TriviA-ck-v2

Designers : Avik Chakraborti and Mridul Nandi

Submitters : Avik Chakraborti and Mridul Nandi

avikchkrbrti@gmail.com

August 28, 2015

Chapter 1

Specification

The specification for the older version TriviA-ck-v1 for round 1 is available at [3]. The conference version of the specifiction and hardware implementation of TriviA-ck-v1, with $\mathsf{ck} = 0$ is available at [2]. This chapter contains the specification of the newer version TriviA-ck-v2.

1.1 Notation

- Throughout this note, word represents 32 and w represents 64. Any element of $\{0,1\}^{\mathsf{w}}$ is called **block** and any element of $\{0,1\}^{\mathsf{word}}$ is called **word**. Given any element $x \in \{0,1\}^{\mathsf{w}\times\ell}$, we write $\|x\| := \ell$, the number of blocks of x.
- Let $a \in \{0,1\}^n$ then we write $a = a_{n-1}a_{n-2}\cdots a_0$ where $a_i \in \{0,1\}$. Moreover, we write $a_{[j-1..0]} = a_{j-1}a_{j-2}\cdots a_0$ which returns the least significant j bits of a.
- $a << i$ represents a, i left shift of an n-bit string a. Similarly, $a >> i$ represents i right shift of a.
- $r = a \mod n$ represents the remainder when a is divided by n, i.e., $0 \leq r < n$ such that n divides $(a - r)$.
- For any set S, $S^+ = \cup_{i=1}^{\infty} S^i$ and $S^* = \{\lambda\} \cup S^+$ where λ denotes the empty string.
- The padding function maps $x \in \{0,1\}^*$ to $\mathsf{pad}(x) := x^* = x\|10^p$ where $p = w - (|x| \mod \mathsf{w}) - 1$. Note that $x^* \in \{0,1\}^{wm}$, i.e., $\|x^*\|_{\mathsf{w}} = m$, where $m = \lceil \frac{|x|+1}{\mathsf{w}} \rceil$.

1.1.1 Underlying Finite Field $\mathbb{F}_{2^n}$

Let $\mathbb{F}_{2^n}$ denote the binary Galois field of size 2^n, for a positive integer n. Field addition and multiplication between $a, b \in \mathbb{F}_{2^n}$ are represented by $a \oplus b$ (or $a + b$

whenever understood) and $a \cdot b$ respectively. Any field element $a \in \mathbb{F}_{2^n}$ can be represented by any of the following equivalent ways for $a_0, a_1, \ldots, a_{n-1} \in \{0,1\}$.

- An n bit string $a_n a_{n-1} \cdots a_0 \in \{0,1\}^n$.
- A polynomial $a(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ of degree at most $(n-1)$.

1.1.2 Choice of Primitive Polynomials

In our construction, the primitive polynomials [14, 6] used to represent the field $\mathbb{F}_{2^{32}}$ and $\mathbb{F}_{2^{64}}$ are respectively

1. $p_{32}(x) = x^{32} + x^{22} + x^2 + x + 1$ and
2. $p_{64}(x) = x^{64} + x^4 + x^3 + x + 1$

We denote the primitive element $0^{n-2}10 \in \mathbb{F}_{2^n}$ by α_n, where $n \in \{32, 64\}$. Whenever n is understood from the context, we simply write α to mean α_n for notational simplicity.

32-bit String	Polynomial
$0^{30}10$	x or α
$0^{30}11$	$x+1$ or $\alpha+1$
$0^{29}100$	x^2 or α^2

Table 1.1: Various representations of some elements in $\mathbb{F}_{2^{32}}$

Thus, the field multiplication $a(x) \cdot b(x)$ is the polynomial $r(x)$ of degree at most $(n-1)$ such that $a(x)b(x) \equiv r(x) \mod p_n(x)$.

Multiplication by Primitive Element α. We first see an example how we can multiply by α_{32}. Multiplying an element $a := a_{31}a_{30}\cdots a_0 \in \mathbb{F}_{2^{32}}$ by the primitive element α_{32} of $\mathbb{F}_{2^{32}}$ can be done very efficiently as follows:

$$\begin{aligned} a \cdot \alpha_{32} &= a << 1, \text{ if } a_{31} = 0 \\ &= (a << 1) \oplus 0^9 10^{19} 111, \text{ else} \end{aligned}$$

Let $c = 0^9 10^{19} 111$ and $d = 0^{59}11011$. Hence, we can also write the multiplication as $a \cdot \alpha_{32} = (a << 1) \oplus a_{31}c$. One can similarly express the multiplication of the other powers of α.

$$\alpha_{32}^2 \cdot a = (a << 2) \oplus a_{31}(c << 1) \oplus a_{30}c = (a << 2) \oplus (s_{31} \cdots s_1 s_0)$$

where $s_0 = s_{22} = a_{30}$, $s_3 = s_{23} = a_{31}$, $s_1 = s_2 = a_{30} \oplus a_{31}$ and all other $s_i = 0$. Similar simplification can be made for other power of α multiplications. This representation is useful when we implement the power of α multipliers in hardware.

Examples

1. $$\begin{aligned} x^2(x^{31}+x^{30}+x+1) &= (x^{33}+x^{32}+x3+x^2) \mod p_{32}(x) \\ &= (x^3+x^2)+(x^{23}+x^3+x^2+x)+(x^{22}+x^2+x+1) \\ &= (x^{23}+x^2+1) \end{aligned}$$

2. $$\begin{aligned} x^2(x^{63}+1) &= (x^{65}+x^2) \mod p_{64}(x) \\ &= x^2+(x^5+x^4+x^2+x) \\ &= (x^5+x^4+x) \end{aligned}$$

1.1.3 Vandermonde Matrix and Horner's Rule

We define a special form of vandermonde matrix, denoted $V_{n,d,\ell}$, where α denotes the primitive element α_n (i.e., $x \mod p_n(x)$).

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

Example

$$V_{16,4,7} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha & 1 \\ \alpha^{12} & \alpha^{10} & \alpha^8 & \alpha^6 & \alpha^4 & \alpha^2 & 1 \\ \alpha^{18} & \alpha^{15} & \alpha^{12} & \alpha^9 & \alpha^6 & \alpha^3 & 1 \end{pmatrix}$$

where α is the primitive element of $\mathbb{F}_{2^{16}}$.

Multiplying $V_{n,d,l}$ to a vector can be done in an online manner without requiring much memory. For example, when we multiply $V_{16,4,1000} \cdot h$ where h is 4×1 vector we can compute it using only 4 states for h by using standard Horner's rule. The general algorithm is described below.

The $\mathsf{VHorner}_{n/d}$ subroutine is described in the Algorithm 2. This subroutine actually implements the Horner's rule. The subroutine will be implemented later in $\mathsf{VHorner}_{n/d}$ circuit described in Chapter 5.

To implement the Algorithm 1, we have to implement α_n^j-multipliers for $1 \leq j < d$. We have seen before that one can efficiently describe these multiplications by shift and bit-wise xor operations. Functionally, the above algorithm is same as multiplying $(x_1, \ldots, x_\ell)$ with the Vandermonde matrix $V_{n,d,\ell}$, i.e.,

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{pmatrix} = \begin{pmatrix} 1 & \cdots & 1 & 1 & 1 \\ \alpha^{\ell-1} & \cdots & \alpha^2 & \alpha & 1 \\ \alpha^{2(\ell-1)} & \cdots & \alpha^4 & \alpha^2 & 1 \\ \vdots & \cdots & \vdots & \vdots & \vdots \\ \alpha^{(d-1)(\ell-1)} & \cdots & \alpha^{2(d-1)} & \alpha^{(d-1)} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_\ell \end{pmatrix}$$

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

```
1    $y_1 = \cdots = y_d = 0^n$
2    for $i = 1$ to $\ell$
3        $(y_1, \ldots y_d) = \mathsf{VHorner}_{\mathsf{n/d}}(x_i)$
4  return $(y_1, \ldots y_d)$;
```

Algorithm 1: $\mathsf{VMult}_{n/d}$ multiplies a ℓ-dimensional vector $x = (x_1, \ldots, x_\ell)$ by Vandermonde matrix V to convert into d-dimensional vector $x \cdot V_{n,d,\ell}$. We apply Horner's rule to implement the algorithm.

Subroutine $\mathsf{VHorner}_{\mathsf{n}/d}$
Input: $(x, (y_1, y_2, \ldots, y_d)) \in \mathbb{F}_{2^n} \times \mathbb{F}_{2^n}^{d}$
Output: $y := (y'_1, y'_2, \ldots, y'_d) \in \mathbb{F}_{2^n}^{d}$

```
1    for $j = 1$ to $d$
2        $y'_j = \alpha_n^{j-1} \cdot y_j \oplus x_i$ ;
3  return $(y'_1, \ldots y'_d)$;
```

Algorithm 2: $\mathsf{VHorner}_{\mathsf{n/d}}$ subroutine.

Suppose in Algorithm $\mathsf{VMult}_{n/d}$ we denote the value of y_j (in line 3) by y_j^k when *"for-loop"* is executed up to $i = k$. Thus, the final output is $(y_1^l, \ldots, y_d^l)$. Note that for any k, we have

$$\begin{pmatrix} y_1^k \\ y_2^k \\ \vdots \\ y_d^k \end{pmatrix} = \begin{pmatrix} 1 & \cdots & 1 & 1 & 1 \\ \alpha^{k-1} & \cdots & \alpha^2 & \alpha & 1 \\ \alpha^{2(k-1)} & \cdots & \alpha^4 & \alpha^2 & 1 \\ \vdots & \cdots & \vdots & \vdots & \vdots \\ \alpha^{(d-1)(k-1)} & \cdots & \alpha^{2(d-1)} & \alpha^{(d-1)} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix}$$

In other words, with an appropriate tapping of y values we can compute $\mathsf{VMult}_{n/d}(x_1, \ldots, x_k)$, for all $1 \le k \le l$, during the computation of $\mathsf{VMult}_{n/d}(x_1, \ldots, x_l)$.

1.2 External Parameters

1.2.1 Recommended Parameter Choice

In this paper We propose a construction TriviA. However, it has some other parameters which can be chosen by user following the recommendation.

1. ck: The possible choices of ck is from 0 to $\mathsf{L} := 2^{32}$. Here $\mathsf{ck} = 0$ means that there are no intermediate tag in the output of Trivia-ck but the computation of authentication is same for $\mathsf{ck} = \mathsf{L}$. Roughly, we need to store 64ck bits of messages for the intermediate authentication before we release it. Based on the buffer availability of implementation environment one can choose the value of the parameter ck.

We recommend two values for ck, namely 0 (when the complete message can be stored before we authenticate whole message, i.e., no intermediate tag is required) and $\mathsf{ck} = 128$ (if the buffer size is at least 1KB).

1.3 Input and Output Data

To encrypt a message M with an associated data D and a nonce N, one needs to provide the informations given below.

- An encryption key $K \in \{0,1\}^{128}$, the seed for the underlying streamcipher Trivia-SC.
- A Public message no. $\mathsf{pub} \in \{0,1\}^{64}$. This can include the counter to make the nonce non-repeating, if required.
- The parameter set $\mathsf{param} \in \{0,1\}^{64}$. The first 32 bits denote the bit representation of ck. The next 8-bits denote the final tag length (maximum length is 128). The next 8-bits denote the length of each of the intermediate tags (if any) and set as 0^8 is $\mathsf{ck} = 0$ and upper bounded by

128 if $\mathsf{ck} > 0$. The remaining 16 bits are preserved for future parameters which is currently set as 0^{16}. We define nonce $N \in \{0,1\}^{128}$ to be the concatenation $\mathsf{param}\|\mathsf{pub}$.

- Associated data $D \in \{0,1\}^*$, with the following restriction of associated data size : $0 \leq |D| \leq 2^{64}$.
- A message (or plaintext) $M \in \{0,1\}^*$, where $1 \leq |M| \leq 2^{128}$. In this algorithm we do not have any provision of secret message number.

TriviA-ck authenticated encryption produces the following output data:

- Ciphertext $C \in \{0,1\}^{|M|}$.
- Tag $T \in \{0,1\}^{128t}$, where

$$t = \begin{cases} 1 & \text{if } \mathsf{ck} = 0; \\ \lceil \frac{\|\mathsf{pad}(M)\|}{\mathsf{ck}} \rceil & \text{else} \end{cases}$$

1.4 Mathematical Components

1.4.1 Streamcipher Trivia-SC

Trivia-SC is the base stream cipher which is a modified version of Trivium [1] for encryption key and authentication tag generation. Trivia-SC is loaded with 128-bit key and 128-bit IV and generates bitstream. It uses three Non-linear feedback registers (NFSR) A, B and C of size 132 bit, 105 bit and 147 bit respectively. We will also represent the stream cipher internal state by $S_1, S_2, \cdots, S_{384}$, where $A = (S_1, S_2, \cdots, S_{132})$, $B = (S_{133}, S_{134}, \cdots, S_{237})$ and $C = (S_{238}, S_{239}, \cdots, S_{384})$.

Algorithm 3 describes the all basic modules used for the streamcipher Trivia-SC. A proper intergration of these modules can be used to descibe a streamcipher. In this specification, we describe an integrated combinations of these modules and VPV-Hash to describe TriviA.

The 64 bit Modules

Trivia-SC is parallelizable upto 64 bit. This means the stream cipher can produce upto 64 bit stream at a single clock cycle.

Similarly the 64 round updations of Trivia-SC can be done at a single clock cycle due to the parallelism. That is the Update64 subroutine which is equivalent to running the Update subroutine 64 times can be executed in a single clock cycle. More formally the KeyExt64 and the Update64 module is described in the Algorithm 4,

The StateExt64 returns 64 bits and it has already been defined in the Algorithm 3.

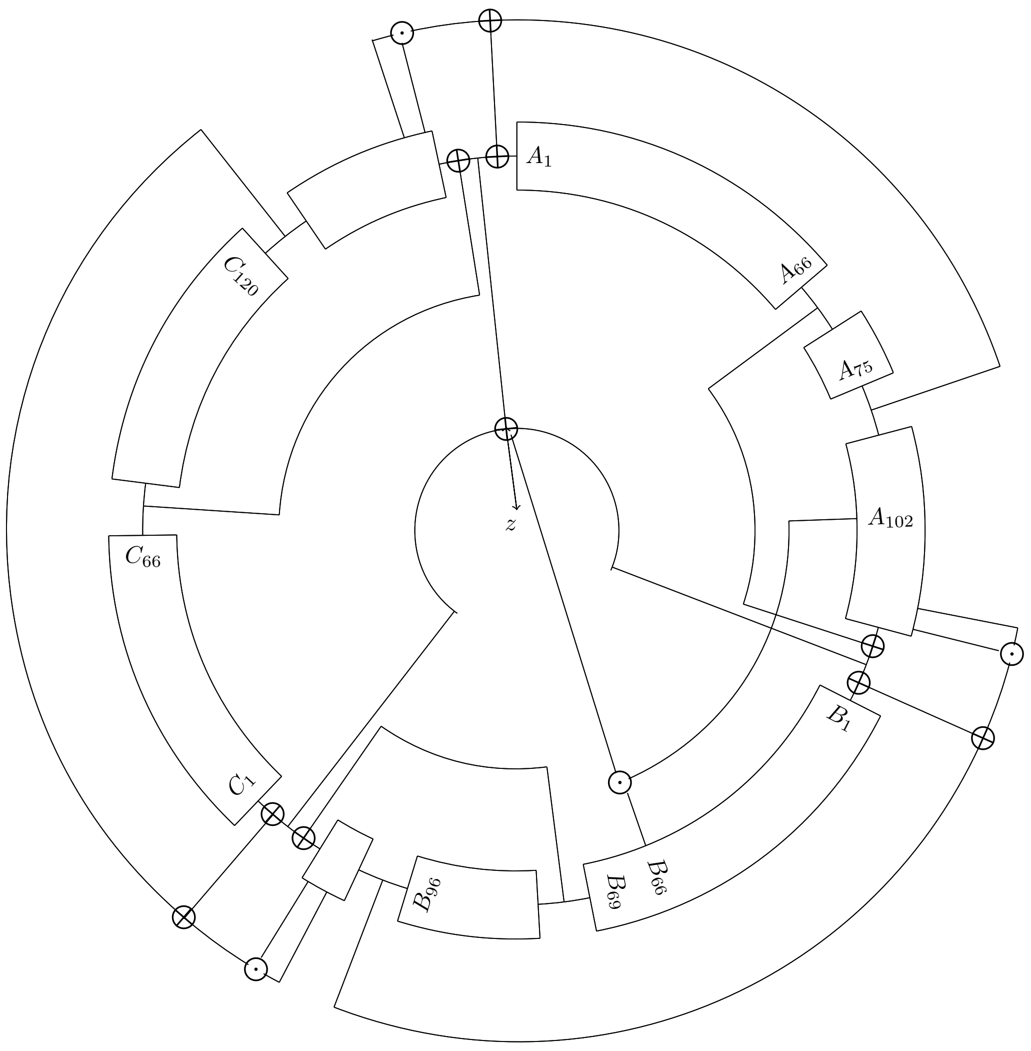

Figure 1.1: Trivia-SC Stream Cipher

```
Modules of Trivia-SC

Load $(K, IV)$ /* Key and IV Loading */
1    $(A_1, A_2, A_3, ..., A_{132}) \leftarrow (K_1, K_2, K_3, ..., K_{128}, 0, 0, 0, 0)$;
2    $(C_1, C_2, C_3, ..., C_{147}) \leftarrow (IV_1, IV_2, IV_3, ..., IV_{128}, 0, 0, ..., 0)$;
3    $(B_1, B_2, B_3, ..., B_{105}) \leftarrow (0, ..., 0, 1, 1, 1)$;

Update /* Update a Single Round */
4    $t_1 \leftarrow A_{66} \oplus A_{132} \oplus (A_{130} \wedge A_{131}) \oplus B_{96}$;
5    $t_2 \leftarrow B_{69} \oplus B_{105} \oplus (B_{103} \wedge B_{104}) \oplus C_{120}$;
6    $t_3 \leftarrow C_{66} \oplus C_{147} \oplus (C_{145} \wedge C_{146}) \oplus A_{75}$;
7    $(A_1, A_2, A_3, ..., A_{132}) \leftarrow (t_3, A_1, A_2, ..., A_{131})$;
8    $(B_1, B_2, B_3, ..., B_{105}) \leftarrow (t_1, B_1, B_2, ..., B_{104})$;
9    $(C_1, C_2, C_3, ..., C_{147}) \leftarrow (t_2, C_1, C_2, ..., A_{146})$;

KeyExt /* Extract a Key Bit */
10   $\mathsf{z} = A_{66} \oplus A_{132} \oplus B_{69} \oplus B_{105} \oplus C_{66} \oplus C_{147} \oplus (A_{102} \wedge B_{66})$ ;
11   Output z ;

StExt64 /* Extract 64 Key Bits */
12   Output $A_1, A_2, \cdots, A_{64}$;

Insert (T) /* Insert T into State Registers */
13   $(S_1, S_2, \ldots, S_{|\mathsf{T}|}) = (S_1, S_2, \ldots, S_{|\mathsf{T}|}) \oplus \mathsf{T}$ ;
```

Algorithm 3: Modules of Trivia-SC. Here $\wedge$ represents "and" between two bits

```
KeyExt64 /* Extract First 64 Bits from A After 64 Rounds */
1    $\mathsf{t} = A_{[3\cdots66]} \oplus A_{[69\cdots132]} \oplus B_{[6\cdots69]} \oplus B_{[42\cdots105]} \oplus C_{[3\cdots66]} \oplus C_{[84\cdots147]} \oplus$
         $A_{[39\cdots102]} \wedge B_{[3\cdots66]}$ ;
2    Output t ;

Update64 /* Update 64 Rounds */
3    $t_1 \leftarrow A_{[3\cdots66]} \oplus A_{[69\cdots132]} \oplus A_{[67\cdots130]} \wedge A_{[68\cdots131]} \oplus B_{[33\cdots96]}$ ;
4    $t_2 \leftarrow B_{[6\cdots69]} \oplus B_{[42\cdots105]} \oplus B_{[40\cdots103]} \wedge B_{[41\cdots104]} \oplus C_{[57\cdots120]}$ ;
5    $t_3 \leftarrow C_{[3\cdots66]} \oplus C_{[84\cdots147]} \oplus C_{[82\cdots145]} \wedge C_{[83\cdots146]} \oplus A_{[12\cdots75]}$ ;
6    $(A_1, A_2, A_3, ..., A_{132}) \leftarrow (t_3, A_1, A_2, ..., A_{68})$ ;
7    $(B_1, B_2, B_3, ..., B_{105}) \leftarrow (t_1, B_1, B_2, ..., B_{41})$ ;
8    $(C_1, C_2, C_3, ..., C_{147}) \leftarrow (t_2, C_1, C_2, ..., A_{83})$ ;
```

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

1.4.2 VPV-Hash

In this section we describe our second component, VPV-Hash [15] defined to compute tag. It first applies Vandermonde based an error-correcing code, called ECCode^* and then applies Pseudo-dot-product and again Vandermonde-based linear transformation.

ECCode

ECCode_d is an error correcting code of systematic form having minimum distance d, when $d = 4$ and it expands $(d-1)$ elements, called checksum. Thus, it is an optimum or MDS code in terms of minimal expansion. In our constructions, we use ECCode_d for $d = 4$. Our error-correcting code has systematic form and so it would be sufficient to describe the checksum elements. In other words, given any input of ℓ-tuple of field elements $(x_1, \ldots, x_\ell)$, the codeword is $(x_1, \ldots, x_\ell, y_1, \ldots, y_{d-1}) \in \mathbb{F}_{2^{\mathsf{w}}}^{\ell+d-1}$ where $(y_1, \ldots, y_{d-1}) = \mathsf{VMult}_{\mathsf{w}/(d-1)}(x_1, \ldots, x_\ell)$. We also denote as

$$\mathsf{ECCode}_d(x_1, \ldots, x_\ell) = (x_1, \ldots, x_\ell, y_1, \ldots, y_{d-1}). \qquad (1.1)$$

We recall the constant $\mathsf{L} = 2^{32}$ to be the maximum number of field elements can be fed as an input to ECCode_d. In other words, we restrict ℓ for ECCode_d to be less than or equal to L. After this length the code may not have desired minimum distance which we have smaller length as described in Proposition 4.1 in Chapter 3 later.

Arbitrary Length Error Correcting Code:

The above algorithm works for at most $\mathsf{L} = 2^{32}$ blocks. We define an error correcting code, denoted $\mathsf{ECCode}^*_{d,\mathsf{ck}}$ which works for any arbitrary length blocks $x \in \mathbb{F}_{2^{\mathsf{w}}}^{+}$ with an additional parameter $\mathsf{ck} \le \mathsf{L}$. We first sparse x as $(X_1, \ldots, X_{\mathsf{m}})$ where all X_i's, possible except the last one, are ck-block elements. We call these X_i's **chunk**. More formally $X \in \mathbb{F}_{2^{64}}^{\mathsf{ck}}$ is a complete chunk and $X \in \mathbb{F}_{2^{64}}^{i}$, with $i < \mathsf{ck}$ is called an incomplete chunk. The last chunk may be incomplete. $\mathsf{ECCode}^*_{d,\mathsf{ck}}$ first parse the input string then apply ECCode_d individually.

Definition 1.1 We define $\mathsf{ECCode}^*_{d,\mathsf{ck}}$ for $0 < \mathsf{ck} \le \mathsf{L}$ as

$$\mathsf{ECCode}^*_{d,\mathsf{ck}}(x) = (\mathsf{ECCode}_d(X_1), \ldots, \mathsf{ECCode}_d(X_{m-1}),\ \mathsf{ECCode}_d(X_m)). \qquad (1.2)$$

We also define $\mathsf{ECCode}^*_{d,0}(x) = \mathsf{ECCode}^*_{d,\mathsf{L}}(x)$. Given $Z \in \{0,1\}^*$, we denote ℓ_Z by $\|\mathsf{ECCode}^*_{d,\mathsf{ck}}(\mathsf{pad}(Z))\|$. Thus,

$$\ell_Z = \begin{cases} \|\mathsf{pad}(Z)\| + (d-1)\lceil \frac{\|\mathsf{pad}(Z)\|}{\mathsf{L}} \rceil & \text{if } \mathsf{ck} = 0 \\ \|\mathsf{pad}(Z)\| + (d-1)\lceil \frac{\|\mathsf{pad}(Z)\|}{\mathsf{ck}} \rceil & \text{if } \mathsf{ck} > 0. \end{cases}$$

Algorithm $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$, $0 \le \mathsf{ck} \le \mathsf{L} = 2^{32}$.
Input: $x \in \{0,1\}^*, (k_1, \ldots, k_{\ell_x}, k^*) \in \{0,1\}^{\mathsf{w} \times \ell_x} \times \{0,1\}^{128}$
Output: $\mathsf{Tag} \in (\{0,1\}^{128})^t$, $t = \lceil \frac{\|x\|}{\mathsf{L}} \rceil$ if $\mathsf{ck} = 0$, else $t = \lceil \frac{\|x\|}{\mathsf{ck}} \rceil$

```
1   $x^* = \mathsf{pad}(x)$ ;
2   $(x_1, \ldots, x_{\ell_x}) = \mathsf{ECCode}^*_{4,\mathsf{ck}}(x^*)$;
     $x_i = (x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}) \in \mathbb{F}^4_{2^{16}}$ and $k_i = (k_{i_1}, k_{i_2}, k_{i_3}, k_{i_4}) \in \mathbb{F}^4_{2^{16}}$, $1 \le i \le \ell_x$
3   for $i = 1$ to $\ell_x$ /* 32-bit Field Multiplication for $\ell_x$ Blocks */
4       $g_i = ((x_{i_1}||x_{i_3}) \oplus (k_{i_1}||k_{i_3}))((x_{i_2}||x_{i_4}) \oplus (k_{i_2}||k_{i_4}))$;
5   $c = \lceil \frac{\|x\|}{\mathsf{ck}} \rceil$ if $\mathsf{ck} > 0$, $c = 1$ if $\mathsf{ck} = 0$ ;
6   for $i = 1$ to $c - 1$
7       $\mathsf{T}_i = \mathsf{VMult}_{\mathsf{word}/4}(g_1, g_2, \ldots, g_{i(\mathsf{ck}+3)})$;
8   $k^* = k^*_1||k^*_2||k^*_3||k^*_4$ where, $k^*_1, k^*_2, k^*_3, k^*_4 \in \mathbb{F}_{2^{\mathsf{word}}}$;
9   $k^{**}_2 = \alpha^2_{32}.k^*_2$ ;
10  $\mathsf{T}_c = \mathsf{VMult}_{\mathsf{word}/4}(g_1, g_2, \ldots, g_{\ell_x}) \oplus k^*_1||k^{**}_2||k^*_3||k^*_4$;
11  return $(\mathsf{T}_1, \ldots, \mathsf{T}_c)$;
```

Kindly note that, $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}(x, (k_1, \ldots, k_\ell, k^*)) = \mathsf{T}_c$.

Algorithm 5: $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$: A Δ-universal hash for variable length binary strings which would essentially help to prove the unforgeability of our authenticated encryption.

1.5 TriviA-ck

The components needed to construct TriviA-ck have been defined in the previous subsections.

1.5.1 TriviA-ck-[Auth-Enc]

```
Select($\mathbf{M}, \mathbf{K}, \mathsf{Len}$)
1  if $|M| \mod 64 = 0$ then
2  $K' = ((K_1, \cdots, K_{\mathsf{ck}}), (K_{\mathsf{ck}+4}, \cdots, K_{2\mathsf{ck}+3}) \cdots, (K_{\mathsf{q}(\mathsf{ck}+3)+1}, \cdots, K_{\mathsf{Len}} - 4))$;
3  else
4  $K' = ((K_1, \cdots, K_{\mathsf{ck}}), (K_{\mathsf{ck}+4}, \cdots, K_{2\mathsf{ck}+3}), \cdots, (K_{\mathsf{q}(\mathsf{ck}+3)+1}, \cdots, K_{\mathsf{Len}} - 3))$;
5  output $K'$ ;
```

Algorithm 6: The Select subroutine. Here, $\mathsf{q} = \lceil \frac{\|pad(M)\|}{\mathsf{ck}} \rceil - 1$ if $\mathsf{ck} > 0$ and $\mathsf{q} = \lceil \frac{\|pad(M)\|}{2^{32}} \rceil - 1$ $\mathsf{ck} = 0$

TriviA-ck-[Auth-Enc] authenticated encryption algorithm is described by Algorithm 7. The algorithm uses the subroutine Select described in the Algorithm 6 to extract the key bits corrosponds to the padded message position (Excluding checksum positions computed by ECCode*) from the ℓ_M words.

We have already mentioned that $\mathsf{ck} = 0$ denotes the final tag doesn't contain the sequence of intermediate tags. Hence TriviA-0 represents the presence of L sized buffer and Tag is $32 \times 4 = 128$ bit long.

Note that TriviA-L and TriviA-0 are not same. If the message has more than L blocks then TriviA-L produces intermediate tags but TriviA-0 does not. If the message has less than L blocks then both TriviA-L and TriviA-0 are functionally same.

Algorithm TriviA-ck-[Auth-Enc]

Input: (M, D, (param, pub), Key) $\in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^{2\times 64} \times \{0,1\}^{128}$

Output: (C, Tag) $\in \{0,1\}^{|M|} \times \{0,1\}^{128t}$, $t = 1$ if $\mathsf{ck} = 0$, o.w. $t = \lceil \frac{\|\mathsf{pad}(M)\|}{\mathsf{ck}} \rceil$.

```
1   $N = \mathsf{param}\|\mathsf{pub}$ ;
2   $\mathsf{Load}(Key, N)$;
3   for $i = 1$ to $18$  $\mathsf{Update64}$ ;

4   for $i = 1$ to $\ell_D$ ;
5       $k_i = \mathsf{StExt64}$;
6       if $i > (\ell_D - 3)$   Then $k_D^{(i-\ell_D-3)} = \mathsf{KeyExt64}$;
7       $\mathsf{Update64}$;

8   $K = (k_1, k_2, \ldots k_{\ell_D})$ ;
9   $K_D^* = (k_D^1, k_D^3)$ ;
10  $T = \mathsf{final\text{-}VPV\text{-}Hash}_{\mathsf{ck}}(D, (K, K_D^*))$;
11  $\mathsf{Insert}(T)$;
12  for $i = 1$ to $18$  $\mathsf{Update64}$ ;

13  for $i = 1$ to $\ell_M$;
14      $k_i^C = \mathsf{KeyExt64}$;
15      $k_i^T = \mathsf{StateExt64}$;
16      if $i > (\ell_M - 3)$   Then $k_M^{(i-\ell_M-3)} = \mathsf{KeyExt64}$;
17      $\mathsf{Update64}$;

18  $K^C = (k_1^C, k_2^C, \ldots k_{\ell_M}^C)$ ;
19  $K^T = (k_1^T, k_2^T, \ldots k_{\ell_M}^T)$ ;
20  $K_M^* = (k_M^1, k_M^3)$ ;
21  $K = \mathsf{Select}(M, K^C, \ell_M)$ ;
22  $\mathsf{C} = K_{[|M|-1\cdots 0]} \oplus M$;
23  $\mathsf{Tag} = \mathsf{VPV\text{-}Hash}_{\mathsf{ck}}(M, (K^T, K_M^*))$ ;

24  return (C, Tag);
```

Algorithm 7: TriviA-ck-[Auth-Enc]

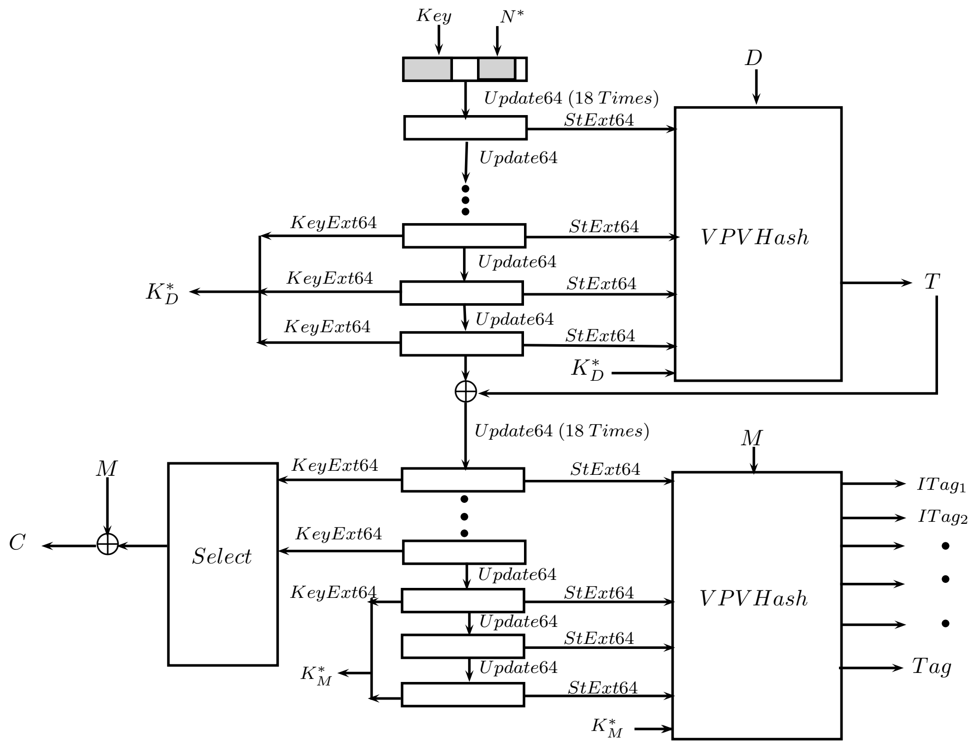

Figure 1.2: Circuit Diagraim for TriviA when messige size is not multiple of 64. When it is multiple of 64 we need to run 4 StExt64 final calls instad of three as we get 4 expanded message blocks whose ciphertext is not required.

1.5.2 TriviA-ck-[Ver-Dec]

In this section we will describe the verified-decryption algorithm TriviA-ck-[Ver-Dec]. The algorithm takes as input the $(\mathsf{C},\ \mathsf{Tag})$ tuple in addition with $D, \mathsf{param}\|\mathsf{pub}$. Let us describe how it works when $\mathsf{ck} = 0$. As there is no intermediate tag we run exactly same as encryption (we use ℓ_{C} instead of ℓ_M) to obtain the key stream K^C and then computes message as

$$K = \mathsf{Select}(C, K^C, \ell_C)$$

$$M = K_{[|C|-1\cdots 0]} \oplus \mathsf{C}$$

The verified-decryption algorithm when $\mathsf{ck} = 0$, returns

- M if $\mathsf{Tag} = \mathsf{VPV\text{-}Hash}_{\mathsf{ck}}(M, (K^T, K^*_M))$.
- $\perp$ (rejects and no message is released) otherwise.

Now we describe verified-decryption algorithm when $\mathsf{ck} > 0$. We write $\mathsf{Tag} = (\mathsf{Itag}'_1, \cdots \mathsf{Itag}'_{\lceil \frac{\mathsf{r}}{\mathsf{ck}} \rceil - 1}, \mathsf{Tag}' := \mathsf{Itag}'_{\lceil \frac{\mathsf{r}}{\mathsf{ck}} \rceil})$, for $\mathsf{r} = \lceil \frac{|C|+1}{64} \rceil$. Using the similar manner we compute Itag_i for all $1 \leq i \leq \lceil \frac{\mathsf{r}}{\mathsf{ck}} \rceil$ after decrypting the message $M = (M_1, \ldots, M_{\lceil \frac{\mathsf{r}}{\mathsf{ck}} \rceil})$ where M_i represents the i^{th} chunk of the message M. The verified-decryption algorithm returns

- M if $\mathsf{Itag}_i = \mathsf{Itag}'_i$, for all $1 \leq i \leq \lceil \frac{\mathsf{r}}{\mathsf{ck}} \rceil$
- $(M_1, \ldots, M_{i-1}, \perp)$ (rejects and and the first $(i-1)$ message chunks are released) if there exists any index $i \geq 1$ such that $\mathsf{Itag}_i \neq \mathsf{Itag'}_i$ but $\mathsf{Itag}_j = \mathsf{Itag'}_j$, $1 \leq j < i$.

Chapter 2

Modular Description of TriviA

This chapter describes the working principle of different algorithimic modules of TriviA for processing a block of data. The description may be useful for hardware implementation of TriviA. The modular algorithms give a rough idea about how the different modules VPV-Hash, TriviA can be described in hardware platform by the corresponding programming language. We describe three main algorithms used in TriviA, namely VPV-Hash-Block described by Algorithm 8, TriviA-Block described by Algorithm 9 and TriviA-Top-ck described by Algorithm 10. VPV-Hash-Block algrithm describes how the $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$ algorithm processes a data block. The algorithm is described below.

The above algorithm shows how $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$ processes a message or associated data block and updates the tag. isFinalChecksum indicates whether the current block is in the final 3-checksum blocks. CScount denotes the index of the checksum block inside the final 3-checksum blocks.

TriviA-Block algorithm describes how the TriviA algorithm processes a message block or an associated data block or a checksum block. It uses the VPV-Hash-Block algorithm to update the intermediate tag values at each of the clock cycle. The algorithm also shows how the output keystreams and the state extracted key streams are used at each of the clock cycles for encryption and authentication. The algorithm is described below.

The signals isFinalChecksum, CScount are already described above. Signal isAD indicates whether the currently processed blcok is a an associated data block or a message block. isCS signal indicates whether the currently processed block is a checksum block or not. Note that, depending on the length of the associated data or message the $\mathsf{ECCode}^*_{4,\mathsf{ck}}$ algorithm expands the message and produces a series of three checksum blocks after each of the chunk of ck blocks.

```
Algorithm VPV-Hash-Block
Input: $x \in \{0,1\}^{64}, K^V \in \{0,1\}^{64}, K^T \in \{0,1\}^{64}$,
       $\mathsf{InTag} \in \{0,1\}^{128}, \mathsf{isFinalChecksum} \in \{0,1\}, \mathsf{CScount} \in \{0,1\}^2$
Output: $\mathsf{OutTag} \in \{0,1\}^{128}$

1  $x = (x_1, x_2, x_3, x_4) \in \mathbb{F}_{2^{16}}^4$ and $k^T = (k_1, k_2, k_3, k_4) \in \mathbb{F}_{2^{16}}^4$
2  $g$=$((x_1||x_3) \oplus (k_1||k_3))((x_2||x_4) \oplus (k_2||k_4))$;
3  $\mathsf{InTag} = (\mathsf{InTag}_1, \mathsf{InTag}_2, \mathsf{InTag}_3, \mathsf{InTag}_4) \in \mathbb{F}_{2^{32}}^4$
4  $\mathsf{InTag}_1 = \mathsf{InTag}_1 \oplus g, \mathsf{InTag}_2 = \alpha_{32}.\mathsf{InTag}_2 \oplus g$,
   $\mathsf{InTag}_3 = \alpha_{32}^2.\mathsf{InTag}_3 \oplus g, \mathsf{InTag}_4 = \alpha_{32}^3.\mathsf{InTag}_4 \oplus g$
5  If $\mathsf{isFinalChecksum} = 1$ and $\mathsf{CScount} = 1$ then
6      $\mathsf{InTag}_1||\mathsf{InTag}_2 = \mathsf{InTag}_1||\mathsf{InTag}_2 \oplus K^V$
7  If $\mathsf{isFinalChecksum} = 1$ and $\mathsf{CScount} = 3$ then
8      $\mathsf{InTag}_3||\mathsf{InTag}_4 = \mathsf{InTag}_3||\mathsf{InTag}_4 \oplus K^V$
9  $\mathsf{OutTag} = \mathsf{InTag}$
10 return $\mathsf{OutTag}$;
```

Algorithm 8: VPV-Hash-block processing module

Thus, isFinalChecksum denotes whether the current block belongs to the final three checksum blocks or not, where as isCS indicates whether the current block belongs to any one of the three checksum blocks or not.

Both the modules VPV-Hash-Block and TriviA-Block process a data block. However, they are needed to be controlled by a top module, which controls the input signals to these algorithms. TriviA-Top-ck algorithm use to control the isFinalChecksum, CScount, isAD and isCS signals and the values of the checksum counter CScount. It calls the submodule TriviA-Block along with the signal values. Finally it produces the output ciphertext and the tag. The decryption algorithm can be defined accordingly. The TriviA-Top-ck algorithm is described below.

Algorithm TriviA-Block

Input: $x \in \{0,1\}^{64}, \mathsf{isAD} \in \{0,1\}, \mathsf{isCS} \in \{0,1\}, \mathsf{isFinalChecksum} \in \{0,1\}, \mathsf{InTag} \in \{0,1\}^{128}, \mathsf{CScount} \in \{0,1\}^2$

Output: $(C \in \{0,1\}^{64}, \mathsf{OutTag} \in \{0,1\}^{128})$ if $\mathsf{isCS} = 0$, $\mathsf{OutTag} \in \{0,1\}^{128}$ if $\mathsf{isCS} = 1$

```
1  $K^1 = \mathsf{KeyExt64}, K^2 = \mathsf{StExt64}$ ;
2  if $\mathsf{isAD} = 1$ then /–associated data block–/
3    $\mathsf{OutTag} = \mathsf{VPV\text{-}Hash\text{-}Block}(x, K^1, K^2, \mathsf{InTag}, \mathsf{isFinalChecksum}, \mathsf{CScount})$ ;
4  if $\mathsf{isAD} = 0$ then /–message block–/
5    if $\mathsf{isCS} = 0$ /–not a checksum block–/
6      $\mathsf{OutTag} = \mathsf{VPV\text{-}Hash\text{-}Block}(x, K^1, K^2, \mathsf{InTag}, 0, 0)$ ; /–$K^1$ Ignored–/
7      $C = x \oplus K^1$ ; /–ciphertext block–/
8    if $\mathsf{isCS} = 1$ /–checksum block–/
9      $\mathsf{OutTag} = \mathsf{VPV\text{-}Hash\text{-}Block}(x, K^1, K^2, \mathsf{InTag}, \mathsf{isFinalChecksum}, \mathsf{CScount})$ ;
10 if $\mathsf{isCS} = 0$ then
11   **return** $(C, \mathsf{OutTag})$ ;
12 else
13   **return** $\mathsf{OutTag}$ ;
```

Algorithm 9: TriviA-block processing module

Algorithm TriviA-Top-ck

Input: (M, D, (param, pub), Key) $\in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^{2\times 64} \times \{0,1\}^{128}$

Output: (C, Tag) $\in \{0,1\}^{|M|} \times \{0,1\}^{128t}$, $t = 1$ if $\mathsf{ck} = 0$, o.w. $t = \lceil \frac{\|\mathsf{pad}(M)\|}{\mathsf{ck}} \rceil$.

```
1   $C = \phi$, $T = 0$, $\mathsf{Tag} = 0$, $\mathsf{isAD} = 1$, $\mathsf{isCS} = \mathsf{isFinalChecksum} = \mathsf{CScount} = 0$,
    $\mathsf{L} = 2^{32}$;
2   $\mathsf{chunk} = \mathsf{L}$ if $\mathsf{ck} = 0$, $\mathsf{chunk} = \mathsf{ck}$ if $\mathsf{ck} > 0$ ;
3   $(D_1, \cdots, D_{\ell_D}) = \mathsf{ECCode}^*_{4,\mathsf{ck}}(D)$, $(M_1, \cdots, M_{\ell_M}) = \mathsf{ECCode}^*_{4,\mathsf{ck}}(M)$ ;
4   $\mathsf{Load}(Key, (\mathsf{param}\|\mathsf{pub}))$ ; /* Key and Nonce Load */
5   for $i = 1$ to 18   Update64 ; /* Initialize */
6   for $i = 1$ to $\ell_D$ /* AD and Checksum Block Processing */
7     if $i > (\ell_D - 3)$ then /* Final Checksum Blocks */
8       $\mathsf{isCS} = 1$, $\mathsf{isFinalCS} = 1$, $\mathsf{CScount} = i - \ell_D - 3$ ;
9     $\mathsf{T} = \mathsf{TriviA\text{-}Block}(D_i, \mathsf{isAD}, \mathsf{isCS}, \mathsf{isFinalChecksum}, \mathsf{T}, \mathsf{CScount})$ ;
10    Update64 ;
11  $\mathsf{isAD} = \mathsf{isCS} = \mathsf{isFinalChecksum} = \mathsf{CScount} = 0$;
12  $\mathsf{Insert}(T)$;
13  for $i = 1$ to 18   Update64 ; /* Reinitialize */
14  $c = \lceil \frac{\|x\|}{\mathsf{ck}} \rceil$ if $\mathsf{ck} > 0$, $c = 1$ if $\mathsf{ck} = 0$ ;
15  for $i = 1$ to $\ell_M$ /* Message and Checksum Block Processing */
16    $K^1 = \mathsf{KeyExt64}$, $K^2 = \mathsf{StExt64}$ ;
17    if $(i \;\%\; (\mathsf{ck} + 3)) > \mathsf{ck}$ or $(i \;\%\; (\mathsf{ck} + 3)) = 0$ then /* Checksum Blocks */
18      $\mathsf{isCS} = 1$ ;
19      if $i > (\ell_M - 3)$ then /* Final Checksum Blocks */
20        $\mathsf{isFinalCS} = 1$, $\mathsf{CScount} = i - \ell_M - 3$ ;
21      $\mathsf{Tag} = \mathsf{TriviA\text{-}Block}(M_i, \mathsf{isAD}, \mathsf{isCS}, \mathsf{isFinalChecksum}, \mathsf{Tag}, \mathsf{CScount})$ ;
22      if $(i \;\%\; (\mathsf{ck} + 3)) = 0$ then $\mathsf{Tag}_i = \mathsf{Tag}$ ; /* Intermediate Tags */
23    else
24      $\mathsf{isCS} = 0$ ;
25      $(C_i, \mathsf{Tag}) = \mathsf{TriviA\text{-}Block}(M_i, \mathsf{isAD}, \mathsf{isCS}, \mathsf{isFinalChecksum}, \mathsf{Tag}, \mathsf{CScount})$
;
26      $C = C||C_i$ ;
27    Update64 ;
28  $\mathsf{Tag}_c = \mathsf{Tag}$ ;
29  **return** $(\mathsf{C}, (\mathsf{Tag}_1, \mathsf{Tag}_2, \cdots, \mathsf{Tag}_c))$;
```

Algorithm 10: TriviA-Top-ck Encryption Module

Chapter 3

Security Goals

Recommended parameter sets	confidentiality for the plaintext	integrity for the plaintext	integrity for the associated data
TriviA-0	128	124	124
TriviA-128	128	124	124

Table 3.1: Table quantifying for all choices of ck including 0, the intended number of bits of security with the assumption that nonce can repeat at most 2^{32} times. Note that our recommendation choice is ck =128 and ck = 0. However, for any other choices of ck, the security remains same. It is suggested to choose depending on the application requirement.

We call the concatenation of the public message number and parameter nonce. However we do not allow the repetition of nonce. Our constructions remain secure as given in the above table, as long as nonce remains distinct over every execution. Moreover, if nonce repeats then still we have but lesser bits of security for privacy and integrity. The privacy and the integrity proof can be found in Theorem 4.4 and Theorem 4.5.

Chapter 4

Security Analysis

4.1 Empirical Results

Cube Attack [4] is best known algebraic attack on reduced round version of Trivium [1]. It has been tested for the updated versions of Trivia-SC with 1152 round initialization, no maxterms of size less than equal to 29 has been found. Moreover for the 896 and 832 round initialization version we have havn't found any maxterm of size 29 or less. But for 768 round initialization version this attack finds some linear superpoly with cube size 20.

Trivia-SC with 1152 initialization rounds has also been tested for the output bit polynomial density. We have used Moebious Transform [7] to compute the polynomial density. The polynomial has been restricted to 29 IV variables and density of the monomials of degree less than 29 in the restricted polynomial has been calculated. The result is given in the table 4.1. Trivia-SC with 960 initialization rounds has also been tested for the output bit polynomial density and the result is given in the table 4.2. Output bit polynomial density for Trivia-SC with 768, 832 and 896 rounds initialization is given in table 4.5 , table 4.4 and table 4.3 respectively.

It has been observed from the results that the output bit polynomial for Trivia-SC with 896 or more rounds of initialization behaves as random polynomial as each of the monomials of size less than 29 have density very close to 0.5 and the average density is also very close to 0.5. For 832 rounds of initialization version, monomials of size less than 29 have density far from 0.5 but the average density is close to 0.5. Thus, the polynomial may behave in a non random manner for 832 or less rounds of initialization version. From the result, it is clear for 768 or less rounds of initialization version the stream cipher output behaves in a non random manner as monomials of size less than 25 have density far from 0.5 as well as the average density is also far from 0.5.

We have also observed the behaviour of the polynomials corresponding to the statebits of the internal state register of Trivia-SC. The statebit polynomial

Monomial Size	22	24	25	26	27	28
Density	0.499	0.499	0.498	0.500	0.495	0.586

Table 4.1: Trivia-SC with 1152 Rounds

Monomial Size	22	24	25	26	27	28
Density	0.501	0.498	0.496	0.498	0.500	0.379

Table 4.2: Trivia-SC with 960 Rounds

Monomial Size	24	25	26	27	28
Density	0.498	0.498	0.520	0.497	0.448

Table 4.3: Trivia-SC with 896 Rounds

Monomial Size	17	18	19	20	21	22	24	25	28
Density	0.494	0.481	0.456	0.407	0.320	0.206	0.033	0.003	0.000

Table 4.4: Trivia-SC with 832 Rounds

Monomial Size	13	15	16	17	18	20	21	22	24
Density	0.308	0.197	0.144	0.097	0.059	0.011	0.003	0.000	0.000

Table 4.5: Trivia-SC with 768 Rounds

density has been observed to be very similar to that of the output bit polynomial. We are doing extensive analysis on the properties of statebit polynomials and the results will be given later in more details.

The statistical tests on Trivia-SC were performed using the NIST Test Suite [19]. Tests were performed on the on the output bit-stream and no weaknesses were found.

4.1.1 Security against Biryukov and Maximov Attack [12]

Trivia-SC is an extended version of Trivium, has been tested to output data streams indistinguishable from independent and uniform random strings for distinct inputs. Hence the ciphertext C is indistinguishable from *one time pad* of plaintext P. The constrained (over 30 randomly chosen IV bits) output polynomial over *key* and IV bits behaves as random polynomial for Trivia-SC with 1152 round initialization.

Biryukov and Maximov Attack [12] in 2007 have constructed an attack on Trivium which aims to recover the whole internal state with known keystream by guessing one third of the internal state bits. Since the output bit equation

$z = S_{\mathsf{A}}^{66} + S_{\mathsf{A}}^{132} + S_{\mathsf{B}}^{69} + S_{\mathsf{B}}^{105} + S_{\mathsf{C}}^{66} + S_{\mathsf{C}}^{147}$ has no nonlinearity the attacker can easily get 22 linear equations from the output and the complexity of the attack reduces to $2^{\frac{\theta}{3}-22}$, where θ is the internal state size. This leads to a key recovery attack with complexity $2^{96-22} = 2^{74}$, where the key size is 80 bit.

The above attack can recover the internal state and thus the secret key with complexity less than 2^{128} for Trivia-SC when the ouput bit equation has no nonlinearity. The attack can obtain $\min\{\frac{66}{3}, \frac{69}{3}, \frac{66}{3}\}$ i.e, 22 linear equations on the statebits as mentioned in the attack. Hence the total complexity for the state recovery which in tern leads to the key recovery attack reduces to $2^{\frac{\theta}{3}-22} = 2^{106}$, where $\theta = 384$ and the key size is 128 bit, making it an efficient distinguisher for key recovery.

ScTriviA has removed this disadvantage of trivium by introducing nonlinearity in the output bit equation. This resists the attacker from getting some linear equation from the output bitstreams. Hence the complexity of the state recovery attack which in tern led to key recovery attack is $2^{\frac{\theta}{3}}$, where $\theta = 384$ is the total size of the internal state. Hence this attack does not help the attacker since the key in Trivia-SC is 128 bit long.

4.2 Main Theorems

In this section, we prove the privacy and authenticity against all adversaries which can make encryption queries with distinct nonces. Before we these prove we first describe universal hash property of VPV-Hash.

4.2.1 Δ-Universal Property of VPV-Hash

Proposition 4.1 *Let $d \leq 4$. For any fixed $\ell \leq \mathsf{L} := 2^{32}$, the output of ECCode_d, which is $(x_1, x_2, \cdots, x_\ell, y_1, y_2, \cdots y_{d-1})$ has minimum distance d. More precisely, for any fixed $\ell \leq \mathsf{L}$, and $x \neq x' \in \mathbb{F}_{2^{64}}^{\ell}$, the hamming distance between $\mathsf{ECCode}_d(x)$ and $\mathsf{ECCode}_d(x')$ is at least d.*

Proof. Proof for $\mathsf{d} = 4$ is already given in [15].

Corollary 4.2 *For any positive integer ℓ, $\mathsf{ECCode}^{*}_{d,\mathsf{ck}}$ mapping ℓ-block elements to codewords, has minimum distance d. Moreover, the distance d can be observed in a single chunk of codewords.*

The proof of the above corollary is trivial from the above Proposition 4.1.

Remark 1 *Note that two different length inputs of ECCode^{*}_{d} can map to codewords (with different length) having minimum distance just one. In other words, one codeword can be simply prefix of the longer which has one extra block or field element. To handle variable length inputs, we will process length independently as described in Section 1.4.2.*

Now we state that VPV-Hash is Δ-universal hash function. A keyed hash function h_k is called **ϵ-ΔU (universal) hash function** if for all distinct inputs x and x' from input space and for all difference δ, the following *δ-differential probability*

$$\mathsf{diff}_{\mathcal{H},\delta}[x, x'] := \mathsf{Pr}_{\mathbf{K} \xleftarrow{\$} \mathcal{K}}[h_{\mathbf{K}}(x) - h_{\mathbf{K}}(x') = \delta] \tag{4.1}$$

is at most ϵ. The notation "$\mathbf{K} \xleftarrow{\$} \mathcal{K}$" means that the random variable $\mathbf{K}$ is uniformly distributed over the key space $\mathcal{K}$.

Proposition 4.3 *Suppose the keys for VPV-Hash is $(K_1, \ldots, K_\ell)$ and $(K_1^*, \ldots, K_\ell^*)$ (used for hashing length). Moreover, assume that we only use $K_{\ell-2}^*$, $K_{\ell-1}^*$, K_ℓ^* as described in TriviA. Then,*

1. *VPV-Hash$_{ck}$ is $1/2^{128}$-Δ-universal hash function for arbitrary length messages.*

2. *Even when the K_i^*'s are fixed for a length suitable for one message, VPV-Hash$_{ck}$ is $1/2^{124}$-Δ-universal hash function.*

Proof. As $\mathsf{ECCode}^*_{d,\mathsf{ck}}$ has distance d, we can apply the result from [15] to prove the bounds for fixed length even for a fixed key corresponding to length. To prove for variable length we need to argue in two cases. If the length key is not leaked and different for two messages whose differential probability is computed then due to full entropy of length key we will have $1/2^{128}$ differential probability. When length key is revealed, we still have 64 bit entropy in length key due to different lengths of two messages. This can happen when two padded message differ by at least two blocks. Let us assume that the last 64 bit of 128 bit hash output has the full entropy and so differential probability is $1/2^{64}$ for these 64 bits. Now for the first 64 bits we can still use the entropy of hash key. Note that we should have at least two blocks of 64-bit hash keys which is present in longer messages. Due to $1/2^{31}$-regular property of pseudo-dot product hash for those keys (independent with length keys) we have $1/2^{62}$ differential probability for the the first 64 bits of hash. This completes the proof.

4.2.2 Privacy of TriviA

We give a particularly strong definition of confidentiality or privacy, one asserting indistinguishability from random strings. Consider an adversary A who has access of one of two types of oracles: a "real" encryption oracle or an "ideal" authenticated encryption oracle. A real authenticated encryption oracle, F_K, takes as input (D, M) and returns (C, Tag) $= F_K$(D, M). Whereas an ideal authenticated encryption oracle \$ returns a random string R with $|\mathsf{R}| = |\mathsf{M}| + 1$ for every fresh pair (D,M). Given an adversary A (w.o.l.g. we assume a **deterministic adversary**) and an authenticated encryption scheme F, we define the (full) **privacy-advantage** of A by the distinguishing advantage of A distinguishing F from \$. More formally,

$$\mathbf{Adv}_F^{\mathrm{priv}}(A) := \mathbf{Adv}_F^{\$}(A) = \mathsf{Pr}_K[A^{F_K} = 1] - \mathsf{Pr}_{\$}[A^{\$} = 1].$$

Note that TriviA has similarity with AEAD-5 in [18]. So we adopt the proof of the theorem stated in that paper. However, we have better bound due to insertion of hash value T of nonce and associated data in the current state of streamcipher. In [18], T (in [18], it was denoted by V in Table-4) is initialized and the streamcipher is freshly started. So the collision probability for T is about $q^2/2^{128}$. However, if we insert it and if we assume that the nonce can not be repeated then the collision probability of state is about $q^2 \times 2^{-384}$ which is neglible. So we have our following modified theorem: Let A be an adversary which can make q queries (including both encryption and decryption queries) at an aggregate of total σ associated data and message blocks. Moreover, nonce can not be repeated. Then the privacy-advantage of the adversary A against TriviA is given by,

Theorem 4.4

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathit{TriviA}}(A) \;\leq\; \eta + \frac{qn}{2^{128}}.$$

where η denotes the maximum distinguishing advantage over all adversaries B making at most σ block queries to Trivia-SC and running in time T_0 (which is about time of the adversary A plus some insignificant overhead).

4.2.3 Authenticity of TriviA

We say that an adversary A **forges** an authenticated encryption F if A outputs (D, C) where $F_K(\mathsf{D}, \mathsf{C}) \neq \perp$ (i.e. it accepts and returns a plaintext), and A made no earlier query (D, M) for which the F-response is C. It can make s attempts to forge after making q queries. We define that A forges if it makes at least one forges in all s attempts and the **authenticity-advantage** of A by

$$\mathbf{Adv}^{\mathrm{auth}}_{F}(A) = \mathsf{Pr}_K[A^{F_K} \text{ forges}].$$

We can argue similarly for authenticity also. Suppose adversary makes q forgery attempts. After removing the collision probability for state and distinguishing advantage, i.e., $\eta + \frac{qn}{2^{128}}$, we can consider differential probability for final tag. Note that even though intermediate tag is leaked, the final tag is actually a linear combination of all intermediate tag and so adversary must forge one of the intermediate tag. For any such forging attempt, the differential probability of intermediate tag (for matching nonce-associated data) is bounded by 2^{-124}. So we can have similar following theorem: Let A be an adversary which can make q queries (including both encryption and decryption queries) at an aggregate of total σ associated data and message blocks. The authenticity-advantage of the adversary A against TriviA is given by,

Theorem 4.5

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathit{TriviA}}(A) \;\leq\; \eta + \frac{qn}{2^{128}} + \frac{q}{2^{124}}.$$

where η denotes the maximum distinguishing advantage over all adversaries B making at most σ block queries to Trivia-SC and running in time T_0 (which is about time of the adversary A).

Chapter 5

Features

5.1 Main Features of the Cipher

5.1.1 Efficient and Nonce Misuse Resistant

One of the most important requirement for most of the nonce based authenticated encryption scheme is the nonce should be distinct for every invocation. Otherwise the privacy of those schemes can be compromised easily. Nonce can be chosen as a counter value or random integer (such that repitation occurs with negligible probability) to ensure the distinctness. But in practical scenario such as lightweight applications or some other applications it is very challenging to to ensure distinct nonces since in lightweight applications either it needs to store in a nontamperable state or require some hardware source of randomness. Again in case of GCM-AES [13] and CCM-AES [5] occurance of same nonce for two different invocations under the same key, but with distinct plaintext compromises the confidentiality of the plaintexts as well as the authenticity and integrity under the key. Hence *Nonce Misuse Resistance* is an important criteria for designing the AE scheme.

TriviA permits nonce misuse with degradation in its security bounds, though the tuple of nonce and associated data should be distinct. However, we recommend not to repeat nonce because of this security degradation. Thus, TriviA provides *Nonce Misuse Resistance* with degradation in security.

Various Nonce Misuse Resistant AE Schemes like SIV [17], BTM [8], HBS [9] are deterministic in nature and they don't use nonce. Instead they use distinct IVs which are processed from the message and associated data with the requirement that the message and associated data tuple should be distinct. But these constructions are less efficient since they are two pass construction (they have to process the message twice), where one pass is reserved for encryption another is for authentication. TriviA produce encryption key and authentication key in one pass (after the intermediate state is produced) and message is processed with this keys only once and hence it is more efficient.

5.1.2 Presence of Intermediate Tag

TriviA computes a sequence of intermediate tags before computing the final tag. Since creating a single authentication tag requires additional memory to store the complete message, TriviA creates a sequence of intermediate tags where each tag in the sequence can be computed from the previous tag without storing all messages. The final tag will be computed from the last intermediate tag from the sequence.

This construction is useful for limited buffer implementation and has been proven secure in this implentation structure. The main disadvantage of the scheme is that the presence of sequence of tags makes the ciphertext, authentication tag tuple a bit longer. Hence the user has been given the flexibilty to make the computation of the intermediate tags optional. We would like to note that intermediate tag in authenticated encryption has been described in [16].

5.1.3 Low Hardware Area in the Implementation

The base component for TriviA is the stream cipher Trivia-SC, which needs low hardware requirement, since it is a variant of Trivium, which has low hardware requirement. Beside this TriviA uses the same VPV-Hash [15] universal hash for both the intermediate state value computation and the authentication tag generation, hence both the operation uses the same circuit.

Moreover VPV-Hash universal hash requires low hardware area than that of the hardware efficient Topelitz construction [11]. For example, $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$ uses only one 32 bit multiplier to compute 128 bit tag, which is much better than compared to that of Toeplitz construction which requires four 32 bit multiplier to compute 128 bit tag. The figure below gives a abstract view of the circuit required for VPV-Hash.

Clearly VPV-Hash requires two VHorner Circuit and one 32 bit multiplier. The $\mathsf{VHorner} - n/d$ circuit executes the inner for loop of the $\mathsf{VMult}_{n/d}$ algorithm. The figure represents the circuit for the VPV-Hash Universal hash which is run by TriviA-ck algorithm in two phases for internal state generation and authentication tag generation respectively.

5.1.4 Advantage and Drawback of the Implementation

TriviA uses Trivia-SC to produce the encryption key, authentication key and VPV-Hash to produce the authentication tag. Trivia-SC is very fast and it is parallelizable since it can process 64 bit at a time in a single clock cycle. VPV-Hash also uses very low hardware area. We also generate a intermediate state from the associated data and the nonce and the construction does not permit nonce misuse. Moreover the scheme also produces intermediate tags in the limited buffer scenario.

TriviA-ck uses two invocation of Trivia-SC, thus two initializations for Trivia-SC both of 1152 rounds respectively occurs. But the main drawback of the scheme

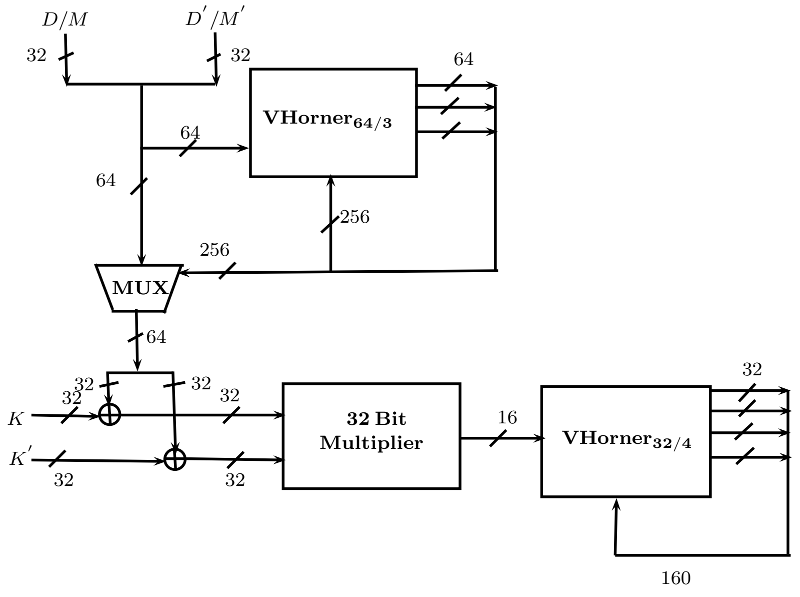

Figure 5.1: Circuit for VPV-Hash

is that TriviA generates intermediate tag (optional) after processing a chunk of data, size of the ciphertext and tag pair becomes larger.

5.2 Implementation Issues

The base component of TriviA for generating encryption and authentication key is Trivia-SC. which is parallelizable upto 64 bit. This means the stream cipher can produce upto 64 bit stream at a single clock cycle. This parallelizibilty has been used by the KeyExt64 andUpdate64 subroutines. Moreover Trivia-SC is an extended version of Trivium which performs good in both software and hardware, thus depicts that Trivia-SC is one of the best candidate for generating keystreams for encryption and authentication.

VPV-Hash uses ECCode error correcting code of minimum distance d, for $d = 4$ to expand the data. It has been verified that $\mathsf{ck} \leq L$ where $\mathsf{L} = 2^{32}$. This result says that the maximum possible size of a chunk can be L which is quite large and ECCode can process more data in a single invocation.

VPV-Hash algorithm generates the authentication tag by using the hashkey generated by Trivia-SC. It also produces a sequence of intermediate tags (optional if the buffersize is limited) after processing a chunk/s of data. This requires storing a chunk of message and then applying the VPV-Hash over the chunk/s. This is done in the incremental manner. That is if the first intermediate tag ITag_1 is computed for G_1 then the next intermediate tag ITag_2 will be calculated for (G_1, G_2) where $G_1 = (g_1, \cdots g_\ell)$ and $G_2 = (g_1, \cdots g_{2\ell})$ with $|g_i| = 32$ bits $\forall i$ and $\ell = \mathsf{ck} + 3$. But this doesn't require to store both the chunks because the second intermediate tag can be computed by using G_2 and ITag_1. Note that More precisely.

$$
\begin{aligned}
\mathsf{ITag}_1 &= V_{\mathsf{word},d,\mathsf{l}} \cdot G_1 \\
\mathsf{ITag}_2 &= V_{\mathsf{word},d,2\mathsf{l}} \cdot \begin{pmatrix} G_1 & G_2 \end{pmatrix} \\
&= \alpha^{\mathsf{l}} \cdot V_{\mathsf{word},d,\mathsf{l}} \cdot G_1 + V_{\mathsf{word},d,\mathsf{l}} \cdot G_2 \\
&= \alpha^{\mathsf{l}} \cdot \mathsf{ITag}_1 + V_{\mathsf{word},d,\mathsf{l}} \cdot G_2
\end{aligned}
\tag{5.1}
$$

Where

$$
V_{\mathsf{word,d,l}} = \begin{pmatrix}
1 & \cdots & 1 & 1 & 1 \\
\alpha^{\mathsf{l}-1} & \cdots & \alpha^2 & \alpha & 1 \\
\alpha^{2(\mathsf{l}-1)} & \cdots & \alpha^4 & \alpha^2 & 1 \\
\vdots & \cdots & \vdots & \vdots & \vdots \\
\alpha^{(\mathsf{l}-1)(d-1)} & \cdots & \alpha^{2(d-1)} & \alpha^{d-1} & 1
\end{pmatrix}
$$

The above description of intermediate tag generation confirms the optimization of the buffer size. Hence it has great advantage for low-end devices (keeping in mind that, block-wise adversaries are considered only when buffer size is limited implying low-end device).

5.3 Advantages Over AES-GCM

TriviA has the following advantages over AES-GCM :

- AES-GCM needs distincts nonce for every invocation under the same key where in TriviA we can repeat nonce with degradation in its security. Note that, we always need distinct nonce and associated data tuple for each invocation.

- AES-GCM isn't security against blockwise adaptive adversaries since it leaks some partial information by decrypting a invalid ciphertext when buffer is limited. TriviA has overcome this disadvantage by incorporating a sequence of intemediate tags before the final tag.

- Our constructions has much higher bit security for authencity compare to AES-GCM.

- AES-GCM uses a 128 bit field multiplier to process 128 bit data in a single clock cycle. our construction uses a 32 bit field multiplier to process 64 bit data. Hence it can process 128 bit data by using two 32 bit field multiplier. Clearly two 32 bit multiplier takes less hardware area than a 128 bit multiplier with a factor less than $\frac{1}{2}$.

Chapter 6

Changes from TriviA-v1

The changes made for the updated version is listed below.

1. Beside key and Nonce loading, the stream cipher state is loaded with 0 bits except for three positions, instead of all 1 bits.
2. Intermediate state is reduced from 160-bit to 128-bit and the encryption queries should be made with distinct nonce.
3. PDP hash reorders the bits of input block and the key block for the 32-bit field multiplication.
4. A 32-bit portion of the variable key is multiplied with a nonzero constant.

6.1 Changes in TriviA-SC Module

The version TriviA-v1 uses the stream cipher TriviA-SC, with all one binary string loaded into the 384 internal state along with the 128-bit key and the 128-bit IV. The updated TriviA-SC in the updated version of TriviA is initialized by loading an 128-bit key and an 128-bit IV into the 384-bit initial state, and fixing all remaining bits to 0, except for B_{103}, B_{104}, and B_{105}. This updation is done to resist slide attack on the stream cipher state, as all 1 constant is helps the attacker to construct a valid start state with high probability after some rounds.

6.2 Change in TriviA Mode

The version TriviA-v1 generates a 160-bit intermediate state after the associated data processing, and this permits the relaxed nonce-respecting adversary to make atmost 2^{32} queries with the same nonce. The updated version of TriviA recommends not to misuse nonce and reduces the intermediate state size from

160 to 128-bits. Moreover, the use of reduced 128-bit intermediate state will optimize the hardware area by preserving the security bound under the restriction that the adversary can not repeat the nonce during the encryption queries.

This updation makes the circuit for TriviA more uniform, as the algorithm uses $\mathsf{VPV\text{-}Hash}_{\mathsf{ck}}$ to process both the message and the associated data (TriviA-v1 used $\mathsf{VPV\text{-}Hash}_{5,\ \mathsf{ck}}$ to process the associated data for producing an 160-bit internal state and uses $\mathsf{VPV\text{-}Hash}_{4,\ \mathsf{ck}}$ to process the message for producing an 128-bit tag). This also reduces the hardware register requirement by 32-bit (from 160-bit to 128-bit). It removes the overhead for switching between $\mathsf{VPV\text{-}Hash}_{5,\ \mathsf{ck}}$ and $\mathsf{VPV\text{-}Hash}_{4,\ \mathsf{ck}}$.

6.3 Changes in VPV-Hash Module

6.3.1 Changes in the Blockwise Hash Module PDP.

VPV-Hash module in Trivia-v1 uses PDP blockwise hash to process a message-key pair $(x = (x_1, x_2), k = (k_1, k_2))$ by $(x_1 \oplus k_1)(x_2 \oplus k_2)$. The updated version uses a blockwise hash PDP^* to process a message-key pair $(x = (x_1, x_2, x_3, x_4), k = (k_1, k_2, k_3, k_4)) \in (\mathbb{F}_{2^{16}}^4)^2$ in a different way as $((x_1||x_3) \oplus (k_1||k_3))((x_2||x_4) \oplus (k_2||k_4))$. PDP^* is same as PDP, except it reorders the message and key bits before the 32-bit field multiplication.

This reordering actually increases the efficiency of the implementation of Trivia in 32-bit platform. Note that, we can use Karatsuba algorithm for the 32-bit field multiplication. In the 32-bit platform, $x_1||x_2$ and $k_1||k_2$ are received in the first clock cycle and $x_3||x_4$ and $k_3||k_4$ are received in the second clock cycle. Thus, we can efficiently compute the 16-bit polynomial multiplications $(x_1 \oplus k_1)(x_2 \oplus k_2)$ and store this value along with $(x_1 \oplus k_1)$ and $(x_2 \oplus k_2)$ for its use in the next clock cycle. These stored values are used in the second clock cycle to get the 32-bit field multiplication output. Hence, this reordering makes the 32-bit field multiplication efficient and faster in a 32-bit platform. Moreover, we assume that the stream cipher produces random keystream corresponding to PDP . Thus, reordering key bits does not change the security bound.

6.3.2 Changes in the Variable Key Addition Technique.

The VPV-Hash algorithm described in TriviA-v1 adds an input variable key K^* to compute T_c. In the updated version of TriviA, the variable key is addition function is same except the second 32-bit component of the variable key is multiplied with a nonzero constant α_{32}^2 before the addition. Thus, if $K^* = (k_1^*||k_2^*||k_3^*||k_4^*)$ then the updated key will be $K^{**} = k_1^*||\alpha_{32}^2.k_2^*|k_3^*||k_4^*$. Note that, this technique removes the need to store the variable key in a separate register (Described by Algorithm 8). Here, we can add each of the 64-bit components of the variable key with the intermediate result in a online manner (add the 64-bit key whenever it is extracted from the stream cipher). Note that, K^* and K^{**} follows the same distribution and the security bound will be preserved.

The above change is done to reduce the hardware footprint for TriviA. In the hardware implementation for TriviA-v1, we need to store the variable key $K^* = (K_x^1, K_x^3)$ while processing a message x, as they are extracted at different clock cycles by KeyExt64 (line 16, Algorithm 7) and they are also used in a different clock cycle. The KeyExt64 algorithm extracts key in the iteration $\ell_x - 1$ (extracts K_x^3) and $\ell_x - 3$ (extracts K_x^1) when the last block of x is not full or in the iteration $\ell_x - 2$ (extracts K_x^3) and $\ell_x - 4$ (extracts K_x^1) when the last block of x is full. The extracted key is finally added to the intermediate value. We have observed that, if we extract the 64-bit key K_x^1 (in the iteration $\ell_x - 3$) and immediately add it with the intermediate result, then the second 32-bit component of K_x^1 will be propagated linearly towards the end of the tag computation in the iteration ℓ_x with a factor of α_{32}^2. In the final clock cycle, we extract K_x^3 and add it with the final tag value. This is actually equivalent to XORing K^{**} with the intermediate value to produce T_c. Note that, this technique makes XORing variable key online and eleminates the need to store K^{**}.

Chapter 7

Design Rationale

TriviA-ck uses Trivia-SC and VPV-Hash$_{\mathsf{ck}}$ as the mathematical components and it can be viewed as a integration of these two components. Trivia-SC is an extended version of Trivium [1], which is one of the eStream finalists and can be efficiently implemented both in software and hardware. The subroutines of Trivia-SC are almost equivalent to Trivium except the non-linearity in the output bit equation. We add this non-linearity to resist the attack in [12]. Hence Trivia-SC requires low hardware area and it is comparable to Trivium. Moreover Trivia-SC gives equivalent software performance as Trivium since it provides same 64 bit parallelism as in Trivium and can process 64 bit in a single clock cycle.

We have also observed that after 896 round initializations the output bit polynomial (over key and IV bits) behaves like random functions. Hence it functions like one time pad when XORed with a message after 896 round initalization. Thus TriviA-ck runs atleast 960 round initialization of Trivia-SC in both the phases.

The second component of TriviA-ck is VPV-Hash$_{\mathsf{ck}}$ which is an efficient universal hash function with minimum number of multiplication. As we have mentioned earlier VPV-Hash$_{\mathsf{ck}}$ performs better than efficient Toeplitz construction in terms of the number of multiplications and the hardware area.The number of multiplications in VPV-Hash$_{\mathsf{ck}}$ is optimum [15] and it is 4 times less than that of the Toeplitz construction.

VPV-Hash module in this updated version is same as that in the older version TriviA-v1 except it reorders the bits of the key and input blocks during the PDP* blockwise hash call. This technique makes the 32-bit field multiplication in PDP* more efficient in a 32-bit platform. The reordering technique helps to compute a 16-bit polynomial multiplication in a clock cycle and then use it in the next clock cycle. More specifically, it process a message-key pair $(x = (x_1, x_2, x_3, x_4), k = (k_1, k_2, k_3, k_4)) \in (\mathbb{F}_{2^{16}}^4)^2$ in a different way as $((x_1||x_3) \oplus (k_1||k_3))((x_2||x_4) \oplus (k_2||k_4))$.

This reordering actually increases the efficiency of the implementation of Trivia in 32-bit platform. In case of Karatsuba algorithm for the 32-bit field multiplication, $x_1||x_2$ and $k_1||k_2$ are received in the first clock cycle and $x_3||x_4$ and $k_3||k_4$ are received in the second clock cycle. Thus, we can efficiently compute the 16-bit polynomial multiplications $(x_1 \oplus k_1)(x_2 \oplus k_2)$ and store this value along with $(x_1 \oplus k_1)$ and $(x_2 \oplus k_2)$ for its use in the next clock cycle. These stored values are used in the second clock cycle to get the 32-bit field multiplication output. This make the 32-bit field multiplication more efficient in a 32-bit platform.

Due to limited buffer implementation such as low end devices the decryption algorithm has to release some part of the plaintext before the authentication is done. This can cause some attacks on some constructions [10] since the adversaries against authenticity called blockwise adaptive adversary would have access of partial decryption oracles. To resist such attacks, we recommend to use a sequence of intermediate tags (along with the final tag), generated in a such a manner that during decryption, the plaintext computation is independent of the intermediate tags.

We have generated a intermediate state using the associated data and Trivia-SC generated bit streams. The generated intermediate state is then mixed internal state registers of Trivia-Sc and the final tag is produced. This technique has been implemented to make the scheme nonce-misuse resistant. This is indeed an important requirement since schemes like AES-GCM [13], AES-CCM [5] occurance of same nonce for two different invocations under the same key compromises confidentiality oe authenticity of the plaintext. TriviA-ck requires that the nonce may be repeated but not the tuple of associated data and nonce.

The designers have not hidden any weaknesses in this cipher. One can analyze the polynomials corresponding to state bits for the further analysis of weaknesses in the cipher. Polynomial density for the polnomials corresponding to state bits have been checked and found that they behave like random polynomials. Still it may be a good area for analyzing the weaknesses in the cipher.

the designers have tried to exploit the dependency between state bits (from StateExt64) and key streams (from KeyExt64) mathematically but couldn't find any. One may try to find dependencies between the state bits and keystream bits and try to explore attacks.

Chapter 8

Intellectual Property

This cipher or any parts of the cipher, doesnt have an kind of patents. Existance of any kind of patent on any parts of the cipher is not known to the submitters. If any of this information changes, the submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

Chapter 9

Consent

The submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

Bibliography

[1] Christophe De Cannire, Bart Preneel, *"Trivium," New Stream Cipher Designs* **4986** (2005), 244–266, *The eSTREAM Finalists*, Lecture Notes in Computer Science, 2005. Citations in this document: §1.4.1, §4.1, §7.

[2] Avik Chakraborti, Anupam Chattopadhyay, Muhammad Hassan, Mridul Nandi, *TriviA: A Fast and Secure Authenticated Encryption Scheme* (2015), *CHES 2015*, Lecture Notes in Computer Science, 9293, 2015. Citations in this document: §1.

[3] Avik Chakraborti, Mridul Nandi, *TriviA-ck-v1* (2015). URL: `http://competitions.cr.yp.to/round1/triviackv1.pdf`. Citations in this document: §1.

[4] Itai Dinur, Adi Shamir, *Cube Attacks on Tweakable Black Box Polynomials* (2009), 278–299, *EUROCRYPT 2009*, Lecture Notes in Computer Science, 5479, 2009. Citations in this document: §4.1.

[5] Morris Dworkin, *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality* (2004), *NIST Special Publication 800-38C*, 2004. Citations in this document: §5.1.1, §7.

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

[7] Pierre-Alain Fouque, Thomas Vannet, *Improving Key Recovery to 784 and 799 rounds of Trivium using Optimized Cube Attacks* (2013), *FSE 2013*, Lecture Notes in Computer Science, 8424, 2013. Citations in this document: §4.1.

[8] Tetsu Iwata and Kan Yasuda, *BTM : A Single-Key, Inverse-Cipher-Free Mode for Deterministic Authenticated Encryption* **5867** (2009), 313–330, *Selected Areas in Cryptography*, Lecture Notes in Computer Science, 2009. Citations in this document: §5.1.1.

[9] Tetsu Iwata and Kan Yasuda, *HBS : A Single-Key mode of Operation for Deterministic Authenticated Encryption* **5665** (2009), 394–415, *Fast*

Software Encryption, Lecture Notes in Computer Science, 2009. Citations in this document: §5.1.1.

[10] Antoine Joux, Gwenlle Martinet and Fredric Valette, *Blockwise-Adaptive Attackers: Revisiting the (In)Security of Some Provably Secure Encryption Models: CBC, GEM, IACBC* **2442** (2002), 17–30, *CRYPTO*, Lecture Notes in Computer Science, 2002. Citations in this document: §7.

[11] 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: §5.1.3.

[12] Alexandar Maximov, Alex Biryukov, *Two Trivial Attacks on Trivium.* (2007), 36–55, *Selected Areas in Cryptography*, Lecture Notes in Computer Science, 4876, 2007. Citations in this document: §4.1.1, §4.1.1, §7.

[13] David A. McGrew, John Viega, *The Galois/Counter Mode of Operation (GCM)* (2005). URL: `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf`. Citations in this document: §5.1.1, §7.

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

[15] Mridul Nandi, *On the Minimum Number of Multiplications Necessary for Universal Hash Constructions* (2013), *IACR Cryptology ePrint Archive 2013:574*, 2013. Citations in this document: §1.4.2, §4.2.1, §4.2.1, §5.1.3, §7.

[16] Josef Pieprzyk, Pawel Morawiecki, *Parallel authenticated encryption with the duplex construction*, IACR Cryptology ePrint Archive 2013 (2013). Citations in this document: §5.1.2.

[17] P. Rogaway and T. Shrimpton, *Deterministic Authenticated-Encryption : A Provable-Security Treatment of the Key-Wrap Problem* **4004** (2006), 373–390, *Advances in Cryptology - Eurocrypt*, Lecture Notes in Computer Science, 2006. Citations in this document: §5.1.1.

[18] Palash Sarkar, *Modes of Operations for Encryption and Authentication Using Stream Ciphers Supporting an Initialisation Vector*, Cryptography and Communications (2014). Citations in this document: §4.2.2, §4.2.2, §4.2.2.

[19] National Institute of Standards and Technology. URL: `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`. Citations in this document: §4.1.