Implementation and Evaluation of Authenticated Encryption Algorithms on Java Card Platform

MASTER THESIS

Rajesh Kumar Pal

Brno, Spring 2017
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Rajesh Kumar Pal

**Advisors:** RNDr. Petr Švenda, Ph.D.
Dr. Chester Rebeiro, Ph.D.
Acknowledgments

First and foremost, I would like to express my profound gratitude and sincere thanks to my supervisors RNDr. Petr Švenda and Dr. Chester Rebeiro. Without their guidance, support, encouragement, and faith in me, it would not have been possible to complete this thesis. RNDr. Petr Švenda taught me how to program and manage the nuisances of a small embedded system such as smart card. I am indebted to the pains he took for reviewing all my codes and suggesting measures to make them elegant and fool-proof. Dr. Chester Rebeiro provided help on security aspects and technical writing. I am also thankful to Prof Matyas for providing timely-advice, encouragement, and imparting deep-knowledge on computer security fundamentals.

I am thankful to my organization and Government of India who has given me the opportunity to pursue Master studies at Masaryk University. I express thanks to my Indian friends who accompanied me to the course at Masaryk. I am grateful to Martin Ukrop for making our stay at Brno, a comfortable and memorable one.

Rajesh Kumar Pal
Keywords

authenticated encryption, CAESAR, smart card, Java card, embedded system, ACORN, AEGIS, ASCON, CLOC, MORUS
Abstract

In this thesis, five authenticated ciphers namely ACORN, AEGIS, ASCON, CLOC, and MORUS from the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) are implemented and evaluated on Java card platform. The Java implementation using Java Card Technology is made for each cipher. The ciphers are optimised to exploit the onboard resources such as cryptographic coprocessor, RAM, EEPROM of Java card platform. The main focus of the evaluation is authenticated cipher’s timing performance and memory requirements. Their performance is determined through timing measurements. The general performance of each cipher is quantified, and compared with one another. The results are helpful in selection of authenticated encryption algorithm for low compute-power, memory-constrained, embedded devices. This work enables availability of authenticated ciphers on Java card.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Algorithm Specific Optimisations</td>
<td>21</td>
</tr>
<tr>
<td>4.3.1</td>
<td>ACORN</td>
<td>21</td>
</tr>
<tr>
<td>4.3.2</td>
<td>AEGIS</td>
<td>21</td>
</tr>
<tr>
<td>4.3.3</td>
<td>ASCON</td>
<td>22</td>
</tr>
<tr>
<td>4.3.4</td>
<td>CLOC</td>
<td>22</td>
</tr>
<tr>
<td>4.3.5</td>
<td>MORUS</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Github Repository of Authenticated Ciphers</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation of Authenticated Ciphers on Java Cards</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Test Cases</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>Timing Performance</td>
<td>24</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Authenticated Encryption on Java Card</td>
<td>24</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Software Versus Hardware Implementation</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Memory Footprint</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions and Future Work</td>
<td>29</td>
</tr>
<tr>
<td>References</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

1.1 Authenticated encryption. .............................................. 1
1.2 Smart card. .......................................................... 3

2.1 The concatenation of 6 LFSRs in ACORN-128 (adapted from [10]).
   $f_i$ indicates the overall feedback bit for the $i^{th}$ step; $m_i$ indicates the
   message bit for the $i^{th}$ step. ..................................... 6
2.2 AEGIS encryption. ...................................................... 7
2.3 The state update function of AEGIS (adapted from [11]). ........ 8
2.4 The sponge mode for ASCON encryption (adapted from [12]). ... 9
2.5 One round of Permutation used in ASCON. ......................... 9
2.6 CLOC authenticated encryption. ..................................... 10
2.7 MORUS encryption. .................................................. 11
2.8 The state update function of MORUS (adapted from [15]). In Rotl
   xxx_yy, xxx yy is 128 32 for MORUS-640 and 256 64 for MORUS-1280. 12

4.1 Java card. .............................................................. 16
4.2 Communication protocol. ............................................. 17
4.3 Communication protocol for authenticated encryption. ........... 18

5.1 Timing results of authenticated encryption. ......................... 25
5.2 Timing results of authenticated decryption. ......................... 25
5.3 Comparison of CLOC using AES in software and hardware. ....... 26
5.4 Time-Memory efficiency matrix. ..................................... 28
List of Tables

1.1 Secret-key cryptography .............................................. 2

2.1 The recommended parameter sizes for ASCON .................. 9

2.2 The recommended parameter sizes for MORUS ................ 11

2.3 Comparison of selected candidates. The type of cipher is shown as
stream for stream-cipher, block for block-cipher, sponge for key-less
permutation based encryption, dedicated for encryption structure
similar to Type-3 Feistel schemes. ¹ signifies that AEGIS is parallel-
elizable only for encryption, not for decryption ................ 13

5.1 Memory footprint of authenticated algorithms. * represents CLOC
using hardware AES .................................................. 27
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>Authenticated Encryption</td>
</tr>
<tr>
<td>AEAD</td>
<td>Authenticated Encryption with Associated Data</td>
</tr>
<tr>
<td>APDU</td>
<td>Application Protocol Data Unit</td>
</tr>
<tr>
<td>CAESAR</td>
<td>Competition for Authenticated Encryption: Security, Applicability, and Robustness</td>
</tr>
<tr>
<td>IOT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>SHA-3</td>
<td>Secure Hash Algorithm-3</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter presents an overview of authenticated encryption and smart cards. We describe the motivations, objective and contributions made by the thesis. The chapter also presents the organization of this thesis.

1.1 Authenticated Encryption

Authenticated Encryption (AE) [Figure 1.1] or Authenticated Encryption with Associated Data (AEAD) is a form of symmetric key encryption which provides confidentiality, integrity, and data authenticity in one go. In a single step, the encryption is combined with generation of integrity-verifiable tag while decryption is done with integrity verification. Confidentiality ensures protection of information by converting input to indistinguishable random bits, while authentication guarantees originality and integrity of data by facilitating easy detection of any tampering / change in the data. The unique property of authenticated encryption is that it simultaneously provides confidentiality and authentication. Most of the authenticated algorithms use block or stream cipher as a base to encrypt the data while utilise a structure to preserve the encryption state. The state update function is fed with some tweak also called authenticated data and all the input to generate a tag which assures authenticity. The authentication tag helps detection of possible forgery attempt. It is advantageous to combine confidentiality and integrity assurances into a single scheme. However, practical attacks into widely used protocols and applications (including Secure Socket Layer (SSL)/Transport Layer Security (TLS)) show that securely combining a confidentiality mode with

![Figure 1.1: Authenticated encryption.](image-url)
an authentication mode is challenging and error prone.

Authenticated encryption is a type of secret-key cryptography (Table 1.1) which provides confidentiality and integrity of the messages using a shared key by sender and receiver. Due to the performance advantages over public-key cryptography, secret-key cryptography remains the high-performance workhorse of cryptography. The variants of secret key cryptography are:

- **Block Cipher**: A short fixed-length message is encrypted using the secret key shared by the sender and receiver. For example, a popular block cipher AES [1] encrypts a 16-byte (128-bit) block with a 128-bit, 192-bit, or 256-bit key.

- **Stream Cipher**: A variable-length message is encrypted using a public nonce and a secret key shared by the sender and receiver. For example, Rabbit cipher from eSTREAM portfolio.

- **Message-authentication Code**: A variable-length message is reduced to an authenticator using a public nonce and a secret key shared by the sender and receiver. Cryptographic hash functions such as Secure Hash Algorithm-3 (SHA-3) and variants of block ciphers are often used to construct message-authentication codes.

- **Authenticated Cipher**: A variable-length message is encrypted as well as authenticated using a public nonce and a secret key shared by the sender and receiver.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Message length</th>
<th>Encrypts</th>
<th>Authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block cipher</strong></td>
<td>Fixed</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Stream cipher</strong></td>
<td>Variable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Message-authentication code</strong></td>
<td>Variable</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Authenticated cipher</strong></td>
<td>Variable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The various combinations of block ciphers, stream ciphers, message authentication codes, and hash functions are used to implement authenticated ciphers. In ISO/IEC 19772:2009, six different authenticated encryption modes (namely OCB 2.0, Key Wrap, CCM, EAX, Encrypt-then-MAC (EtM), and GCM) have been standardized. The commonly used AE based on block ciphers are Offset Codebook Mode (OCB) [2] or Galois/Counter Mode (GCM) [3].

1.2 **Smart Cards**

Smart cards are very attractive security options to security practitioners because of it being self-sufficient, isolated and physically-securable hardware having small attack surface. Smart card, as shown in Figure 1.2, is like a micro-computer having
its processor, coprocessor, memory (transient: RAM and persistent: EEPROM), and I/O ports for interfacing with other devices. It often contains special cryptographic coprocessors for AES encryption, RSA, DH, etc. Its strength lies in the fact that it can securely store secret keys and perform cryptographic operations. These tiny computers having their own memories and processors are widely used in telecommunication as SIM (Subscriber Identity Module), payment and banking systems as credit / debit cards, transportation, and healthcare.

A smart card does not contain its own power supply, display, or keyboard. It interacts with a Card Acceptance Device (CAD) or card reader through using a communication interface, provided by a collection of eight electrical or optical contact points, as shown in Figure 1.2b. Smart cards are constrained embedded devices with limited memory and processing capability. To make any implementation really useful in practice, it must be efficient in both time and space. Optimizing algorithms for efficiency on smart cards is challenging because fancy programming constructs of full-fledged programming languages are not supported. The memory limitation requires careful optimization like reusing existing memory storages. To exploit processing capability of smart cards, data reorganizations are often required. Further the transfer of data from computer to smart cards being bandwidth limited adds on to the delays.

1.3 Motivations and Objective

Authenticated encryption algorithms will be the main workhorse for secret-key cryptography in future. The Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [6] will facilitate a portfolio of AE ciphers. The ever increasing use of smart cards requires that authenticated encryption schemes are also available on smart cards. We expect that by porting AE on Java Card, data security will be enhanced on small and embedded devices. This work may be useful to users who are looking for authenticated encryption on smart cards. Further this work may also help in analysis of selected algorithms towards selection in final portfolio of CAESAR competition.

The objective of this work is to efficiently implement authenticated encryption algorithms, which will be the future work-horse for secret-key cryptography, on
Java Card.

1.4 Thesis Contributions

The key contributions of the thesis are as follows:

(i) Implementation and optimization of authenticated encryption schemes on Java Card.
The main contribution of this thesis is to implement authenticated encryption algorithms from CAESAR competition on Java Card.

(ii) Evaluation of authenticate encryption on Java Card.
The evaluation and analysis of different authenticated encryption schemes provides a fair idea about their strengths, weaknesses and opportunities for employing each one.

(iii) Enable availability of selected authenticated encryption on smart card.
We enable availability of selected authenticated encryption implementations on smart card to the world through github.

1.5 Thesis Organization

The thesis is organized into the following chapters:

- Chapter 1 starts with a overview of authenticated encryption schemes and discusses challenges in porting AE algorithms on Java Card. We also discuss the objective and contributions made by the thesis.

- Chapter 2 presents an overview of CAESAR competition for selecting a portfolio of authenticated ciphers. We describe selected candidate algorithms used in this thesis.

- Chapter 3 surveys the earlier work on implementation of authenticated encryption on embedded platforms and in particular, on smart cards. We list the significant differences between the different approaches and compare our work with them.

- Chapter 4 presents our implementation approach and the pitfalls we faced while porting on Java Card. This chapter lists the recommended optimisations for Java Cards. In addition, we describe the algorithm specific optimisations carried out for different schemes.

- Chapter 5 presents the evaluation of authenticated ciphers on Java Cards. We focus on timing performance and memory footprint, which are vital for any embedded system.

- Chapter 6 concludes the thesis and explains how this work can be extended in the future.
In this chapter, we give the background of CAESAR competition and describe the candidate authenticated algorithms that are ported to Java Cards.

2.1 CAESAR Competition

CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [6] is a contest to select a portfolio of authenticated encryption algorithms that are superior over Advanced Encryption Standard (AES) used in Galois/counter mode [2, 3] and can efficiently be implemented in software and hardware.

The CAESAR contest follows the rich tradition of focused cryptography competitions. In 1997, United States National Institute of Standards and Technology (NIST) held the first open competition for a new Advanced Encryption Standard [1]. The competition received 15 block-cipher entries. NIST selected Rijndael [7] as AES. Similarly, in 2004, ECRYPT conducted contest to select new stream ciphers [8]. Around 34 stream-cipher submissions were received from cryptographers around the world. In 2007, NIST initiated competition for selecting a new hash standard. The contest was participated by 64 submissions. Finally Keccak [9] was selected as SHA-3.

With the call for submission on 16 April 2013, CAESAR attracted 57 submissions from more than 150 cryptographers around the world. The second round selected 30 submission whereas only 15 submissions could make their way in the third round. The final portfolio for authenticated encryption is expected to be announced in Dec 2017.

2.1.1 Candidates requirements

CAESAR specifications require plaintext, symmetric encryption key, associated data, secret message number and public message number as input to produce
the ciphertext and accompanying authentication tag. The brief description of parameters are as follows.

- **Key**: This is the mandatory symmetric key of fixed (generally 80, 128 or 256-bit) length.
- ** Plaintext**: The mandatory input of variable-length.
- **Associated data**: The mandatory input of variable-length whose integrity must be preserved by the cipher.
- **Public message number**: An optional input field of fixed-length whose integrity must be retained. This is basically a nonce.
- **Secret message number**: An optional input field of fixed-length whose integrity and confidentiality must be maintained.
- **Ciphertext**: The output of variable-length which provides confidentiality of data.
- **Authentication Tag**: The output of variable-length (mostly 128-bit) which assures authenticity of data.

### 2.2 Candidate Authenticated Encryption Algorithms

We have selected five authenticated algorithms from the third-round of CAESAR competition. It is our estimation that these have potentially high chances to be in the final portfolio of the CAESAR as they are efficient on software as well as on hardware and especially attractive for smart cards. We give a brief overview of our selected AE candidates for implementation and evaluation on Java Card.

#### 2.2.1 ACORN

ACORN [10], developed at the Nanyang Technological University in Singapore, is a stream cipher based authenticated encryption algorithm. As per its primary recommendations, it uses 128-bit key, 128-bit nonce, and 128-bit tag. It has been principally designed to run on resource constrained environments. It also supports high performance applications.

![The concatenation of 6 LFSRs in ACORN-128 (adapted from [10]).](image)

*Figure 2.1: The concatenation of 6 LFSRs in ACORN-128 (adapted from [10]). $f_i$ indicates the overall feedback bit for the $i^{th}$ step; $m_i$ indicates the message bit for the $i^{th}$ step.*

The encryption state of ACORN-128 is 293-bit length. It utilises six Linear Feedback Shift Register (LFSR) for performing linear bit shifting of its state. Figure 2.1 shows concatenation of 6 LFSRs in ACORN-128. The initialization of ACORN
happens with feeding the key and IV into the state and running the cipher for 1792 steps. After initialization, the state is updated with the associated data. The encryption is performed after processing the AD. At each encryption step, one plaintext bit is fed for updating the state, and the plaintext bit is encrypted to ciphertext bit. At this stage the cipher is run for 256 steps. The authentication tag is generated in the finalization step.

ACORN uses three main functions. Function to generate keystream bit from the state at each step, function to calculate the overall feedback bit at each step, and a state update function at each step. The state update function starts with updating the state using 6 LFSRs. The keystream bit is generated in next step. Thereafter non-linear feedback bit is generated. In the final phase, the 293-bit register is shifted with the feedback bit.

The decryption process is similar to encryption whereas the verification process is akin to tag generation. It is recommended that on tag verification failure, the ciphertext and newly generated tag should not be given out as output.

### 2.2.2 AEGIS

AEGIS [11], developed by researchers from Nanyang Technological University in Singapore and KU Leuven at Ghent, is a dedicated authenticated cipher. Dedicated ciphers do not use block or stream ciphers as basic building block, but rather it uses a message to update the encryption state and in the process message authentication is achieved for free. AEGIS is constructed from AES round. AEGIS-128 passes a 16-byte message block through 5 AES rounds. AEGIS is considered fast as it operates at half the speed of AES. As per the recommendations, AEGIS-128 works upon 128-bit key, 128-bit nonce, 640-bit state, and 128-bit tag. It is mainly designed for high performance applications.

Figure 2.2 shows the main stages of AEGIS encryption. At the core of the encryption is a state update function. At initialization stage, the key, IV and two constants are fed to initialize the state. The associated data is pumped into the encryption state to update it. The message blocks in chunk of 16 bytes are fed to

![Figure 2.2: AEGIS encryption.](image-url)
the state update function. The ciphertext is generated by XORing the message with the output state of this stage. Finally, the tag is generated by updating the state with length of associated data and message length. Specifically, bytes 64 to 80 of state in this stage are used as the authentication tag.

![Figure 2.3: The state update function of AEGIS (adapted from [11]).](image)

The AES encryption round function used in AEGIS is an ordinary AES round (not the last round). A message is encrypted and authenticated with a 128-bit key and a 128-bit initialization vector. The encryption state of AEGIS consists of 80 bytes. The 16-byte message block is fed to update the state. The logic of state update function is depicted in Figure 2.3. In the figure, R indicates the AES encryption round function without XORing the round key and w is a temporary 16-byte word.

For successful decryption and verification process, the exact values of key size, IV size and tag size should be known. Similar to encryption, the decryption also starts with initialization and followed by processing of associated data. The ciphertext is decrypted in the next stage. The finalization stage outputs the plaintext. It is recommended that the ciphertext and newly generated authentication tag should not be provided as output if the verification fails.

### 2.2.3 ASCON

ASCON [12], developed at the University of Technology in Graz, Austria, aims to implement a moderate speed cipher with side-channel resistance features.

ASCON is founded on duplex sponge modes, similar to MonkeyDuplex [13]. The sponge operates on a state of 320 bits, with injected message blocks of 64 or 128 bits. The encryption process is split into four phases, shown in Figure 2.4:

1. **Initialization:** The state is updated with the key K and nonce N.
2. **Associated Data Processing:** The state is updated with associated data blocks.
3. **Plaintext Processing:** The plaintext blocks are injected into the state and ciphertext blocks are extracted.
4. **Finalization:** The key K is injected again and extracts a tag T for authentication.
There are two recommended parameters for ASCON, shown in Table 2.1. Both versions offer 128-bit confidentiality, integrity and authenticity.

**Table 2.1: The recommended parameter sizes for ASCON.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Key-size</th>
<th>Block-size</th>
<th>Public message number size</th>
<th>Tag-size</th>
<th>Rounds begin and end</th>
<th>Rounds datablock</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCON-128</td>
<td>128</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>ASCON-128a</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

The internal state of ASCON is 320 bit arranged in five sub-states of 64-bit each. It is memory-efficient as no other storage is required. After the initialization, the AD and plaintext are processed in 64-bit blocks in ASCON-128 (which is implemented in this thesis). The AD and plaintext are packed into block length by padding with a single “1” bit followed by zeros. Due to smaller sizes of initialization data and keys, more rounds are used for these phases. The data processing phase has less rounds due to abundance of input data fed to it. The encryption state is XORed with data to transit the state to next stage. The state passes through a number of rounds, which finally generates ciphertext.

**Figure 2.5: One round of Permutation used in ASCON.**

Figure 2.5 shows one round of the permutation of the ASCON cipher. Each round consists of three operations— two XORs, a S-box and a shift operation, thus forms a short critical path. In one permutation, rounds between 6 and 12 are repeated.

The decryption mode is similar in structure of encryption. Unlike encryption, the ciphertext block is directly inserted in the permutation function while decryption. The XOR between this block and the state yields the plaintext which is the output. Therefore encryption and decryption can be achieved from a single
implementation with a small additional logic.

ASCON is fast and lightweight in hardware and software. Being inverse-free, same code can facilitate encryption as well decryption. ASCON is efficient for short messages and long messages by employing differential number of rounds.

2.2.4 CLOC

Compact Low-Overhead CFB (CLOC) [14], developed at Nagoya University Japan, is provably secure authenticated encryption scheme which uses blockcipher modes of operation. CLOC builds over the previous schemes namely CCM, EAX, and EAX-prime by reducing the overhead over the block cipher, and optimising the precomputation complexity and the memory requirement. The block diagram of CLOC is shown in Figure 2.6. Its main components are a hash function (HASH), an encryption engine (ENC), and a pseudo random function (PRF). The HASH and PRF functions are variants of CBC-MAC. The ENC block uses AES encryption in CFB encryption mode. For processing the associated data and ciphertext CBC-MAC is called twice whereas ciphertext is generated by a single call to CFB mode. It works with two state blocks (i.e. 2n bits). The tweak functions are used to update the internal state at several points in the encryption and the decryption. Tweak functions consist of word-wise permutations and XOR’s.

![Figure 2.6: CLOC authenticated encryption.](image)

CLOC is suitable for handling short input data on embedded devices. To optimize the performance for small devices, the algorithm completely eliminates bit-wise operations. CLOC with AES on Intel Haswell family shows a performance of 4.56 cpb, which is considered efficient. CLOC is provably secure up to the standard birthday bound. It guarantees privacy against nonce-respecting adversaries. The authenticity is ensured against nonce-reusing adversaries.

2.2.5 MORUS

MORUS [15], developed at the Nanyang Technological University in Singapore, aims to be fast in hardware and software by keeping a short critical path for the
CAESAR Competition and Candidate Algorithms

encryption. MORUS has 3 different recommended parameters which are shown in Table 2.2. In this thesis, MORUS-640 is implemented on smart card. Figure 2.7 shows high-level view of encryption.

![MORUS Encryption Diagram](image)

Figure 2.7: MORUS encryption.

<table>
<thead>
<tr>
<th>Name</th>
<th>Key-size</th>
<th>IV</th>
<th>State</th>
<th>Tag-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MORUS-640-128</td>
<td>128</td>
<td>128</td>
<td>640</td>
<td>128</td>
</tr>
<tr>
<td>MORUS-1028-128</td>
<td>128</td>
<td>128</td>
<td>1280</td>
<td>128</td>
</tr>
<tr>
<td>MORUS-1028-256</td>
<td>256</td>
<td>128</td>
<td>1280</td>
<td>128</td>
</tr>
</tbody>
</table>

MORUS has a 2-dimensional state array that is at the centre of the algorithm. MORUS has long initialization and finalization phases. MORUS uses a scheme similar to a type 3 Feistel scheme to update its state [16]. The multi-block state is updated using a round function over number of rounds using a state update function which is shown in Figure 2.8. The ciphertext is generated by XORing the internal state with the plaintext to achieve confidentiality. The state update function is injected with the plaintext to generate the tag which assures authenticity and integrity.

The internal state is made up of 5 blocks of 128-bit, making a total of 640 bits. It requires another 128-bit storage to hold the input during finalization. As a result, MORUS has small memory footprint. The encryption starts with initialization of state. The AD and plaintext are inserted in the state update function in 128-bit blocks. The last block is padded with zeros to make it full 128 bits.

The decryption is similar to encryption mode. The ciphertext is processed in the same way as the plaintext (in encryption), and then this plaintext is fed as input to the state update function. The initialization and finalization stages require 16 and 8 iterations respectively to achieve strong 128-bit security.

The state update function is based on type 3 Feistel scheme [17]. It has 5 rounds that use XOR, shift and AND operations. It heavily uses rotation function by splitting 128-bit into smaller parts of 32-bit. The amount of cyclic rotation is
dependent on the current round. The analysis of MORUS shows that it provide strong security [18].

Figure 2.8: The state update function of MORUS (adapted from [15]). In Rotl xxx_yy, xxx yy is 128 32 for MORUS-640 and 256 64 for MORUS-1280.
2.3 Comparison of Selected Candidates

The CAESAR candidates can be classified on a number of parameters [16]. We consider the following parameters to classify our selected candidates.

- **Parallelizable**: It means the encryption or decryption of a block can be done independent of any other block.

- **Online**: The encryption or decryption of a block depends on its previous blocks only i.e., the $i^{th}$ block depends on $0, 1, ..., (i - 1)$ blocks.

- **Inverse-Free**: This parameter implies that underlying primitive’s inverse operation is not required i.e., encrypt function is sufficient for encryption as well as decryption.

- **Intermediate Tag**: The intermediate authentication tag helps to detect early if parts of a decrypted message are invalid.

- **Robustness**: An algorithm is robust if it provides security in both nonce-misuse and decryption-misuse settings.

Table 2.3: Comparison of selected candidates. The type of cipher is shown as stream for stream-cipher, block for block-cipher, sponge for key-less permutation based encryption, dedicated for encryption structure similar to Type-3 Feistel schemes. $^1$ signifies that AEGIS is parallelizable only for encryption, not for decryption.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Type</th>
<th>Parallelizable</th>
<th>Online</th>
<th>Inverse-Free</th>
<th>Intermediate Tag</th>
<th>Robustness</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN</td>
<td>stream</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AEGIS</td>
<td>dedicated</td>
<td>✓$^1$</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ASCON</td>
<td>sponge</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLOC</td>
<td>block</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MORUS</td>
<td>dedicated</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Chapter 3

Literature Review

In this chapter, we survey different authenticated encryption schemes and the work on implementation of authenticated ciphers on embedded systems. Smart cards being a type of embedded system, the survey provides an overview on the recent work done in this area.

3.1 Authenticated Encryption Schemes

Bogdanov et. al evaluates performance of block cipher modes of operation for AE in parallel software [19]. The paper proposes the scheduling of multiple messages in parallel to speed up both inherently sequential modes and to the scenario of having many but shorter messages. Significant speed-ups for serial modes as well as parallelizable modes is achieved. The first optimized AES-NI implementations for the AE modes OTR, CLOC, COBRA, SILC, McOE-G, POET and Julius are also provided.

Correct authenticated decryption requires the receiver to buffer the decrypted message until the authenticity check has been performed. In high-speed networks, which must handle large message frames at low latency, this behavior becomes practically infeasible. To overcome this, Abed et. al proposes pipelineable online ciphers, that combines a block cipher and a family of hash functions, as a practical alternative to AE schemes [20].

Online ciphers encrypt an arbitrary number of plaintext blocks and output ciphertext blocks which only depend on the preceding plaintext blocks. All online ciphers proposed so far are essentially serial, which significantly limits their performance on parallel architectures such as modern general-purpose CPUs or dedicated hardware. Andreeva et. al propose parallelizable online ciphers, that is fully parallelizable in both encryption and decryption [21].

Basel Alomair proposed an authenticated encryption scheme [22]. By advancing the hashing phase before block cipher encryption, a hashing function that is not universal can be used without affecting security of authentication. Since the hash function does not have to be universal, it can be computed faster than universal hash function. The paper shows that hashing the plaintext instead of the ciphertext can secure the hashing keys against key recovery attacks.
3.2 Authenticated Ciphers on FPGA

Homsirikamol and Gaj provide the first hardware implementation of AEZ on Xilinx Virtex-6 FPGA [23]. The paper brings out that implementing AEZ, contrary to its name AE made Easy, is significantly difficult on hardware because AEZ is complex, optimized for software, and impossible to implement in a single pass.

Saarinen implemented WhirlBob and Keyak AE on FGPGA fabric of Xilinx Zynq 7010 [24]. The AE is realized as a coprocessor and integrated with System-on-Chip (SoC) similar to Cortex-A9 CPU. By offloading authenticated encryption on a dedicated hardware, aim is to extend battery life and performance of the mobile device such as smart phones, tablets, and Internet of Things (IOT).

Michael Fivez provides energy efficient hardware implementations of CAESAR submission [25]. Joltik, MORUS, and ASCON algorithms were built on Spartan 6 FPGA board and evaluated for energy consumption and area requirements. The thesis shows that MORUS is fastest, followed by ASCON and then Joltik. However area wise Joltik is the most space efficient, followed by ASCON and MORUS.

Groβ et. al present hardware implementations of ASCON which is suitable for RFID tags, Wireless Sensor Nodes, Embedded Systems [26]. They show that ASCON is fast and small as well as can also be easily protected against differential power analysis attacks. They present three variants with different design goals implemented in VHDL and evaluated using a Cadence-based ASIC design-flow.

Kotegawa et. al performed hardware implementations of authenticated ciphers with VIVADO High-Level Synthesis which is a tool of Xilinx [27]. The paper shows various optimization techniques on the point of speed, area size and the clock frequency. They work on two nonce-based algorithms (AES-OTR and SILC) and two Nonce-misuse Resistant algorithms (AES-COPA and POET).

Jasper Gorissen evaluates three CAESAR candidates (Trivia-ck, Ketje and MORUS) on FPGA [28]. They chose area and speed as the optimisation targets. They show that Ketje has the smallest area usage and MORUS has an excellent speed/area ratio whereas Trivia-ck, while having good throughput in the speed optimised version, underperforms in area usage compared to others.

3.3 Authenticated Ciphers on Java Card

On Java Card Platform, Classic Edition 3.0.5, javacardx.crypto.Cipher package provides the AEADCipher class which is the abstract base class for authenticated encryption with associated data ciphers [29]. Examples of AEAD algorithms are the GCM and CCM modes of operation for AES. AEAD ciphers can be created by the Cipher.getInstance method using the ALG_AES_GCM and ALG_AES_CCM algorithm constants. The returned Cipher instance should then be cast to AEADCipher.

Beside GCM and CCM modes of AES support on Java card, we could not find any other authenticated encryption algorithm on Java card. To the best of our knowledge, this work is the first attempt to provide five authenticated encryption from CAESAR submission on Java card platform.
Chapter 4

Implementation of Authenticated Ciphers on Java Cards

This chapter describes framework for development of authenticated ciphers on Java Card. The implementation details and algorithm specific optimisations done for each cipher in making them suitable for Java Card are also described.

4.1 Framework for Authenticated Encryption on Java Card

This section gives an insight into Java Card, the APDU based communication protocol, and the protocol we implemented for authenticated encryption. The development tools are listed and we also describe the general optimisations for Java Cards.

4.1.1 Java Card

Java Card\textsuperscript{1} [30] refers to a software technology that allows Java-based applications (applets) to be run securely on smart cards and similar small memory footprint devices. Java Card is the tiniest of Java platforms targeted for embedded devices.

\textsuperscript{1}In this thesis Oracle SDK: Java Card Development Kit 2.2.2 is used
Java Card gives the user the ability to program the devices and make them application specific. It is widely used in SIM cards (GSM mobile phones) and ATM cards. Figure 4.1 gives an insight into the CPU capability and memory sizes of Gemalto Java Card. The GlobalPlatform specifications are used for the secure management of applications on the card (download, installation, personalization, deletion).

### 4.1.2 Smart Card Communication Protocol

The communication between a smart card and a computer follows a well defined communication protocol shown in Figure 4.2. The communication happens in form of data packages called Application Protocol Data Unit (APDU)\(^2\) which contain either command or response messages. Communication model is master/slave in which smart card is slave and computer is master. The smart card (attached to a reader) always wait for a command APDU from host computer, processes as per the received command and returns result in a response APDU.

![Communication protocol](image)

**Figure 4.2: Communication protocol.**

### 4.1.3 Communication for AE

The communication protocol realized for implementation of all authenticated encryption is shown in Figure 4.3. To start interaction with the Java Card, the host computer starts by selecting an Applet. On successful selection, the host communicates the secret key, associated data and other parameters required as per CAESAR specifications. For the session, the key is stored securely on the card. Thereafter input is packeted into APDU frames and submitted to the Java Card for encryption. The output—ciphertext for encryption process and plaintext for decryption process, is returned for each submission.

### 4.1.4 Development Tools

The development infrastructure is build on Macbook pro with OS X 10.11. The following software are installed:

1. **JDK 1.8.0_73 and Netbeans 8.1 IDE**: NetBeans is a software development platform that helps applications development using Java.

---

\(^2\)APDU protocol is an application layer standard and its specifications are given in ISO 7816-4.
2. Apache-ant 1.9.6: Apache Ant is a software tool for automating software build processes.

3. Applet Playground: AppletPlayground is an educational repository for getting to know JavaCard development by learning from existing open source software.

4. Global platform Pro v0.3.7: GPShell is a script interpreter that talks to a smart card which complies to the GlobalPlatform Card Specification. It is written on top of the GlobalPlatform Library. It uses the PC-SC Connection Plug-in for accessing smart cards. It can establish a secure channel with a smart card, load, instantiate, delete and list applications on supported smart cards.

5. PCSC lite: PC/SC is the international standard for PC to smart card reader communication, which is implemented by PCSC-Lite.

### 4.1.5 Applet Implementation

The native algorithm and source code in C has been provided by the inventors of the respective authenticated ciphers. We implement the algorithms in Java and test byte-level compatibility of our output with the results produced by native C source code. We then modify our code and test with Java Card simulator. On successful execution with the simulator, we prepare the code for porting on real Java Card. We port our Java implementations on Java Card and measure the time taken to find out efficiency of our implementations. We also measure the memory footprint of each cipher. We optimise the implementations to improve upon the timing or to reduce the memory consumption.
A main applet class (e.g. AEGISApplet.java) along with the supporting classes (e.g. AEGISCore.java) are prepared. The core class contains the algorithmic implementations. The constructor of applet declares variables, initialises various methods and keep them ready for subsequent use. The install method installs the applet. The select and deselect methods are used to clean up important parameters such as key. We define two new instructions (e.g. INS_AEGISEncryption (0x61) and INS_AEGISDecryption (0x62)) respectively for encryption and decryption functions of authenticated ciphers. To cater to encryption instruction a dedicated encryption method (e.g. AEGISEncryption) is created which does the following:

1. Copy the message (plaintext) from APDU buffer to fast RAM memory byte array.
2. Call Core class (e.g. AEGISCore.java) for encryption.
3. Copy back the ciphertext from RAM memory to APDU buffer.
4. Return the encrypted bytes to the host application.

The decryption instruction calls the decryption (e.g. AEGISDecryption) method to perform the following steps for deciphering:

1. Copy the ciphertext from APDU buffer to fast RAM memory byte array.
2. Call Core class (e.g. AEGISCore.java) for decryption.
3. Copy back the plaintext from RAM memory to APDU buffer.
4. Return the decrypted bytes to the host application.

We also prepare the user program that runs on the PC. This program acts as an interface between the smart card and the users. It receives instructions and data from users and communicates with smart card using APDU packets. The APDU packet has a header of 5 bytes followed by the payload. The max size of APDU is 256 bytes. Different types of APDU can be send by properly setting up the header fields and its interpretation on the card. This function performs the following:

1. Uses cardManager to connect to the card.
2. Selects an applet (e.g. AEGISAPPLET) using cardManager.
3. Prepares APDU for sending data with required headers.
4. Receives user input regarding mode (encryption or decryption) and the file to transform.
5. Read file and prepare for encryption/decryption by dividing its data into packet sizes of 232 bytes payload for encryption and 248 bytes payload for decryption. For decryption the last 16 bytes carries the authentication tag used in verification.
6. Pads the last payload to a multiple of 16 bytes.
7. Measures timings for encryption and decryption.

The class file named CardMngr.java has the code for connecting/disconnecting a smart card to the PC. It also has methods for sending the APDU to the smart card.

4.2 Common Optimisation for Java Card Applets

Java Card platform is a constrained device which does not support Unicode characters, 32-bit and 64-bit integers, float and double data types, threads, and multidimensional arrays. Based on the experience of this work and guidance given at [31, 32], we list the common optimisations that are generally applied on Java Card Applets. We have applied these best practices in our implementations.

- **Allocation and Initialisation of objects in constructor versus methods at the time of call**: The allocation and default initialisation of objects (memory arrays, keys, pins, ciphers, etc.) in constructor when a card is installed is a good optimisation as it allocates memory and keep the objects ready for usages. This is much more efficient than creating the objects when required as time gets wasted. It is recommended to exit the process if all required resources are not allocated.

- **Reusing objects**: Most of the Java Card Runtime Environments do not contain garbage collector. Therefore any object instantiation using `new` permanently consumes the storage space as it can never be reclaimed due to absence of garbage collection. Reusing objects is a necessity to have best usage of scarce memory resource. The care should be taken to reduce the side-effects of reusing objects by proper initialization before reuse.

- **Memory allocation**: There are two types of memory on Java card— RAM and EEPROM. The RAM is transient whereas EEPROM is persistent. EEPROM is consumed to allocate memory for objects created using `new`, and all global variables of basic types (byte, short). RAM is consumed for allocating memory for local variables, parameters of methods, and objects created with Java card system call such as `JCSystem.makeTransientByteArray`.

- **Generating and keeping the key on card versus transferring key from host**: The smart card is capable of generating and securely storing secret keys. It is much better from security and performance perspective to generate and keep the key on the smart card rather then transferring the key from host. However the use case of an application may influence this decision. Sensitive data must be stored in transient memory with initialization at the beginning and clearance at the end of the session.

- **Working data stored on RAM not on EEPROM**: Working data must be stored on transient RAM. Keeping data on EEPROM is slow (sometimes 1000 times slower than RAM) as well as not recommended from security perspective as it remains persistent (may be done with due thought and additional protection like encrypted with PIN) on card.
• **No object creation in Applet**: In the interest of efficiency, all objects must be created in constructor but not in other functions in the Applet. The constructor is called once resulting in memory allocation once only. A careless allocation of memory in a local method used frequently results in memory wastage as unclaimable memory chunks get allocated.

• **Copy-free methods**: Copy-free methods are faster than copy based methods and recommended to be used on memory constrained smart cards.

• **Payload size**: The payload size decides the amount of data transferred to the smart card in one APDU. The max size of an APDU is 256 bytes. The maximum payload size may be used to pack the headers and data to save on the communication time.

• **Constants**: Using the qualifiers static and final to a variable in Java makes it constant. The use of constants result in smaller program size and better performance.

• **Avoid storing intermediate values**: The use of compound arithmetic statement in program saves memory which otherwise gets consumed with multiple separate assignments.

### 4.3 Algorithm Specific Optimisations

#### 4.3.1 ACORN

ACORN is a stream cipher based authenticated encryption algorithm. Though it has been designed for resource constrained embedded environment, we found it to be one of the misfit for Java Card platform. ACORN converts all bytes of input message to bits, stores each bit in a byte and performs heavy bit shift operations. As bit shift operations are extremely slow on Java Cards, we received extremely poor timing performance. The timing results can be improved on the platform which have ISA support for bit shift operations. Due to no direct support for bit shift operations on Java Card, optimisation improving timing results is not possible.

#### 4.3.2 AEGIS

AEGIS is a dedicated authenticated encryption algorithm that uses AES round for its construction. The following optimisation in memory is carried out.

• **Memory Optimisation**: To improve upon the performance of AEGIS, lookup arrays are used for AES operation. There are 4 arrays (TE[0]...TE[3]) each of 1024 bytes. These are read-only arrays i.e., write-once read-multiple times. Due to shortage of RAM memory on Java Card it could not be build on the RAM. Further considering the fact that EEPROM has large size and read operations are efficient on it, the lookup arrays are allocated with memory space on EEPROM. Initially it created a big size (37 KB) cap file which could
not be uploaded on the Java Card. We found that the cap file occupies too much of EEPROM causing the upload to fail. To mitigate this problem we modified the qualifier of lookup arrays to `private final static` as a result the compiler considered the elements of arrays as constant and the space got saved. In addition to this, we also reused a few variables to save on the memory consumption. The resulting cap size reduced to 12 KB and uploaded successfully.

### 4.3.3 ASCON

ASCON has sponge-based mode of operation with custom-tailored SPN permutation. The sponge (keyless permutation) operates on a state of 320 bits, with injected message blocks of 64 or 128 bits. ASCON uses heavy bit manipulation in its operations. We optimised bit rotation as it was becoming a major performance killer.

- **Cyclic bit rotation:** In the permutation function, right cyclic bit rotation is required. In the native code, the bit rotation was done at integer level. As integers are not supported on Java Card Technology, we realized cyclic bit rotation by manipulation at bit level. While it worked correctly, we find it adds extensive delay as each bit in input data undergo manipulation. Therefore, we optimized our implementation for achieving cyclic bit rotation as combination of left and right shift at byte level, which improved the performance.

### 4.3.4 CLOC

CLOC uses `state`, a 16 byte array, to keep the state of encryption. For the core encryption, it uses AES. The state passes through 3 stages sequentially—Associate Data processing, Nonce processing, and Message processing. The tag and ciphertext are derived out in the process. The following two are optimised to improve performance.

- **CTX Structure:** The CTX structure of CLOC keeps the context information at all times. This structure uses pointers to track memory arrays. The CTX is fed with associated data, nonce, and message to generate the ciphertext and the tag. We simplified the CTX structure to store required data in single dimension arrays.

- **XOR operation:** In CLOC the XOR operates at integer level. We modified the XOR operation to operate at byte level. In addition, all variable of type long are declared as byte array of size 8. The block structure is also simplified as byte array of size 16.

### 4.3.5 MORUS

MORUS is a dedicated authenticated cipher. The design of MORUS makes it parallelizable. The core of MORUS is the state update function which updates
the state of the cipher. In each step of MORUS, there are 5 rounds with similar operations to update the state. MORUS heavily uses bit rotation for diffusion. The AD, and input data are operated on the state in a phased manner. The endianness of the input parameters is corrected before further processing. The following optimisations undertaken for MORUS.

- **2D state array as 1D array:** MORUS authenticated encryption uses a 2-dimensional state array for its core function. The state is organized in 5 blocks of 128-bit each, totalling a total of 640-bit. As 2-dimensional arrays are not supported on Java card Technology, we converted the structure of state into 1-dimensional array. We changed the state update function and other methods to accommodate the new structure of state.

- **Cyclic bit rotation:** The state update function performs XOR, AND and cyclic bit rotation operations. The cyclic bit rotation is generally implemented using bitwise shift operators. In the native implementation of C reference code, the state update function uses shift operator on integers to effect cyclic bit shift by predefined constants. But as integers are not supported on Java card Technology we had to manipulate 4 consecutive bytes to effect a shift in an integer. The naive implementation of cyclic bit rotation by manipulating each bit was found to be very expensive in terms of time. Therefore we implement cyclic bit rotations as combination of left and right shift at byte level. We also encountered error in cyclic rotation of negative numbers. The error occurred because byte (8-bit) is upgraded to integer (32-bit) in Java before shift, and 2’s complement (of negative number) adds 1s in all MSB bits for negative numbers. This we resolved using logical right shift (\(\gg\gg\)) and by ANDing with suitable constants.

- **Endianness Issue:** The reference C implementation uses little-endian to represent the data. Java is big-endian by default i.e., it stores high-order bytes of a word at MSB places. To have byte-level compatibility with outside world, we manual convert byte-ordering of words. We change the endianness of input data manipulate it as per MORUS algorithm and re-change the endianness before delivering the output.

### 4.4 Github Repository of Authenticated Ciphers

A repository on github with the title *Authenticated Encryption on Java Card* [33] is created that contains the source of the authenticated ciphers ported on Java card platform. This enables availability of authenticated ciphers on Java card to the public.
Chapter 5

Evaluation of Authenticated Ciphers on Java Cards

The evaluation of authenticated ciphers on Java cards is done in this chapter. Java cards are constrained embedded devices which should give reasonable timing performance while consuming as less memory as possible. We look at timing performance for encryption and decryption, and memory footprint of different schemes.

5.1 Test Cases

We prepare test cases to ensure correctness of our implementations. We use files with different data sizes namely 256 B, 512 B, 1 KB, and 2 KB as input to the algorithms. By comparing the output from the reference implementations submitted by the inventors of authenticated algorithms with the output from our implementations we perform byte-level integrity check. All our implementations produce the same output as the native algorithms. For evaluation of authenticated ciphers Java card NXP JCOP CJ2A081 on USB reader Gemalto USB Shell Token V2 is used.

We use the native implementation of AES with 128 bit key encryption (10 rounds) as the baseline cipher for comparison against authenticated ciphers.

5.2 Timing Performance

We measure the time taken for encryption and decryption for different chunks of data. This gives an idea on the capability of algorithms to efficiently handle different data sizes. We also compare candidate’s performance to guage their relative performances.

5.2.1 Authenticated Encryption on Java Card

Figure 5.1 and 5.2 plot the time taken for encryption and decryption respectively by the authenticated algorithms. We find that AEGIS and CLOC are the most
time efficient ciphers from selected authenticated algorithms because they do not use costly operations such as bit-shifts or complex precomputations, and employ simple structure to keep the encryption state. Among the two, AEGIS produces the best timing results because at its core it uses 5 rounds of AES (without key XORing) implemented with lookup arrays. Obtaining values from lookup arrays reduces computation on Java card and makes AEGIS faster. We note that AEGIS has comparable performance to AES (baseline cipher) with bigger chunks of data. For smaller chunks, AEGIS has pre-computation and AD data processing overheads, which slows it down against AES. For big data sizes, AEGIS tends to do marginally
better than AES as AEGIS only uses 5 rounds. We find that AEGIS and CLOC are suitable for mobile embedded devices, RFID tags, and IoT devices. MORUS takes significant amount of time for encrypting (e.g. for 1 KB data it takes more than 20 minutes) which makes it unsuitable for practical use. The performance killer for Morus is the heavy use of bit-shift operations. Bit shift operations are costly and its implementation with bit manipulations results in poor performance. Though we optimised bit-shift by a combination of byte and bit shifts, but it certainly needs to be further optimised. In future, we plan to use the bit rotation instructions provided by ISAs such as ARM. ASCON has poor performance than MORUS. ASCON uses extensive bit-rotations which decreases its performance. We obtain the worst timing performance with ACORN because its structure is that of a stream cipher which converts each byte to 8 bits, and performs linear bit-shift operations throughout. The bit-shift operation is performance dampner. As all input bytes are dealt at bit-level, ACORN becomes unsuitable for practical use on Java card. On hardware / architectures supporting bit-shift operations, ACORN may be optimized to improve timing performance.

Comparing Figure 5.1 and 5.2, we find that the decryption operation takes similar amount of time as encryption operation in most of the ciphers. The reason for this behavior can be seen from Table 2.3 which shows that all the selected algorithms are inverse-free.

![Figure 5.3: Comparison of CLOC using AES in software and hardware.](image)

### 5.2.2 Software Versus Hardware Implementation

CLOC uses the standard AES without any modification. So it provided an opportunity to compare the performance difference between CLOC using AES in software versus AES inbuilt on cryptographic coprocessor of Java card. Figure 5.3 shows that using onboard cryptographic coprocessor (hw-enc) provides 21x speedup
over the software implementation (sw-enc). This motivates the case to build authenticated algorithms on cryptographic coprocessor. To facilitate wider use of authenticated algorithms on smart cards, it becomes a necessity to provide AE on smart cards.

5.3 Memory Footprint

The memory on Java card is limited and therefore must be used carefully. The RAM is 1000 times faster than EEPROM but many times smaller in capacity than EEPROM. The EEPROM has limitation on maximum number of writes supported. The writes on EEPROM are costly but reads are fast. So a carefully implemented authenticated encryption balancing the Java card constraints can give required performance and can result in memory saving. We measure the RAM, EEPROM utilization (in Bytes) by each cipher. We also measure the cap\(^1\) size of the ported cipher.

*Table 5.1: Memory footprint of authenticated algorithms. * represents CLOC using hardware AES.*

<table>
<thead>
<tr>
<th>Cipher</th>
<th>RAM (B)</th>
<th>EEPROM (B)</th>
<th>Cap Size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN</td>
<td>845</td>
<td>1690</td>
<td>5760</td>
</tr>
<tr>
<td>AEGIS</td>
<td>468</td>
<td>4176</td>
<td>12623</td>
</tr>
<tr>
<td>ASCON</td>
<td>476</td>
<td>190</td>
<td>5742</td>
</tr>
<tr>
<td>CLOC</td>
<td>832</td>
<td>1247</td>
<td>10149</td>
</tr>
<tr>
<td>CLOC*</td>
<td>832</td>
<td>223</td>
<td>7365</td>
</tr>
<tr>
<td>MORUS</td>
<td>192</td>
<td>180</td>
<td>8302</td>
</tr>
</tbody>
</table>

Table 5.1 shows that MORUS is the most memory efficient and therefore can fit on small memory embedded devices. MORUS uses 192 B in RAM to store 80 byte encryption state and a few arrays for intermediate values. MORUS consumes only moderate size EEPROM. ASCON is also found to be memory efficient as it mainly performs permutation operations using ten 8 bytes arrays. CLOC using hardware AES saves 1024 bytes in EEPROM over CLOC using software implementation of AES. The EEPROM utilization by ACORN is 1690 B out of which a 1536 bytes array is used by a supporting structure to the encryption state. The RAM consumption of 845 B (including 293 bytes for encryption state) by ACORN is considered to be high. AEGIS uses more than 4 KB of EEPROM to store lookup arrays for AES rounds. As these lookup arrays need to be frequently read (but write once), EEPROM proves to be most suitable. The fast reads and availability of adequate size on EEPROM motivates memory allocation for big size arrays on it. Note that even though RAM is faster but this much capacity is not available on most Java cards. This also caused creation of big cap file for AEGIS which could not be

---

\(^1\)cap is the file format of an applet which is loaded on Java card.
loaded and installed on Java card. When we reused a few variables the cap file size reduced to 12 KB and got loaded/installed on Java card.

![Figure 5.4: Time-Memory efficiency matrix.](image)

Figure 5.4 gives an overview of timing performance and memory consumption which may be helpful in selection of algorithm for particular type of smart cards suiting onboard memory and performance requirements. We observe that even though AEGIS has requirement of large EEPROM, it is the fastest cipher among the selected authenticated encryption. MORUS has the minimum memory requirement and moderate timing performance. ACORN shows the worst timing performance as well as the worst memory consumption for Java cards. ASCON is memory efficient but time inefficient while CLOC is memory inefficient but has good timing performance.
Chapter 6

Conclusions and Future Work

Authenticated ciphers provide confidentiality, integrity and authentication at the same time. The ongoing CAESAR competition is an effort to select a portfolio of authenticated encryption algorithms after thorough evaluation of security and performance on software as well as on hardware. It is expected that in future the authenticated ciphers will be the main workhorses for cryptographic solutions. Smart cards being popular in forms of credit / debit cards, SIMs, iKeys, secret tokens are also expected to be needing them.

In this dissertation, we have implemented five Authenticated Ciphers on Java Card platform. We optimized these algorithms for making them suitable and efficient for embedded systems specifically to smart cards. We explored the designs and reference code provide by the inventor of each algorithm. We identified regions of bottleneck that pose challenge to their porting on smart cards. Accordingly, the algorithms have been optimized for Java Card platforms.

This work reports the evaluation of selected CAESAR candidates for their timing performance and memory footprint. We hope these results will be useful to security practitioner working on smart cards.

The major areas in which this work can be extended are as follows:

- **Optimization to use AES coprocessors available on Java Cards**: The contemporary smart cards have coprocessor that support AES and other cryptographic functions. We used the hardware coprocessor for AES encryption in CLOC. A few other authenticated ciphers used AES with certain modifications and therefore coprocessor could not be used directly. Optimizations or tweaking to the algorithm to employ AES coprocessor available on smart cards will boost the performance.

- **Optimization to exploit Instructions provided in Instruction Set Architecture (ISA)**: There are a few architectures such as ARM which provide instructions for cyclic bit rotation in one cycle in their ISA. We expect that the performance will improve drastically if these can be exploited. Our experience showed that doing bit manipulation at bit-level is costly and therefore we optimized rotation at byte level.
References


