Software Implementation and Analysis of Low Cost Hash Functions

Desabattula Sreecharan

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Rourkela – 769 008, Odisha, India

Software Implementation and Analysis of Low Cost Hash Functions

Dissertation submitted in

May 2015

to the department of

Computer Science and Engineering

of

National Institute of Technology, Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology (Dual Degree)

in

Information Security

by

Desabattula Sreecharan

(Roll 710cs2040)

under the supervision of

Dr. Ashok Kumar Turuk

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Rourkela – 769 008, Odisha, India

Department of Computer Science and Engineering
National Institute of Technology, Rourkela
Rourkela – 769 008, Odisha, India

May, 2015

Certificate

This is to certify that the work in the thesis entitled ***Software Implementation and Analysis of Low Cost Hash Functions*** by ***Desabattula Sreecharan*** is a record of an original work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Dr. Ashok Kumar Turuk

Acknowledgement

I would like to express my earnest gratitude to my project guide, Prof. A.K. Turuk for believing in my ability. His profound insights has enriched my research work. I am indebted to all the professors, batch mates and friends at National Institute of Technology Rourkela for their cooperation. My full dedication to the work would have not been possible without their blessings and moral support. Finally, I am grateful to my family for continuous motivation and encouragement without which the present research would not have been possible.

Desabattula Sreecharan

Author’s Declaration

I hereby declare that all work contained in this report is my own work unless otherwise acknowledged. Also, all of my work has not been submitted for any academic degree. All sources of quoted information has been acknowledged by means of appropriate reference.

Desabattula Sreecharan

Roll: 710cs2040

Department of Computer Science and Engineering,

National Institute of Technology, Rourkela

Abstract

The need for light-weight cryptographic algorithms has been increasing in recent times with the birth of RFID-enabled Internet of Things (IoT). The main challenge everyone faces when designing security protocols for RFID systems is the resource constrained environment in which these cryptographic algorithms have to be deployed. These resources include the limited memory and processing capabilities, limited power supply and short range of operation. The higher the memory offered, the costlier the system gets. Traditional cryptographic algorithms developed so far are rendered useless in such an environment and new algorithms have to be developed from scratch keeping in mind the limited resources available.

Many such security protocols have been proposed which often assume that a hash function is present inside the RFID-tag. These hash algorithms form an additional overhead to the processing and hence several low cost functions have been researched. Two of such functions are PHOTON and SPONGENT which were primarily hardware oriented designs based on the hermetic sponge construction. The present work involves implementation of these two hardware designs in software to be deployed on to the RFID tags and analyzing these implementations.

List of figures

List of tables

List of Abbreviations

RFID: Radio Frequency Identification

IC: Integrated circuits

LF: Low Frequency

HF: High Frequency

UHF: Ultra High Frequency

MHz: Mega Hertz

KHz: Kilo Hertz

DoS: Denial of Service

MD5: Message Digest Algorithm - 5

SHA-3: Secure Hash Algorithm - 3

EPC: Electronic Product Code

AES: Advanced Encryption Standard

RC (*v*): Round Constant for round v

IC (i): Internal Constant for element i

GE: Gate Equivalents

GHz: Giga Hertz

GNU: GNU's not UNIX!

GCC: GNU C Compiler

Contents

Chapter
Introduction

1.1 RFID Systems

Communication in Radio Frequency Identification (RFID) systems occurs through radio waves in order to identify objects/persons automatically. Readers and tags are the main parts of an RFID System. Readers send queries to tags in their particular transmission ranges for some data contained in the tags and tags reply with the queried information such as identification (ID) numbers.

RFID tags are constrained devices in terms of storage and computing capability. Some tags don't even have their own power source. They are instead powered wirelessly by the readers.

Tags can be divided based on:

- Their operating frequency: low, high and ultra-high.
- Power source: passive and active.
- Processing ability: there are tags with a limited amount of processing and storage and tags without any ICs (called chip-less tags)

Passive tags use the radio energy transmitted by the reader whereas active tags have their own power source and transmitters. Most RFID systems utilize one of three general bands:

- Low Frequency (LF) i.e., 125 kHz to 134 kHz
- High Frequency (HF) i.e., 13.56 MHz

- Ultra High Frequency (UHF) i.e., 860 to 930 MHz

Any RFID tag contains at least the following components:

1. An IC for processing and modulation/demodulation of signals
2. An antenna for communication purposes

The basic setup of any RFID system is as shown below.

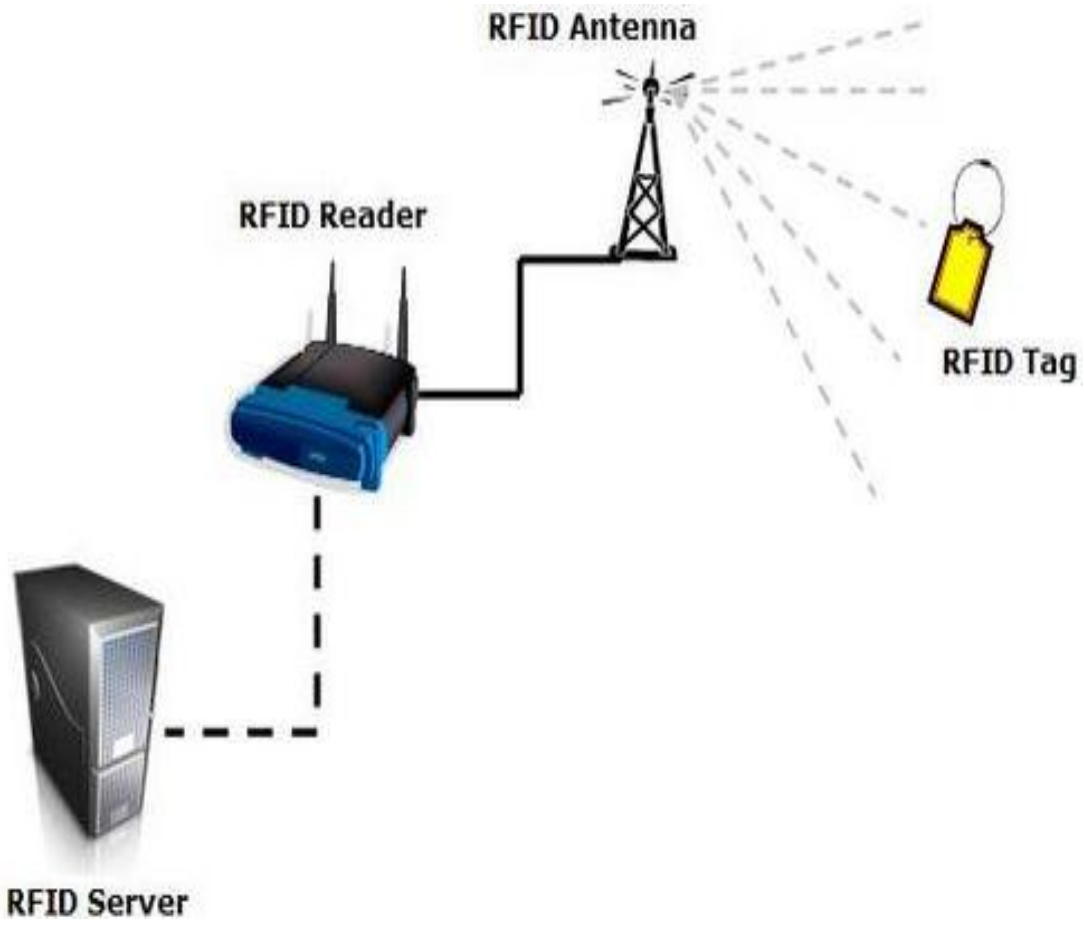

Figure 1 A typical RFID System

Every tag has a unique code for identification. In a passive tag environment, the RFID reader generates a low-level radio frequency magnetic field that works as a power source for the tag. The tag responds to the reader's query and broadcasts its existence in the field via the same medium and transmits its unique identification data. The reader captures this data and decodes it before passing it to an application software that resides on a system other than the reader. The application software will then search and compare this data with the necessary information stored in the backend system which will typically be a database. Depending on the result produced by the backend system, access and/or authorization can be ensured in an RFID system.

1.2 Security Aspects

Due to the high vulnerability of RFID tags and the ease of identifying/tracking the RFID-tagged objects, security and privacy form an important factor in successful deployment of the systems. Most of the existing RFID systems do not provide adequate security and can give away valuable information regarding the object/person to whom the tags belong. Passive attacks like tracking/identifying can be done by any adversary which are hard to detect. The most common attacks on an RFID system are eavesdropping on the communication channel, replay attack where duplicate data is sent again after some time interval, man-in-the-middle attack where an adversary masquerades as genuine tag/reader, loss of data/unavailability due to denial of service attack (DoS), forgery, hijacking and skimming as well as physical attacks like tampering with the tags.

Addressing the security concerns invariably is difficult given the wide range of application scenarios in which the RFID systems may be used along with the limited resources including computing capabilities. Commercial market demands lower cost tags with high user functionality while security/privacy aspects add additional cost and hinders response times. Even if all of resources offered by the tag are utilized for security protocols, the traditional cryptographic

algorithms cannot be used to provide proper protocols in passive tags because of their limited resources. A higher-end passive tag can offer at most 2000 gates for security purposes whereas any standard cryptographic algorithm requires gates in tens of thousands [1]. In addition, the storage requirement of these algorithms is very high where only limited memory is offered by the RFID tags.

1.3 Hash Functions

The constrained environment, limited memory and processing capabilities thus dictate the cryptographic algorithm to be implemented so as to meet the necessary security standards of the RFID systems. Many cryptographic professionals gave considerable attention to device authentication and user privacy and various new protocols have been proposed. In most of these protocols, especially the ones designed to protect privacy of users and to hide interactions of the tag, it is generally assumed that a cryptographic hash function is provided on the tag and can be used without significant overhead on response times, memory or computational time.

However, the hash function to be used has not been specified in reality and a general assumption is made that the existing hash functions with high throughput are also efficient enough to be implemented in constrained environments. This has been proven wrong by Feldhofer and Rechberger [2]. Traditional hash functions such as MD5 or SHA3 offer 256-bit security which is the main reason for their high memory and processing requirements. The electronic product code (EPC) is one type of data that can be found on RFID tags which is 96-bit long. Therefore, a security level of 256 bits is certainly overkill when only 96-bit data is present and in most scenarios a security of 80 or 64 bits sufficient. Shamir followed the same view while proposing SQUASH to be used in RFID systems [3].

Thus, there is a need for designing hash functions that have fewer resource requirements than the traditional ones so that they can be deployed in a constrained environment such as an RFID tag. Next chapter explores one such design called the sponge construction based on which many hash functions have been proposed with proven security and less resource hunger.

Chapter 2
Sponge Construction

2.1 Basic concept

The sponge construction operates on a padding rule and fixed-length permutation/transformation producing a function called ***sponge function*** which maps an input of variable length to an output of variable-length. The input to this function is a binary string of any length, and output is also a binary string of requested length. The sponge function is a generalization of fixed input length and fixed output length ciphers/hash functions. A finite state is present and the function operates on it with interleaved application of permutation function for input as well as output.

A random oracle is a theoretical black-box which satisfies all the known security criteria for hash functions and no security criteria can be found anew that a random oracle does not satisfy. It maps an input of variable length to an output string of infinite length. This output string is purely random which implies that this string has bits that are uniformly distributed. The only rule of the random oracle is that identical inputs generate identical hash values. On the other hand, the output of a hash function has finite bits (n). So, an ideal hash function should represent a random oracle with output reduced to n bits. To find a collision, the expected number of queries to a random oracle is $2^{n/2}$. In order to find a (second) pre-image, the expected number of calls is 2^{n}. A hash function is said to be compromised if the complexity of an attack is less than the bounds stated for the random oracle.

2.2 Construction

The sponge construction can be imagined as a function **F** whose input is a fixed length message and whose output length can be chosen by the user as per the requirement. There is an internal permutation/transformation, ***f***, of fixed-length. This function works on an internal state of length **b-bits** called the width of the sponge. The internal state is divided logically into **r** and **c** where **c** is the capacity and **r** is the bitrate.

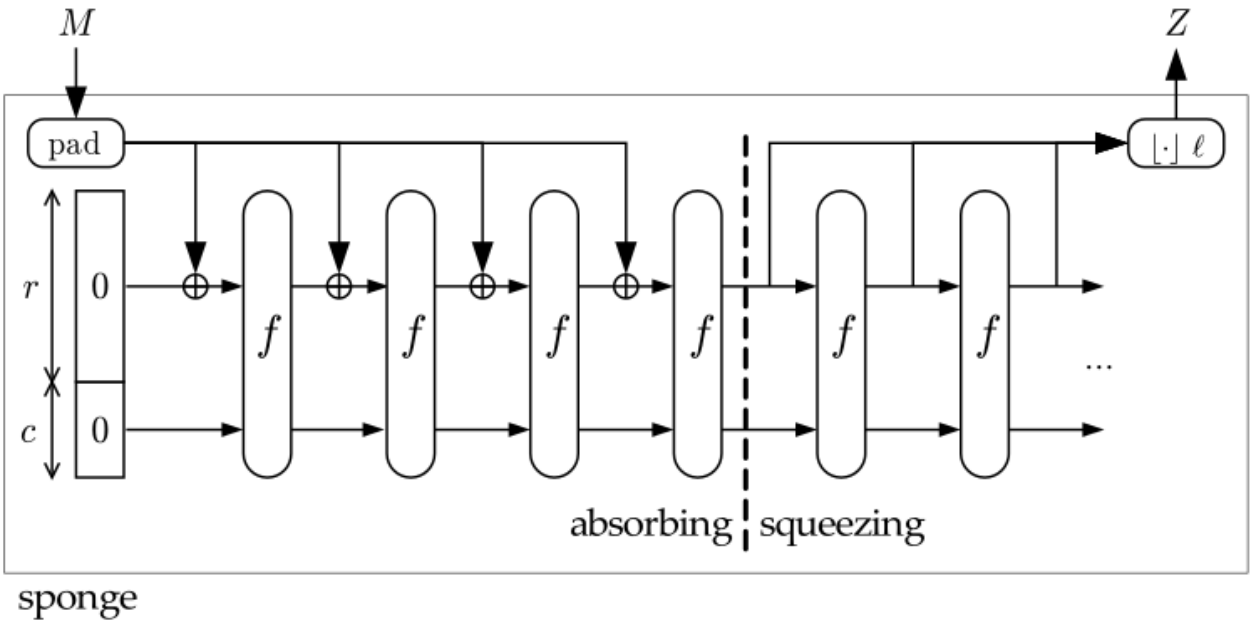

Figure 2 A Sponge Construction Setup

A reversible padding rule is used to pad the input message such that it is a multiple of the bit rate **r**. Then the message is divided into several blocks of **r-bits**. All the bits of the state are then initialized to some value and the sponge operates in the following phases:

- Absorbing phase: the most significant *r* bits of the state are XORed with the *r*-bit input blocks sequentially, while applying the internal function ***f*** between the XOR

operations. Once the entire input is consumed, the sponge goes to the squeezing phased to produce the output hash values.

- Squeezing phase: The most significant *r* bits of the state are output in blocks while applying the function ***f*** between each squeeze. The will choose the number of output blocks.

The *c* least significant bits of the state called the capacity are never directly operated upon neither by the absorption nor the squeezing phase.

2.3 As a security reference

A finite memory is used in any iterated function to process the message and store it. The state of the iterated function at any point of time can represent the input blocks received so far. Collisions can happen in the state as it contains a finite number of bits. Random oracles do not incur collisions in their internal state because such a concept of finite state does not exist. Thus, for functions with variable output length, a random oracle cannot be a reference to claim security as there are no collisions in a random oracle.

An alternative to the random oracle model for expressing security exist which is called **random sponge**. A random sponge is a sponge construction with ***f*** chosen randomly from the set of transformations/permutations over ***b*** bits. Apart from the effects due to finite memory, it has been proven that a random sponge function is as strong as a random oracle. Thus a reference model has been created in the form of random sponge construction to express security claims of hash functions and stream ciphers.

If the permutation used by the sponge function can be distinguished from a randomly-chosen function, then there is a high probability of an attack. A new design called as hermetic sponge strategy in which the sponge construction has a permutation ***f*** that does not show any properties called **structural distinguishers** which can be exploited and used to attack.

Chapter 3
Low Cost Hash functions

As stated above, a random sponge is as strong as a random oracle if the underlying sponge function is chosen randomly from the all transformation/permutation functions. Thus, various hash functions with proven security have been proposed based on the sponge construction and using different sponge functions. Two of such hardware proposals are SPONGENT and PHOTON. These are primarily designed to reduce the area occupied on the RFID tag so that the hardware footprint of these hash functions is as low as possible.

3.1 SPONGENT

The construction of SPONGENT follows the same sponge construction. Initial state $\boldsymbol{b=r+c}$ is 0 and the input message is padded with a 1000…..0 sequence such that the size is a multiple of $\boldsymbol{r}$. The input message is then divided into r-bit blocks. $\boldsymbol{n}$ is the length of the output hash value.

The sponge construction follows the phases:

- Initialization phase in which the message is padded by 100…..0 until it is multiple of $\boldsymbol{r}$. Then the input message is divided into blocks of $\boldsymbol{r}$-bits.
- Absorbing phase in which the most significant $\boldsymbol{r}$-bits of the internal state are XORed with the $\boldsymbol{r}$-bit input message blocks while applying the permutation π_b between the XOR operations.

- Squeezing phase in which the most significant ***r***-bits of the state are given as output, with application of the permutation π_b between the outputs as shown in the figure, until ***n*** bits are returned. This ***n*** can be chosen to be any number as required

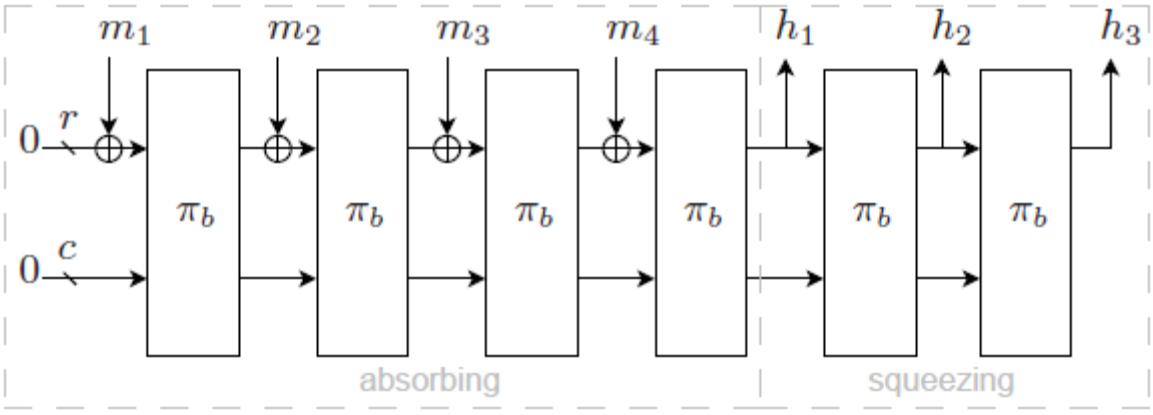

Figure 3 Sponge Construction of SPONGENT

The permutation π_b: $F^b_2 \rightarrow F^b_2$ is a function that can be roughly stated as:

for i = 1 to R **do**

State ← REVERSE (lCounter$_b$ (i)) ⊕ ***State*** ⊕ lCounter$_b$ (i)

State ← sBoxLayer$_b$ (***State***)

State ← pLayer$_b$ (***State***)

end for

The following are the definitions of the sbox and pbox as stated in the original work [5]

1. **sBoxLayer$_b$**: This is a substitution box that operates on 4-bits and is defined as $S : F^4_2 \rightarrow F^4_2$. The S-box fulfills the PRESENT S-box criteria [4]. The following table describes the substitutions to be made in hexadecimal format

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	E	B	D	0	2	1	4	F	7	A	8	5	9	C	3	6

Table 1PRESENT type SBox

2. **pLayer$_b$**: This layer moves bit ***j*** to a new bit position defined by $P_b(j)$ as below. It is a slight extension of the (inverse) PRESENT bit-permutation

$$P_b(j) = \begin{cases} j.\frac{b}{4} mod(b-1) \; if \; j \in \{0,1,2, \dots \dots \dots .. b-2\} \\ \\ b-1 \; if \; j = b-1 \end{cases}$$

3. **lCounter$_b$ (i):** This is a $\lceil \log_2 R \rceil$-bit LFSRs. Say ζ is the root of unity in the binary finite field, the 6-bit LFSR used in SPONGENT-88 is defined by the primitive trinomial $\zeta^6 + \zeta^5 + 1$ and is initialized with the value 000101.

The following parameters are used for SPONGENT-88 which has the lowest cost.

- Length of hash value, ***n = 88***
- Capacity, ***c = 80***
- Bit rate, ***r = 8***
- No. of Rounds, ***R = 45***

3.2 PHOTON

PHOTON works in the same way as SPONGENT except that the internal permutations here are AES-like. This choice is particularly advantageous as we have the confidence of proven security of AES and all AES based hash functions while proving security of PHOTON.

Apart from the internal permutation, PHOTON operates in the same way as any other sponge construction would. It consists of a t=r+c bits of internal state and n bits of output digest length. Absorption phase consists of dividing the m-bit input message into several r-bit blocks and padding with 1000….00 at the end to exact multiple of r blocks. Each r-bit block is then absorbed into the leftmost bits of internal state S of the sponge construction which was previously initialized as $S_0 = \{0\}^{t-24} \,||\, n/4 \,||\, r \,||r`$ and the permutation function is applied on the entire state.

Once all the r-bit blocks are absorbed, squeezing phase occurs in which, r`-bit blocks of data are extracted from the least significant bit positions of the state S while applying the permutation function in between. Thus, from [6]

$$S_{i+1} = P(S_i \oplus (m_i || \{0\}^c))$$

The permutation function used here is similar to that of AES (Advanced Encryption Standard): The internal state can be thought of as a matrix of dimensions ***d x d*** where each element is of ***s*** bits. Therefore the total size of internal state would be $d^2 \, x \, s$ bits. Similar to AES, it has four stages as described below in each round and the total number of rounds being 12.

AddConstants:

Two constants are used in this layer. Round constants RC (*v*) where v is the round number starting from 1 is XORed with the first column of the internal state. Different internal constants

IC_d (i) are XORed into the same first column of the internal state. The overall operations for round v are $S'[i; 0] = S[i; 0] \oplus RC(v) \oplus IC_d(i)$ for all $0 \leq i < d$.

RC (v) = [1; 3; 7; 14; 13; 11; 6; 12; 9; 2; 5; 10].

IC_d (i) = [0; 1; 3; 6; 4]

SubCells:

An ***s*** bit substation is done at this layer to each of the cells of the internal state using the given Sbox. This Sbox is PRESENT type and labelled SBOXPRE [4]. The transformation function in this layer is

$S' [i; j] = SBOX(S [i; j])$ for all $0 \leq i, j < d$.

SBOXPRE = [0xc; 0x5; 0x6; 0xb; 0x9;

0x0; 0xa; 0xd; 0x3; 0xe;

0xf; 0x8; 0x4; 0x7; 0x1; 0x2]:

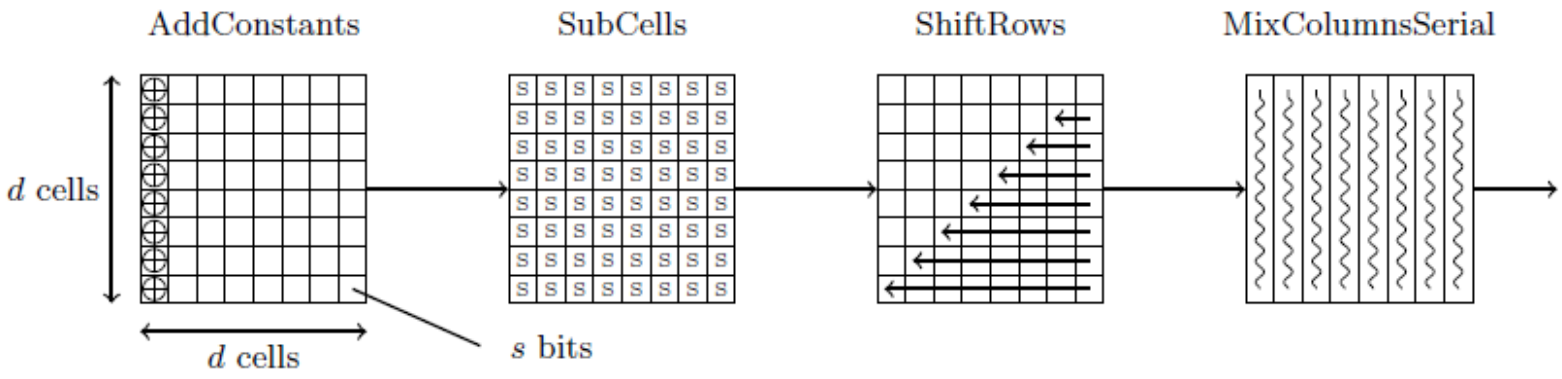

Figure 4 Single Round of Permutation in PHOTON

ShiftRows:

Just like the AES cipher, all the cells of every i^{th} row are rotated to the left by i positions of the column. The function would be,

S' [i; j] = S [i; (j + i) mod d] for all $0 \leq i, j < d$.

MixColumns:

This final mixing is applied to each column of the state separately. For each j input vector (S [0, j] . . . . , S [d – 1, j]) ^{T} representing a column, the matrix $\mathbf{A_t}$ =Serial ($Z_0 \ldots Z_{d-1}$). The function is for all $0 \leq j < d$:

$$(\boldsymbol{S}`[\boldsymbol{0}, \boldsymbol{j}], \ldots\ldots\ldots\ldots \boldsymbol{S}`[\boldsymbol{d-1}, \boldsymbol{j}])^{\boldsymbol{T}} = \boldsymbol{A_t^d} \times (\boldsymbol{S}[\boldsymbol{0}, \boldsymbol{j}], \ldots\ldots\ldots\ldots \boldsymbol{S}[\boldsymbol{d-1}, \boldsymbol{j}])^{\boldsymbol{T}}$$

and A_t^d =

1	2	9	9	2
2	5	3	8	13
13	11	10	12	1
1	15	2	3	14
14	14	8	5	12

The following are the parameters for the present implementation:

- $\boldsymbol{n}$ = 88
- $\boldsymbol{r}$ = 20
- $\boldsymbol{r'}$ = 16
- $\boldsymbol{s}$ = 4
- $\boldsymbol{d}$ = 5

Chapter 4

Motivation and Present Work

4.1 Motivation

The two hash functions discussed, SPONGENT and PHOTON have the lowest cost in terms of hardware implementations. The internal permutation/transformation function has been kept simple and requires less memory while simultaneously offering acceptable security levels. This principle of design is focused primarily to reduce the hardware footprint i.e., the gate equivalents (GE) required but the same can be applied when to software implementations as well.

A simple sponge function with no additional lookup tables has to have a low footprint in software implementation as well. This gave us the motivation to explore the possibilities of a software implementation of the same hash functions and their footprint in terms of memory occupied, run time over different sized input etc.,

There are several advantages of a software implementation over a hardware design. Consider a situation where a large number of RFID tags with hardware cryptographic algorithms have been used in some particular application. Suppose, the present hash function has been compromised by generic attacks, i.e., attacks which ignore the strength of internal structure but concentrate on other weaknesses in the hashes. All the tags would then need another cryptographic algorithm that offers higher security. But as the tags themselves are hardware etched, they are rendered completely useless and new tags with the new hash functions are to be placed. A huge cost should bared to replace all the tags and is sometimes infeasible if there are large number of

tags or if the application is critical. Such a situation needs a software implementation of the hash function to provide easy updates to newer and better algorithms.

The minimum amount of data a tag needs to hold is currently non-standardized as the scope of application of RFID systems is huge. This minimum data may include the unique identification codes, privacy policies, passwords, hash values etc. The electronic product code is currently 96-bits long. But there is no guarantee that this length would be fixed forever. As we have witnessed in the case of IPv4, the 32 bit address space is used up and now IPv6 is being explored. The same will be the case in RFID systems but the growth will be much faster with exponential usage.

Although the software implementation seems to be a logical choice, there are few disadvantages too. The hardware design offers the most compact function which can be fit in less than 2000 GE as evident in [5] and [6]. In addition, the software implementation adds additional overhead of managing bits and preprocessing.

Thus the manufacturer must make a choice between compactness and the ability to update regularly both of which offers the same level of security.

4.2 Present work

The above two algorithms have been implemented in the language C. A major challenge during the implementation was managing the bits. C does not offer manipulation of individual bits. There are third party libraries that do manage individual bits but in a redundant way which will further dump addition overhead on the program.

Declaring an integer variable for each bit would remove the overhead of managing bits but would increase the memory required. There is a tradeoff between memory and processing

overhead. As the implementation is intended for constrained devices, memory is of highest value. Compromising on the storage area defeats the whole purpose of low cost implementations. Therefore, a choice has been made to manage the individual bits while reducing the storage requirements to as low as possible.

The size of the smallest data type that can be used in C is ***8 bits*** for a ***char*** type. The permutation of the discussed hash constructions operate on ***4-bits.*** Hence, a structure has been used that contains a char type out of which only 4 bits will be used for all operations of the internal permutation. To ensure only 4 bits shall be used, the structure is declared in the form:

```
struct element {
        unsigned char val : 4;
};
```

Another viable option to save memory is to use all the ***8 bits*** of the char data type and manage the bits according to which bits form the least significant part and which form the most significant part. This choice has been explored and it was found that the program becomes too complex to operate in real time.

The slight wastage incurred by using only 4 bits out of the 8 available bits of the ***char*** type was simple. Hence, the choice.

4.2.1 Algorithms

SPONGENT

Pad the input M with 1000….001 such that it is a multiple of r

State = [0000….00]

1. **for each block of r-bit sized input (m_i) do**
2. **State = State $\oplus$ m_i**
3. **for i = 1 to R do**
4. **state ← (reverse ($lCounter_b(i)$))$\oplus$ state $\oplus$ $lCounter_b(i)$**
5. **state ← $sBoxLayer_b$(state)**
6. **state ← $pLayer_b$(state)**
7. **end for**
8. **end for**

9. **for each block of desired output (n-bits/r-bits):**
10. **output r-bits from state**
11. **for i = 1 to R do**
12. **state ← (reverse ($lCounter_b(i)$)) $\oplus$ state $\oplus$ $lCounter_b(i)$**
13. **state ← $sBoxLayer_b$(state)**
14. **state ← $pLayer_b$(state)**
15. **end for**
16. **end for**

PHOTON

/* S is the internal state represented by a dxd matrix of t bits total size with s bits at * each cell*/

PHOTON:

1. **For each block of r-bit input:**
2. **Absorb(m_i,S)**
3. **Permute(S)**
4. **End**
5. **For each block of r`-bit output required:**
6. **Squeeze(S)**
7. **Permute(S)**
8. **End**

Absorb:

9. **S = S ^ m_i**

Squeeze:

10. **Print r` bits from S**

Permute

11. **For i = 0 to N_r:**
12. **AddConstants()**
13. **SubCells()**
14. **ShiftRows()**
15. **MixColumns()**
16. **End**

Chapter 5

Results and Analysis

5.1 Results

The hash values of the following messages have been calculated using both the functions and the results are as shown in the screen shots below. The programs are run on a 2.67 GHz Intel core i5 processor using GNU GCC compiler.

Messages:

1. 96 bit messages
    a. “NIT ROURKELA”
    b. “NIT ROURKFLA”
    c. “NIT RPURKFLA”
    d. “NIT SPURKFLA”
2. 304 bit message
3. 7808 bit message

5.1.1 Hash function SPONGENT

1. 96 bit message

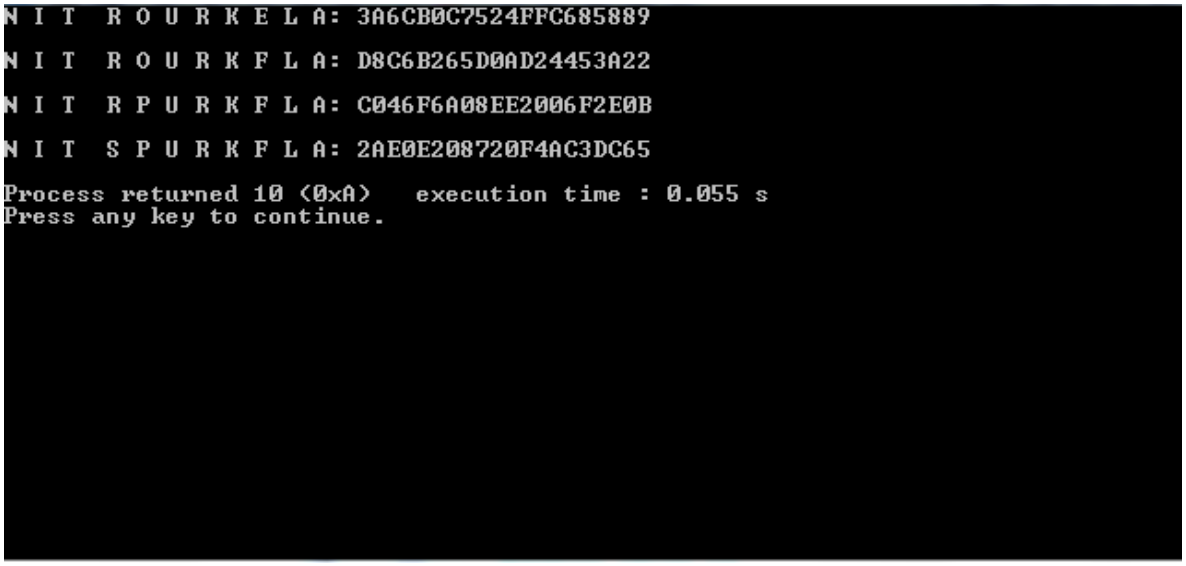

Figure 5 SPONGENT hash values for 96 bit messages

2. 304 bit message

Figure 6 SPONGENT hash value for 304 bit message

3. 7808 bit message

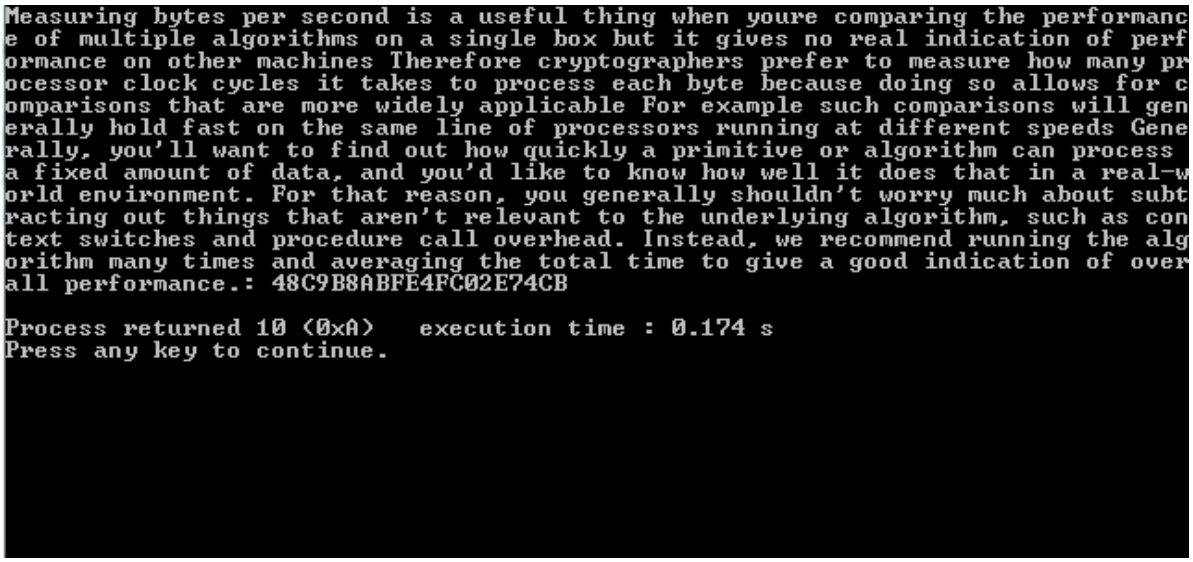

Figure 7 SPONGENT hash value of 7808 bit message

5.1.2 Hash function PHOTON

1. 96 bit messages

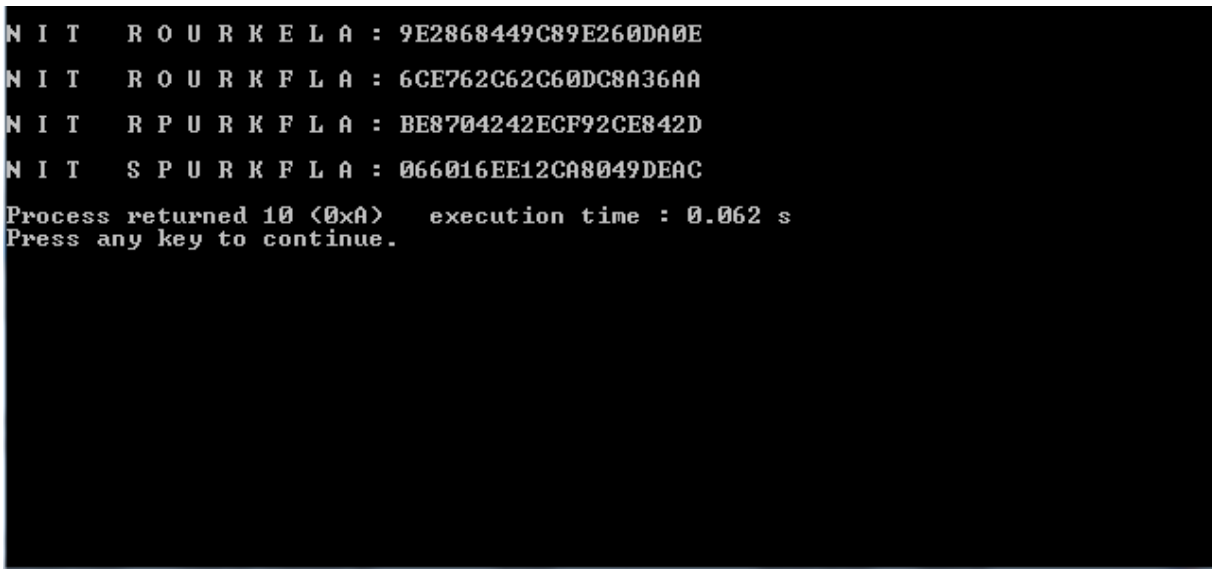

Figure 8 PHOTON hash values for 96 bit messages

2. 304 bit message

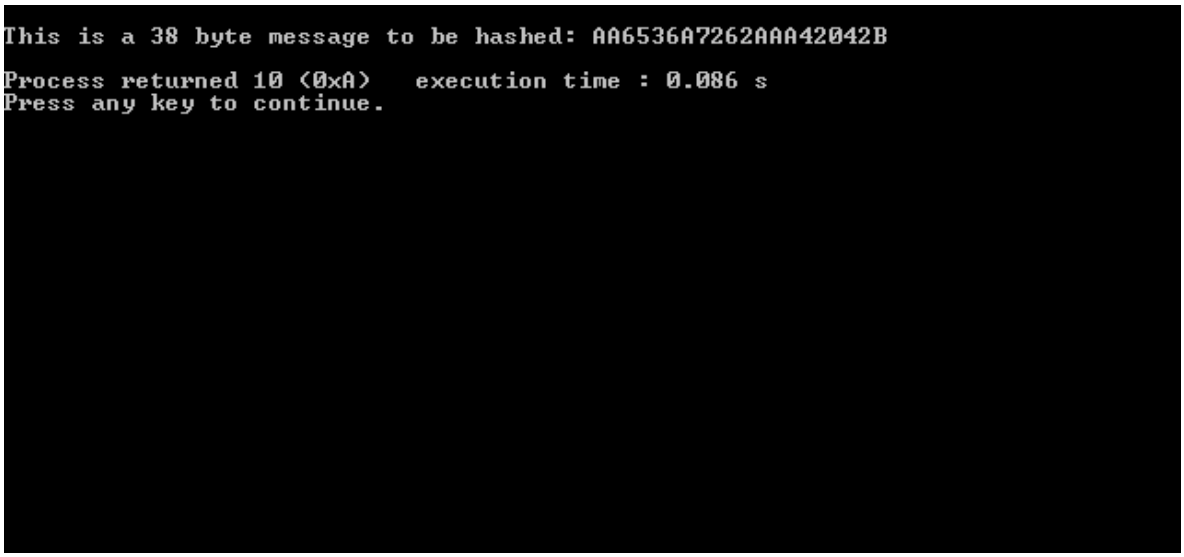

Figure 9 PHOTON hash value for 304 bit message

3. 7808 bit message

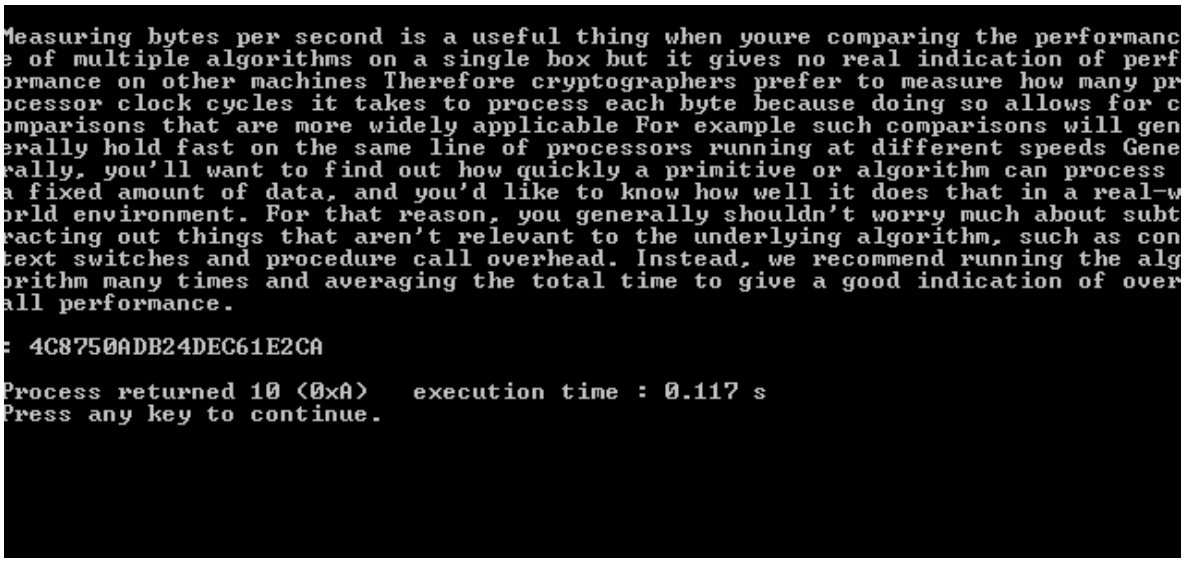

Figure 10 PHOTON hash value for 7808 bit message

5.2 Analysis

The following are the results of the analysis performed on both of the implementations using same bit messages to encrypt.

Table 2 A comparison of the software implementations

PARAMETERS	***SPONGENT***	***PHOTON***
Memory	5014	4136
Runtime for general use case (96 bits)	0.003132	0.001986
Cycles/Byte	25.38	19.53

Cycles/Byte measures the number of cycles the CPU has spent to process one byte of data. It is calculated using the following formula

$$Cycle\ per\ Byte = \frac{Cycles\ per\ Second}{Bytes\ per\ Second}$$

Cycles are the CPU instruction cycles. Thus Cycles per second is the clock speed of the processor. Bytes per second is the number of Bytes processed by the algorithm in one second.

Bytes per second is calculated by running the programs N times over B byte input messages and calculating the time taken, E for the entire operation. Thus bytes per second will be N*B/E.

As the result suggests, SPONGENT needs more cycles to process one byte of data than PHOTON. This is reverse of what the hardware analysis done by [5] and [6] suggests. SPONGENT occupies only 728 GE where PHOTON occupies more than 1120 GE.

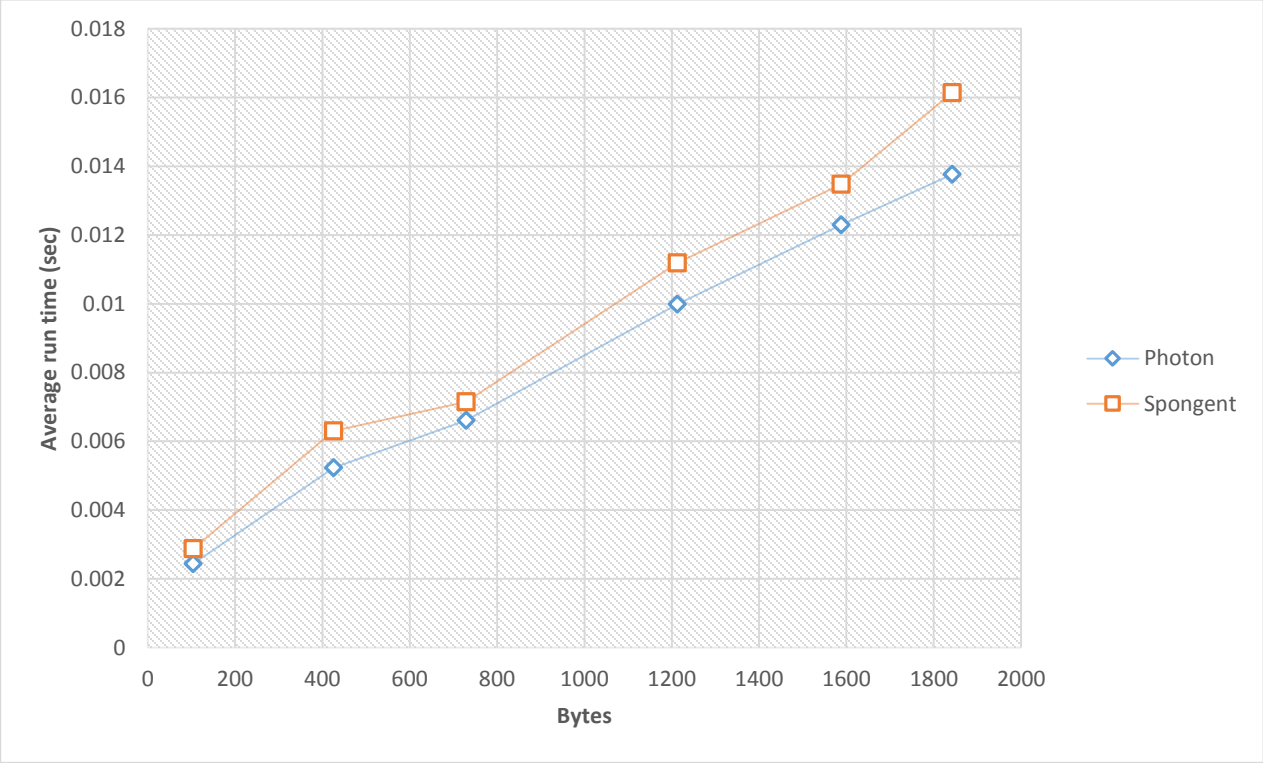

Figure 11 Bytes vs Runtime of PHOTON and SPONGENT

The above graph clearly confirms the fact that on a minute scale, PHOTON outperforms SPONGENT. This difference seems insignificant due to the optimizations by the processor on which the programs are run. This difference will be more hard pressed if the size of input message

increases and/or if the program is deployed in a constrained environment where the processing capability and buffer space are limited.

Chapter 6
Conclusion and future work

Although SPONGENT has the lowest hardware footprint according to its implementers [5], its software cost is higher than PHOTON. In the hardware implementations PHOTON occupied approximately 1120 GE [6] whereas SPONGENT required 728 GE [5] only. But the size and runtime of SPONGENT is greater than PHOTON and thus, in a software implementation of hash functions PHOTON is a better choice. Other analysis including the variation of runtime with message size further proves that software implementations behave differently than that of hardware designs and a careful analysis has to be carried out before making a choice.

The programs are implemented in language C and compiled using a GNU GCC compiler. Execution of the programs is done on an Intel Core i5 processor with clock speed of 2.67 GHz. This does not represent even closely the constrained environments in which the programs are to be deployed but a measure of Cycles/Byte gives us an approximate metric independent of processors. PHOTON again proves that it is a better choice independent of the underlying hardware.

The sponge construction is a promising design which can offer high levels of security while leaving the overhead and complexity involved in block ciphers. New hash functions can be proposed using the same sponge construction and different internal permutations. The security of the new proposals can also be proved with proper research. The process of proving that cryptographic algorithm is completely secure and safe to use takes several years and hundreds of

cryptographers. Thus, new proposals and analysis of their security has been left for future work involving a lot of research.

Bibliography

1. A. Juels and S.A. Weis. “Authenticating Pervasive Devices With Human Protocols”, In V. Shoup, editor, CRYPTO 2005, volume 3126 of LNCS, , 293–198, Springer-Verlag, 2005.
2. M. Feldhofer and C. Rechberger. “A Case Against Currently Used Hash Functions in RFID Protocols”, In First International Workshop on Information Security (IS’06), volume 4277 of LNCS, pages 372-381, Springer-Verlag, 2006.
3. A. Shamir. “SQUASH - a New MAC With Provable Security Properties for Highly Constrained Devices Such As RFID Tags”, In K. Nyberg, editor, FSE 2008, to appear. Springer.
4. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C. “PRESENT: An Ultra-Lightweight Block Cipher”, In: Paillier, P., Verbauwhede, I. (eds.) CHES. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007).
5. Andrey Bogdanov , Miroslav Knežević , Gregor Leander , Deniz Toz , Kerem Varici , Ingrid Verbauwhede, “SPONGENT: a lightweight hash function”, Proceedings of the 13th international conference on Cryptographic hardware and embedded systems, September 28-October 01, 2011, Nara, Japan.
6. Jian Guo , Thomas Peyrin , Axel Poschmann, “The PHOTON family of lightweight Hash functions”, Proceedings of the 31st annual conference on Advances in cryptology, August 14-18, 2011, Santa Barbara, CA.

7. Rieback. M, Crispo. B, Tanenbaum. A, “The Evolution of RFID Security”, Pervasive Computing. Jan 2008.
8. Chandramouli. R, Grace. T, Kuhn. R, Landau. S, “Security Standards for the RFID Market”, The IEEE Computer Society, 2005.