TriviA: A Fast and Secure Authenticated Encryption Scheme *

Avik Chakraborti[1], Anupam Chattopadhyay[2], Muhammad Hassan[3], and Mridul Nandi[1]

[1] Indian Statistical Institute, Kolkata, India
avikchkrbrti@gmail.com, mridul.nandi@gmail.com
[2] School of Computer Engineering, NTU, Singapore
anupam@ntu.edu.sg
[3] RWTH Aachen University, Germany
muhammad.hassan@rwth-aachen.de

Abstract. In this paper, we propose a new hardware friendly authenticated encryption (AE) scheme TriviA based on (i) a stream cipher for generating keys for the ciphertext and the tag, and (ii) a pairwise independent hash to compute the tag. *We have adopted one of the ISO-standardized stream ciphers for lightweight cryptography, namely Trivium, to obtain our underlying stream cipher.* This new stream cipher has a state that is a little larger than the state of Trivium to accommodate a 128-bit secret key and IV. *Our pairwise independent hash is also an adaptation of the EHC* or "Encode-Hash-Combine" hash, that requires the optimum number of field multiplications and hence requires small hardware footprint. We have implemented the design in synthesizable RTL. Pre-layout synthesis, using 65 nm standard cell technology under typical operating conditions, reveals that TriviA is able to achieve a high throughput of 91.2 Gbps for an area of 24.4 KGE. We prove that our construction has at least 128-bit security for privacy and 124-bit security of authenticity under the assumption that the underlying stream cipher produces a pseudorandom bit stream.

Keywords: Trivium, stream cipher, authenticated encryption, pairwise independent, EHC, TriviA.

1 Introduction

The emergence of Internet-of-Things (IoT) has made security an extremely important design goal. A huge number of embedded devices are online and this online presence opens myriads of possibilities to a third party intruder to alter the communication between two devices. Hence, a critical information transfer requires a secure channel. Symmetric-key encryption provides privacy by securing the channel, whereas message authentication codes (MACs) are used to provide integrity and authenticity assurances. Using an appropriate, efficient

* This is the full version of the paper. The original version of the paper will appear in the proceedings of CHES-2015.

combination of symmetric key encryption and MAC, also called authenticated encryption [10, 25], one can achieve both privacy and authenticity. An interest in new efficient and secure solutions of authenticated encryption is manifested in the recently launched competition called CAESAR [3].

Authenticated encryption can be achieved using either a stream cipher or a block cipher or both. The general opinion seems to be that stream ciphers can be designed to offer high throughput/ area ratios, a desired performance metric for embedded devices. Trivium [16], Grain [23], Mickey [9] etc. are prominent examples of implementation-friendly stream ciphers from the eSTREAM project [4]. Trivium has been specified as an International Standard under ISO/IEC 29192-3 for the lightweight cryptography category [5].

Authenticated Encryption based on Stream Cipher. A method of using stream cipher for the construction of authenticated encryption scheme is described by Bernstein in [12]. The authenticated encryption scheme HELIX [19] and later PHELIX [35] are designed based on a stream cipher. Both were later attacked [31, 36]. Grain has been modified to Grain-128 [24] to support an integrated authentication mechanism. To the best of our knowledge, in the ETSI specification [1], combining the stream cipher SNOW-3G [2] with polynomial hash, and later in [33] by Sarkar, a definitive study on constructions of authenticated encryptions using stream cipher and ΔU hash have been made. Integrating universal hash with other cryptographic primitives has also been studied by Bernstein [11].

Our Contribution. In this paper, we propose a new stream cipher TriviA-SC which is a modification of Trivium [16], a well-studied and efficient (both in terms of software and hardware) stream cipher. Moreover, our new stream cipher has a key and a initial value of 128 bits. We introduce non-linearity in the output stream which helps to resist some known approaches of finding the key for Trivium. We also study a Δ-Universal hash EHC [32], parametrized by a parameter d, which requires a minimum number of field multiplications and can be implemented with small hardware footprint. In the paper by Nandi [32], the ΔU property (a close variant of pair-wise independent property) of EHC is shown for $d \leq 4$. Here we extend their result and show that the same hash function is a ΔU hash for $d = 5$. This choice of d helps us to make a higher security claim.

Finally, we describe an efficient integration of these primitives to construct a new authenticated encryption scheme-TriviA constructed as a variant of the stream cipher based modes described by Sarkar [33]. We would like to point out that EHC requires a variable key to incorporate variable length messages and the security of it relies on the assumption that all the keys are chosen independently. However, when we use it in an authenticated encryption mode, we have to leak the key through ciphertext and the independence assumption is no longer true. We have shown that TriviA achieves 128-bit security for privacy and 124-bit security for authenticity, assuming that the underlying stream cipher Trivia-SC produces pseudorandom bit stream. We would like to remark that our scheme provides no less security than most of the authenticated encryption schemes.

We also report the hardware performance of TriviA on both FPGA and ASIC platforms and make a comparative study with other authenticated encryption schemes implemented in a similar platform. We have observed that, TriviA is very efficient in terms of throughput, cycles per byte and area-efficiency. For area-efficiency metric TriviA is at least 3.8 times better than the closest candidate Ascon from our list.

2 Preliminaries

Notation. We represent a tuple $(X_a, \cdots, X_b)$ by $X[a \cdots b]$, when the X_i's are bit strings, we also identify the tuple as the concatenation $X_a \| \cdots \| X_b$. For a set S, let $S^+ = \cup_{i=1}^{\infty} S^i$, $S^{\leq n} = \cup_{i=1}^{n} S^i$ and $S^* = S^+ \cup \{\lambda\}$ where λ is the empty string. The usual choice of S is $\{0,1\}$. For $\forall x = x_1 \dots x_n \in \{0,1\}^*$, denote n by $|x|$. The string with one followed by n zeroes is denoted by 10^n. Let the number of n-bit blocks in a Boolean string x be denoted by $\ell_x^n = |x|/n$.

Finite Field. Let $\mathbb{F}_{2^n}$ denote the finite field over $\{0,1\}^n$, for a positive integer n. In this paper, we consider the primitive polynomials [18, 29] $p_{32}(x) = x^{32} + x^{22} + x^2 + x + 1$ and $p_{64}(x) = x^{64} + x^4 + x^3 + x + 1$ to represent the fields $\mathbb{F}_{2^{32}}$ and $\mathbb{F}_{2^{64}}$ respectively. We denote the corresponding primitive elements by α and β which are binary representations of 2. The field addition or bit-wise addition is denoted as "xor" $\oplus$. We note that multiplication by powers of α, β are much simpler than multiplication between two arbitrary elements. For example, multiplication between an arbitrary element $a := (a_0, \dots, a_{31}) \in \{0,1\}^{32}$ and α is $a \cdot \alpha = (b_0, \dots, b_{31})$ where $b_0 = a_{31}$, $b_1 = a_0 \oplus a_{31}$, $b_2 = a_1 \oplus a_{31}$, $b_{22} = a_{21} \oplus a_{31}$ and for all other i, $b_i = a_{i-1}$. Similarly, one can express the multiplication of other powers of primitive elements by some simple linear combinations of the bits a_i's. This representation is useful when we implement power of α and β multipliers in hardware.

2.1 Authenticated Encryption and Its Security Definitions

An authenticated encryption F_K is an integrated scheme that provides both privacy of a plaintext $M \in \{0,1\}^*$ and authenticity or data integrity of the plaintext M as well as the associate data $D \in \{0,1\}^*$. Thus, on the input of a public variable nonce N (it can be considered as an arbitrary number distinct for every encryption), associate data $D \in \{0,1\}^*$ and a plaintext $M \in \{0,1\}^*$, F_K produces a tagged-ciphertext (C,T) where $|C| = |M|$ and $|T| = t$ (tag-size, usually 128). Its inverse or decryption algorithm F_K^{-1} returns $\perp$ for all those (N, D, C, T) for which no such M exists, otherwise it returns M for which (C,T) is the tagged-ciphertext.

Privacy. A distinguishing advantage of A against two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ is defined as $\Delta_A(\mathcal{O}_1; \mathcal{O}_2) = |\mathsf{Pr}[A^{\mathcal{O}_1} = 1] - \mathsf{Pr}[A^{\mathcal{O}_2} = 1]|$. Given a nonce-respecting adversary A (nonces for every encryption are distinct) we define the **privacy** or

PRF-advantage of A against F as $\mathbf{Adv}^{\mathrm{prf}}_{F}(A) := \Delta_A(F_K; \$)$ where $\$$ returns a random string of appropriate size. The PRF-advantage of F is defined as

$$\mathbf{Adv}^{\mathrm{prf}}_{F}(q, \sigma, t) = \max_{A} \mathbf{Adv}^{\mathrm{prf}}_{F}(A)\,,$$

where the maximum is taken over all adversaries running in time t and making q encryption queries with total bit-size of all responses at most σ.

Authenticity. We say that an adversary A **forges** an authenticated encryption F if A outputs a fresh (not obtained before through an F-query) (N, D, C, T) where $F_K^{-1}(N, D, C, T) \neq \perp$. Note that in the forging attempt N can repeat. In general, a forger can make q_f attempts to forge. We denote the forging advantages as

$$\mathbf{Adv}^{\mathrm{auth}}_{F}(A) := \mathsf{Pr}[A^F \text{ forges}], \quad \mathbf{Adv}^{\mathrm{auth}}_{F}(q, q_f, \sigma, t) = \max_{A} \mathbf{Adv}^{\mathrm{auth}}_{F}(A)\,,$$

where the maximum is taken over all adversaries running in time t making q encryption queries and q_f forging attempts with total bit-size of all responses at most σ.

2.2 Examples of Universal Hash Functions

A keyed hash function $h(K; \cdot)$ over D is called an **ϵ-ΔU (universal) hash function** if for all δ, the δ-*differential probability*

$$\mathsf{Pr}[h(K; x) - h(K; x') = \delta] \leq \epsilon \text{ for all } x \neq x' \in D\ .$$

In this paper, we conventionally assume the hash keys are uniformly chosen from the key-space. A hash function h is called **ϵ-universal** (or ϵ-U) if the 0-differential probability (or *collision probability*) is at most ϵ for all $x \neq x'$. We call h **ϵ-balanced** if for all a, b, $\mathsf{Pr}[h(K; a) = b] \leq \epsilon$.

Examples. The Multi-linear hash $\mathsf{ML}(k_1; x_1) := k_1 \cdot x_1$ and Pseudo-dot-product (or PDP) hash $\mathsf{PDP}(k; x) := (x_1 \oplus k_1) \cdot (x_2 \oplus k_2)$, with $k = k_1 || k_2$ and $x = x_1 || x_2$ are two popular examples of 2^{-32}-ΔU hash where $x_1, x_2, k_1, k_2 \in \mathbb{F}_{2^{32}}$ and $k, x \in \mathbb{F}_{2^{64}}$. One can check that the ΔU property of a hash using independent keys is closed under summation. This is a useful technique to define a hash for larger domain, e.g. we can add the invidual PDP values corresponding to the message and key blocks to obtain a ΔU hash $\bigoplus_{i=1}^{m} (k_{2i-1} \oplus x_{2i-1}) \cdot (k_{2i} \oplus x_{2i})$ for $x = (x_1, \cdots, x_{2m})$ and $k = (k_1, \cdots k_{2m})$.

3 EHC Hash

This section describes a Δ-Universal hash EHC or *Encode-Hash-Combine* hash, which is constructed using an error correcting code ECCode_d of distance d. We first describe a Vandermonde matrix of size $d \times \ell$, denoted $V_\gamma^{(d)}$, γ as a primitive

element of $\mathbb{F}_{2^n}$ is defined below. For $n = 32$ and 64 the matrices are denoted by $V_\alpha^{(d)}$ and $V_\beta^{(d)}$ respectively.

$$V_\gamma^{(d)} = \begin{pmatrix} 1 & \cdots & 1 & 1 & 1 \\ \gamma^{\ell-1} & \cdots & \gamma^2 & \gamma & 1 \\ \gamma^{2(\ell-1)} & \cdots & \gamma^4 & \gamma^2 & 1 \\ \vdots & \cdots & \vdots & \vdots & \vdots \\ \gamma^{(\ell-1)(d-1)} & \cdots & \gamma^{2(d-1)} & \gamma^{d-1} & 1 \end{pmatrix}.$$

We have observed that, whenever $1, \gamma, \ldots, \gamma^{\ell-1}$ are distinct, any $s \leq d$ columns of V are linearly independent. We next describe the VMult algorithm for *multiplying $V_\alpha^{(d)}$ to a vector $h = (h_1, \ldots, h_\ell) \in \mathbb{F}_{32}^\ell$ in an online manner using Horner's rule* without requiring **any additional memory**.

Algorithm $\mathsf{VMult}_{\alpha,d}$
Input: $x := (x_1, x_2, ..., x_\ell) \in \mathbb{F}_{2^{32}}^\ell$
Output: $y := (y_1, y_2, \ldots, y_d) \in \mathbb{F}_{2^{32}}^d$ such that $y = V_\alpha^{(d)} \cdot x$

$y_1 = \cdots = y_d = 0^{32}$
for $i = 1$ to ℓ
    for $j = 1$ to d: $y_j \leftarrow \alpha^{j-1} \cdot y_j \oplus x_i$; * VHorner module *\
return $(y_1, \ldots y_d)$;

Algorithm 1: $\mathsf{VMult}_{\alpha,d}$ multiplies an ℓ-dimensional column vector $x = (x_1, \ldots, x_\ell)$ by a Vandermonde matrix $V_\alpha^{(d)}$ to output a d-dimensional vector $y := V_\alpha^{(d)} \cdot x$. Similarly we define $\mathsf{VMult}_{\beta,d}$ for 64-bit field elements. Note that, α and β are the primitive elements of $\mathbb{F}_{2^{32}}$ and $\mathbb{F}_{2^{64}}$, respectively described in Sect. 2. When we implement this algorithm in hardware we only need to implement VHorner.

3.1 ECCode

We next describe the efficient instantiation of an error correcting code ECCode_d with systematic form over $\{0,1\}^{64}$.

$$\mathsf{ECCode}_d(x_1, \ldots, x_\ell) = (x_1, \ldots, x_\ell, x_{\ell+1}, \ldots, x_{\ell+d-1}), \tag{1}$$

where $(x_{\ell+1}, \ldots, x_{\ell+d-1}) = \mathsf{VMult}_{\beta,(d-1)}(x_1, \ldots, x_\ell)$.

Example 1. Let $(x_1, x_2, x_3) \in (\{0,1\}^{64})^3$ be the input to ECCode_4. The output is $\mathsf{ECCode}_4(x_1, x_2, x_3) = (x_1, x_2, x_3, x_4, x_5, x_6)$ where, $x_4 = x_1 + x_2 + x_3$, $x_5 = \beta^2 x_1 + \beta x_2 + x_3$ and $x_6 = \beta^4 x_1 + \beta^2 x_2 + x_3$.

In [32], it has been shown that for $d = 4$, the above code has minimum distance 4. We next extend their result for $d = 5$ and show that it has minimum distance 5 for all $\ell \leq 2^{30}$. The result is described in Proposition 1 below.

Proposition 1. *ECCode_5 has minimum distance 5 over $\{0,1\}^{64\ell}$ for any fixed $\ell \leq 2^{30}$.*

Proof. ECCode_5 is a linear code with systematic form in which the expansion is determined by the matrix $V := V_\beta$. So it suffices to show that V_β is an MDS matrix, i.e., all square submatrices are non-singular. Clearly, any square submatrix of size 1 or 4 is a Vandermonde matrix and hence non-singular. Each of the submatrices of size 2 can be converted to a Vandermonde matrix by elementary column operations (multiplying the columns with non-zero constants).

We now consider the submatrices of size 3. If we consider the submatrices corresponding to the $1^{st}, 2^{nd}$ and the 3^{rd} row or the $2^{nd}, 3^{rd}$ and the 4^{th} row, then these submatrices can be transformed to a Vandermonde matrix by elementary column operations (by non-zero constant multiplications). If we consider the submatrices corresponding to the $1^{st}, 2^{nd}$ and the 4^{th} row or the $1^{st}, 3^{rd}$ and the 4^{th} row then the matrices have the form

$$\begin{pmatrix} 1 & 1 & 1 \\ \beta^i & \beta^j & \beta^k \\ \beta^{3i} & \beta^{3j} & \beta^{3k} \end{pmatrix} \text{ or } \begin{pmatrix} 1 & 1 & 1 \\ \beta^{2i} & \beta^{2j} & \beta^{2k} \\ \beta^{3i} & \beta^{3j} & \beta^{3k} \end{pmatrix}.$$

One can check that the submatrix corresponding to the $1^{st}, 2^{nd}$ and the 4^{th} row is non-singular if and only if $1 + \beta^{(i-j)} + \beta^{(k-j)} \neq 0$, by computing the determinant. We have experimentally verified that the above condition holds for all $i < j < k \leq 2^{30}$. This completes the proof. □

3.2 EHC Hash

EHC hash [32] is a 2^{-128}-ΔU hash which requires fewer multiplications than the Toeplitz hash [26] to process a message block. For a fixed length $\ell \leq 2^{30}$, the definition of $\mathsf{EHC}^{(d,\ell)}$ is given in the Algorithm 2 for all $d \leq 5$. PDP hash used in this construction is described in Sect. 2.

The variable length hash $\mathsf{EHC}^{(d)}$ defined over all messages of sizes 64ℓ, $1 \leq \ell \leq 2^{30}$, is computed as follows:

$$\mathsf{EHC}^{(d)}((K,(V_1,V_2));x) = \mathsf{EHC}^{(d,\ell)}(K;x) \oplus b_1 \cdot V_1 \oplus b_2 \cdot V_2\,,$$

where $x \in s^{64\ell}$, $K \in \{0,1\}^{64(\ell+d-1)}, V_1, V_2 \in \{0,1\}^{32d}$ and $(b_1, b_2) \in \{0,1\}^2$ is the binary representation of $\ell \mod 4$.

Example 1 (continued). We have already seen how $\mathsf{ECCode}_4(x_1,x_2,x_3) = (x_1,x_2,x_3,x_4,x_5,x_6)$ has been defined. Let $k = (k_1,\ldots,k_6) \in (\{0,1\}^{64})^6$ be the corresponding key. For $1 \leq i \leq 6$, denote, $x_i = x_{i1}||x_{i2}$ and $k_i = k_{i1}||k_{i2}$ with $x_{i1}, x_{i2}, k_{i1}, k_{i2} \in \{0,1\}^{32}$. For $1 \leq i \leq 6$, denote $g_i = \mathsf{PDP}(x_i, k_i) = (x_{i1} + k_{i1})(x_{i2} + k_{i2})$. Thus, $\mathsf{EHC}^{(4,3)}(k; x_1, x_2, x_3) = (o_1, o_2, o_3, o_4)$ where,

- $o_1 = g_1 + \ldots + g_5 + g_6$, $o_2 = \alpha^5 g_1 + \ldots + \alpha g_5 + g_6$,
- $o_3 = \alpha^{10} g_1 + \ldots + \alpha^2 g_5 + g_6$ $o_4 = \alpha^{15} g_1 + \ldots + \alpha^3 g_5 + g_6$

Algorithm $\mathsf{EHC}^{(d,\ell)}$
Input: $(k_1, \ldots, k_{\ell+d-1}) \in \{0,1\}^{64(\ell+d-1)}, x \in \{0,1\}^{64\ell}$

1 $(x_1, \ldots, x_{\ell+d-1}) \leftarrow \mathsf{ECCode}_d(x)$;
2 `for` $i = 1$ `to` $\ell + d - 1$: $g_i = \mathsf{PDP}(k_i, x_i)$;
3 `return` $\mathsf{VMult}_{\alpha,d}(g_1, g_2, \ldots, g_{\ell+d-1})$;

Algorithm 2: $\mathsf{EHC}^{(d,\ell)}$ [32] hash for a fixed length message.

3.3 Discussions

ECCode_4 and ECCode_5 are MDS codes for $\ell \leq 2^{32}$ and $\ell \leq 2^{30}$ respectively (see [32] and Proposition 1). To incorporate arbitrary length messages, we define ECCode^*_d as follows. It first parses $x \in \mathbb{F}^+_{2^{64}}$ as $(X_1, \ldots, X_{\mathsf{m}})$ such that all X_i's, possibly excluding the last one, are 2^{30}-block elements. We call these X_i's **chunk**. The last one is possibly an incomplete chunk. Next, apply ECCode_d to all of these chunks individually. More formally,

$$\mathsf{ECCode}^*_d(x) = (\mathsf{ECCode}_d(X_1), \ldots, \mathsf{ECCode}_d(X_{m-1}),\ \mathsf{ECCode}_d(X_m))\,. \qquad (2)$$

We next extend the definition of $\mathsf{EHC}^{(d,\ell)}$, denoted as $\mathsf{xEHC}^{(d,\ell)}$, which works the same as $\mathsf{EHC}^{(d,\ell)}$ except it runs ECCode^*_d instead of ECCode_d (line 1 of Algorithm 2), i.e., the first step is executed as $(x_1, \ldots, x_{\ell+d-1}) \leftarrow \mathsf{ECCode}^*_d(x)$.

Table 1. # 32-bit field multiplications needed for EHC-Hash, Toeplitz-Hash and Poly-Hash (with $d = 4$) to process a 64ℓ-bit message, $\ell \leq 2^{30}$. In case of Poly-Hash we need to apply 40-bit field multiplications for a 160-bit hash.

Tag Size	d	# Multiplications EHC Hash	# Multiplications Toeplitz-PDP Hash	# Multiplications Poly-Hash
128	4	$\ell + 3$	4ℓ	4.5ℓ
160	5	$\ell + 4$	5ℓ	4.5ℓ [40]

Comparison with $\mathsf{EHC}^{(d)}$ for arbitrary length and other hashes. We have chosen EHC hash as it requires much less multiplications than others (see Table 1). We have modified the processing of EHC for variable length messages. EHC uses a fixed length dependent key to deal with variable length messages, and the key needs to be stored, but we generate all the keys in run-time through the stream cipher so we do not need to store it. To achieve authenticity, one needs to apply a pairwise independent hash. By adding a length dependent key to the output of a Universal hash we can construct a Δ-Universal hash. Construction of

a pairwise independent hash can be achieved by masking one more independent key to the output of a Δ-Universal hash. However, as we generate keys on the fly, our hash becomes pairwise independent and this further saves more storage. We provide a detailed discussion of hardware implementation along with the block diagram in Sect. 6.

4 TriviA Authenticated Encryption

We first propose a stream cipher TriviA-SC[4] which has a similar design as the popular stream cipher Trivium [16]. Trivium is well studied and efficient both in terms of hardware and software. It uses an 80-bit secret key, an 80-bit nonce and a 288-bit internal state and provides 80-bit security. We aim to provide higher security while maintaining the simplicity and without increasing the state size much. In particular, we have made the following modifications:

1. We keep the size of state S to be 384 bits and increase the size of key K and nonce N to 128 bits.
2. We introduce a non-linear effect in the key stream computation.

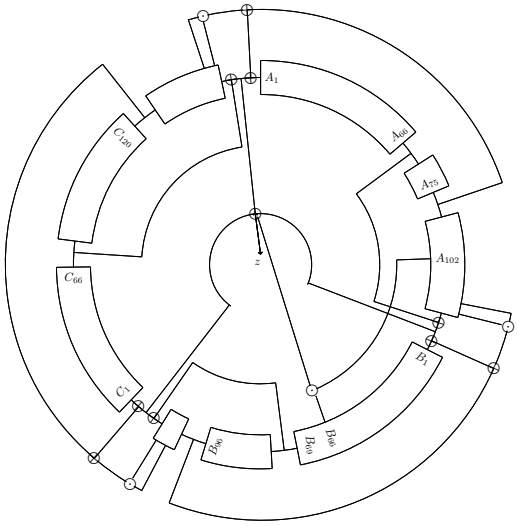

Fig. 4.1. TriviA-SC Stream Cipher

[4] Our authenticated encryption TriviA (a shorthand notation for Trivium-Authenticated Encryption) is based on the stream cipher TriviA-SC.

Modules of TriviA-SC: The state $S := (S_1, S_2, \ldots, S_{384}) \in \{0,1\}^{384}$ is represented by $A = (S_1, \ldots, S_{132})$, $B = (S_{133}, \ldots, S_{237})$ and $C = (S_{238}, \ldots, S_{384})$.

Load (K, N) / * **Key and IV Loading** * /

1 $A = K \| 1^4, \quad B = 1^{105}, \quad C = N \| 1^{19}$;

Update(S) / * **Update a Single Round** * /

2 $t_1 \leftarrow A_{66} \oplus A_{132} \oplus (A_{130} \wedge A_{131}) \oplus B_{96}$;

3 $t_2 \leftarrow B_{69} \oplus B_{105} \oplus (B_{103} \wedge B_{104}) \oplus C_{120}$;

4 $t_3 \leftarrow C_{66} \oplus C_{147} \oplus (C_{145} \wedge C_{146}) \oplus A_{75}$;

5 $(A_1, A_2, A_3, ..., A_{132}) \leftarrow (t_3, A_1, A_2, ..., A_{131})$;

6 $(B_1, B_2, B_3, ..., B_{105}) \leftarrow (t_1, B_1, B_2, ..., B_{104})$;

7 $(C_1, C_2, C_3, ..., C_{147}) \leftarrow (t_2, C_1, C_2, ..., C_{146})$;

Algorithm 3: Modules of TriviA-SC: Note that, the update and key-extract module has parallelism and so we can run KeyExt and Update 64 times in one clock-cycle which would be denoted as KeyExt64 and Update64 described in Algorithm 4.

Algorithm 3 describes all the basic modules used for the stream cipher TriviA-SC (see Fig. 4.1). A proper integration of these modules need to be defined to obtain a stream cipher or an authenticated encryption. For example, when we want to use it in stream cipher mode, we first run Load(K, IV) for an initial value IV, then Update for some reasonable rounds (to make the state random) and finally, both KeyExt and Update to obtain the key stream. However, in case of authenticated encryption we additionally need to process the associate data and need to produce a tag.

Trivia-SC is also parallelizable up to 64 bits, i.e., the stream cipher can produce upto 64 output bits at a single clock cycle (described in KeyExt64). Similarly, the 64 round updates of Trivia-SC can also be computed in a single clock cycle (described in Update64) due to the parallelism. KeyExt64 and Update64 are described in Algorithm 4.

4.1 Specification of TriviA

Algorithm 5 describes our authenticated encryption algorithm TriviA.

Sarkar in [33] has proposed several generic methods of combining ΔU hash and a stream cipher SC_K. Formally, a stream cipher supporting an n-bit initial value IV is a keyed function $SC_K : \{0,1\}^n \times \mathbb{N} \to \{0,1\}^+$ such that $SC_K(N; \ell) \in \{0,1\}^\ell$. Whenever understood, we simply skip ℓ as an input for the sake of notational simplicity. We mention a scheme which is close to our design paradigm and state its security guarantee (in a revised and simplified form appropriate to our notation).

Theorem [33] *Suppose H_τ is an ϵ-ΔU n-bit hash function and SC_K is a stream cipher. Let AE be an authenticated encryption scheme defined as*

$$\mathsf{AE}_{K,\tau}(N, D, M) = (C := M \oplus Z, \quad T := H_\tau(M) \oplus R)\,,$$

KeyExt64 / $*$ **Extract 64 Bit Key Stream** $*$ /

$\mathtt{t} = A_{[3\cdots66]} \oplus A_{[69\cdots132]} \oplus B_{[6\cdots69]} \oplus B_{[42\cdots105]} \oplus C_{[3\cdots66]} \oplus C_{[84\cdots147]} \oplus A_{[39\cdots102]} \wedge B_{[3\cdots66]}$;

Output t ;

Update64 / $*$ **Update 64 Rounds** $*$ /

$t_1 \leftarrow A_{[3\cdots66]} \oplus A_{[69\cdots132]} \oplus (A_{[67\cdots130]} \wedge A_{[68\cdots131]}) \oplus B_{[33\cdots96]}$;

$t_2 \leftarrow B_{[6\cdots69]} \oplus B_{[42\cdots105]} \oplus (B_{[40\cdots103]} \wedge B_{[41\cdots104]}) \oplus C_{[57\cdots120]}$;

$t_3 \leftarrow C_{[3\cdots66]} \oplus C_{[84\cdots147]} \oplus (C_{[82\cdots145]} \wedge C_{[83\cdots146]}) \oplus A_{[12\cdots75]}$;

$(A_1, A_2, A_3, ..., A_{132}) \leftarrow (t_3, A_1, A_2, ..., A_{68})$;

$(B_1, B_2, B_3, ..., B_{105}) \leftarrow (t_1, B_1, B_2, ..., B_{41})$;

$(C_1, C_2, C_3, ..., C_{147}) \leftarrow (t_2, C_1, C_2, ..., A_{83})$;

Algorithm 4: 64-bit modules of Trivia-SC. Here $\wedge$ denotes "bitwise-and" of two 64-bit variables.

Algorithm TriviA

Input: $(K, (N, D, M)) \in \{0,1\}^{128} \times (\{0,1\}^{128} \times \{0,1\}^* \times \{0,1\}^*)$, $\ell_M, \ell_D \leq 2^{30}$.

Processing N : Load(K, N), Update64 18 times;

Processing D : $(z, SK) \leftarrow$ KeyGen$(\ell_D + 4)$;

$T' = \mathsf{EHC}^{(5,\ell_D)}(SK\ ; \overline{D}) \oplus (z_{\ell_D+2} \| z_{\ell_D+3} \| z_{\ell_D+4}[1..32])$;

$S[1..160] = S[1..160] \oplus T'$, Update64 18 times;

Processing M : $(z, SK) \leftarrow$ KeyGen$(\ell_M + 3)$;

`if` 64 `divides` $|M|$ `then` $V = z_{\ell_M} \| z_{\ell_M+2}$;

`else` $V = z_{\ell_M+1} \| z_{\ell_M+3}$;

$C = M \oplus z$, $T = \mathsf{EHC}^{(4,\ell_M)}(SK\ ; \overline{M}) \oplus V$;

`return` (C, T) ;

Module KeyGen(ℓ):

`for` $i = 1$ `to` ℓ: $z_i =$ KeyExt64, $SK_i = A[1..64]$, Update64 ;

`return` $(z_1 \| \cdots z_\ell, SK_1 \| \cdots SK_\ell)$;

Algorithm 5: TriviA Authenticated Encryption Scheme: Given a binary string x, we define $\overline{x} := x \| 10^d$ where d is the smallest non-negative number such that $|x| + d + 1$ is a multiple of 64 and we write $\ell_x = |\overline{x}|/64$. The nonce N is chosen unique for each encryption. Here $C = M \oplus z$ means that we xor M with the first $|M|$ bits of z.

where $(R, Z) = SC_K(H_\tau(N, A),\ |M| + n)$. *Then, we have*

1. $\mathbf{Adv}^{\mathrm{prf}}_{AE}(q,\ \sigma,\ t) \leq \mathbf{Adv}^{\mathrm{prf}}_{SC}(q,\ \sigma,\ t') + q^2\epsilon$ *and*
2. $\mathbf{Adv}^{\mathrm{auth}}_{AE}(q,\ q_f,\ \sigma,\ t) \leq \mathbf{Adv}^{\mathrm{prf}}_{SC}(q+1,\ \sigma,\ t') + (1+q^2)\epsilon$.

Where $t' \approx t + t_H$ *and* t_H *is the total time required for hashing all queries.*

How Our Construction Differs from $\mathrm{AE}_{K,\tau}$. The above construction requires two keys K and τ. In our construction, we generate the key τ from the stream cipher and hence we require only one key K. As the stream cipher generates run time output bit stream, we can apply those universal hash functions requiring variable length keys, which are more efficient than those hash functions based on a single small key. For example, Poly-Hash [14, 15, 34] is not as hardware efficient as EHC and provides a weaker security bound.

4.2 Discussions

Authenticated Encryption for larger message/associate data. We can further extend TriviA for computing the intermediate data and the tag for arbitrary length message and associated data. A brief description of the extended version of TriviA is given as follows. The extended algorithm of TriviA for handling larger messages is functionally almost the same as TriviA, except that it uses $\mathsf{xEHC}^{(d,\ell)}$ to compute the intermediate data (line 3 of Algorithm 5) and the tag, and the KeyGen algorithm will generate keystream according to the length of the codeword computed by ECCode^*_d. The algorithm also selects the part of the key z that appears in the same clock cycles with the message blocks so that we do not need to hold the key. This part of z is xored with the message to produce the ciphertext as before.

Nonce Misuse Scenario. We generalize the EHC hash to incorporate 160 bits hash for processing associate data. This would allow some room for repetition of the nonce (but no repetition of nonce, associated data pair) without degrading the security (see the privacy and authenticity bound for TriviA in Sect. 5).

5 Security Analysis

5.1 Security Against Known Attacks

Cube-Attack and Polynomial Density: The Cube Attack [17] is the best known algebraic attack on reduced round versions of Trivium [16]. Note that the output bits from the stream cipher can be described by a polynomial over the key and the nonce bits. The cube attack tries to analyze the polynomial $P(k_1, \cdots, k_n; iv_1, \cdots, iv_p)$ corresponding to the first output bit, where $k_1, \cdots, k_n$ are the secret key bits and $iv_1, \cdots, iv_p$ are the public nonce bits. Given a subset $S = \{iv_{v_1}, \cdots, iv_{v_k}\}$ of the set of all public nonce bits, P can be written as $P = iv_{v_1} \cdots iv_{v_k} P_S + P_R$, where no monomial of P_R is divisible by $iv_{v_1} \cdots iv_{v_k}$. P_S is called the *superpoly* yielded by S and $iv_{v_1} \cdots iv_{v_k}$ is called the *maxterm* if

Table 2. The estimation of monomial densities of TriviA-SC with 1152, 960, 896 and 832 Rounds.

Monomial Size	25	26	27	28	29
1152	0.49	0.49	0.5	0.52	0.4
960	0.5	0.5	0.5	0.51	0.36
896	0.5	0.49	0.5	0.47	0.5
832	0.43	0.36	0.29	0.14	0.03

P_S is linear. The TriviA-SC with the recommended 1152-rounds initialization has no maxterm of size less than or equal to 29. Moreover, for the 896 and 832-round initialization version we have not found any maxterm of size 29 or less. But for the 768-round initialization version we have found some linear superpoly with cube size 20. This justifies our recommendation of the 1152-round initialization for TriviA-SC. We have also applied the Moebious Transform technique described by Fouque et al. [20] to estimate the polynomial density of the output boolean function. We restrict polynomial to 30 IV variables and the density of the monomials of degree less than 30 in the restricted polynomial has been calculated. The result is given in Table 2. For a random Boolean function, we expect 50% density. The statistical tests on TriviA-SC have been performed by observing the output bit stream using the NIST Test Suite [6] and no weeknesses were found. We have also performed the same tests on a version of TriviA-SC where the key is a random 384-bit string and no weaknesses were found.

Resistance against guess-then-find attack [27]: The attack presented by Maximov et al. [27] works in two phases. The first phase guesses some internal state and makes linear approximations of some of the nonlinear state updation. This would help to produce a set of linear equations (and also several second degree equations) on the unguessed state bit using the observed output stream. In the second phase we simply solve all state bits provided we have sufficient number of equations. This idea is applicable for both reduced round versions of Trivium and Trivia-SC.

One possible approach of the first phase for the Trivium is an exhaustive guess on one-third of the state (96 bits with the indices that are a multiple of 3 stored in a set $\tau_0^{(t)}$ out of 288 bits). As the output bits are linear in the state bits, it is sufficient to guess 72 state bits and the remaining 32 state bits can be recovered easily. This actually happens, as indices of the bits in the output polynomial are multiple of 3 as well as the lifetime of a state bit in the internal state is at least 66 rounds before it is mixed with other bits. Using this guesses, we can obtain $n_1 = 100$ linear equations and $n_2 = 61$, 2-degree equations on the remaining internal state bits by observing the output stream. So the complexity for the second phase, denoted c, would be costly as we do not have sufficient linear equations.

There is an optimized version of the first phase which further makes linear approximation of the nonlinear terms in the state update functions to construct

several other linear equations on the state bits in $\tau_0^{(t)}$. The complexity of the first phase for this version of the attack is $2^{83.5}$ and it forms $n_1 = 192$ linear equations. Thus, the complexity in the second phase would be small.

Unlike Trivium, TriviA-SC has a nonlinear function in the output stream so to obtain $n_1 = r$ linear equations one has to approximate r nonlinear equations ("AND" gate). In fact, as long as $r \leq 96$, the indices involved in these linear approximations are completely disjoint. Thus, the probability that all of these linear approximations hold is $(3/4)^r$. Now if we follow a similar approach mentioned above, we first make a guess of one-third of the internal state (128 bits out of 384). However, one can simply guess 106 state bits and the remaining 22 bits can be recovered from the output stream. As the output is nonlinear, we have to make a linear approximation for 22 round outputs. So the complexity of the first phase would be about $2^{128-22} \times (4/3)^{22}$. Now if we want to obtain $n_1 = 32$ linear equations for the second phase (which is in fact much less than sufficient linear equations to recover all unguessed state bits), the total complexity for the first phase becomes $2^{106} \times (4/3)^{22} \times (4/3)^{32} > 2^{128}$. So we can not perform the above guess-then-find attack strategy in our stream cipher.

5.2 Privacy of TriviA

An adversary is called nonce-respecting if it makes encryption queries with all distinct nonce. A relaxed nonce-respecting adversary makes queries such that the pairs (N, D) are distinct over all q encryption queries. In the following two theorems we assume that the stream cipher Trivia-SC generates a pseudorandom bit stream.

Theorem 1. *Let A be a relaxed nonce-respecting adversary which makes at most q encryption queries. Moreover we assume that A can make at most 2^{32} queries with a same nonce. Then,* $\mathbf{Adv}^{\mathrm{priv}}_{\mathit{TriviA}}(A) \leq \frac{q}{2^{129}}$.

Proof. Suppose A makes q queries $(N_1, D_1, M_1), \ldots, (N_q, D_q, M_q)$ such that (N_i, D_i)'s are distinct and let Z_i and (C_i, T_i) be the respective key stream (including the state bit extraction) and final responses. Moreover let T_i' denote the intermediate tag obtained from the associated data which are inserted in the state after processing associated data. Let $\mathcal{N} = \{N : \exists i \ N = N_i\}$ denote the set of all distinct nonces. We denote $m = |\mathcal{N}|$. For each $N \in \mathcal{N}$, we write $\mathcal{I}_N = \{j : N_j = N\}$ and $|\mathcal{I}_N| = q_N$. Note that, $q_N \leq 2^{32}$ for all nonces N and $\sum_{N \in \mathcal{N}} q_N = q$. By our assumption on the stream cipher output, the key stream Z_i's would be independently distributed whenever we have distinct nonces. Thus, we define an event coll: there exists $i \neq j$ such that $N_i = N_j$ and $T_i' = T_j'$. If the coll event does not hold, then by using the ideal assumption of the stream cipher, all key streams Z_i's (even with same nonce) are independent and uniformly distributed. As (C_i, T_i)'s are injective functions of the key-stream Z_i, the distribution of (C_i, T_i)'s are independent and uniform. So the privacy advantage is bounded by the probability of the event coll. In Proposition 2 below, we show that the collision probability is bounded above by $\frac{q}{2^{129}}$ and hence the result follows. □

Proposition 2. $Pr[coll] \leq \frac{q}{2^{129}}$.

Proof. Fix a nonce $N \in \mathcal{N}$. The probability that there exists $i \neq j$ with $N_i = N_j = N$ such that $T'_i = T'_j$ is bounded by $\binom{q_N}{2} \times 2^{-160}$. This actually holds as this collision implies that $\mathsf{EHC}^5(D_i) = \mathsf{EHC}^5(D_j)$ and EHC^5 is a 2^{-160}-ΔU hash (the underlying code ECCode_5 is MDS as shown in Proposition 1). Summing up the probability for all choices of nonce N, we have

$$Pr[\mathsf{coll}] = \sum_{N \in \mathcal{N}} q_N^2/2^{161} \leq 2^{32} \sum_{N \in \mathcal{N}} q_N/2^{160} = q/2^{129} .$$

5.3 Authenticity of TriviA

Now we show the authenticity of TriviA.

Theorem 2. *Let A be a relaxed nonce-respecting adversary which makes at most q queries such that nonce can repeat up to 2^{32} times. In addition, A is making at most q_f forging attempt. If the stream cipher Trivia-SC is perfectly secure then*

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathit{TriviA}}(A) \;\leq\; \frac{q}{2^{129}} + \frac{q_f}{2^{124}} .$$

Proof. As before, let the q queries be (D_i, N_i, M_i) and the corresponding responses be (C_i, T_i) with intermediate tags T'_i, $1 \leq i \leq q$. We also denote the key stream for the ith query be Z_i. By applying the privacy bound, which is $q/2^{129}$, we may assume that the all q key streams $Z_1, \ldots, Z_q$ are uniformly and randomly distributed. Now we consider two cases depending on a forging attempt (N^*, D^*, C^*, T^*).

Case A. The adversary makes a forging attempt (N^*, D^*, C^*, T^*) with a fresh (N^*, D^*). In this case, let $\mathcal{I} = \{i : N_i = N^*\}$. By the restriction, $|\mathcal{I}| \leq 2^{32}$. Note that for all $j \notin \mathcal{I}$, the Z_j's are independent from Z^* the key-stream for the forging attempt. For all $i \in \mathcal{I}$, the Z_i's also would be independent from Z^* provide that the intermediate tag T'_i's do not collide with the intermediate tag T^* for the forging attempt. This can happen with probability at most $2^{32}/2^{160} = 2^{-128}$. Whenever Z^* behaves like a random string, the forging probability will be 2^{-128} (as the tag size is 128). So the total forging probability, in this case, will be at most $2^{-128} + 2^{-128} = 2^{-127}$.

Case B. Suppose the adversary makes a forging attempt with $(N^*, D^*) = (N_i, D_i)$ for some i. Note that one of the key-streams Z_i and Z^* would be a prefix of the other (depending on the length of the ciphertext). Note that for all other Z_j's, $j \neq i$ would be independent of Z^* and so we can ignore the responses of the other queries. So the forging probability is the same as

$$p := \mathsf{Pr}[(N_i, D_i, C^*, T^*) \text{ is valid } \mid (C_i, T_i) \text{ is response of } (N_i, D_i, M_i)] . \quad (3)$$

Claim $p \leq 2^{-124}$.

We postpone the proof of the claim. Assuming this claim, any forging attempt is successful with probability at most 2^{-124} (as the Case-A has lower success probability). Since A makes at most q_f attempts and adding the privacy advantage the forging probability would be bounded by $\frac{q}{2^{129}} + \frac{q_f}{2^{124}}$. This completes the proof.

Proof of the Claim. Let M^* be the message corresponding to C^*. We prove it by considering different cases based on $\ell := \ell_{M_i}$ and $\ell^* := \ell_{M^*}$. For simplicity, we assume that both M_i and M^* are complete block messages. The proof for incomplete message blocks is similar. We also write Z into a pair (SK, z) where SK denotes the state key and z denotes the output stream. Note that, the z-values can be leaked through the ciphertext and some of the z-values may be also used to compute the tag. We mainly need to handle different cases depending on how the z-values are leaked.

Case 1: $\ell^* = \ell$ In this case, the conditional forging event can simply be written as $\mathsf{EHC}^{4,\ell}(SK; M_i) \oplus \mathsf{EHC}^{4,\ell}(SK^*; M^*) = \delta := T_i \oplus T^*$. As, $\ell^* = \ell$, thus $SK = SK^*$. By using the known fact that EHC is a 2^{-128}-ΔU hash [32] we have $p \leq 2^{-128}$.

For the case 2 and 3, we denote, $\mathsf{EHC}^{4,\ell}(z; M_i) = (H_1, H_2)$ and similarly $\mathsf{EHC}^{4,\ell^*}(z^*; M^*) = (H_1^*, H_2^*)$ where H_i and H_i^*s are 64-bit strings. We similarly parse T and T^* as (T_1, T_2) and (T_1^*, T_2^*).

Case 2: $\ell^* = \ell + 1$ In this case, all of the variable keys z are distinct and are not leaked through the ciphertext C_i. So the forging probability is equivalently written as

$$p = \mathsf{Pr}[H_1^* \oplus z_{\ell+1} = T_1^*, H_2^* \oplus z_{\ell+3} = T_2^* \mid H_1 \oplus z_\ell = T_1, H_2 \oplus z_{\ell+2} = T_2].$$

Thus, by using the entropy of $z_\ell, z_{\ell+1}, z_{\ell+2}$ and $z_{\ell+3}$, we get the bound.

Case 3: $\ell^* > \ell + 1$ Except the case $\ell^* = \ell + 2$, this case is same as before as all variable keys z are distinct and are not leaked through the ciphertext C_i. When $\ell^* = \ell + 2$ we have three variable keys $z_\ell, z_{\ell+2}$ and $z_{\ell+4}$ which are masked to define the tags. So the forging probability is equivalently written as

$$p = \mathsf{Pr}[H_1^* \oplus z_{\ell+2} = T_1^*, H_2^* \oplus z_{\ell+4} = T_2^* \mid H_1 \oplus z_\ell = T_1, H_2 \oplus z_{\ell+2} = T_2].$$

The independence of the z values implies,

$$\begin{aligned} p &= \mathsf{Pr}[H_2^* \oplus z_{\ell+4} = T_2^*] \times \mathsf{Pr}[H_1^* \oplus H_2 = T_1^* \oplus T_2 := \delta] \\ &= 2^{-64} \times \mathsf{Pr}[H_1^* \oplus H_2 = T_1^* \oplus T_2 := \delta]. \end{aligned}$$

Now the effect of the state keys $SK_{\ell+4}, SK_{\ell+5}$ is not present in H_1^* but they influence H_2. By using 2^{-31}-balancedness of the pseudo-dot-product hash, we conclude that $\mathsf{Pr}[H_1^* \oplus H_2 := \delta] \leq 2^{62}$ and so $p \leq 2^{-126}$.

Case 4: $\ell^* < \ell - 2$ In this case, the variable keys are different for both computations. Since one set of variable keys are leaked through the ciphertext and the other has full entropy we use the fact that EHC is 2^{-124}-balanced. Using this one can show that $p \leq 2^{-124}$.

Case 5: $\ell^* = \ell - 1$ Again, all four variable keys are distinct and one of them is leaked. So we can apply the argument (using balancedness of one 64-bit equation) to show that $p \leq 2^{-126}$.

Case 6: $\ell^* = \ell - 2$ Again, by simplifying the forging event with the notation described in case 3 we have

$$p = \mathsf{Pr}[H_1^* = z_{\ell-2} \oplus T_1^*, H_2^* \oplus z_\ell = T_2^* \mid H_1 \oplus z_\ell = T_1, H_2 \oplus z_{\ell+2} = T_2].$$

Here note that, unlike in case 2, the value of $z_{\ell-2}$ is leaked in the ciphertext. The above probability is the same as $\mathsf{Pr}[H_1^* = c_1, H_2^* \oplus H_1 = c_2]$ for some 64-bit constants c_1 and c_2. Based on the balanced property of H_1^* (based on the state key $SK_1, \ldots, SK_{\ell+1}$) and the balanced property of H_1 (based on the state key $SK_{\ell+2}, SK_{\ell+3}$) we can conclude that $p \leq 2^{-124}$.

By considering all the above cases, we prove that $p \leq 2^{-124}$ which concludes the proof of the claim. □

6 Hardware Implementation of TriviA-ck

6.1 Cycles Per Byte (cpb) Analysis

The TriviA design targets high speed implementation and requires 47 clock cycles to authenticate and encrypt one message block of 64 bits. 18 cycles are required for the initialization phase where the state register is updated in every cycle along with Z, the associated data AD is loaded and processed in 1 cycle, and during the checksum phase instead of loading the block, the checksum computed in an earlier stage is used as the input. The overall computation requires 4 cycles and an additional cycle is required to update the tag and the state register during the processing of AD. For message (msg) encryption, again, the same number of cycles are required but AD is replaced by msg. The rest of the process is the same with one minor difference, now the checksum is calculated only 3 times instead of 4 before the tag update. Analytically, the cycle count can be represented in the following manner

$$cycle\ count = (init\_count * 2) + \frac{adlen}{8} + \frac{msglen}{8} + 4 + 1 + 3 + 1\,, \quad (4)$$

where $init\_count$ is 18 in TriviA, $adlen$ and $msglen$ are in bytes instead of bits. The corresponding cpb can be calculated using the following formula

$$cpb = \frac{cycle\ count}{msglen}\,. \quad (5)$$

Pipelining: So far, our analysis is done based on a design without any pipeline. Pipelining is a well-known technique to improve the throughput for a digital design. A three-stage pipelined design is employed for TriviA, which is explained later in this section. Pipelining affects latency adversely. In our case, the cycle count to authenticate and encrypt one message block of 64 bits increases to 49. Two additional clock cycles are required to flush the pipeline registers. The rest of the data processing flow remains the same. Similarly the cycle count for a pipelined design can be represented in the following manner.

$$cycle\ count = \ (init\_count * 2) + \frac{adlen}{8} + \frac{msglen}{8} + 9 + (pipe\_stages - 1)\,, \quad (6)$$

where *pipe_stages* is equal to 3 in our case. As the number of *pipe_stages* increases, the corresponding cycle count will increase accordingly. The *cpb* can be calculated using Eq. (5).

6.2 Hardware Architectures

We have implemented two different architectures of TriviA: a base implementation without any pipelining, and a three-stage pipelined implementation. The implementation is performed in a modular manner, which offers excellent scalability. Due to the similarity in the operations for processing AD, and msg, the same hardware modules are used to process both kinds of data. A single bit switch is used to distinguish between the type of input data. Following the different operations in TriviA algorithm, the TriviA architecture consists of the following modules:

1. *State Registers*: The state registers are used to store the intermediate states after each iteration. The state registers are used for 384-bit State_Update, 256-bit Z register, 64-bit *block*, 160-bit *tag*, and 256-bit *checksum*.
2. State_Update: The State_Update module is nothing but a combination of Updated64, KeyExt64 which are used to update the current state of the stream cipher and generate key-stream. This module is used in each iteration during initialization, encryption, and finalization. It takes 128-bit key, 128-bit nonce (which is further divided into two 64-bit parts, namely pub and param) and 384-bit stream cipher state register as inputs and updates the stream cipher state.
3. Field Multiplication: The Field multiplication module takes two 32-bit inputs, calculates the pseudo dot product on the input, and produces a 32-bit output.
4. VHorner32: This module is used for Horner's multiplication for the 32-bit Vandermonde Matrix Multiplication (i.e. in the computation of $\mathsf{VMult}_{\alpha,5}$). It takes two inputs, a 32-bit value from the field multiplication, and 160-bit tag value. It processes the input to generate a new tag of 160-bits. During the processing of AD, it processes all the 160-bits of the tag to give the output, whereas, for the message processing, only 128 bits are used.

5. VHorner64: This module is used for Horner's multiplication for the 64-bit Vandermonde Matrix Multiplication while computing the checksum for the error correcting code. It takes an input block of 64-bits, and the current checksum value of 256 bits as input. It generates a 256-bit checksum value as output. This modules executes its operations on 256 bits of the checksum when working on AD, otherwise it uses only 192 bits.

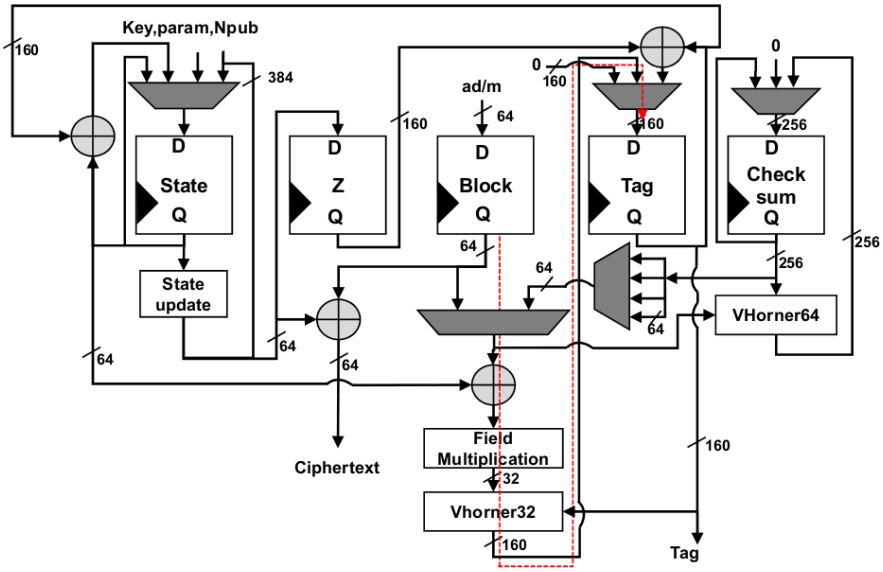

Fig. 6.1. TriviA basic hardware implementation

Base Implementation: We start with a base implementation of TriviA that exploits the parallelism inherent in the algorithm and processes 64 bits in each cycle, as shown in Fig. 6.1. The critical path is shown using a dotted line. Prior to initialization, the state register is loaded with Key, param, pub and 1 in the remaining bits on reset. Once the state registers are initialized, the initialization process starts where *state_register* is updated in each cycle with State_Update operation. After the initialization process, 8 bytes of AD are fetched to the *block* register, which feeds the Field Multiplication module after performing an *exclusive OR* between the 64-bit block and the 64-bit *state_register*. In parallel, an exclusive OR is also performed between the 64-bit block and the 64-bit Z to produce the *ciphertext*. The Field Multiplication module is followed by the VHorner32 module which generates the new tag. The Field Multiplication module has 64 many 2×1 32-bit MUXes in series. The VHorner32 block diagram can be seen in Fig. 6.2.

The checksum is updated in the VHorner64 module which is also executed in parallel. When the AD is finished, the checksum is calculated hence we require a 4×1 64-bit MUX which fetches 64 bits of the checksum at a time to update the tag. Since the checksum is calculated at the end, we can share resources using a 2×1 64-bit MUX. The first input of the MUX comes from the *block* register, while the second input comes in chunks of 64 bits from the checksum. All these operations are parallel, hence one round can be executed in a single cycle. At the end, after processing the checksum, the *tag* and the *state_register* are updated in single cycle. We require a 4×1 MUX at the input of *state_register* and a 3×1 MUX at the input of the *tag* register. The *state_register* takes input from four sources, initialization values on reset, State_Update after each cycle, the result of an exclusive OR between the *state_register* and the *tag*, and a feedback path. Similarly, *tag* takes values from three sources, initialization on reset, output of VHorner32, and Z exclusive ORed with *tag*.

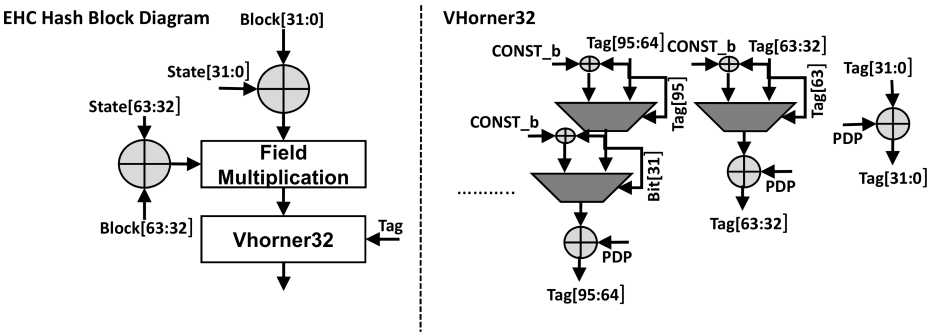

Fig. 6.2. EHC Block Diagram

The control of the complete design is implemented using a finite state machine (FSM), not shown in the schematic for the sake of simplicity. The FSM consists of 6 states, starting with an idle state followed by initialization. After initialization, FSM enters the processing state and stays there until all the data has been processed. Then it jumps to checksum processing, followed by tag update, and pipeline flush. A 3-bit register is required to store the state of the FSM.

The combinational logic can be easily reduced for lowering the area further by first, sharing computing resources and second, performing computation on smaller bit-widths. Consequently, the throughput will decrease leading to an area-delay trade-off. In the following, we implement a pipelined architecture to boost the throughput.

Pipelined Implementation: After analyzing the basic implementation, we identified the critical path, and split that to achieve higher operating frequency. All the operations are unit operations except for the *tag* generation. *Tag* generation requires two operations in series, which are using multiple MUXes in

series. This long chain of MUXes reduces the clock speed of the whole design, hence other modules which can operate at higher frequencies are also limited by this. We break the critical path and insert a pipeline register after the Field Multiplication module. To balance the design, we also insert a pipeline register after the Z register. Using the pipelined architecture, as shown in Fig. 6.3, we could achieve higher throughput for TriviA.

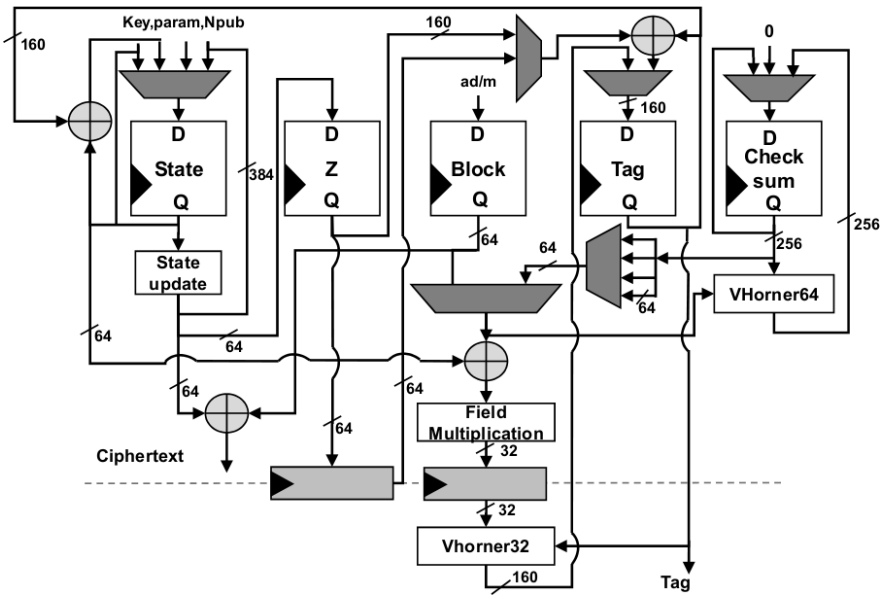

Fig. 6.3. TriviA pipelined hardware implementation

Enc/Dec Implementation: Due to the similar structure of encryption and decryption algorithms, a combined hardware can also be designed with a small increase in area, while getting the same throughput. The encryption or decryption mode is selected using a *mode_select* signal. When the *mode_select* is set to *0*, the hardware operates in encryption mode, whereas when *mode_select* is set to *1*, the hardware operates in decryption mode.

6.3 Performance Results and Comparison

TriviA Results. The architectures of TriviA are described in Verilog HDL and synthesis is done with the Synopsys Design Compiler J-2014.09 using Faraday standard cell libraries in topographical mode. We used UMC 65nm logic SP/RVT Low-K process technology node for synthesis. The implementation is performed till gate-level synthesis hence, the reported results are pre-layout. The area results are reported in terms of equivalent NAND gates. The area for the base

implementation of TriviA was 23.6 KGE at a frequency of 1150 MHz, with 7.2 KGE required for sequential logic and 16.4 KGE required for combinational logic. The corresponding throughput turns out to be 73.9 Gbps, and the area-efficiency (throughput/area) is 3.13 Mbps/GE. The area utilization is shown in Table 3 where each module and its respective area is shown. The registers for *Tag*, *block*, and *checksum* are instantiated in the top-module, hence their distribution is not listed in the table.

The synthesis for pipelined implementation was performed under similar operating conditions, tools, and libraries. The design was successfully synthesized at 1425 MHz for an area of 24.4 KGE, with 7.7 KGE in sequential and 16.7 KGE in combinational logic. The design successfully achieved a throughput of 91.2 Gbps with an area-efficiency of 3.73 Mbps/GE. The module-wise breakdown of area is shown in Table 3. The registers for *Tag*, *block*, *checksum*, and pipeline registers are instantiated in the top-module, hence their distribution is also not listed in the table. For performance measures, different message lengths

Table 3. Area utilization without pipeline stage, TriviA

Module	Base implementation		Pipelined implementation	
	Area (GE)	%	Area (GE)	%
Field Multiplication	6275	26.0	6890	28.0
Update State	7214	30.0	7208	29.5
FSM	1260	5.3	1296	5.3
VHorner32	675	2.8	387	1.5
VHorner64	573	2.4	576	2.4

were considered to calculate cycles per byte, taking associated data length as a small value of 8 bytes. When the message length is 8 bytes, the overhead of initialization is very large in both the cases, base implementation and pipelined implementation, giving a high cycles per byte count. As we increase the message length, the overhead becomes smaller resulting in 0.12 cycle per byte for a very long message. This is shown in the Table 4 and graphically in Fig. 6.4. Hence, as we increase the message length the cycles per byte of both designs converge. Note that, if we assume that the associated data is of the same length as the message, then the cycle per byte doubles.

Example. If both the *message* and *AD* are of 8192 bytes then 0.25 cycles per byte are required.

Comparison: The results for algorithms marked with a (*) in Table 5 have been scaled assuming a 2× improvement of achievable maximum clock frequency for every two generations of CMOS technology node, thus roughly following Moore's law [30]. Admittedly, this is a very rough comparison, without considering the

Table 4. TriviA performance (in clocks per byte or cpb)

Algorithm	Message length (Bytes)										
	8	16	32	64	128	256	512	1024	2048	8192	16384
TriviA-pipelined	6.12	3.12	1.62	0.87	0.50	0.31	0.21	0.17	0.14	0.13	0.12

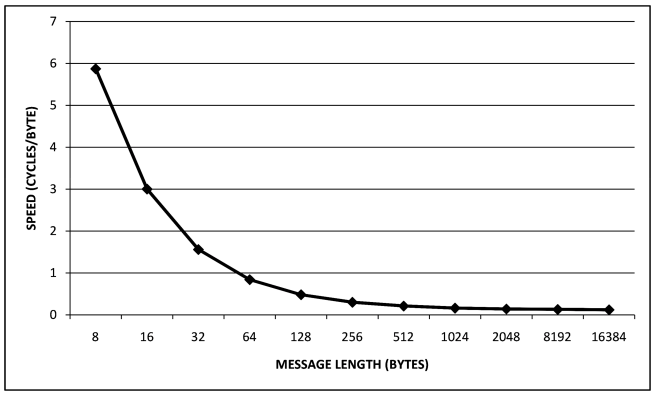

Fig. 6.4. Hardware performance of pipelined design of TriviA for different message lengths

effects of physical synthesis, diversity of cells in different technology libraries and synthesis constraints. The unavailability of the RTL code and the synthesis details in the presented papers makes the task even harder. Nevertheless, the performance gap is at least an order of magnitude in the area-efficiency, hence unlikely to close.

To the best of our knowledge, not all CAESAR candidates have hardware implementations in ASIC so far, and there is no other hardware implementation of TriviA as well, so, the comparison done with the known results listed in Table 5 shows that, the TriviA has a better throughput, cycles per byte and area-efficiency. Particularly, for the area-efficiency, the TriviA is at least 3.8 times better compared to the closest entry Ascon. We will offer the hardware implementation to the CAESAR candidates and also request corresponding RTL for a fair evaluation in similar technology settings.

Besides CAESAR candidates, it is also interesting to benchmark TriviA against state-of-the-art authenticated encryption engines that are commercially available. We take one particular example from [7], which provides high-speed authenticated encryption hardware based on AES-CCM mode. Based on the data provided at [7], for 130 nm CMOS technology, AES-CCM achieves > 800 Mbps at $< 19K$ gates. Even with an optimistic technology scaling, this is still one order of magnitude inferior compared to the performance achieved with the TriviA.

More detailed results are provided at [7] for different FPGA platforms. There, the performance results reported are for AES-CCM cores which require 128-bit

Table 5. Benchmarking TriviA in ASIC

AE Schemes		ASIC Implementation			Cycles/ Byte (cpb)
		Area (KGE)	Throughput (Gbps)	Efficiency (Mbps/ GE)	
TriviA Base		23.6	73.9	3.13	0.12
TriviA Pipelined		24.4	91.2	3.73	0.12
Scream, iScream [22]		17.29	5.19	0.30	-
NORX* [8]		62	28.2	0.45	-
Ascon* [21]		7.95	7.77	0.98	0.75
AEGIS [13, 37]	*AO1*	20.55	1.35	0.07	6.67
	AO2	60.88	37.44	0.61	0.33
	TO1	88.91	53.55	0.60	0.20
	TO2	172.72	121.07	0.70	0.07

key, and 48 cycles to generate output cipher text. In order to benchmark, we performed pre-layout logic synthesis done for TriviA under typical conditions using Xilinx ISE 14.7. The results are collectively reported in Table 6. It can be noted that the area occupied for TriviA is generally larger compared to [7], since the computation in TriviA is done in parallel, processing 64 bits in 1 cycle. In comparison, [7] requires 48 cycles. When comparing the area-efficiency figures, it is clear that TriviA is much superior by showing at least 5.4× improvement for TriviA pipelined implementation compared to AES-CCM implementation of [7].

Table 6. Benchmarking TriviA in FPGA

Xilinx FPGA Platform	AES-CCM [7]			TriviA-Base			TriviA-Pipelined
	# Slices	Gbps	Area-Efficiency (Mbps/ Slice)	# Slices	Gbps	Area-Efficiency (Mbps/ Slice)	Area-Efficiency (Mbps/ Slice)
Spartan-6 -3	272	>0.57	2.09	815	7.6	9.3	11.29
Virtex-5 -3	343	>0.78	2.27	637	11.7	18.3	20.3
Virtex-6 -3	295	>0.87	2.95	725	16	22	25
Kintex-7 -3	296	>1	3.38	714	16.89	23.65	24.31
Virtex-7 -3	296	>1	3.38	714	16.89	23.65	24.31

7 Conclusion

This paper introduces a hardware efficient authenticated encryption scheme TriviA. The structure of TriviA is simple and achieves high provable security. Our

proposal uses a stream cipher and a pairwise independent hash function. We have constructed a stream cipher TriviA-SC, which is a variant of a well known stream cipher Trivium [16]. Trivium is well studied and very efficient in both hardware and software. We have also used a hardware efficient Δ-Universal hash function EHC, which requires minimum number of field multiplications to process a message. We have integrated these two primitives in an efficient way, such that the resultant construction is highly efficient in both hardware and software as well as it provides high security of 128-bits for privacy and 124-bits for authenticity. This work provides the details of the hardware implementation of TriviA and hardware comparison between TriviA and some of the CAESAR candidates. We have observed that, TriviA is very hardware efficient in terms of throughput, cycles per byte and area-efficiency. More specifically the area-efficiency of TriviA is at least 3.8 times better than the closest CAESAR candidate Ascon.

Acknowledgement

Avik Chakraborti and Mridul Nandi are supported by the Centre of Excellence in Cryptology and R. C. Bose Centre for Cryptology and Security, Indian Statistical Institute, Kolkata. We would like to thank Bart Preneel for his detailed comments and suggestions on our paper. We would also like to thank the anonymous reviewers for their useful comments on our paper.

References

[1] ETSI/SAGE Specification, *Specification of the 3GPP Confidentiality and Integrity ALgorithms UEA2 and UIA2.* (2006), *Document 5: Design and Evaluation Report, Version 1.1.*, 2006. Citations in this document: §1.

[2] ETSI/SAGE Specification, *Specification of the 3GPP Confidentiality and Integrity ALgorithms UEA2 and UIA2.* (2006), *Document 2: SNOW 3G Specification*, 2006. Citations in this document: §1.

[3] CAESAR, *Competition for Authenticated Encryption: Security, Applicability, and Robustness.* (2014). URL: http://competitions.cr.yp.to/caesar.html. Citations in this document: §1.

[4] eSTREAM, *The ECRYPT Stream Cipher Project.* URL: http://www.ecrypt.eu.org/stream. Citations in this document: §1.

[5] International Organization for Standardization, *ISO/IEC 29192-3:2012, Information technology – Security techniques – Lightweight cryptography – Part 3: Stream ciphers* (2102). URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56426. Citations in this document: §1.

[6] National Institute of Standards and Technology, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* (2010). URL: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html. Citations in this document: §5.1.

[7] Helion Technology, *AES-CCM core.* URL: http://www.heliontech.com/aes_ccm.htm. Citations in this document: §6.3, §6.3, §6.3, §6.3, §6.3, §6.3, §6.

[8] J. P. Aumasson, P. Jovanovic, S. Neves, *NORX: Parallel and Scalable AEAD.* (2014), 19 – 36, *Computer Security-ESORICS*, Springer International Publishing, 2014, 2014. URL: https://eprint.iacr.org/2015/034.pdf. Citations in this document: §5.

[9] S. Babbage, M. Dodd, *The eSTREAM Finalists.* (2008), 191 – 209. Citations in this document: §1.

[10] M. Bellare, C. Namprempre, *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm.* (2000), 531–545, *ASIACRYPT*, Lecture Notes in Computer Science, 2501, 2000.

[11] D. J. Bernstein, *The Poly1305-AES Message-Authentication Code.* (2005), 32–49, *FSE*, Lecture Notes in Computer Science, 3557, 2005. Citations in this document: §1.

[12] D. J. Bernstein, *Cycle counts for authenticated encryption.* (2007), *Workshop Record of SASC 2007: The State of the Art of Stream Ciphers*, 2007. Citations in this document: §1.

[13] D. Bhattacharjee, A. Chattopadhyay, *Efficient Hardware Accelerator for AEGIS-128 Authenticated Encryption.* (2014), 385–402, *Inscrypt*, Lecture Notes in Computer Science, 8957, 2014.

[14] J. Bierbrauer, T. Johansson, G. Kabatianskii, B. Smeet, *On Families of Hash Functions via Geometric Codes and Concatenation.* (1993), 331–342, *CRYPTO*, Lecture Notes in Computer Science, 773, 1993.

[15] B. den Boer, *A simple and key-economical unconditional authentication scheme.* (1993), 65 – 72, *Journal of Computer Security 2*, 1993.

[16] C. D. Cannière, B. Preneel, *"Trivium," New Stream Cipher Designs.* (2005), 244–266, *The eSTREAM Finalists*, Lecture Notes in Computer Science, 4986, 2005. Citations in this document: §1, §1, §4, §5.1, §7.

[17] I. Dinur, A. Shamir, *Cube Attacks on Tweakable Black Box Polynomials.* (2009), 278–299, *EUROCRYPT*, Lecture Notes in Computer Science, 5479, 2009. Citations in this document: §5.1.

[18] X. Fan, G. Gong, *Specification of the Stream Cipher WG-16 Based Confidentiality and Integrity Algorithms.* (2013). URL: http://cacr.uwaterloo.ca/techreports/2013/cacr2013-06.pdf.

[19] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, T. Kohno, *Fast Encryption and Authentication in a Single Cryptographic Primitive.* (2003), 330–346, *FSE*, Lecture Notes in Computer Science, 2887, 2003. Citations in this document: §1.

[20] P. A. Fouque, T. Vannet, *Improving Key Recovery to 784 and 799 rounds of Trivium using Optimized Cube Attacks.* (2013), *FSE*, Lecture Notes in Computer Science, 8424, 2013. Citations in this document: §5.1.

[21] H. Groβ, E. Wenger, C. Dobraunig and, C. Ehrenhofer, *Suit up! Made-to-Measure Hardware Implementations of ASCON.* (2014). URL: https://eprint.iacr.org/2015/034.pdf. Citations in this document: §5.

[22] V. Grosso, G. Leurent, F. Standaert, K. Varici, F. Durvaux, L. Gaspar, S. Kerckhof, *SCREAM and iSCREAM Side-Channel Resistant Authenticated Encryption with Masking.* (2014). URL: http://competitions.cr.yp.to/round1/screamv1.pdf.. Citations in this document: §5.

[23] M. Hell, T. Johansson, W. Meier, *Grain: a stream cipher for constrained environments.* (2007), 86–93, *International Journal of Wireless and Mobile Computing*, Special Issue on Towards Ubiquitous Wireless Communication: the Integration of 3G/WLAN Networks, 2(4), 2007. Citations in this document: §1.

[24] M. Hell, T. Johansson, A. Maximov, W. Meier, *A Stream Cipher Proposal: Grain-128.* (2006), *International Symposium on Information Theory-ISIT*, IEEE, 2006. Citations in this document: §1.

[25] C. Jutla, *Encryption Modes with Almost Free Message Integrity.* (2008), 547–578, *J. Cryptology*, 21, 2008.

[26] Y. Mansour, N. Nissan, P Tiwari, *The Computational Complexity of Universal Hashing.* (1990), 235–243, *Twenty Second Annual ACM Symposium on Theory of Computing*, 1990. Citations in this document: §3.2.

[27] A. Maximov, A. Biryukov, *Two Trivial Attacks on Trivium.* (2007), 36–55, *Selected Areas in Cryptography*, Lecture Notes in Computer Science, 4876, 2007. Citations in this document: §5.1, §5.1.

[28] D. A. McGrew, J. Viega, *The Galois/Counter Mode of Operation (GCM).* (2005). URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf.

[29] T. K. Moon, *Error Control Coding: Mathematical Methods and Algorithms.* (2005), *Wiley*, 2005.

[30] G. E. Moore, *Cramming more components onto integrated circuits.* (1965), *Retrieved 2015-06-07*, 1965. URL: http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf. Citations in this document: §6.3.

[31] F. Muller, *Differential Attacks against the Helix Stream Cipher.* (2004), 94 – 108, *FSE*, Lecture Notes in Computer Science, 3017, 2004.

[32] M. Nandi, *On the Minimum Number of Multiplications Necessary for Universal Hash Constructions.* (2014), 489 – 508, *FSE*, Lecture Notes in Computer Science, 8540, 2014. Citations in this document: §1, §1, §3.1, §3.2, §2, §3.3, §5.3.

[33] P. Sarkar, *Modes of Operations for Encryption and Authentication Using Stream Ciphers Supporting an Initialisation Vector.* (2014), 189–231, *Cryptography and Communications*, 6(3), 2014. Citations in this document: §1, §1, §4.1, §4.1.

[34] R. Taylor, *Near optimal unconditionally secure authentication.* (1995), 244–253, *EUROCRYPT*, 950, 1995.

[35] D. Whiting, B. Schneier, S. Lucks and, F. Muller, *Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive* (2004). URL: http://www.ecrypt.eu.org/stream/. Citations in this document: §1.

[36] H. Wu, B. Preneel, *Differential-Linear Attacks Against the Stream Cipher Phelix.* (2007), 87–100, *FSE*, Lecture Notes in Computer Science, 4593, 2007.

[37] H. Wu, B. Preneel, *AEGIS: A Fast Authenticated Encryption Algorithm.* (2013), 185–201, *SAC*, Lecture Notes in Computer Science, 8282, 2013.