register in commercially available CPUs. Thus, the state needs to be defined as an array of primitive types. Shifting the state which is an array of basic types, such as bytes, requires each cell to be moved onto its adjacent cell in a loop of 288 iterations. If the shift is performed using bitwise operators, then the update and rotate operation become a hassle. Further, reading and updating individual bit within the state is a cumbersome task: One needs to first locate the bit (by doing some calculations) to be accessed or updated, and then perform lots of masking, shifting and other bitwise operations.

The original software implementation of the TRIVIUM [16] defines the state as a byte array with 40 elements. To rotate the state, the array elements are shifted in such a way that overall effect is the same as rotating a single 288-bit register. In fact, shift operations alone are not complex, but shifting all bytes in an array as a single entity makes the whole operation complex. To anticipate the complexity developed, one needs to consider that, when dealing with a Full HD video, even a single extra logic or arithmetic operation can add hundreds of millions of instructions per second to the CPU execution queue. Another drawback is that this software implementation outputs only a single bit at every iteration, resulting in a rigid tool in terms of cipher key width. Finally, for performance considerations, the design is implemented in C code using low-level instructions: All operations are bitwise, and functions are inline macros. Applications coded in this style are tailored to C language, therefore, they are hardly portable to high-level programming languages, such as Java, C# and MATLAB etc.

We propose to implement a new software version of the TRIVIUM to alleviate drawbacks mentioned above. Our implementation has high performance, comparing to the original one, yet it does not rely on low-level instructions such as shifts or rotations, resulting in a highly portable code. Its output can easily be configured to 1, 8, 16, 32, or 64-bit granularity with the same super-fast iterations. This gives us the opportunity to generate cipher keys of the size of the plain text data. For example, a bitmap image with 24-bit per pixel can be encrypted with 32-bit wide cipher keys generated by the TRIVIUM at each iteration. No need to concatenate individual bit in order to construct a 24-bit wide cipher.

Pseudo code of our implementation is given in Listing 2. Our implementation utilizes a circular array with a dynamic head as the internal state. The internal state is rotated by rewinding the head pointer in reverse direction. This pseudo rotation eliminates the need for a 288 steps loop to shift state cells, which makes a big contribution to the optimization of the final code. The head is rewound by decrementing it (lines 18th – 20th) once every rotation, and the cell it points to is regarded as the 1st index of the state array. The array indexes are then calculated relative to the dynamic head. The function IDX shown in the Listing 2 calculates the absolute index of the cell to be accessed or modified, it is, therefore, called every time the state array accessed. For example, "`State[i]`" of the original code is now replaced with "`State[IDX(i)]`".

```
1.  head ← 1
2.  (State[1], State[2], ..., State[93]) ← (Seed[1], Seed[2], ..., Seed[80], 0, ..., 0)
3.  (State[94], State[95], ..., State[173]) ← (IV[1], IV[2], ..., IV[80])
4.  (State[174], State[175], ..., State[288]) ← (0, 0, ..., 0, ~0, ~0, ~0)

5.  function IDX(i)
6.    i ← i + head
7.    if (i > 288)
8.      i ← i - 288
9.    return (i)
10. end

11. k1 ← State[IDX(66)] ⊕ State[IDX(93)]
12. k2 ← State[IDX(162)] ⊕ State[IDX(177)]
13. k3 ← State[IDX(243)] ⊕ State[IDX(288)]

14. Zi ← k1 ⊕ k2 ⊕ k3

15. t1 ← k1 ⊕ (State[IDX(91)] & State[IDX(92)]) ⊕ State[IDX(171)]
16. t2 ← k2 ⊕ (State[IDX(175)] & State[IDX(176)]) ⊕ State[IDX(264)]
17. t3 ← k3 ⊕ (State[IDX(286)] & State[IDX(287)]) ⊕ State[IDX(69)]

18. head ← head - 1
19. if (head < 1)
20.   head ← 288
```

```
21. State[IDX(178)] ← t2
22. State[IDX(94)] ← t1
23. State[IDX(1)] ← t3
```

Listing 2. Pseudo Code for the proposed TRIVIUM implementation

As shown in Listing 2, the state array is setup by loading Seed and IV in 1st – 4th lines (initialization section), while the cipher key is generated in 11th – 14th lines (update section), and the state array is rotated in 15th – 23rd lines (rotate section) in order to prepare the state for the next iteration. The biggest enhancement of our implementation is in the last three lines (21st – 23rd): Shifting 288 elements array is replaced by just three assignments.

3. RESULTS and DISCUSSION

We evaluate our implementation by comparing its bandwidth to the original one. We encrypt a bitmap image of Full HD size (1920 x 1080 pixels, 24bpp) 100 times for each language and take the average as the result to discard transient effects. The image we used and its encrypted version can be seen in Figure 1. Our testbed computer has the following operating system and hardware specifications:

OS: Windows 10, 64 Bit
CPU: Intel I5 7500, 3.40GHz
Memory: 8GB, DDR4, 2400MHz

Compiler/Interpreter versions of the programming languages are shown in the Table 1.

Table 1. TRIVIUM Performance of Different Programming Languages

	Image	C (gcc 6.3.0)	Java (12.0.1)	C# .NET (8.0)	MATLAB (2018B)	PHP (7.3.1)
Original Code	BMP	140ms	1440ms	3410ms	12.40sec.	126.24sec.
	Virtual	45ms	1320ms	860ms	5.84sec.	125.36sec.
Proposed Code	BMP	124ms	150ms	2540ms	7.30sec.	3.47sec.
	Virtual	30ms	30ms	30ms	0.78sec.	2.71sec

Table 1 shows average time to encrypt a BMP image for both the original and proposed implementation. As it can be seen from the table, the encryption speed is affected by the performance of the image manipulation libraries. To assess performance of the TRIVIUM core itself, we also performed tests using a virtual image, i.e. an integer array in main memory, of the same size of a Full HD image, 1920 x 1080. Encrypting a real BMP image is slower because the library used to load/store the image from/to the disk and manipulate individual pixels is a bottleneck for the test application. Moreover, the same image library is not available to all programming languages we compare, therefore, we want to show results for both the real and the virtual image.

Test results in Table 1 show that we achieve a significant bandwidth increase, as low as 33% to as high as 98% across different programming languages. Encrypting or decrypting an image of Full HD resolution within 30ms gives the opportunity to encrypt a Full HD video content with up to 33 frames per second (fps). Video streaming applications can now encrypt or decrypt the live video content while delivering to end parties. Although MATLAB and PHP are far slower comparing to C, C# and Java, they can still encrypt Full HD images or live video streams of smaller resolutions. For example, MATLAB can encrypt 10 video frames per second of 480 x 600 resolution.

Details of performance results is given for each language below.

3.1. C Implementation

C is the closest to machine language right after the assembly. C codes are compiled into binary, which is then executed by the CPU directly. It has ability to access and modify memory locations through pointers, which dramatically increases its speed comparing to other higher-level languages. Although mastering C is time consuming and thus it is less preferred comparing to C#, Java etc. it still shows best performance when it comes to compute or data intensive applications. C is the best language for our purposes, as streaming data such as Full HD video is both compute and data intensive.

As seen from the Table 1, the C implementation is the fastest one in encrypting Full HD images. Besides, our implementation has unique features resulting 33% faster encryption of the virtual image.

3.2. Java Implementation

Java language in terms of distance to machine code, is in the middle of compiled languages such as C, C++ and interpreted languages such as MATLAB and PHP. Java produces an intermediate byte code which runs on the Java Virtual Machine. As the Virtual Machine is farther to the hardware, it executes machine dependent code slower. Our code shows the same performance for C, Java and C#, thanks to our high-level code executes faster. The original code is much slower, by a factor over 40, because of the low-level coding style not efficient for a high-level language.

3.3. C# Implementation

C#.NET codes are compiled in two stages. The first stage is similar to the compilation stage in Java: The user code is compiled into an intermediate code. C# has one more compilation stage executed per demand (aka just in time compiling). This second stage produces a binary, from the intermediary code, fully compatible with the host OS and hardware. Our evaluation shows that the proposed C# code has superior performance against the original implementation ported into C#. Real BMP encryption tests in .NET platform are heavily affected by the highly slow image manipulation library of .NET. Although the .NET image library adds over 2500ms latency, our code shows the same performance as in C for the virtual image while the original implementation port runs much slower.

3.4. MATLAB Implementation

MATLAB is a popular language especially among the scientific community. It is an interpreted language, therefore, user code written in MATLAB is expected to run slower comparing to the popular languages C/C++, Java or C#. Nevertheless, test results still show significant performance gain (over 85%) for encryption of the virtual image comparing to the code ported from the original implementation.

3.5. PHP Implementation

PHP is an interpreted language, generally used to code dynamic web pages. It executes code slowly not only because it interprets the code, additionally, it lacks strong typing, i.e. variables are declared without explicit types, degrading code efficiency. Although the PHP has the lowest speed (among all languages being tested) executing our code, our implementation still reaches a significant achievement, it runs 98% faster comparing the original counterpart.

4. CONCLUSION

TRIVIUM shows best performance on hardware platforms such as FPGA, because all jobs in an iteration are performed within a single clock period. Software implementations on the other hand, are slower as CPUs partition those jobs into basic instruction steps and executes them sequentially. Data and compute intensive applications such as streaming video, leaves nanosecond grade time windows for CPUs to handle all those jobs. Low-level programming languages still perform good, but not all programmers code their applications in them; instead, they use high-level languages for modern applications such as web and mobile. However, the original TRIVIUM code does not allow easy migration to high-level languages.

We restructured the TRIVIUM implementation in a way that we stripped out low-level instructions so it can be coded in high-level languages. Furthermore, we achieve a high optimization degree without sacrificing portability, resulting in a high-performance code. Finally, this version of the TRIVIUM makes it a perfect candidate of software-oriented stream ciphers. Our implementation runs 33% faster even in the C language, where it is difficult to beat the original design.

REFERENCES

[1] Kessler, G. C., An overview of cryptography, https://www.garykessler.net/library/crypto.html, accessed: 2020-19-01, 2020.

[2] eSTREAM, The eSTREAM project by European Network of Excellence in Cryptology (ecrypt), http://www.ecrypt.eu.org/stream/project.html, accessed: 2020-19-01, 2004 – 2008.

[3] De Cannière, C., TRIVIUM: A stream cipher construction inspired by block cipher design principles, in *Information Security*, edited by Katsikas, S.K., López, J., Backes, M., Gritzalis, S., and Preneel, B., pp. 171 – 186, Springer, Berlin, Heidelberg, 2006.

[4] Hell, M., Johansson, T., and Meier, W., Grain: A stream cipher for constrained environments, *Int. J. Wire. Mob. Comput.*, 2 (1), 86 – 93, 2007.

[5] Babbage, S., and Dodd, M., The MICKEY Stream Ciphers, pp. 191 – 209, Springer, Berlin, Heidelberg, 2008.

[6] Babbage, S., De Cannière, C., Canteaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B., Rijmen, V., and Robshaw, M.J.B., The eSTREAM Portfolio, Available via https://www.ecrypt.eu.org/stream/portfolio.pdf, accessed: 2020-19-01, April 2008.

[7] Good, T., and Mohammed, B., Hardware performance of eSTREAM phase III stream cipher candidates, in *State of the Art of Stream Ciphers Workshop SASC 2008*, Lausanne, Switzerland, 2008.

[8] Datta, P., Roy, D., and Mukhopadhyay, S., A probabilistic algebraic attack on the Grain family of stream ciphers, in *Network and System Security*, pp. 558 – 565, Springer International Publishing, Cham, 2014.

[9] Ghafari, V. A., and Hu, H., A new chosen IV statistical attack on Grain-128a cipher, in *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pp. 58 – 62, 2017.

[10] Kazmi, A. R., Afzal, M., Amjad, M. F., Abbas, H., and Yang, X., Algebraic side channel attack on TRIVIUM and Grain ciphers, *IEEE Access*, 5, 23, 958 – 23, 968, 2017.

[11] Quedenfeld, F. M., and Wolf, C., Advanced algebraic attack on TRIVIUM, in *Mathematical Aspects of Computer and Information Sciences*, pp. 268 – 282, Springer International Publishing, Cham, 2016.

[12] Raj, A. S., and Srinivasan, C., Analysis of algebraic attack on TRIVIUM and minute modification to TRIVIUM, in *Advances in Network Security and Applications*, pp. 35 – 42, Springer, Berlin, Heidelberg, 2011.

[13] Rohani, N., Noferesti, Z., Mohajeri, J., and Aref, M. R., Guess and determine attack on TRIVIUM family, in *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 785 – 790, 2010.

[14] Sarkar, S., Banik, S., and Maitra, S., Differential fault attack against Grain family with very few faults and minimal assumptions, *IEEE Transactions on Computers*, 64 (6), 1647 – 1657, 2015.

[15] Zhang, B., Xu, C., and Meier, W., Fast near collision attack on the Grain v1 stream cipher, in *Advances in Cryptology* - EUROCRYPT 2018, pp. 771 – 802, Springer International Publishing, Cham, 2018.

[16] TRIVIUM, The eSTREAM project, eSTREAM phase III, https://www.ecrypt.eu.org/stream/triviumpf.html, accessed: 2020-19-01, 2004-2008.

[17] TRIVIUM Implementations, Stream Cipher Codes for C, C#, Java, MATLAB, PHP, Verilog, http://ceng.harran.edu.tr/msuzer/scr/codes/, accessed: 2020-19-01, 2020