Analysis of Authenticated Encryption Based on Stream Ciphers

by

Md Iftekhar Salam

Bachelor of Engineering
(Multimedia University) – 2008
Master of Science
(Dongseo University) – 2011

Thesis submitted in fulfilment of the requirement for the Degree of Doctor of Philosophy

School of Electrical Engineering and Computer Science
Science and Engineering Faculty
Queensland University of Technology
Australia

2018

Keywords

Authenticated encryption, Stream cipher, CAESAR cryptographic competition, Cryptanalysis, ACORN, Tiaoxin-346, MORUS, AEGIS, Cube attack, State collision, State convergence, State cycles, Rotational cryptanalysis, Fault attack, Forgery attack

Abstract

Authenticated encryption (AE) is a current research topic in cryptography. AE schemes provide a process for simultaneously achieving the security goals of confidentiality and integrity assurance. Secure and efficient AE constructions are important for protecting modern electronic information systems. This thesis examines several recent AE proposals in detail.

In 2013, the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) was announced with the aim to select secure and efficient AE algorithms suitable for widespread adoption. The CAESAR competition received 57 AE proposals, mostly based on either block cipher, or stream cipher or sponge constructions. The CAESAR competition is currently in the third and final round. Of the 57 first round submissions, 15 AE proposals have advanced to the third round. The final portfolio of the AE algorithms is expected to be announced in December 2017. The proposals submitted in the CAESAR competition require extensive public analysis to determine the final portfolio of AE algorithms. This thesis aims to contribute into this analysis.

The main objective of this thesis is to provide a security analysis of selected stream cipher based AE proposals from the third round of the CAESAR competition. In investigating the resistance of the AE ciphers to common cryptanalytic techniques, both security goals of confidentiality and integrity assurance are considered.

This thesis first classifies the stream cipher based AE proposals from the CAESAR competition, using the approaches for classifying authenticated encryption described by Bellare and Namprempere, Katz and Yung, and Al-Mashrafi. These classifications are used to determine the similarity among ciphers and potential cryptanalytic techniques to apply.

Next, this thesis provides a detailed security analysis of three AE stream ciphers: ACORN, Tiaoxin-346 and MORUS. All of these ciphers are included

in the third round of the CAESAR competition. The security analysis includes applying classical techniques such as state collision attack, cube attack, rotational attack, forgery attack and state cycle analysis. In some cases the attacks on a reduced version of these ciphers are considered. Successful attacks on the reduced version of these ciphers are demonstrated. For the full versions, all these ciphers demonstrate a strong security margin against the classical cryptanalytic techniques considered in this thesis.

The results from this thesis demonstrate that successful attacks can be applied to these three ciphers if they are used in the nonce-reuse scenario. To prevent these attacks, the ciphers should not be used in the nonce-reuse scenario.

This thesis also explores the application of physical attacks on these three ciphers. The results of the thesis demonstrate that physical attacks can be powerful against these ciphers. Fault injection based key recovery attacks can be applied to MORUS and Tiaoxin-346, whereas fault based forgery attack can be applied to all three ciphers. This thesis recommends having adequate physical protections to prevent these types of fault attacks.

Finally, this thesis concludes with an overall evaluation of these three ciphers. The overall evaluation is obtained by combining the security analysis described in this thesis and the existing efficiency analysis of these ciphers provided by other researchers. Based on the overall evaluation, these three ciphers appear to be strong candidates for the final portfolios of the CAESAR competition.

Contents

List of Figures

List of Tables

List of Algorithms

Declaration

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: . . QUT Verified Signature **Date:** 06/04/2018

Previously Published Material

The following articles have been published, and contain material based on the content of this thesis.

- Salam, M. I., Wong, K. K-H., Bartlett, H., Simpson, L., Dawson, E., Pieprzyk, J. Finding State Collisions in the Authenticated Encryption Stream Cipher ACORN. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2016)*. ACM, 2016.

- Salam, M. I., Bartlett, H., Dawson, E., Pieprzyk, J., Simpson, L., Wong, K. K-H. Investigating Cube Attacks on the Authenticated Encryption Stream Cipher ACORN. In Batten, L., Li, G. (eds), *Applications and Techniques in Information Security - ATIS 2016*. Vol. 651, pp. 15-26. Springer Singapore, 2016.

- Salam, I., Simpson, L., Bartlett, H., Dawson, E., Pieprzyk, J., Wong, K. K-H. Investigating Cube Attacks on the Authenticated Encryption Stream Cipher MORUS. In *Proceedings of the IEEE Trustcom/BigDataSE/ICESS*. pp. 961-966. IEEE Computer Society, 2017.

- Salam, I., Mahri, H., Simpson, L., Bartlett, H., Dawson, E., Wong, K. K-H. Fault Attacks on Tiaoxin-346. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2018)*. ACM, 2018.

- Salam, I., Simpson, L., Bartlett, H., Dawson, E., Wong, K. K-H. Fault Attacks on the Authenticated Encryption Stream Cipher MORUS. *Cryptography*, 2(1):4, 2018.

Acknowledgements

This thesis would not be possible without the continuous support, encouragement and guidance from my supervisory team, including my principal supervisor Dr Leonie Simpson, and my associate supervisors Dr Harry Bartlett, Prof Ed Dawson, Dr Kenneth Koon-Ho Wong & Prof Josef Pieprzyk. My supervisory team have had regular weekly meetings to discuss the updates on my research, which helped to keep my research on track. They have spent a lot of time with me discussing my research, providing research guidance and constructive feedbacks, reviewing my writings. I am grateful for their kind support throughout my PhD candidature.

Thanks to Leonie and Harry for helping me with the understanding of the concepts of state cycle analysis, state convergence and state collision analysis. Thanks to Ed for helping me to build a background on finite field arithmetic. Thanks to Kenneth for all the tips with the implementations, and the usage of SAGE and MAGMA software. Thanks to Josef for helping me understanding the idea of cube attacks and rotational cryptanalysis.

I would like to thank Dr Matthew McKague and Prof Ian Turner for being panel members (outside my supervisory team) in my PhD final seminar, and taking their time to review this thesis. Also, thanks to the anonymous external examiners for reading, and providing suggestions to improve the quality of this thesis. Thanks to Dr Douglas Stebila and Dr Joanne Hall for working as panel members in my PhD confirmation seminar.

Many thanks to my fellow PhD student Hassan Qahur Al Mahri, with whom I had the opportunity to produce a joint collaborative work on the Fault Analysis of Tiaoxin-346. This collaborative work appears in the Chapter 4 of this thesis.

I am thankful to QUT for funding my research with the QUTPRA scholarship, QUT HDR Tuition Fee Scholarship and QUT Excellence Top Up Scholarship. Thanks to the Asian Workshop on Symmetric Key Cryptography - Cryptology

School (ASK 2014), CORE Funding and QUT Information Security Discipline for providing travel support to attend conferences/ workshops. Also many thanks to Leonie, Josef, Matt, Ernest and Wasana for providing the opportunity to engage in teaching as a sessional academic at QUT.

I have made a lot of friends during my three and a half years candidature, which made the PhD journey much easier. Thanks to all my fellow colleagues and friends from the Information Security Discipline of QUT, including Hassan, Nick, Udyani, Tarun, Basker, Anisur, Jack, Janaka, Raphael, Qinyi, Thomas and Ben. Also thanks to all the fellow PhD students from my office at GP S-1029, including Hamzah, Tusher, Israt, Ifa, Khanh, Edy, Raji, Daniel, Thiru, Wathsala, Gayani, Basker and Gaurangi. Thanks to Nayim and Avijit for providing the initial support when I first arrived in Brisbane. Thanks to my house-mates Rohit and Masina for making my life more enjoyable in Brisbane.

Finally, I would like to express thanks to my parents, wife, brothers and all other family members for their love and unconditional support. Their ongoing support kept me motivated during this PhD candidature.

To my parents

Chapter 1

Introduction

Communication systems have changed significantly compared to last century. Most of today's communication systems are electronic, we communicate using mobile communications, emails, messengers. Also, different types of services are provided using electronic systems, such as government services (e.g., tax return, e-voting, health service), education services (e.g., online education, online teaching) and commercial services (e.g., online banking, online shopping). While using these electronic communications/services, it is important to secure the data that is being transferred from one party to another.

Securing these communication channels in the presence of an adversary may require mechanisms to protect the confidentiality and integrity of the transmitted data. A confidentiality mechanism ensures the secrecy of the transmitted data, whereas an integrity mechanism provides the receiver with an assurance that the transmitted data has not been modified [1]. These security goals can be provided using symmetric cryptographic algorithms.

1.1 Overview of Symmetric Key Algorithms

The security goals of confidentiality and integrity assurance can be achieved using symmetric key algorithms. Symmetric algorithms use the same/shared secret key at both the sender and receiver side.

1.1.1 Confidentiality Algorithms

Confidentiality can be achieved using a secure symmetric key cipher that provides encryption and decryption functionalities. These symmetric ciphers are mainly categorised into two types: block ciphers and stream ciphers. Over the years there have been several developments of secure block ciphers and stream ciphers.

Currently, the most common block cipher algorithm is Rijndael [2]. Rijndael was selected as the Advanced Encryption Standard (AES). Block ciphers have different modes of operation to provide different services. Examples of block cipher mode of operations providing confidentiality include AES Cipher Block Chaining (AES-CBC) mode [3], AES Counter (AES-CTR) mode [3].

Similarly, stream ciphers can also be used to provide confidentiality. Examples of stream cipher based confidentiality algorithms include Salsa20 [4], Trivium [5].

1.1.2 Integrity Assurance Algorithms

Integrity assurance can be achieved using a message authentication code (MAC) algorithm, or using a cryptographic hash function. There have been several developments of secure algorithms, e.g., Hash-based Message Authentication Code (HMAC) [6], Poly1305 [7], to provide integrity assurance of the transmitted data.

Modes of block cipher, e.g., Cipher Block Chaining MAC (CBC-MAC) [8], can also be used to provide integrity assurance. Similarly, there are some stream cipher based constructions, e.g., ZUC [9], which provide integrity assurance.

1.1.3 Authenticated Encryption (AE) Algorithms

There are several cases where both confidentiality and integrity assurance are needed. For example, any changes made to the ciphertext computed using a stream cipher, has predictable impact on the decrypted plaintext; in such cases an adversary can make arbitrary changes to the plaintext if there is no integrity protection mechanism [10]. Vaudenay demonstrated a padding oracle attack [11] on a cipher block chaining (CBC) mode, which relies on the fact that the implementation of the algorithm decrypts the data before performing any integrity check. Several researchers point out that along with the confidentiality scheme one must also use an integrity scheme to ensure secure communication [10, 12].

This is because having only one of these security services can lead to attacks as described above.

The idea of providing both data confidentiality and integrity assurance is not new. A scheme that provides both of these security goals is called an authenticated encryption (AE) scheme. Individual algorithms can be combined to construct an authenticated encryption scheme.

A generic composition to form such authenticated encryption scheme was described by Bellare and Namprempere [13], and Katz and Yung [14]. These generic compositions are two pass schemes, thus require two passes over the data; one providing confidentiality and the other, integrity assurance. These AE schemes also require two different keys, one for each of their component mechanisms. The computational cost of the two pass scheme is about twice of the single pass scheme [15].

1.1.3.1 Dedicated Authenticated Encryption (AE) Algorithms

The efficiency of the AE may be significantly improved if the construction can, in a single pass over the data, provide both confidentiality and integrity assurance simultaneously. There have been some developments in constructing single pass AE schemes.

Some of these developments are in block cipher modes of operation. For example, Integrity Aware Parallelizable Mode (IAPM) [16], Offset Codebook (OCB) Mode [17] and Galois/Counter Mode (GCM) [18] are block cipher modes providing authenticated encryption.

There have also been some dedicated stream cipher based single pass AE schemes. Examples include SNOW 3G [19], Phelix [20], NLSv2 [21], SOBER-128 [22], SFINKS [23] and Grain-128 [24].

Although there are a variety of AE designs available, many of these designs are not widely adopted. This is due to the fact that some of these algorithms are patented which restricts the use cases, or due to the performance or security issues. Additionally, many applications use obscure algorithms due to their performance advantages. These practical reasons demonstrate the need for new designs providing secure and efficient AE. This has led to the beginning of a new cryptographic competition called CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [25].

1.2 Overview of CAESAR Competition

In 2013 the CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [25] cryptographic competition was announced. The CAESAR competition follows the trend of previous cryptographic competitions such as the Advanced Encryption Standard (AES) competition [26], NIST hash function competition [27]. The goal of this competition is to select a portfolio of algorithms for single pass AE schemes, which are faster than AES-GCM [18] while maintaining at least the same level of security and are suitable for widespread adoption.

There were 57 authenticated encryption cipher proposals submitted to the CAESAR competition. The evaluation process has been performed over four rounds. The ciphers advancing from one round to the next are selected based on the available public cryptanalysis. In the second round of the competition, 28 candidates were selected. In the third round of the competition, 15 ciphers were selected. The final portfolio consisting of the best AE ciphers is expected to be announced by the end of 2017. Table 1.1 provides an overview of the number of different types of designs in each round of the CAESAR competition, classified by the type of design.

Table 1.1: Overview of the Cipher Proposal Submitted to CAESAR

Round	No. of Candidates	Design
1	15	Stream Cipher
	27	Block Cipher
	11	Sponge
	4	Other
2	6	Stream Cipher
	11	Block Cipher
	8	Sponge
	3	Other
3	4	Stream Cipher
	7	Block Cipher
	4	Sponge

This research is focused on the security analysis of selected stream cipher based authenticated encryption schemes from the CAESAR competition. There were 15 stream cipher based AE proposals submitted to the first round of CAESAR. However, many of these cipher proposals were broken during the first round

of the competition, and only six of these AE stream ciphers advanced to the second round. For the third round, four stream cipher based AE schemes namely ACORN, Tiaoxin-346, MORUS and AEGIS were selected.

In this thesis, we first provide a classification of all the original stream cipher based proposals in the CAESAR competition. These classifications are based on the order of encryption and authentication, message injection procedure and the number of keys used. These classification techniques are based on the approach of Bellare and Namprempere [13], Katz and Yung [14], and Al-Mashrafi [28]. The classification of these ciphers can be used to determine similarities among cipher designs and potential cryptanalysis techniques that may be applicable for a particular cipher.

In the later chapters of this thesis, investigations into specific authenticated encryption stream cipher algorithms from the CAESAR competition are presented. In particular, results from the analysis of the CAESAR submissions ACORN [29, 30, 31], Tiaoxin-346 [32, 33] and MORUS [34, 35] are presented in Chapter 3, Chapter 4 and Chapter 5, respectively. These ciphers have all advanced to the third round of the CAESAR competition. The security analysis of these three ciphers determine resistance against well-known forms of cryptanalytic attack including cube attacks, fault attacks, state convergence and state collision attacks, forgery attacks, and rotational attacks.

Note that AEGIS is also a third round candidate which has a similar structure to Tiaoxin-346. Due to the time limitation we have not provided an in-depth security analysis of AEGIS. However, the structural similarity between Tiaoxin-346 and AEGIS is discussed in Chapter 4. The implications of these structural similarities with respect to the vulnerability of the proposal to attack is also discussed.

1.3 Justification of Research

At the beginning of the CAESAR competition, the existing literature on authenticated encryption (AE) stream ciphers consisted largely of efficiency analysis of the proposed algorithms with little analysis of their security. The CAESAR competition requested public scrutiny and analysis of the submissions to determine the most secure and efficient authenticated encryption algorithm. Efficiency of the cipher is easy to measure; the CAESAR committee provided a software and

hardware benchmarking of the cipher proposals submitted to the competition. Security analysis of the ciphers is harder and requires extensive investigation.

The research presented in this thesis aims to contribute to the CAESAR competition by analysing specific stream cipher based authenticated encryption algorithms from the competition submissions. In particular, this research investigates the application of known forms of cryptanalytic attack to these stream cipher based AE schemes.

Similar to some of the previous cryptographic competitions, (for example, the AES competition [26], the SHA-3 [27] competition and the eSTREAM competition) the CAESAR competition expects to identify a portfolio of secure and efficient authenticated encryption ciphers. The outcome of this research will benefit the cryptographic community by providing independent security analysis of these submissions to support the objectives of the CAESAR competition.

1.4 Scope and Objectives of the Thesis

This research aims to investigate the security of the authenticated encryption (AE) cipher proposals based on stream ciphers submitted to the CAESAR [25] competition. The efficiency of the cipher proposals in the CAESAR competition is not considered in this research, since all the submissions in the competition are benchmarked by the CAESAR committee.

There are two angles of attack when it comes to the cryptanalysis of authenticated encryption ciphers: attack on the confidentiality mechanism resulting in a breach of confidentiality of the data, or attack on the integrity mechanism resulting in a breach of the integrity assurance of the data. Investigations into the security of AE stream ciphers can be performed by looking at either of these angles separately, or from combined aspects of both. In general, this research aims to explore and apply common cryptanalytic techniques and adapt them to the specific requirements imposed by the particular authenticated encryption structures.

The objectives of this research are listed below:

1. To classify the stream cipher based authenticated encryption proposals on the bases of order of encryption and authentication, message injection procedure and the number of key-IV pairs used.

2. To apply common cryptanalytic techniques and investigate the security of the selected AE stream cipher proposals:
   - ACORN.
   - Tiaoxin-346.
   - MORUS.

1.5 Research Contributions

This thesis has four contributions. These are described below.

1.5.1 Classification of AE Stream Ciphers in CAESAR

The first contribution of this thesis is the classification of the 15 stream cipher based authenticated encryption proposals from the CAESAR [25] competition. The classification of these ciphers can be used to determine the similarity among ciphers and potential cryptanalysis techniques for any particular cipher. For this classification we followed the approaches of Bellare and Namprempere [13], and Katz and Yung [14], and Al-Mashrafi [28]. These classifications are based on the following characteristics:

- the order in which authentication and encryption are performed,
- the message injection procedure, and
- the number of key-IV pairs used.

Encryption and authentication order classifies the cipher based on different ways to combine an authentication and encryption scheme. These are: Encrypt and MAC (E&M), Encrypt then MAC (EtM) and MAC then Encrypt (MtE). For the E&M and MtE schemes, the MAC is computed on the plaintext; whereas, for the EtM scheme the MAC is computed on the ciphertext.

The message injection procedure is classified based on the way the input message is accumulated into the internal states of the cipher. The message accumulation in the internal state of the cipher can be classified as either direct message injection, or indirect message injection.

Classifications based on the number of key-IV pairs considers whether a single key-IV pair is used for both the confidentiality and integrity components, or whether distinct key-IV pairs are used for each component.

1.5.2 Analysis of ACORN

The second contribution of this thesis is a security analysis of the authenticated encryption stream cipher ACORN. The investigation includes finding state collisions in the internal state of ACORN, and the application of cube attacks and fault based forgery attacks on ACORN.

Finding State Collisions in ACORN: This contribution demonstrates a weakness in the state update function of ACORN, which results in state collisions in the internal state. State collisions occur when two different sets of inputs produce identical internal states at some point of operation of the cipher. The collisions in ACORN can be achieved by manipulating either the initialization vector, associated data or the plaintext. This is a known key attack. Finding this sort of known key collisions in the internal state gives some advantage to the sender since they can frame a forged message which will be accepted as legitimate. The results of this investigation were published in the Proceedings of ACSW 2016 [36].

Cube Attacks on ACORN: In this contribution, we applied the cube attack to a reduced version of ACORN. The full initialization of ACORN comprises 2048 rounds, whereas our attack was applied to the 477 rounds. Our application of cube attack to a 477-round initialization phase of ACORN can successfully recover the secret key with negligible complexity. The result of this investigation demonstrates that ACORN has a large security margin against the cube attack, since it has a 2048-round initialization phase.

We have also shown that the cube attack can recover the initial state of the full version of ACORN with complexity less than that of an exhaustive search attack. The attack works under the nonce-reuse scenario, that is, the adversary needs to encrypt/ authenticate multiple sets of input using the same key and initialization vector. This attack validates the designer claims that ACORN should not be used in the nonce-reuse scenario.

The results from both of these analyses were published in the Proceedings of ATIS 2016 [37].

Fault based Forgery Attack on ACORN: In this contribution, we show that fault injection can be a powerful tool to apply forgery attacks on ACORN.

The fault based forgery attack on ACORN works under the bit-flipping fault model. This requires a single bit-flipping fault in the internal state for a single bit modification in the input message. To apply this attack, the adversary needs access to the implementation of the algorithm at the sender's side. This attack results in a forgery attack on ACORN, by getting a modified message accepted as legitimate at the receiver's side.

1.5.3 Analysis of Tiaoxin-346

The third contribution of this thesis is a security analysis of the authenticated encryption stream cipher Tiaoxin-346. The investigation includes state cycle analysis, and the application of cube attacks and fault attacks to Tiaoxin-346.

Tiaoxin-346 State Cycle Analysis: In this contribution we demonstrate the existence of relatively short cycles in the individual components of a toy version of Tiaoxin-346. The experiments for this state cycle analysis were conducted using a toy version, because it is computationally infeasible to explore all possible states in the full version of Tiaoxin-346. The toy version was created by cutting down the size of the internal state words from 128 bits to 8 bits. We also demonstrate that these relatively short cycles in the individual components do not result in a relatively short cycle for the entire state. The analysis is likely to provide good insight into the full version since it has the same structure as the toy version.

Cube Attacks on Tiaoxin-346: In this contribution we applied the cube attacks to a reduced version of Tiaoxin-346. Tiaoxin-346 has a 15-round initialization phase, whereas our attack is successful up to 4 rounds. Using cube attack we demonstrated the construction of a 4-round distinguisher, which can distinguish the output of Tiaoxin-346 from the output of a randomly generated function. The complexity of this distinguishing attack is 2^3. To the best of our knowledge, this is the best cryptanalytic result obtained for Tiaoxin-346, to date.

This contribution also shows that cube attack can be used in the encryption phase of Tiaoxin-346 to recover the initial state. This attack works under the nonce-reuse scenario and requires an adversary to be able to manipulate the internal state of Tiaoxin-346. This attack validates the designer claims that Tiaoxin-346 should not be used in the nonce-reuse scenario.

Fault Attacks on Tiaoxin-346: This contribution demonstrates two different fault attacks on Tiaoxin-346. The first one is a fault based forgery attack on Tiaoxin-346. This attack works under the bit-flipping fault model. The number of required faulty bits is two times the number of modifications made in the input message. To apply this attack, the adversary needs access to the implementation of the algorithm at the sender's side.

The second type of fault attack applied to Tiaoxin-346 can recover the secret key of the cipher using 36 random multi-byte faults. The complexity of this attack is about 2^{36}. This may be applied using a ciphertext only attack model. This is a differential fault attack and therefore it works under the nonce-reuse scenario. The complexity of the attack can be further reduced by increasing the number of faults. Our approach of fault based key recovery attack on Tiaoxin-346 improves the previous analysis [38] by using a random fault model instead of the bit-flipping fault model.

The results of these fault analyses on Tiaoxin-346 were published in the Proceedings of ACSW 2018 [39].

Similarities in Tiaoxin-346 and AEGIS: This contribution discusses some structural similarities in Tiaoxin-346 and AEGIS-128L. Based on this analysis, it is shown that a similar random fault based state recovery is possible for AEGIS-128L. This state recovery approach of AEGIS-128L improves the previous analysis [38] by using a random fault model instead of the bit-flipping fault model.

1.5.4 Analysis of MORUS

The fourth and final contribution of this thesis is a security analysis of the authenticated encryption stream cipher MORUS. This includes investigating the application of cube attacks, rotational attacks, forgery attacks and fault attacks to MORUS.

Cube Attacks on MORUS: In this contribution we applied the cube attacks to a reduced version of MORUS. The full initialization of MORUS comprises 16 steps. This contribution demonstrates the applicability of cube attacks for key recovery and distinguishers on four and five steps of the initialization phase of MORUS, respectively. To date this is the best attack applied to a reduced version

of MORUS. This contribution shows that cube attack on the reduced version of MORUS performs better than the other existing works [40] describing differential and rotational attacks on the cipher. This contribution also demonstrates that the full version of MORUS has a large security margin against the cube attacks. The results of this investigation were published in the Proceedings of TrustCom 2017 [41].

Rotational Attacks on MORUS: This contribution demonstrates the importance of the use of rotational non-invariant constants to prevent rotational attacks in MORUS. We show that all the operations used in MORUS preserve the rotational properties if rotation invariant constants are used in the state update function. The constants used in MORUS are rotational non-invariant, which provides resistance against this type of attack.

Forgery Attack on MORUS: This contribution describes a forgery attack on MORUS using block deletion. To apply this forgery attack, the internal state of MORUS needs to satisfy some specific conditions. The probability of this forgery attack is 2^{-128} and 2^{-256} for MORUS-640 and MORUS-1280, respectively. The key size of MORUS-640 and MORUS-1280 is 128-bit and 256-bit, respectively; therefore the complexity of this forgery attack is the same as the exhaustive search, and the attack would be considered infeasible.

Fault Attacks on MORUS: In this contribution we described three different fault attacks on MORUS. The first type of fault attack describes the process of constructing a forgery for two different input messages. This attack works under the bit-flipping fault model. The number of faulty bits required is four times the number of bits modified in the input message. To apply this attack, the adversary needs access to the implementation of the algorithm at the sender's side.

The second type of fault attack demonstrates partial key recovery of MORUS under the permanent set-to-zero fault model. This is a ciphertext only attack since the adversary only needs to have access to multiple faulty ciphertexts. Note that this attack works under the nonce-respecting scenario.

The third type of fault attack demonstrates a full key recovery of MORUS under the transient set-to-zero fault model. This requires three multi-byte transient set-to-zero faults to recover the entire key for all the variants of MORUS. This is

also a ciphertext only attack, which works under the nonce-respecting scenario. The results of these fault analyses on MORUS were published in Cryptography, 2(1) [42].

1.6 Structure of the Thesis

This thesis first begins with a background and literature review, followed by a discussion of the investigation of three AE stream ciphers: ACORN, Tiaoxin-346 and MORUS. Following this, the thesis summarises the outcomes of this research. The remaining chapters of the thesis are organised as below:

Chapter 2: In this chapter the terminology used in this thesis is introduced. A detailed discussion is provided on the background literature discussing stream cipher based authenticated encryption schemes. Existing literature containing previous cryptanalyses of authentication algorithms, encryption algorithms and AE stream ciphers are included in this chapter. A background on the stream cipher proposals from the CAESAR competition [25], including their existing cryptanalysis is provided in this chapter. This chapter also provides a classification of the 15 AE stream cipher proposals from the CAESAR competition.

Chapter 3: This chapter provides a detailed description of the authenticated encryption stream cipher ACORN. We discuss the theoretical and experimental results from the application of the state collision attacks, cube attacks and fault based forgery attacks on ACORN. We also discuss the security impact of these attacks on ACORN and give some recommendations for avoiding these attacks.

Chapter 4: This chapter provides a detailed description of the authenticated encryption stream cipher Tiaoxin-346. We discuss the theoretical and experimental results of the state cycle analysis, cube attacks and fault attacks on Tiaoxin-346. We also discuss the similarities between the Tiaoxin-346 and AEGIS construction techniques. Finally, we discuss the security impact of different attacks on Tiaoxin-346 and give some recommendations for avoiding these attacks.

Chapter 5: This chapter provides a detailed description of the authenticated encryption stream cipher MORUS. We discuss the theoretical and experimental results of the cube attacks, rotational attacks, forgery attacks and fault attacks

on MORUS. We also discuss the security impact of these attacks on MORUS and give some recommendations for avoiding these attacks.

Chapter 6: This chapter provides an overall conclusion based on the results reported in this thesis. We compare the three AE stream ciphers based on our security analysis and existing performance analysis. Some possible future directions of research in this area are also indicated in this chapter.

Chapter 2

Background and Literature Review

This chapter provides an overview of authenticated encryption. The literature describing different methods for designing an authenticated encryption cipher is reviewed, including relations among the security notions of an authenticated encryption cipher. The ongoing CAESAR competition and the cipher proposals submitted to this competition are also discussed in this chapter. The cryptanalyses of authenticated encryption ciphers are discussed in detail, including attack goals, models and methods.

The rest of the chapter is organized as follows. Section 2.1 provides a generic overview of stream ciphers. Section 2.2 describes the working principle and methods for classification of authenticated encryption ciphers. Section 2.3 describes authenticated encryption cipher constructions using the stream cipher algorithms. Section 2.3.2 describes the details on the ongoing CAESAR competition. In Section 2.3.2, we also focus on the stream cipher based AE algorithms submitted in the CAESAR competition. Section 2.4 describes different cryptanalytic techniques and models that can be applied to an authenticated encryption cipher. This section also provides a description of some existing cryptanalysis of authenticated encryption stream cipher algorithms. Lastly, Section 2.5 concludes this chapter.

2.1 Overview of Stream Ciphers

Stream ciphers are widely used cryptographic algorithms for providing confidentiality. A stream cipher usually divides the message into successive characters and operates on each character separately to encrypt/ decrypt the message. Based on the size of the character, a stream cipher can be either bit based or word based. In the bit based stream cipher, the cipher operates on each bit separately. In the word based stream cipher, each character consists of a group of bits called a word and the cipher operates on these words to encrypt/ decrypt a message. Before going to the details of a stream cipher, we first define some generic terminologies used in this thesis.

2.1.1 Notation and Terminology

The generic terminologies used in this thesis are as follows:

- Keystream generator: A component that generates pseudo-random binary sequences.
- Secret key, K: An input to the keystream generator, which is only known to the sender and receiver.
- Initialization vector (IV), V: An input to the keystream generator, which is usually publicly available information. The initialization vector usually varies from message to message.
- Keystream, Z: Stream of output bits/ words from the keystream generator.
- Plaintext, P: Stream of plaintext message bits/ words before encryption.
- Ciphertext, C: Stream of the ciphertext message bits/ words after encryption.
- Associated Data, D: Stream of the associated data bits/ words. This part of a message does not require confidentiality; but requires integrity assurance.
- Message, M: Message can be either plaintext or ciphertext or associated data.

- Tag, τ: A specific length sequence generated by the tag generation algorithm. The tag is computed on the message value and is used to determine whether the message has been modified during transmission.
- Encryption algorithm: A process that converts the plaintext into ciphertext using a secret key.
- Decryption algorithm: A process that converts the ciphertext into plaintext using a secret key.
- Internal state: Memory locations where information is stored.
- Internal state size: Amount of information that the internal state of the cipher can hold.
- Initialization: The initialization phase loads and diffuses the key and initialization vector into the internal state of the cipher. The state obtained after the initialization procedure is called the initial state.
- State update function: The process to update the contents of the internal state.
- Output function: The process to compute the keystream bits using the contents of the internal state.

2.1.2 Operations to Provide Confidentiality

Confidentiality ensures that the message is not disclosed to an unauthorised entity. This can be achieved using encryption/decryption algorithms. Here, we briefly define the encryption/decryption procedure for achieving confidentiality using a stream cipher. Figure 2.1 shows the general construction of a stream cipher.

At first, the secret key, K and initialization vector, V are loaded into the internal state of the keystream generator as part of some initialization phase. Following this, the keystream generator is operated for a specified number of iterations without generating any keystream bits. After the initialization phase, the internal state of the keystream generator consists of the initial state and is ready to generate the keystream bits.

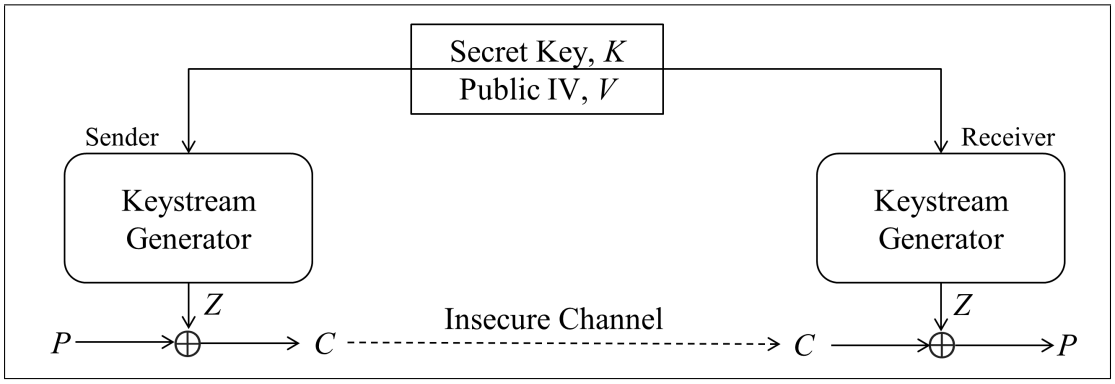

Figure 2.1: Stream Cipher [28]

For keystream generation, the internal state of the keystream generator is updated using a state update function and a keystream bit/ word, Z, is computed at each iteration using the output function of the cipher. The keystream generator is used to generate a keystream sequence by repeating this process.

Lastly, as shown in Figure 2.1, the encryption algorithm uses a combining function to combine the keystream Z with the plaintext P and outputs the ciphertext C. Typically, the keystream is combined with the message using bit-wise XOR function. Stream ciphers using the XOR function as the combining function are called binary additive stream ciphers. Upon encryption of the plaintext, the ciphertext is transmitted through the insecure channel. At the receiver end, the keystream is generated in a similar fashion and then the decryption algorithm combines the ciphertext C with the keystream Z to retrieve the plaintext.

2.1.3 Operations to Provide Integrity Assurance

Integrity assurance of a message provides the receiver with an assurance that the data has not been modified during transmission. Data integrity assurance can be achieved by generating a Message Authentication Code (MAC) tag. Here, we briefly describe a procedure for achieving integrity assurance using a stream cipher.

In a stream cipher based MAC tag generation scheme as shown in Figure 2.2, the input message M is accumulated into the internal state of the cipher after performing the initialization phase. Following this, the cipher is iterated for a specified number of steps in the finalization phases without producing any output bits. At the end of the finalization phase the tag generation function takes input from some of the internal state bits and outputs the MAC tag τ.

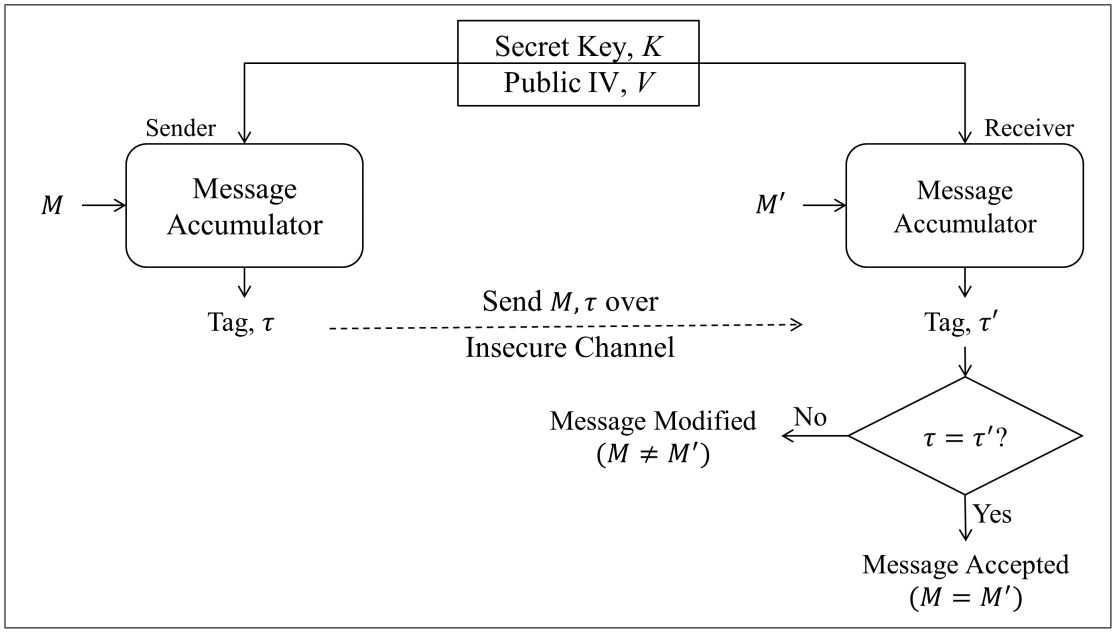

Figure 2.2: Integrity Assurance using Stream Cipher

The MAC tag is appended and then transmitted with the message. Upon receipt of the message, the receiver computes the MAC tag τ' for the received message M' and compares it with the received MAC tag. If the received MAC tag matches the computed MAC tag at the receiver ($\tau = \tau'$), then the receiver assumes that the message has not been modified during transmission. If the received MAC tag does not match the MAC tag computed by the receiver, then the receiver assumes that the message has been modified during transmission through the channel and therefore disregards the received message.

2.2 Overview of Authenticated Encryption

Authenticated encryption is defined as a shared key based transformation where the sender transforms the plaintext into a ciphertext by applying an encryption algorithm and attaches an authentication tag for the message by applying a message authentication algorithm. At the receiver, the decryption and verification process either returns the plaintext by decrypting the ciphertext, or a special symbol indicating that the message authentication has failed [13].

An authenticated encryption scheme needs to provide two functionalities: data confidentiality and assurance of data integrity. Data confidentiality is achieved by transforming the original message into a ciphertext message depend-

ing on the value of a cryptographic key. The ciphertext is transmitted via the insecure channel and the receiver can decrypt to recover the original message by using the corresponding cryptographic key. Data integrity assurance is verified at the receiver by the use of the MAC tag.

The design of message authentication and confidentiality algorithms has been investigated by the cryptographic community for many years. The idea of combining these two primitives to achieve authenticated encryption was first formalized by Bellare and Namprempre [13], and Katz and Yung [14]. Both of these papers independently identify the security notions of AE for a generic scheme, where one combines a standard symmetric encryption scheme with a standard MAC algorithm. Different definitions of security notions for authenticity and confidentiality of an AE scheme were provided by Bellare and Rogaway [43]. These security notions were provided based on the generic composition of AE scheme, where a message authentication algorithm is combined with a confidentiality algorithm to construct the AE scheme and separate keys are used for the authentication and confidentiality algorithms. These schemes are two pass schemes as they operate on the data twice; one providing confidentiality and one providing integrity assurance.

Rogaway [44] introduced the term Authenticated Encryption with Associated Data (AEAD) to address the authentication of the associated data, where one needs to authenticate part of the message but not encrypt it. This is particularly useful in the case of network packets where the payload needs to be encrypted and authenticated, but the header needs to be authenticated only (encrypting the header will fail the packet transfer as the router cannot read the destination address in an encrypted header).

Earlier cryptographic designs to provide confidentiality and authenticity were based on adding redundancy to the message using some checksum function before encryption [45, 46]. However, cipher proposals like this have been found to be vulnerable under chosen plaintext attack [47, 48]. The failure of the redundancy based authentication scheme demonstrates a need for a stronger authentication primitives along with the confidentiality schemes. This realization led to the construction of several new mechanisms to provide authenticated encryption using block ciphers.

Block ciphers are symmetric key cryptographic designs that are applied to a block of data at a time for encryption/decryption. A block cipher can be used

with different modes of operation for providing a specific service, i.e., confidentiality or integrity. In 2001, Jutla [16] constructed a block cipher mode of operation called the Integrity Aware CBC (IACBC) scheme, which provides integrity along with the confidentiality of the message. This requires $m + 2$ number of block encryptions on a plaintext of length m blocks, whereas combining two separate schemes will require $2m$ number of operations. Jutla [16] has also provided a parallelizable mode called Integrity Aware Parallelizable Mode (IAPM), which is a fully parallelizable cipher requiring $m + 1$ block cipher calls. A refined version of IAPM called Offset Codebook Mode (OCB) was constructed by Rogaway et al. [17], which features a single block cipher key, the ability to encrypt arbitrary length of data and with no requirement of random initialization vectors (IVs). Later, two other versions of the OCB scheme were included: OCB2 allows associated data to be included and OCB3 provides minor performance improvement [49, 50]. Other block cipher modes providing authenticated encryption include Counter with CBC-MAC (CCM) mode [51], Galois/Counter mode (GCM) [18], XCBC encryption and XECB authentication mode [52]. Most of the block cipher based authenticated encryption primitives provide confidentiality by turning the block cipher into a stream cipher mode of operation [53]. Dedicated stream ciphers can provide a faster performance. Authenticated encryption primitives using dedicated stream cipher algorithms are discussed later in this chapter.

2.2.1 Classification of Authenticated Encryption Schemes

Authenticated encryption (AE) algorithms have been classified based on different characteristics. These include the order in which encryption and authentication are performed [13, 54], the number of keys and IVs used [28] and the message accumulation procedure [28]. Classifications based on the encryption and authentication order and the number of key-IV pairs used are applicable to the generic composition of AE schemes; however, they can also be used to classify stream cipher based AE schemes. Classification based on the message accumulation procedure is particularly applicable to the stream cipher based AE schemes.

2.2.1.1 Order of Encryption and Authentication

Bellare and Namprempre [13] divided authenticated encryption schemes into three types based on the order in which authentication and encryption occur: Encrypt-and-MAC (E&M), MAC-then-Encrypt (MTE) and Encrypt-then-MAC

(ETM). In an independent work, Krawczyk [54] also analyzed the security properties of authenticated encryption schemes based on the combining methods. They termed their classification as: Encrypt-and-Authenticate (E&A), Authenticate-then-Encrypt (AtE) and Encrypt-then-Authenticate (EtA) respectively. Figure 2.3 shows the diagrammatic form of these approaches. These classifications were provided for generic construction of AE schemes by combining an encryption algorithm with a MAC algorithm; but can also be used for classifying AE stream cipher constructions.

The Encrypt-and-MAC scheme computes the MAC tag for the plaintext message, then encrypts the plaintext to compute the ciphertext and then appends the MAC tag to the ciphertext. This augmented ciphertext is transmitted over the insecure channel. In the case of the MAC-then-Encrypt scheme, the sender computes and attaches the MAC tag of the plaintext to the original message to obtain an augmented message, which is then encrypted to compute the ciphertext. Notice that in this case the MAC is encrypted as well as the plaintext. Lastly, the Encrypt-then-MAC scheme applies the encryption function to the plaintext message first to obtain the ciphertext message, which is then used to compute the value of the MAC tag. The difference between the E&M and ETM scheme is that the E&M computes the MAC tag on the plaintext message where the ETM computes it for the ciphertext message. Note also that only the Encrypt-then-MAC scheme decrypts after verifying the authentication. This approach eliminates the computation of unnecessary decryption if the message is not a valid one, whereas for the other two cases one needs to decrypt the ciphertext first to calculate the MAC tag. Security implications of these classifications are provided in Section 2.4.4.

2.2.1.2 Number of Key-IV Pairs

For the classification approach based on the number of key-IV pairs [28], AE stream ciphers can be classified into single key-IV pair AE schemes and two key-IV pair AE schemes. As the name suggests, the first scheme uses one key-IV pair for both confidentiality and integrity mechanisms, whereas the latter scheme uses two distinct key-IV pairs. It is obvious that for the single key-IV pair AE scheme, compromising one of the services will affect the other one as well.

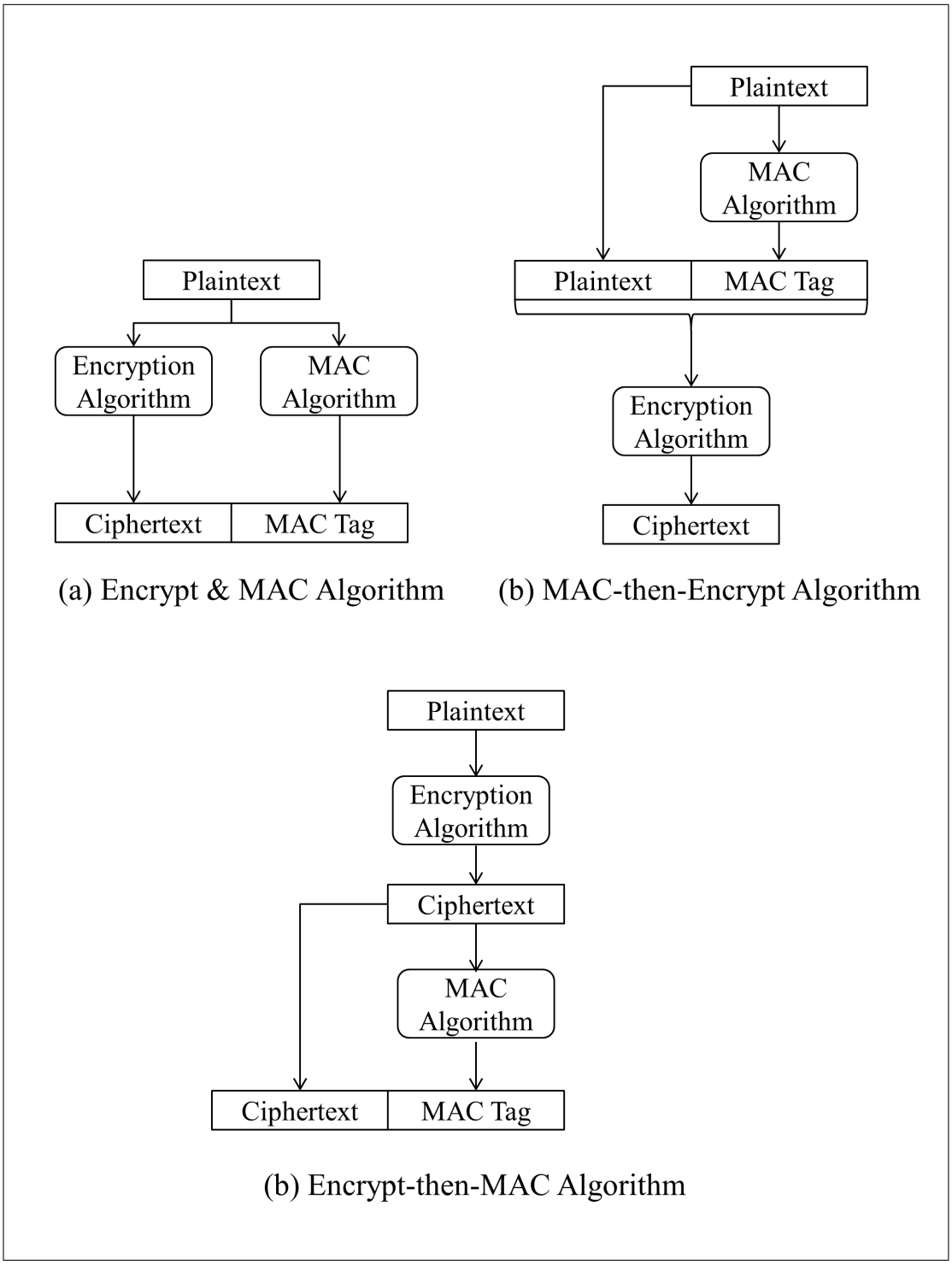

Figure 2.3: Classification of AE Schemes based on the Order of Encryption and Authentication

2.2.1.3 Message Injection Procedure

Al-Mashrafi [28] also classified the message injection procedure for the accumulation component of the MAC as direct or indirect. In the direct message injection case, the plaintext is accumulated in the internal state of the integrity component by using its state update function. Figure 2.4 shows a keystream generator that uses the direct message injection procedure. Figure 2.4 shows only the message

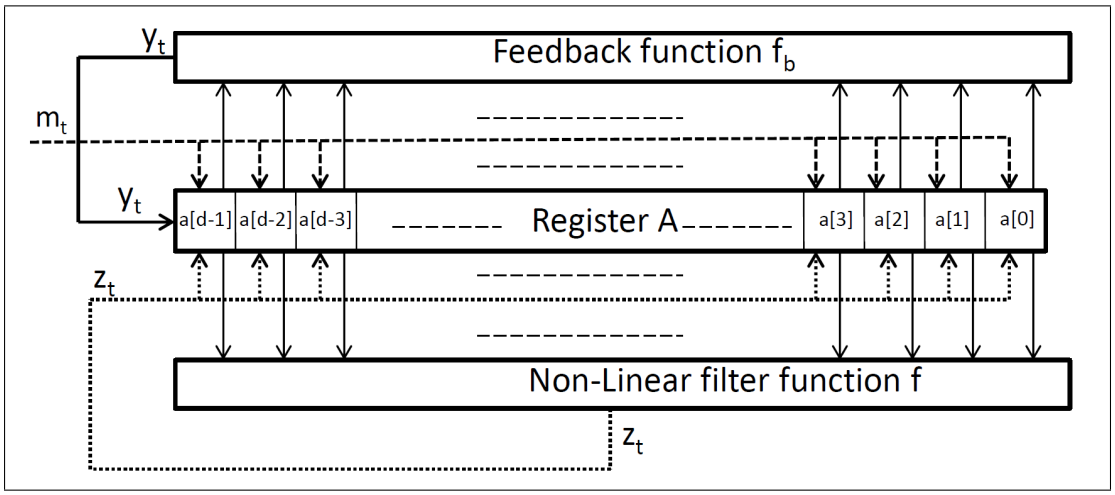

Figure 2.4: Direct Message Injection [28]

accumulation for the direct message injection procedure, it does not show the other two phases of the MAC generation. As shown in Figure 2.4, the output from the state update function and nonlinear function also may be accumulated with the input message into the internal states. This is direct message injection since the message itself is accumulated into the internal state.

For indirect message injection the plaintexts are used to control the accumulation of keystream bits into the states of the integrity component. Figure 2.5 shows a general model for the indirect message injection procedure.

In Figure 2.5, R is a binary shift register, A is the accumulation register, $\otimes$ implies bit-wise multiplication and $\oplus$ implies bit-wise XOR operation. The input message m controls the accumulation of bits from register R to accumulation register A. If the input message bit m_i is zero then there is no change in the contents of the accumulation register, whereas if the input message bit m_i is one then the content of register A is updated by XOR-ing its content with the contents of register R. This is indirect message injection since the input message only controls the accumulation of contents from register R to register A.

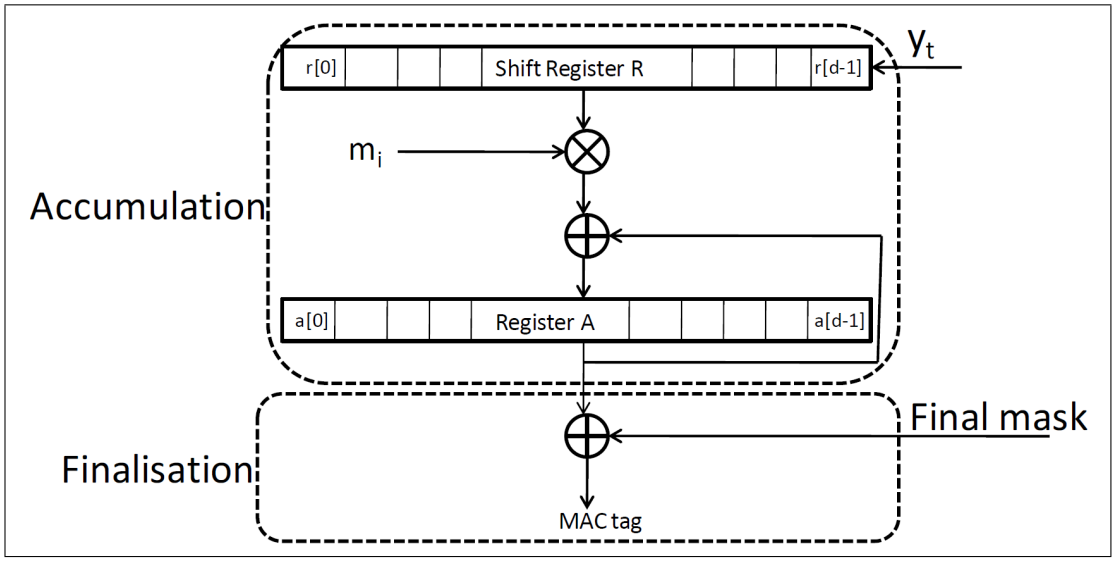

Figure 2.5: Indirect Message Injection [28]

2.3 AE using Stream Cipher Algorithms

An authenticated encryption (AE) stream cipher uses a stream cipher algorithm to provide both assurance of message integrity and confidentiality [13, 55]. The basic construction of an AE stream cipher is shown in Figure 2.6. A pre-arranged shared secret key K is combined with some optional public initialization vector (IV) V to initialize the state of the AE stream cipher at both ends.

At the sender, the AE stream cipher takes the plaintext message P as input and outputs an encrypted message M_E which includes a MAC tag τ. The MAC tag τ is either appended to the ciphertext (for E&M and EtM scheme) or within the encrypted message (for MtE scheme). This encrypted message is then send over the insecure channel.

At the receiving end, M'_E denote the received encrypted message. The AE algorithm at the receiver outputs a decrypted message P along with the computed MAC tag τ' for the message. Finally, the MAC tag τ' computed at the receiver is compared with the received MAC tag τ (appended with the received ciphertext or within the received encrypted message M'_E) to validate the integrity of the message. If the two values are the same then the receiver assumes that the message has not been modified during transmission, otherwise it assumes the message has been modified during transmission and disregards the received message.

The MAC generation and encryption phase of an AE stream cipher usually

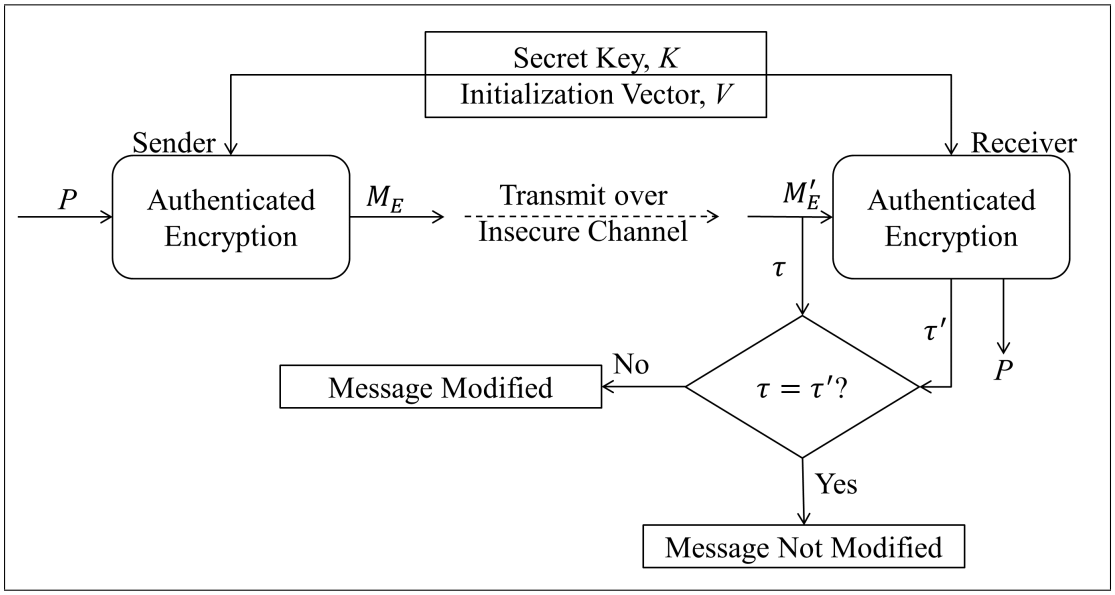

Figure 2.6: AE Stream Cipher

comprises three phases [1, 28]:

- An initialization phase where the secret key and public initialization vector are used to form the internal state of the cipher.
- A phase in which keystream bits are generated from the confidentiality component and combined with the plaintext to encrypt it. At the same time, messages (either plaintext or ciphertext) are accumulated in the integrity component.
- After processing all the plaintext, the contents of the integrity component go through a MAC finalization phase to generate the MAC tag.

For the decryption, the AE stream cipher follows similar steps along with an additional MAC verification phase, where the generated MAC at the receiver is matched with the received MAC tag to verify the integrity of the message.

2.3.1 Historical AE Stream Ciphers

Stream ciphers providing AE include Grain-128a [24], SFINKS [23], Frogbit [56], VEST [57], SOBER-128 [22], Self-Synchronous SOBER (SSS) [58], Non-linear SOBERv2 (NLSv2) [21], Helix [59], Phelix [20], ZUC [9] and SNOW 3G [19]. Among these, ZUC and SNOW 3G are submissions for mobile and telephony application whereas the rest are proposal submissions to the eStream project

[60]. Most of these ciphers are constructed using feedback shift registers and some non-linear function to compute the keystream bits and MAC tag.

Bernstein [61] provided a performance measurement analysis of stream cipher proposals from the phase 2 profile of the eStream project [60]. Apart from those listed above, most of the eSTREAM [60] ciphers do not have an in-built message authentication function and were converted to an authenticated encryption cipher by combining with a Poly1305 message authentication algorithm in the analysis provided by Bernstein [61].

Furthermore, there is not much work done when it comes to the generic composition of constructing an AE scheme by combining a stream cipher with a MAC algorithm. Most of the previous generic framework of AE schemes was constructed from the block cipher instance [13, 14, 54]. Sarkar [62, 63] presented a generic framework to construct standard cryptographic primitives such as MAC, AE, and AEAD using a stream cipher. The generic methods provided in the work by Sarkar [62, 63] construct AE schemes using a stream cipher with a keyed hash function.

2.3.2 AE Stream Cipher Proposals in CAESAR

CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [25] is an ongoing competition that aims to select designs for authenticated encryption suitable for widespread use. Submissions of symmetric cipher proposals (including both block and stream ciphers) in the CAESAR competition have been made available for public analysis. A final portfolio is expected to be announced by the end of 2017. There were 57 cipher proposals submitted to round one of the CAESAR competition. During the first round of the competition, 9 of the candidates were broken and have been withdrawn from the competition [25, 64].

Selected cipher proposals for round two were announced in July 2015. 28 proposals were selected in the second round of the CAESAR competition. The third round of the CAESAR competition was announced in August 2016. In the third round, 15 candidates were selected.

Cipher designs submitted to the CAESAR competition are required to process the following inputs and outputs [25]:

- A variable length plaintext. The designer can specify a maximum plaintext length. The maximum supported length should be at least 65536 bytes.

- A variable length associated data. The designer can specify a maximum associated data length. The maximum supported length should be at least 65536 bytes.
- A fixed length secret message number. This is an optional requirement.
- A fixed length public message number. This is an optional requirement.
- A fixed length key.
- A variable length ciphertext output.

Candidates in the CAESAR competition use different types of primitives in the construction of an authenticated encryption cipher. Among the CAESAR candidates, those cipher candidates which are based on stream cipher primitives include: ACORN [29], AEGIS [65], Enchilada [66], Calico [67], FASER [68], HKC [69], HS1-SIV [70], MORUS [34], PAES [71], PANDA [72], Raviyoyla [73], Sablier [74], Tiaoxin [32], Trivia-ck [75] and Wheesht [76]. A brief summary of the stream cipher based CAESAR candidates is given in Table 2.1 and is documented as follows:

- Cipher: The cipher name and the status of the cipher in the CAESAR competition.
- Construction technique: The construction of these authenticated encryption ciphers use different types of stream cipher primitives, such as a conventional feedback shift register based stream cipher, AES round function based stream cipher, addition-rotation-XOR (ARX) paradigm based stream cipher or a ChaCha stream cipher.
- Parameters: The parameter sets recommended for the ciphers include the key and IV size, the internal state size, and the MAC tag size in bits.
- Classification: The classification is based on the number of keys and IVs used, message injection procedure and order of encryption and authentication. These classifications can be used as a framework to determine similarities among ciphers and potential cryptanalysis techniques that may be applicable to the particular cipher.

More detailed analyses on several ciphers, namely ACORN, Tiaoxin-346 and MORUS are presented in Chapters 3, 4, and 5 of this thesis.

Table 2.1: Summary of AE Stream Cipher Proposals from the CAESAR Competition

AE Cipher	Construction Technique	Parameters	Classification
ACORN (Round 3)	FSR based stream cipher.	Key = 128 IV = 128 State = 293 MAC = 128	Single key-IV pair Direct message injection Encrypt-and-MAC
AEGIS (Round 3)	AES round function based stream cipher.	Key = 128/128/256 IV = 128/128/256 State = 1024/640/768 MAC = 128/128/128	Single key-IV Direct message injection Encrypt-and-MAC
Enchilada (Round 1)	Combination of ChaCha stream cipher, Rijndael block cipher and the authentication mechanism from AES-GCM.	Key = 256 IV = 64 MAC = 128	Single key-IV Direct message injection Encrypt-and-MAC
Calico (Withdrawn) (Round 1)	Stream cipher ChaCha-14 and 20 is used for confideniality algorithm. MAC function Siphash-2-4 is used for the integrity algorithm.	Cipher key = 256 MAC key = 128 IV = 64 MAC = 64 State = 384	Two key-IV pair Direct message injection Encrypt-and-MAC
FASER (Withdrawn) (Round 1)	FSR based stream cipher. Two different registers are used for the integrity and confidentiality component.	Key = 128/256 IV = 64/96 State = 256/384 MAC = 64/96	Single key-IV pair Direct message injection Encrypt-and-MAC
HKC (Withdrawn) (Round 1)	Stream cipher based authenticated encryption.	Key = 256 IV = 128 State = 32768 MAC = 256	Single key-IV pair Direct message injection Encrypt-and-MAC
HS1-SIV (Round 2)	ChaCha Stream cipher based stream cipher.	Key = 256 IV = 96 MAC = 64/128	Single key-IV pair Direct message injection MAC-then-Encrypt
Morus (Round 3)	ARX based stream cipher.	Key = 128/256 IV = 128 State = 640/1280 MAC = 128	Single key-IV pair Direct message injection Encrypt-and-MAC
PAES (Withdrawn) (Round 1)	Stream cipher based on AES round function.	Key = 128 IV = 128 State = 512 MAC = 128	Single key-IV pair Direct message injection Encrypt-and-MAC
PANDA (Withdrawn) (Round 1)	Stream cipher based on round function consisting of a linear transformation and four non-linear transformations.	Key = 128 IV = 128 MAC = 128 State = 448	Single key-IV pair Direct message injection Encrypt-and-MAC
Raviyoyla (Round 1)	Modified version of MAGv2 stream cipher.	Key = 256 IV = 128 MAC = 128	Single key-IV pair Direct message injection Encrypt-and-MAC
Sablier (Round 1)	FSR based stream cipher.	Key = 80 IV = 80 MAC = 32 State = 208	Single key-IV pair Direct message injection Encrypt-and-MAC
Tiaoxin (Round 3)	Stream cipher based on AES round function.	Key = 128 IV = 128 MAC = 128 State = 1664	Singe key-IV Direct message injection Encrypt-and-MAC
Trivia-ck (Round 2)	The keystream generator Trivia-sc is a modified version of Trivium stream cipher. Uses a VPV-Hash algorithm for MAC generation.	Key = 128 IV = 128 MAC = 128 State = 384	Single Key-IV pair Direct message injection Encrypt-and-MAC
Wheesht (Round 1)	Stream cipher based design inspired from Salsa-20 and SipHash.	Cipher Key = 256 MAC key = 256 IV = 128 State = 512 MAC = 256	Two key-IV pair Direct message injection Encrypt-and-MAC

2.4 Cryptanalysis of AE Stream Ciphers

Cryptanalysis is an important aspect of cryptology that deals with the methods of breaching the security of a cryptographic primitive [77]. A cryptosystem needs to go through extensive scrutiny over many years to measure its security strength. We now discuss cryptanalytic techniques, attack models and goals and the security properties required for an authenticated encryption stream cipher.

2.4.1 Attack Goals on AE Stream Ciphers

Attack goals on an authenticated stream cipher may include: secret key recovery, recovery of the initial state of the cipher and MAC forgery. These attack goals are outlined below.

***Secret key recovery*:** The goal of this attack is to recover the secret key, K, shared by the sender and the receiver. A successful secret key recovery will compromise both the confidentiality and integrity components of the cipher.

***Initial state recovery*:** The goal of this attack is to recover the initial internal state of the cipher, which usually is formed in the initialization phase of the cipher by combining the secret key with the public initialization vector. Successful recovery of the initial internal state implies the security breach for that particular key-initialization vector pair only. The security breach compromises both the confidentiality and integrity component. Depending on the initialization process, initial state recovery may also lead to the secret key recovery.

***MAC Forgery*:** This attack tries to create a valid MAC tag for a message that the sender never actually sent. A MAC forgery attack is successful if the attacker can forge a correct MAC with a probability better than trying all the possible MAC values. MAC forgery can be further classified into two types: existential forgery and selective forgery. In existential forgery, an attacker can forge a MAC but need not have any control on the corresponding message contents. On the other hand, for selective forgery the attacker can forge a MAC tag for a message of their choice. This type of attack compromises the integrity component of the cipher.

***Distinguishing attack*:** The goal of this attack is to determine with probability greater than guessing whether a given binary string is a truly random sequence or from a particular keystream generator [78]. This may enable an attacker to identify the cipher from which the keystreams are generated.

2.4.2 Attack Models for AE Stream Ciphers

Cryptanalysis of a cipher is generally based on the assumption that the attacker has knowledge of the cipher design [79] and also has access to the ciphertext. The attacker might have additional resources, for example, access to some plaintext-ciphertext pairs. Based on this, a cryptanalytic technique can be modeled according to the information available to an adversary.

Attack models based on the access to the ciphertext can be classified as [1]:

- Ciphertext-Only Attack (COA)
- Chosen-Ciphertext Attack (CCA)
- Adaptive Chosen-Ciphertext Attack (IND-CCA)

A ciphertext-only attack assumes that the attacker has access to the ciphertext only. This is the weakest model and a cipher is considered very weak if it can be broken by using a COA model. The chosen-ciphertext attack model assumes access to the decryption of arbitrary ciphertext of the attacker's choice. In the adaptive chosen-ciphertext attack, the attacker can obtain the decrypted plaintext for a number of adaptively chosen ciphertexts. That is, the attacker can see the decrypted plaintext for a chosen ciphertext and choose the next ciphertext based on the analysis of the previous ciphertext-plaintext pair.

Attack models assuming access to plaintext can be classified as [1]:

- Known-Plaintext Attack (KPA)
- Chosen-Plaintext Attack (CPA)
- Adaptive Chosen-Plaintext Attack (IND-CPA)

A known-plaintext attack assumes the attacker can mount an attack based on access to some plaintext-ciphertext pairs. For a chosen-plaintext attack, an adversary can obtain encrypted ciphertext for a number of plaintexts of their own choice. For the adaptive chosen-plaintext attack model, the adversary selects the plaintext adaptively based on the observation of the previous plaintext-ciphertext pairs.

2.4.2.1 Security Notions Associated with Confidentiality

Security notions of a symmetric key cryptosystem can be classified as [80, 81, 82]:

- Indistinguishability under a chosen plaintext attack (IND-CPA)
- Indistinguishability under a chosen ciphertext attack (IND-CCA)
- Non-malleability under a chosen plaintext attack (NM-CPA)
- Non-malleability under a chosen ciphertext attack (NM-CCA)

Indistinguishability refers to the inability of an attacker to obtain any information about the plaintext from a challenge ciphertext [82]. Non-malleability implies that an attacker cannot produce a new ciphertext from a challenge ciphertext, that has a meaningful decryption similar to the challenge ciphertext [81]. These security notions are modeled based on the plaintext/ciphertext information available to an adversary.

2.4.2.2 Security Notions Associated with Integrity

Bellare and Namprempre [13] introduced two security notions of integrity:

- Integrity of plaintext (INT-PTXT)
- Integrity of ciphertext (INT-CTXT)

INT-PTXT refers to the inability of an attacker being able to produce a valid message-MAC pair under a chosen plaintext attack. INT-CTXT points to the infeasibility of an attacker to produce a message-MAC pair irrespective of the freshness of plaintext. Similar definitions of INT-CTXT were introduced independently and are called existential unforgeability of encryption [14] and ciphertext unforgeability under a chosen plaintext attack (CUF-CPA) [54].

2.4.3 Attack Methods on AE Stream Ciphers

Cryptanalytic attack methods for secret key recovery or the initial state recovery include: cube attack, differential attack, correlation attack, algebraic attack and rotational attack. Attacks applicable on the integrity component include a MAC forgery attack and a state collision attack. Brute force attacks and side channel

attacks are applicable to both of the components. These attack methods are outlined below.

***Brute force attack*:** Brute force attack is the cryptanalytic method that simply tries all the possible options to find out the correct one. This is also known as an exhaustive search attack. For an n bit key, the complexity of a brute force key recovery attack is computed as 2^n.

***Birthday Attack*:** The birthday attack [83] is a class of brute force attacks that expects to result in a collision in the tag after trying $2^{n/2}$ MAC tag computations for a n bit MAC tag. Success of the collision attack is measured based on the comparison with the birthday attack.

***Differential attack*:** Differential attack [84] is a chosen plaintext attack to recover the secret key of a cipher by manipulating the cipher's non-random behavior. The adversary observes the propagation of differences in the output of the cipher with respect to the differences made in the input and uses these to recover the secret key of the cipher. The differences in the input are usually made by using XOR operations. The attacker then looks into the corresponding differences in the output to detect if any statistical pattern in the distribution exists.

***Correlation attack*:** In the correlation attack [85, 86], an adversary tries to recover the secret key by observing some correlation between the output keystream sequence and the output from a certain component, e.g., a linear feedback shift register (LFSR) used in the cipher. A successful attack is possible when there is a high degree of correlation between the outputs of that certain component and the output of the cipher. In the attack process, the adversary observes the correlation between the keystream sequence and the sequence from one of the underlying LFSRs. This is done by generating the keystream sequence for all the initial states from the underlying LFSR. If a high correlation is obtained between the output keystream and the output from any initial state, then that initial state is chosen as one of the candidates. Similarly, the rest of the LFSRs are analysed. The idea is to recover the initial state of each LFSR separately.

***Slide Attack*:** A slide attack identifies pair of inputs which generate identical states up to a clock difference. These are called slid pairs. The attack was first described by Biryukov and Wagner [87]. Having a slid pair of distance r means that, given a key-IV pair (K, V) and the initial state S associated with the pair, there exists another pair (K', V') such that its initial state S' is equivalent to S

clocked r times.

***Algebraic Attack*:** An algebraic attack [88, 89] represents the relationship between the internal state/secret key and the output keystream bits of the cipher by a set of overdefined algebraic equations. These equations are formed in a pre-computation phase of the attack. In the online phase, the adversary needs to substitute specific keystream bits into the the equations generated in the pre-computation phase. These equations are then solved to recover the initial internal state/secret key of the cipher. The complexity of an algebraic attack is closely related to the maximum degree of the generated equations. Algebraic attacks may require a large number of observed keystream bits to solve the system of equations.

Attacks based on algebraic analysis include a fast algebraic attack [90], an algebraic IV differential attack [91], a higher order differential cryptanalysis [92], a cube attack [93], and rotational cryptanalysis [94]. In this thesis, we have investigated the application of cube attacks and rotational cryptanalysis on several authenticated encryption ciphers. A more detailed description of cube attacks and rotational cryptanalysis are presented in Section 2.4.3.2 and Section 2.4.3.4, respectively.

The application of cube attacks to ACORN, Tiaoxin-346 and MORUS is discussed in Chapters 3, 4 and 5, respectively. The application of rotational cryptanalysis to MORUS is discussed in Chapter 5.

***Side-channel attack*:** A side-channel attack [95] exploits the physical leakage of information from the implementation of a cryptographic algorithm to find out the secret key of the system. It uses implementation specific characteristics to apply an attack. A side channel attacks include a power analysis, an electromagnetic radiation analysis, a timing attack or a fault attack. In this thesis, we have investigated the fault attacks on several authenticated encryption ciphers. More details on the fault attack is provided in Section 2.4.3.3.

The application of fault based forgery attacks on ACORN, Tiaoxin-346 and MORUS is discussed in Chapters 3, 4 and 5, respectively. The application of fault based key recovery attacks on Tiaoxin-346 and MORUS is discussed in Chapter 4 and 5, respectively.

2.4.3.1 State Convergence and State Collision Attack

Convergence [96] occurs when two or more distinct states at a given time are mapped into the same state after α iterations, for some $\alpha > 0$. In the operation of a general stream cipher, state convergence can occur during any phase, i.e., initialization, encryption or the tag generation, if the state update function is not one-to-one. We consider state convergence for the situation in which the keystream generator has no external input, or in which the external input is fixed and an attacker therefore has no opportunity to influence the values of M.

State collisions occur when two different sets of inputs produce identical internal states at some point of operation of the cipher. Here, the input combination implies different possibilities for the key, initialization vector and external input message combinations. State collisions may be exploited in a forgery attack or used in secret key recovery attacks.

Both state convergence and state collision result in identical internal states at some point in the operation of a cipher. The difference is that state convergence occurs when the cipher runs autonomously (or with fixed input), whereas state collisions can be engineered by manipulating the external inputs to the cipher.

In stream cipher based authenticated encryption schemes, a secret key, K, and an initialization vector, V, are used to generate the initial state of the cipher. In general there are three possible scenarios where a state convergence or state collision may occur.

- State collision, or convergence may occur when two input pairs have the same key and different initialization vector. That is, the input pair K, V and K, V' result in identical states at some point in the operation of the cipher.

- State collision, or convergence may occur with two input pairs with a different key and the same initialization vector. That is, the input pair K, V and K', V result in identical states at some point in the operation of the cipher.

- State collision, or convergence may occur when two input pairs have different key and different initialization vector. That is, the input pair K, V and K', V' result in identical states at some point in the operation of the cipher.

The straightforward application of a state collision or state convergence in an authenticated encryption algorithm may enable one to construct a MAC tag forgery attack. The MAC tag is generated as a function of the internal state bits and it will be the same for two different states, given that a collision can be achieved for these states and the inputs to the state do not change once a collision is obtained. Therefore one may construct the same tag for two different states resulting in a forgery attack.

The state collisions, or state convergence may also leak information about the secret key or the internal state of the cipher. In such cases, these attacks can be further used to recover the secret key of the cipher.

2.4.3.2 Cube Attack

The cube attack was introduced by Dinur and Shamir at EUROCRYPT 2009 [93]. The attack can be seen as a generalization of the Higher Order Differential attack [92] and the Algebraic IV differential attack (AIDA) [91]. The goal of the attack is to recover the secret key of a cryptosystem. In the original attack model, the adversary is given a blackbox that evaluates an unknown polynomial, Q constructed over l_k secret variables and l_v public variables. The adversary is also assumed to have access to a single output bit. This is a chosen plaintext attack, which initially was applied to a reduced round version of the Trivium stream cipher [5, 93]. Later, several other symmetric ciphers were analyzed based on this attack [97, 98, 99].

A cube attack is a kind of algebraic attack that aims to recover the secret variable of a cryptographic scheme by manipulating and solving the polynomial equations defined by the scheme. Most of the symmetric cryptographic schemes can be defined by a single master polynomial over $GF(2)$, which contains some secret variables (e.g., secret key) and public variables (e.g., plaintext, ciphertext, initialization vector). The variants of the equations can be derived by changing these secret and public variables. The idea of the cube attack is to generate a sufficient number of closely related low degree equations by manipulating the public variables. An adversary can then solve the generated low degree equations to recover the secret variables of the cryptosystem. The low degree equations are derived by evaluating the master polynomial over all the possible values of some specific public variables and then summing the resultant equations. This is called cube attack, since it requires a sum over all the possible values of the

n-dimensional Boolean cube.

The main observation on the cube attack is that summing the output polynomial over some specific public inputs (cube) might cancel out all the higher degree terms except terms associated with the monomials involving the cube variables, thus resulting in a linear equation if the cube size is appropriately chosen. These polynomials constructed over the cube summation are called superpolys.

The size of the cube is closely related to the degree of the output polynomial. For an output polynomial of degree d a cube of size $d-1$ is guaranteed to produce a linear superpoly. Note that there may exist cubes of size less than $d-1$ that also result in linear superpolys.

The process of cube attack is illustrated with a simple example. Consider the following degree three polynomial consisting of the public variables v_1, v_2, v_3 and secret variables k_1, k_2, k_3:

$$f(v_1, v_2, v_3, k_1, k_2, k_3) = v_1k_1 \oplus v_3k_3 \oplus v_1v_2k_2 \tag{2.1}$$

Evaluating cubes of size two guarantees to result in linear superpolys for this example. For instance, summing the output polynomial over all the possible values of the cube v_1, v_2 results in the linear superpoly k_2.

$$\sum_{(v_1,v_2)\in\{0,1\}^2} f(v_1, v_2, v_3, k_1, k_2, k_3) = k_2 \tag{2.2}$$

Observe that in this example there also exist other cubes of size one, e.g., v_3, that result in a linear superpoly.

In general, the cube attack is performed in two phases: a preprocessing phase and an online phase.

Preprocessing Phase In the preprocessing phase, an adversary finds cubes resulting in linear superpolys. Let $f(K, V)$ define the underlying cryptographic polynomial, constructed over l_k secret variables in $K = \{k_0, \cdots, k_{l_k-1}\}$ and l_v public variables in $V = \{v_0, \cdots, v_{l_v-1}\}$. Generally, the full symbolic representation of $f(K, V)$ is very complex and can not be constructed for the full version of a well designed scheme. Therefore the cube attack needs to estimate the degree of $f(K, V)$ with a complex step in the preprocessing phase.

The degree estimation involves selecting cubes of different sizes and testing them using the linearity test. This is a probabilistic test which indicates whether

the superpoly f_c is linear. The linearity test is performed using the BLR test [100] which selects two random inputs x, y and verifies that $f(0,V) + f(x,V) + f(y,V) = f(x+y,V)$. The superpoly f_c is nonlinear with probability 2^{-j}, if $f(K,V)$ passes the BLR test j times.

If a superpoly is linear with high probability, an adversary then needs to determine whether the linear superpoly will be useful for recovering the secret variables. Note that the linear superpoly can be a constant, which is not very useful. Therefore cubes that pass the linearity test must be checked to see whether they generate linear superpolys in terms of the secret variables, or are just constants. The construction process of the linear superpoly f_c is carried out as follows:

- Find the constant: To compute the constant in the linear superpoly, an adversary sets all the secret variables to zero. The public variables are set to zero everywhere except the cube bits. Summing over all the possible values of the cube bits results in a one bit output, which is the constant for the corresponding linear superpoly.

- Find the coefficient of k_i: To compute the coefficient of a secret variable k_i, an adversary sets all the secret variables to zero except k_i. The public variables are set to zero everywhere except the cube bits. Summing over all the possible values of the cube bits results in a one bit output, which is the coefficient of k_i.

The steps involved for finding a valid cube in the preprocessing phase are summarized below:

1. Estimate the degree d and select the cube size as $l_c = d - 1$.

2. Select a random cube: Randomly select a subset of l_c public variables from $V = \{v_0, \cdots, v_{l_k-1}\}$ as the cube.

3. Perform the linearity test for the selected cube.

4. If the superpoly fails the linearity test then increase the cube size l_c by one and repeat from step 2.

5. If the superpoly passes the linearity test then construct the linear superpoly by computing its coefficients using the technique described above.

6. If the constructed superpoly is a constant then decrease the cube size l_c by one and repeat from step 2.

An adversary needs to find sufficient independent linear superpolys to recover all the secret variables. The number of linearly independent superpolys should be equal to the total number of secret variables.

Online Phase In the online phase, an adversary needs to evaluate $f(K, V)$ for all of the cubes found in the preprocessing phase and use this to determine the values of the corresponding linear superpolys. Values of the linear superpolys are obtained by observing and summing the ciphertext/keystream bit over all the possible values of the corresponding cube. The adversary can then construct and solve the linear equations using Gaussian elimination to recover the secret variables. The steps involved in the online phase of the attack are outlined below:

1. Repeat steps 2 to 3 for all the cubes found in the preprocessing phase.
2. Evaluate $f(K, V)$ over all the possible values of a selected cube and sum the output of these evaluations. This sum represents the value of the corresponding linear superpoly.
3. Construct the linear equation by substituting the value of the linear superpoly.
4. Solve the resulting equations to recover the secret variables.

Depending on the attack model the adversary is assumed to have access to the ciphertext or keystream, or both. The plaintext does not have any effect on the cube summation if it is the same for all evaluations of a given cube and in such cases, an adversary can perform a ciphertext only attack. Alternatively the adversary needs to have access to both the ciphertext and plaintext (to access the keystream) if the plaintext is not the same for different evaluations of a given cube. In this case, an adversary can perform a known plaintext attack. Both of these attack models require an adversary with the capability of manipulating the public variables. Algorithm 2.1 provides a pseudo-code for the generic cube attack.

The application of cube attacks to ACORN, Tiaoxin-346 and MORUS is discussed in Chapters 3, 4 and 5, respectively. Recall that for a degree d polynomial, selecting a cube of size $d - 1$ guarantees to generate linear equations.

Algorithm 2.1 Algorithm for Cube Attack

Inputs: Output keystream bits, Number of linearity tests, Initial cube size, Number of cubes tested
Output: Secret variables of the cryptosystem
Preprocessing Phase
Select a random cube: Estimate the degree, d of the polynomial, choose an initial cube size $l_c \leq d-1$ and select a subset of l_c public variables v_i
Do the linearity test and construct the linear equation
for Number of Cubes Tested **do**
  for Number of linearity tests **do**
    if nonlinear **then**
      if Cubes Tested < Number of Cubes Tested **then**
        Select another cube of size l_c and do the linearity test
      else
        Increase the number of cube variables l_c
      end if
    else
      Compute the coefficients of the secret variables by summing over all the possible values of the cube
      if all the coefficinets are zero **then**
        Select another subset of l_c public variables
      else
        Output the coefficients and construct the linear equation
      end if
    end if
  end for
end for
Do the preprocessing phase until a sufficient number of linear equations are generated
Online Phase
Find the right hand side of the linear equations:
for Each possible cube found in prepreocessing phase **do**
  Compute the output bit
  Sum all the output bits for all the possible values of the cube
end for
Solve the linear equations to recover the secret variables

However, due to the large algebraic degree of the outputs of these ciphers makes it computationally infeasible to evaluate the cubes of size $d-1$.

There may also exist low degree cubes resulting in linear equations. In our application we investigated the existence of cubes which are much smaller than size $d-1$. This process may or may not generate sufficient linear equations. The

application of our cube attack is verified experimentally to determine whether the attack is successful or not. Firstly the number of equations generated is counted. If there are more equations than variables then they are checked for linear independence. Given sufficient linear independent equations an unique solution is guaranteed.

Cube Testers Cube testers were introduced as an extension of the cube attack [101]. Unlike the cube attack, the goal of cube testers is to distinguish the cipher output from the output of a random function. This is detected by observing whether the cube summation always results in a constant. The summation of the underlying polynomial over all the possible values of the cube will always result in zero if the monomial containing the cube does not appear anywhere in the underlying output polynomial. For the example in Section 2.4.3.2, the cubes $\{v_1, v_3\}$ or $\{v_2, v_3\}$ or $\{v_1, v_2, v_3\}$ do not appear anywhere in the example polynomial, thus summing the polynomial over all the possible values of any of these three cubes will always result in the cube sum being equal to zero. For example,

$$\sum_{(v_1,v_3)\in\{0,1\}^2} f(v_1, v_2, v_3, k_1, k_2, k_3) = 0 \tag{2.3}$$

For this example output polynomial this is true regardless of the value of the key. Therefore an adversary can use these cubes to distinguish between the output from a cipher and some randomly generated output.

2.4.3.3 Fault Attack

A fault attack [102] is a powerful cryptanalytic technique applied to the physical implementation of a cryptographic algorithm. This is an active attack which introduces errors in the underlying cryptographic algorithm. An adversary can compare the faulty output with the original output, which may reveal information related to the secret key or internal states, thus compromising the security of the cryptographic algorithm. The goal of the attack can be key recovery, state recovery, or to create a forgery.

Fault attack model The fault attack can be modelled based on the required number of faulty bits, fault type, control on the fault location and time and

duration of the fault [103] . These are discussed below.

***Number of faults*:** This determines the number of faults required to perform the attack. Based on the fault model, the number of bits affected may be a single bit or multiple bits or may be even multiple words.

***Fault type*:** The fault type defines the type of the error introduced in the underlying cryptographic implementation. This can be mainly divided into three types: Stuck-at fault, bit flipping fault and random fault.

- Stuck at fault sets the faulty bits to a specific value of zero or one. This type of fault can also be referred as set-to-zero or set-to-one fault.
- Bit flipping fault flips/ complements bit(s) to introduce error(s) in the cryptographic algorithm.
- Random fault introduces a random error in the cryptographic algorithm. With this type of fault the faulty bits can be set to either zero or one with equal probability. This is the most practical fault type.

***Control on the Fault Location and Time*:** The adversarial model based on the precision of the fault location and time can be categorised in three types: precise, loose and random.

- In the precise control model, the adversary is assumed to have access to fault injection techniques that can inject faults at a precise location of the internal components. The adversary can introduce these faults at a specific time of the operation of the algorithm. This is the strongest model based on the fault location.
- In the loose control model, the adversary have some control on the fault injection location and time.
- In the random model, the adversary does not have any control on the fault injection location and time. In this model, the adversary can just introduce fault(s) at a random time in a random location of the internal components of the cryptographic algorithm. This is the most practical model based on the fault location.

***Duration of the Fault*:** The duration of the fault determines the time period that the fault remains active. The fault model can be categorised into transient and permanent.

- For a transient fault, the faulty bits are changed for a certain period of time. This is also known as a temporary fault, since the fault remains active temporarily.
- For a permanent fault, the faulty bits are changed for the entire duration in the underlying hardware platform. This is also known as a hard fault.

2.4.3.4 Rotational Cryptanalysis

Rotational cryptanalysis investigates the propagation of rotational pairs in the outputs of a cryptographic scheme for given rotational input pairs. This was applied [94] to ciphers composed of three operations: addition, rotation and XOR (ARX). The term rotational cryptanalysis was coined by Khovratovich and Nikolić in 2010 [94]; however, the attack technique was known and applied prior to this work by Biham [104].

Let X and $\overleftarrow{X}^r$ denote two input vectors where $\overleftarrow{X}^r$ is equivalent to the r-bit left rotated version of the input vector X. The input vector pair $(X, \overleftarrow{X}^r)$ is called a rotational input pair. In rotational cryptanalysis, the adversary inputs such a rotational pair $(X, \overleftarrow{X}^r)$ into the underlying cryptographic algorithm and observes the behaviour of the output pairs. Let Z and Z' denote the resulting output pair for the inputs X and $\overleftarrow{X}^r$, respectively and $\overleftarrow{Z}^r$ denote the r-bit left rotated version of Z. Having $Z' = \overleftarrow{Z}^r$ implies that the operations involved in the cryptographic algorithm do not affect the rotational relations in the output pair (Z, Z'). That is, in such a scenario, the rotational relation is preserved in the output pair (Z, Z') for the corresponding input pair $(X, \overleftarrow{X}^r)$. Clearly, if the rotational relation is preserved, the adversary can observe the rotational relation in the output pair to build a distinguisher for the underlying cryptographic algorithm.

In the attack model, the adversary needs to select a pair of inputs. In the initialization phase of stream cipher based scheme, adversary can select the rotational input pairs from the key or the initialization vector or combination of both key-initialization vector. Therefore, the attack model follows the chosen key attack, or the chosen initialization vector attack, or the chosen key-initialization vector attack. Also note that the inputs in the rotational pair are related to each other; thus, this can also be considered as a related key-initialization vector attack.

2.4.4 Security Analysis of AE Stream Ciphers

Bellare and Namprempre's [13] work relates the security notions of integrity with the security notions of confidentiality. This analyzes the generic composition of the AE scheme by indicating whether the composition achieves the security notions of confidentiality and integrity under the assumption that the confidentiality scheme is secure against IND-CPA and the authentication scheme is unforgeable against a chosen plaintext attack. They considered both a weak and stronger version of the integrity notion. According to their work, only the ETM scheme meets with all of the above security notions for a strongly unforgeable MAC, whereas the MTE scheme complies with the IND-CPA and INT-PTXT. The E&M scheme meets the security notions of INT-PTXT only. A similar analysis was done by Krawczyk [54]. These analyses were done under the assumption of generic composition of two primitives using two keys.

Recent work revisits this classification to find a few more subtleties. From the classification based on message accumulation, Al-Mashrafi et. al. [28, 105] suggest that the procedure of input message accumulation plays an important role in direct message accumulation as an attacker might manipulate the input message during the accumulation phase to generate a collision attack. They also point out that for indirect message injection [28, 106, 107], initialization and finalization phases play critical roles in preventing MAC forgery attacks.

2.4.4.1 Cryptanalysis of Historical AE Stream Ciphers

Table 2.2 summarizes a few of the existing authenticated encryption stream ciphers prior to the CAESAR competition. The summary lists the classification of these ciphers and existing attacks on the ciphers. This classification shows that more recent proposals since the eStream project, i.e., Grain-128a, ZUC and Snow-3G are all ETM and indirect injection based schemes. The rest of these proposals are MTE schemes except to Phelix, which uses the E&M scheme. A survey of the literature of these ciphers shows that the common forms of attack against these type of ciphers are mostly based on differential and algebraic techniques.

Table 2.2: Summary of Historical AE Stream Cipher Proposals

AE Cipher	Classification	Cryptanalysis
Grain-128a (2011)	Encrypt-then-MAC (ETM) Indirect message injection Single Key-IV pair	Differential attack [108].
SFINKS (2005)	MAC-then-Encrypt (MTE) Indirect message injection Single Key-IV pair	An algebraic attack was applied by Courtois [109], which requires $2^{79.2}$ keystream bits with an attack complexity of 2^{222}. A fast algebraic attack was applied by Courtois [109], which requires 2^{49} keystream bits with an attack complexity of 2^{70}.
SOBER (2003)	MAC-then-Encrypt (MTE) Direct message injection Single Key-IV pair	Watanabe and Furuya [110] shows that the integrity component of SOBER 128 is vulnerable against differential cryptanalysis. They also applied a MAC forgery attack which has a success probability of 2^{-6}. Cho and Pieprzyk [111] claim that a distinguishing attack is applied to SOBER 128 with the observation of $2^{103.6}$ words.
SSS (2005)	MAC-then-Encrypt (MTE) Direct message injection Single Key-IV pair	A chosen ciphertext attack was applied to SSS which can recover the secret state in 10 seconds by using about 10kB of chosen ciphertext [112]. An algebraic analysis of SSS is described [113] in view of the vulnerability of similarly structured SOBER-t32 against algebraic attack [114].
NLSv2 (2005)	MAC-then-Encrypt (MTE) Direct message injection Single Key-IV pair	Cho and Pieprzyk [115] presented a distinguishing attack against NLSv2 where an attacker can distinguish keystream of NLSv2 from random after observing 279 words of keystream.
ZUC (2011)	Encrypt-then-MAC (ETM) Indirect message injection Two key-IV pair	Two methods [116] of differential attack were constructed against ZUC 1.4 where an adversary can recover the key with an average complexity of $2^{99.4}$ and 2^{67}, respectively. There also exists different algebraic analysis [9, 117] of the ZUC cipher which obtains valid relationship between the internal state and output keystream; however, the attack complexity is found to be infeasible for all the instances. A MAC forgery attack was formulated [118] against the integrity algorithm 128-EIA3 ver.1.4 with a success probability of 1/2.

Phelix (2004)	Encrypt and MAC (E&M) Direct message injection Single Key-IV pair	Wu and Preneel [119] presented a differential linear cryptanalysis which requires $2^{41.5}$ operations to recover the key. However, this attack assumes reuse of the nonce, which breaks all the eStream submissions [61].
SNOW 3G (2006)	Encrypt-then-MAC (ETM) Indirect message injection Two key-IV pair	A fault attack against the SNOW 3G was presented which claims to recover the secret key of the cipher with 22 fault injections [120].

2.4.4.2 Cryptanalysis of AE Stream Ciphers in CAESAR

A summary of the available cryptanalysis and status of the AE stream cipher proposals from the CAESAR competition; dated at October 2017, is presented in Table 2.3. This shows that of the 15 stream cipher based candidates, Calico, FASER, HKC, PANDA, PAES and Raviyoyla are completely broken and withdrawn from the competition. Also, Table 2.3 notes an attack exists against Sablier that recovers the state of the cipher with a complexity of 2^{44} operations. This attack shows that Sablier is far from the goal of its security claim (80-bit). The designer of Sablier proposed some modifications in a new version of Sablier v1.1 [121] to provide resistance against this attack. For Wheesht, a distinguishing attack was presented with an attack complexity of $2^{70.3}$ which is far less than the security claimed by the designer (256-bit). Furthermore, Table 2.3 shows a key recovery attack on Wheesht with an attack complexiy less than that of an exhaustive search attack. Among the other first round candidates, there is no published analysis of Enchilada (round 1). None of these ciphers have advanced to the second round of the competition.

Note that there has not been any publicly available analysis of HS1-SIV (Round 2). During the second round of the competition Baksi et al. presented some observations on constructing cube based distinguishers for Trivia-ck. These two ciphers were not included in the third round of the competition.

The remaining four stream cipher based candidates: ACORN, AEGIS, MORUS and Tiaoxin-346 have advanced to the third round of the CAESAR competition.

Table 2.3: Cryptanalysis of the AE Stream Cipher Proposals in CAESAR

AE Cipher	Cryptanalysis
ACORN (Round 3)	Liu and Lin [122] showed that there exist slid pairs for ACORNv1 for a single (key, IV) pair and two related (key, IV) pairs. Their state recovery attack requires approximately 2^{180} and 2^{130} CPU cycles for the single (key, IV) pair and two related (key, IV) pair setting. Chaigneau et al. [123] applied a key recovery attack to ACORNv1 under the nonce-reuse settings. This attack recovers the internal state of ACORNv1 when three or more messages are encrypted using the same key and IV. Lafitte et al. [124] and Dwivedi et al. [125] provided some observations on the SAT based cryptanalysis of ACORN. Both of their attack require a complexity higher than the exhaustive search attack. Lafitte et al. [124] also generalize some of the results of this thesis described in Section 3.4 of Chapter 3, to find state collisions in ACORN. Todo et al. [126] described a variant of cube attack on a reduced version of ACORN. The complexity of this attack is 2^{122}. Dey et al. [127] described a fault based key recovery attack on ACORN, which requires a complexity of $2^{55.85}$. Probabilistic linear relations among the input message and output ciphertext bits of ACORN were investigated by Roy et al. [128]. Jiao et al. [129] also presented a state recovery attack on ACORN by finding linear approximations of the output function of ACORN. Their analysis requires 2^{157} tests to find the right solution for state recovery.
AEGIS (Round 3)	Minaud [130] described the existence of linear biases in different variants of AEGIS keystream, but noted that the bias described in their analysis does not threaten the security of AEGIS due to the high data complexity. Dey et al. presented differential fault based key recovery attack on AEGIS-128, AEGIS-256 and AEGIS-128L which require 384, 512 and 512 bit-flipping faults, respectively. This attack works under the nonce-reuse scenario for which designer of AEGIS does not claim any security.
Enchilada (Round 1)	No published analysis.
Calico (Withdrawn) (Round 1)	Dobraunig et al. [131] presented a forgery and key recovery attack against Calico. Their attack requires 2^{64} online queries with an overall complexity of 2^{64}. The claimed security is 127 bits for the confidentiality and 63 bits for the MAC.
Tiaoxin-346 (Round 3)	Dey et al. [38] presented a fault based attack on Tiaoxin-346. The attack uses differential fault analysis of Tiaoxin-346 under the nonce-reuse scenario. The attack requires to introduce 384 bit-flipping faults in the internal state of Tiaoxin-346 to recover the 128 bit secret key. To date this is the only published analysis of Tiaoxin-346.

FASER (withdrawn) (Round 1)	Xu et al. [132] presented a correlation attack on FASER which leads to the initial state recovery with a time complexity of 2^{36}; much less than the exhaustive search. They also presented a distinguishing attack using only 16 keystream words. Feng and Zhang [133] further provided a key recovery attack which can recover the secret key by using a real time PC and only 64 keystream words.
HKC (Withdrawn) (Round 1)	Saarinen [134] showed that the update function of HKC is linear in the upper 55 bits of the state which can lead to trivial message forgery attack.
HS1-SIV (Round 2)	No published analysis.
MORUS (Round 3)	Mileva et al. [135] described the existence of a distinguisher for MORUS-640, which can analogously be used for MORUS-1280, under the assumption of nonce-reuse scenario. Mileva et al. [135] also analysed the state update function of MORUS; reporting collisions in the internal state, if the adversary is capable of injecting differences into the internal state. Dwivedi et al. [125] analysed the complexity of SAT based state recovery for MORUS-640 which requires a complexity of 2^{370}. Dwivedi et al. [40] also applied the differential and rotational cryptanalysis on a reduced version of MORUS. Their application of differential cryptanalysis on MORUS-1280-256 with 18 initialization rounds (3.6 steps) can recover the 256-bit key with a complexity of 2^{253}. They have also described a rotational attack on MORUS-1280-256 with 8 initialization rounds (1.6 steps) which can recover the key with a complexity of 2^{251}.
PANDA (Withdrawn) (Round 1)	Sasaki and Wang [136] presented a forgery attack against nonce-repeating version of PANDA which can be applied with 2^5 chosen plaintexts and a negligible memory. The complexity of this attack is 2^{64} which is less than the claimed 128-bit security. Also, Feng et al. [137] presents a state recovery attack against PANDA-s which requires 137 known plaintext/ciphertext pairs and about 2GB memory. The complexity of this attack is 2^{41}.
PAES (Withdrawn) (Round 1)	Jean et al. [138] presented a state recovery and forgery attack against PAES with a complexity of 2^{11}, where the designer claimed a 128 bit security.
Raviyoyla (Round 1)	Yao et al. [139] presented a MAC forgery attack against Raviyoyla by using a single query and claimed that the complexity is negligible.
Sablier (Round 1)	Feng and Zhang [140] presented a state recovery attack against the encryption algorithm of Sablier. This requires about 2^{44} operations with 24 keystream words. Based on the state recovery attack they have also presented a key recovery attack and forgery attack against Sablier. This is less than the 80 bits security claimed by the designer.

Trivia-ck (Round 2)	Baksi et al. [141] showed that distinguishers can be obtained for modified versions of Trivia-ck with reduced initialization rounds. Their distinguishing attack works up to 900 rounds of the keystream generator of Trivia-ck.
Wheesht (Round 1)	Canteaut and Leurent [142] presented a distinguishing attack against Wheesht with $2^{70.3}$ known plaintext words. They also presented a key recovery attack with a complexity of 2^{192} for versions of Wheesht with a single finalization round.

2.5 Summary

This chapter reviewed the existing literature related to stream cipher based authenticated encryption designs. The security goals and construction techniques of authenticated encryption were outlined. Classification of various stream cipher based construction in the CAESAR competition, including the third round candidates ACORN, AEGIS, MORUS and Tiaoxin-346, were presented in this chapter. Also different attack methods applicable to authenticated encryption schemes were examined.

In the later chapters of this thesis, investigations of specific authenticated encryption stream cipher algorithms from the third round of the CAESAR competition are presented. The cipher proposals submitted in the CAESAR competition require public analysis to select the best authenticated encryption algorithm. The research presented in this thesis addresses this gap by analysing specific authenticated encryption algorithms from the CAESAR competition. In particular, the applicability of different known forms attacks on the CAESAR submissions ACORN, Tiaoxin-346 and MORUS was investigated. All of these cipher proposals are included in the third round of the CAESAR competition. The attack methods used in this thesis include cube attacks, fault attacks, state convergence and state collision attacks, and rotational attacks.

The rest of the chapters are organized as follows. Analysis of the authenticated encryption algorithm ACORN is presented in Chapter 3. Analysis of Tiaoxin-346 is presented in Chapter 4. Analysis of MORUS is presented in chapter 5. Finally, the conclusion of this thesis is presented in Chapter 6.

Chapter 3

Analysis of ACORN

This chapter investigates the security of the authenticated encryption stream cipher ACORN. The investigation includes the applicability of cube attacks and forgery attacks on ACORN.

The results presented in Section 3.4 of this chapter have been published in the Proceedings of the Australasian Computer Science Week Multiconference 2016 (ACSW 2016) [36]. The results presented in Section 3.5 of this chapter have been published in the Proceedings of the 7^{th} Conference on Applications and Techniques in Information Security - ATIS 2016 [37].

ACORNv1 [29] is a binary feedback shift register based authenticated encryption stream cipher which was submitted to the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [25]. To date there are three versions of ACORN: ACORNv1 [29], ACORNv2 [30] and ACORNv3 [31].

All versions of ACORN use a 128-bit secret key and a 128-bit initialization vector. This proposed cipher provides authentication and encryption functionalities for the input message. Encryption is performed by XOR-ing the plaintext message bits with the keystream bits output by the keystream generation function. Message authentication is provided by including a tag computed from the message. The cipher also provides authentication but not encryption of the associated data (AD).

The rest of the chapter is organised as follows. Section 3.1 describes the notations and operations used in this chapter. Section 3.2 provides a detailed

description on the different versions of ACORN. Section 3.3 discusses the existing public cryptanalysis of ACORN. Section 3.4 provides our analysis on finding the state collisions and state convergence in ACORN. Section 3.5 describes our application of cube attack on ACORN. Section 3.6 illustrates a fault based forgery attack on ACORN. Finally, Section 3.7 provides an overall summary on the security analysis of ACORN.

3.1 Notations and Operations

- $\oplus$: Bit-wise XOR operation.
- $\otimes$: Bit-wise AND operation.
- $||$: Concatenation
- $\overline{X}$: Bitwise complement of X.
- $K = k_0k_1...k_{127}$: The secret key of size 128 bits.
- $V = v_0v_1...v_{127}$: The initialization vector of size 128 bits.
- $M^t = m_0m_1 \cdots m_{l_d-1}$: The external input to the state at time t.
- $P^t = p_0p_1 \cdots p_{l_d-1}$: The input plaintext block at time t.
- $D^t = d_0d_1 \cdots d_{l_d-1}$: The input associated data block at time t.
- C^t: The output ciphertext block at time t.
- τ: Authentication tag.
- l_p: Length of the plaintext/ ciphertext in bits where $0 \leq l_p < 2^{64}$.
- l_d: Length of the associated data in bits where $0 \leq l_d < 2^{64}$.
- l_{tag}: Length of the tag in bits where $64 \leq l_{tag} \leq 128$.
- $S^t = s_0^t s_1^t ... s_{292}^t$: The internal state of 293 bits, at time t.
- f_z: Keystream generation function, also referred as output function.
- Z^t: Output keystream bit at time t.

3.2 Description of ACORN

ACORN uses a 128-bit key, $K \in \{0,1\}^{128}$ and a 128-bit initialization vector, $V \in \{0,1\}^{128}$. The cipher takes a plaintext message $P \in \{0,1\}^*$ of arbitrary length l_p within the range $0 \leq l_p \leq 2^{64}$. The input to the cipher may also include associated data $D \in \{0,1\}^*$, again of arbitrary length l_d within the range $0 \leq l_d \leq 2^{64}$. The associated data does not require confidentiality and so is not encrypted, but an integrity mechanism is applied. The output of ACORN consists of ciphertext $C \in \{0,1\}^{l_p}$ and an authentication tag $\tau \in \{0,1\}^{l_{tag}}$, where $64 \leq l_{tag} \leq 128$ denote the size of the tag. The designer of the cipher strongly recommends the use of a 128-bit tag. The structure of ACORN is based on six binary linear feedback shift registers (LFSRs) of lengths 61, 46, 47, 39, 37 and 59, respectively, and an additional 4-bit register. This gives the cipher a total internal state, $S = \{s_0, \cdots, s_{292}\}$ of 293 bits.

Operations performed in the ACORN stream cipher can be divided into four phases: Initialization, Encryption, Tag Generation, and Decryption and Tag Verification. The differences between ACORNv1 and ACORNv2 occur only in the initialization phase. On the other hand, ACORNv3 differs from ACORNv2 in most of the phases because of the changes made in one of its component functions.

The generic description of the cipher provided in this chapter is based on ACORNv1. Changes made in the later versions of the cipher, i.e., ACORNv2 and ACORNv3, are specified where it is due. For the rest of this chapter unless specifically noted, the term ACORN refers to the initial version ACORNv1.

The operations performed by the cipher during the different phases are based on several component functions. These component functions are described below.

3.2.1 ACORN Component Functions

ACORN uses three functions: an output function which generates a single bit from the internal state of the cipher, a nonlinear feedback function used to update the internal register stage s_{292} and a state update function used to update the register stages of the LFSRs. In the original description of the cipher, the designer describes the state update function as follows:
The functions f' and Y^t in this algorithm are defined below.

Algorithm 3.1 Wu's [29] Pseudo Code for ACORN State Update Function

$s^t_{289} = s^t_{289} \oplus s^t_{235} \oplus s^t_{230}$
$s^t_{230} = s^t_{230} \oplus s^t_{196} \oplus s^t_{193}$
$s^t_{193} = s^t_{193} \oplus s^t_{160} \oplus s^t_{154}$
$s^t_{154} = s^t_{154} \oplus s^t_{111} \oplus s^t_{107}$
$s^t_{107} = s^t_{107} \oplus s^t_{66} \oplus s^t_{61}$
$s^t_{61} = s^t_{61} \oplus s^t_{23} \oplus s^t_{0}$
Compute Output Function, Y^t
Compute Feedback Function, f'
for $i := 0$ **to** 291 **do**
  $s^{t+1}_i = s^t_{i+1}$
end for
$s^{t+1}_{292} = f' \oplus M^t$

3.2.1.1 Output Function

At time instant t, the output function of ACORN described by Wu [29] takes input from five of the stages: s_{12}, s_{61}, s_{154}, s_{193} and s_{235}. His version of the output bit Y^t at time t is defined as:

$$Y^t = s^t_{12} \oplus s^t_{154} \oplus maj(s^t_{61}, s^t_{193}, s^t_{235}) \tag{3.1}$$

where the majority function is defined as:

$$maj(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_1x_3 \tag{3.2}$$

Note that in Equation 3.1, the contents of register stages s_{61}, s_{154} and s_{193} have undergone an intermediate update before they are used as inputs. This is not highlighted in Wu's version of the output function but is implicit in the state update function given in Algorithm 3.1.

We wish to express the output function directly in terms of the contents of the register prior to update. To do this, we substitute the updated values of stages s_{61}, s_{154} and s_{193} into Equation 3.1 and also use the definition of the majority function to obtain the following output function:

$$\begin{aligned} Z^t = {} & s^t_{12} \oplus s^t_{154} \oplus s^t_{111} \oplus s^t_{107} \oplus s^t_{61}s^t_{193} \oplus s^t_{61}s^t_{160} \oplus s^t_{61}s^t_{154} \oplus s^t_{23}s^t_{193} \oplus s^t_{23}s^t_{160} \\ & \oplus s^t_{23}s^t_{154} \oplus s^t_{0}s^t_{193} \oplus s^t_{0}s^t_{160} \oplus s^t_{0}s^t_{154} \oplus s^t_{193}s^t_{235} \oplus s^t_{160}s^t_{235} \oplus s^t_{154}s^t_{235} \oplus s^t_{61}s^t_{235} \\ & \oplus s^t_{23}s^t_{235} \oplus s^t_{0}s^t_{235} \end{aligned} \tag{3.3}$$

Note that including the intermediate updates in the output function increases the number of input register stages by five, as shown in Equation 3.3. However, it does not affect the degree of the output equation, which remains quadratic. In total, the output function takes input from ten register stages: s_0^t, s_{12}^t, s_{23}^t, s_{61}^t, s_{107}^t, s_{111}^t, s_{154}^t, s_{160}^t, s_{193}^t and s_{235}^t.

3.2.1.2 Feedback Function

The generic form of the feedback function f' of ACORN, as described by Wu, uses the contents of the fourteen register stages s_0, s_{12}, s_{23}, s_{61}, s_{66}, s_{107}, s_{111}, s_{154}, s_{160}, s_{193}, s_{196}, s_{230}, s_{235}, s_{244} and two control bits a, b. The values for a and b vary depending on the phase: initialization, encryption or tag generation. Wu describes the feedback function as:

$$f' = s_0^t \oplus \overline{s_{107}^t} \oplus maj(s_{244}^t, s_{23}^t, s_{160}^t) \oplus ch(s_{230}^t, s_{111}^t, s_{66}^t) \oplus a^t s_{196}^t \oplus b^t Y^t \quad (3.4)$$

where $\overline{s_{107}^t}$ denote the complement value of the content of register stage s_{107}^t. In Equation 3.4, the choice function is defined as:

$$ch(x_1, x_2, x_3) = x_1 x_2 \oplus \overline{x_1} x_3 \quad (3.5)$$

where $\overline{x_1}$ denote the complement of x_1. Similar to the output function, Wu's version of the feedback function given in Equation 3.4 incorporates implicitly the intermediate updates of the register stages s_{61}, s_{107}, s_{154}, s_{193} and s_{230}. After substituting the updated values of these register stages and using algebraic expressions for the majority function, choice function and output function, we can express the generic form of the feedback function as follows:

$$\begin{aligned} f = {} & 1 \oplus s_0^t \oplus s_{107}^t \oplus s_{61}^t \oplus s_{244}^t s_{23}^t \oplus s_{23}^t s_{160}^t \oplus s_{160}^t s_{244}^t \oplus s_{230}^t s_{111}^t \oplus s_{196}^t s_{111}^t \\ & \oplus s_{193}^t s_{111}^t \oplus s_{230}^t s_{66}^t \oplus s_{196}^t s_{66}^t \oplus s_{193}^t s_{66}^t \oplus a^t s_{196}^t \oplus b^t \left(s_{12}^t \oplus s_{154}^t \oplus s_{111}^t \oplus s_{107} \right. \\ & \oplus s_{61}^t s_{193}^t \oplus s_{61} s_{160}^t \oplus s_{61}^t s_{154}^t \oplus s_{23}^t s_{193}^t \oplus s_{23}^t s_{160}^t \oplus s_{23}^t s_{154}^t \oplus s_0^t s_{193}^t \oplus s_0^t s_{160}^t \\ & \left. \oplus s_0^t s_{154}^t \oplus s_{193}^t s_{235}^t \oplus s_{160}^t s_{235}^t \oplus s_{154}^t s_{235}^t \oplus s_{61}^t s_{235}^t \oplus s_{23}^t s_{235}^t \oplus s_0^t s_{235}^t \right) \end{aligned} \quad (3.6)$$

Equation 3.6 is quadratic for given constant values of a and b. We fix the values of a and b to specify four possible feedback functions. We use the notation f_T, f_A, f_E and f_I to denote the specific feedback functions for the cases: $(a, b) = (0, 0), (0, 1), (1, 0)$ and $(1, 1)$ respectively. These four feedback functions are

presented explicitly in Table 3.1.

Table 3.1: Feedback Functions for Different Operation Phases of the Cipher

a	b	f	Feedback Function
0	0	f_T	$1 \oplus s_0^t \oplus s_{61}^t \oplus s_{107}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right)$
0	1	f_A	$1 \oplus s_0^t \oplus s_{12}^t \oplus s_{61}^t \oplus s_{111}^t \oplus s_{154}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right) \oplus \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t\right) \left(s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right) \oplus s_{235}^t \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t \oplus s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right)$
1	0	f_E	$1 \oplus s_0^t \oplus s_{61}^t \oplus s_{107}^t \oplus s_{196}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right)$
1	1	f_I	$1 \oplus s_0^t \oplus s_{12}^t \oplus s_{61}^t \oplus s_{111}^t \oplus s_{154}^t \oplus s_{196}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right) \oplus \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t\right) \left(s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right) \oplus s_{235}^t \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t \oplus s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right)$

3.2.1.3 State Update Function

In each iteration of the state update function the register stages are updated by shifting, except for seven stages s_{60}, s_{106}, s_{153}, s_{192}, s_{229}, s_{288} and s_{292}. The last register stage, s_{292}, is updated by combining the output of the nonlinear feedback function with the input message. Register stages s_{60}, s_{106}, s_{153}, s_{192}, s_{229} and s_{288} are updated as linear combinations of the contents of selected register stages.

The state of ACORN at time $t+1$ is defined as:

$$s_i^{t+1} = \begin{cases} M^{t+1} \oplus f\left(S^t, a^t, b^t\right) & \text{for } i = 292 \\ s_{289}^t \oplus s_{235}^t \oplus s_{230}^t & \text{for } i = 288 \\ s_{230}^t \oplus s_{196}^t \oplus s_{193}^t & \text{for } i = 229 \\ s_{193}^t \oplus s_{160}^t \oplus s_{154}^t & \text{for } i = 192 \\ s_{154}^t \oplus s_{111}^t \oplus s_{107}^t & \text{for } i = 153 \\ s_{107}^t \oplus s_{66}^t \oplus s_{61}^t & \text{for } i = 106 \\ s_{61}^t \oplus s_{23}^t \oplus s_0^t & \text{for } i = 60 \\ s_{i+1}^t & \text{otherwise} \end{cases} \tag{3.7}$$

where $0 \leq i \leq 292$. Figure 3.1 shows the state update process for ACORN in diagrammatic form. Depending on the phase the cipher is in, the input M can denote either the key, initialization vector, associated data, plaintext or fixed length padding, and the feedback function f will be one of the four functions given in Table 3.1.

3.2.2 Initialization

The initialization procedure uses the key, initialization vector and associated data as inputs to construct the initial state of ACORN. The initialization process is

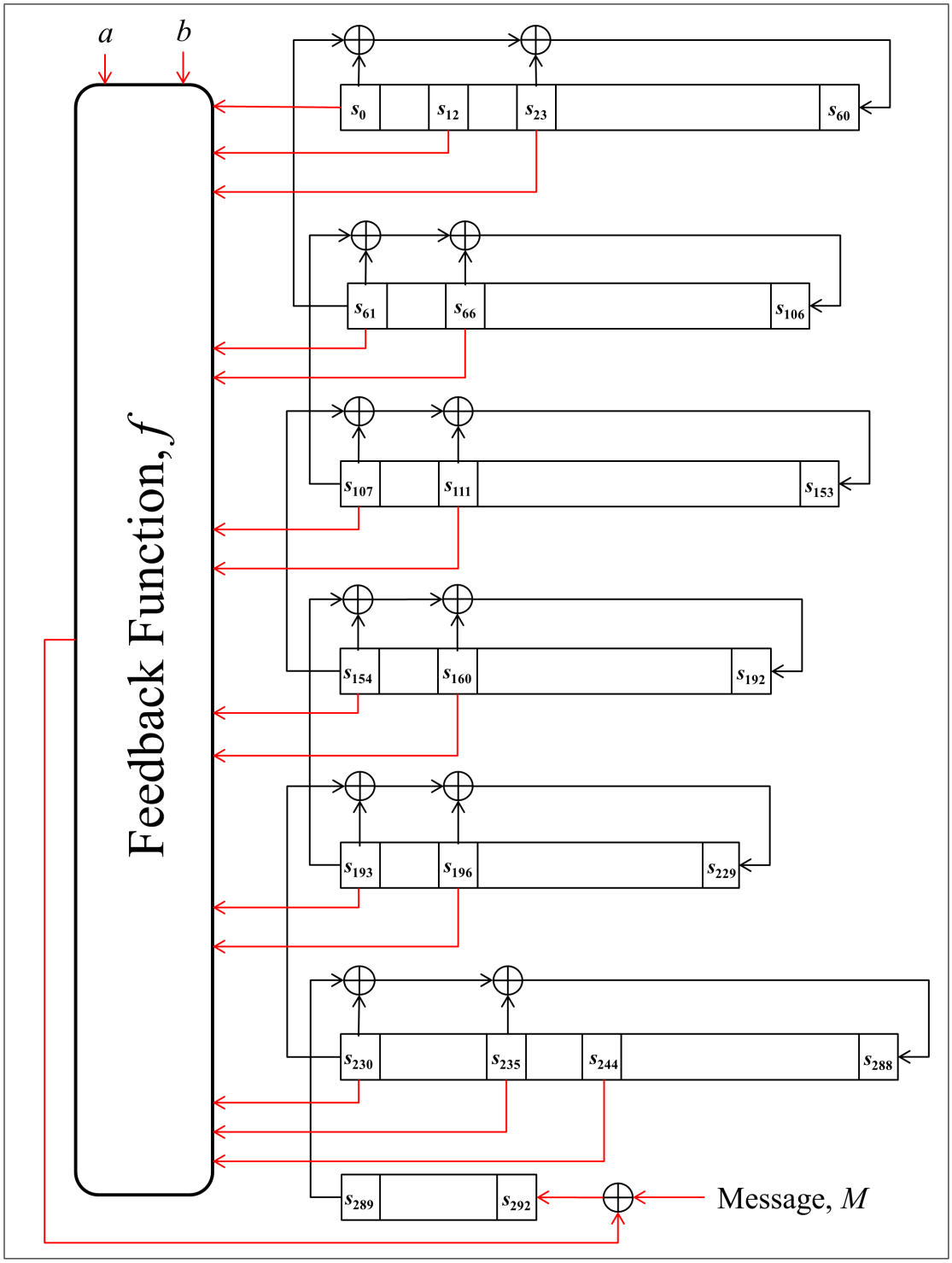

Figure 3.1: ACORN State Update

performed in two phases: first the key and initialization vector are loaded and then the associated data is loaded. Figure 3.2 illustrates the initialization phase of ACORN.

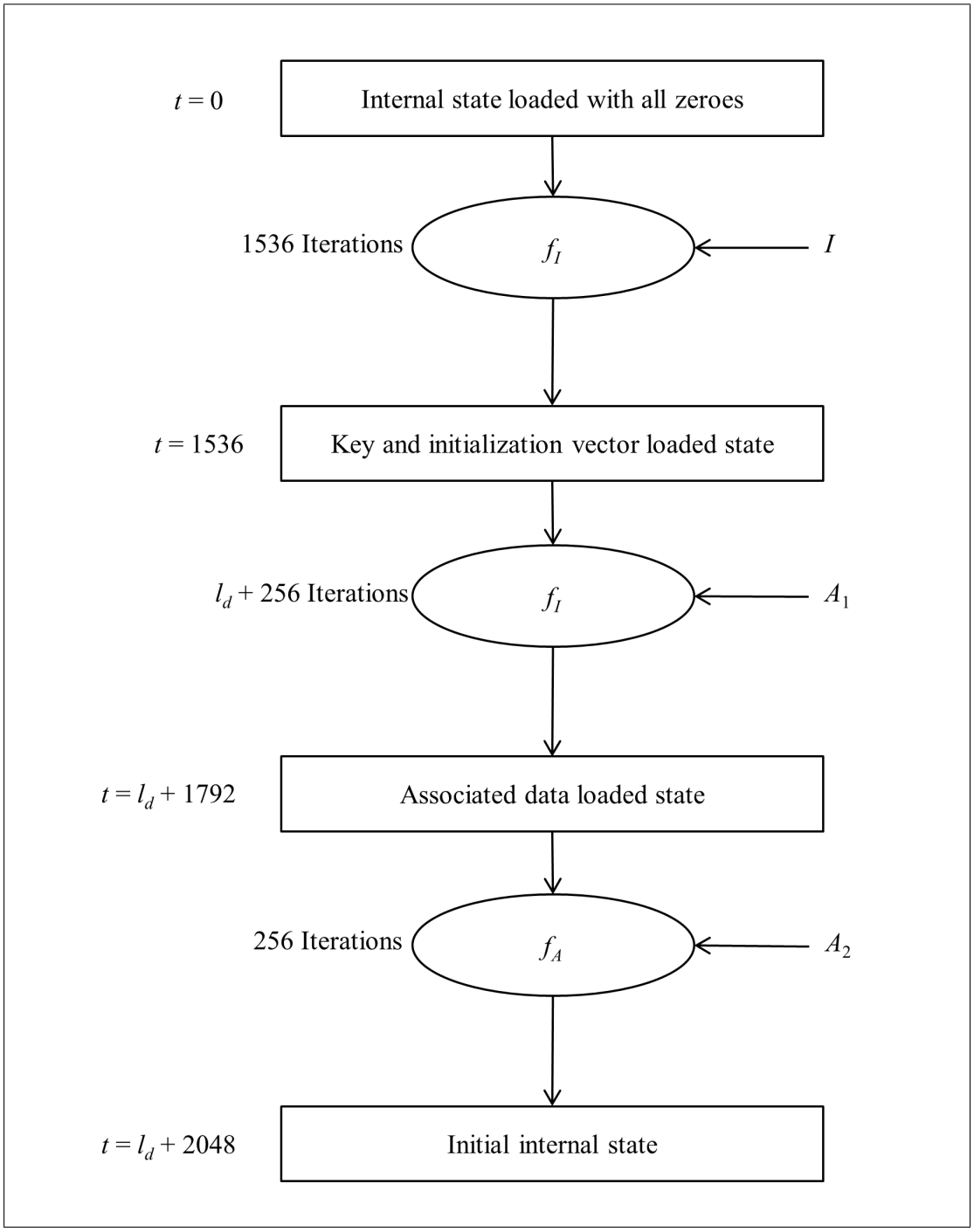

Figure 3.2: Initialization

Note that in the original description of ACORN the designer considered only loading of the key, initialization vector and the padding bits as part of the initialization phase. However, we consider the associated data loading process as part of the initialization phase, because no keystream bits are output until after

that process is complete.

3.2.2.1 Key and Initialization Vector Loading

To begin, the internal state of the cipher is loaded with all zero bits, i.e, $s_i = 0$ for $i = 0, \cdots, 292$. The cipher takes an input, I, of 1536 bits which consists of the concatenation of the 128-bit key, K, the 128-bit initialization vector, V, and 1280 bits of padding, π_I. The padding consists of one bit with value 1, followed by 1279 bits of 0.

$$\begin{aligned} I &= (k_0, \cdots, k_{127}||v_0, \cdots, v_{127}||\pi_{I0}, \pi_{I1}, \cdots, \pi_{I1279}) \\ &= (k_0, \cdots, k_{127}||v_0, \cdots, v_{127}||1, 0, 0, 0, \cdots, 0) \end{aligned} \tag{3.8}$$

During the key and initialization vector loading, both of the control bits are set to 1, i.e, $a = b = 1$. Therefore, during the key and initialization vector loading phase the nonlinear update function is the function f_I as per Table 3.1. The cipher is run for 1536 clocks. At each clock, the register stages are updated using the state update function given in Equation 3.7, with the input I used as the message M.

3.2.2.2 Associated Data Loading

After loading the key, initialization vector and subsequent padding, the cipher next processes the associated data. In this phase, the cipher takes an input vector A consisting of l_d bits of associated data, D, with 512 bits of padding, π_A. The first padding bit is set to one and the rest of the padding bits are set to zeroes, i.e, $\pi_A = 1, 0, 0, 0, \cdots, 0$.

$$\begin{aligned} A &= (d_0, \cdots, d_{l_d-1}||\pi_{A0}, \pi_{A1}, \cdots, \pi_{A511}) \\ &= (d_0, \cdots, d_{l_d-1}||1, 0, 0, 0, \cdots, 0) \end{aligned} \tag{3.9}$$

The cipher is run for l_d+512 clocks to update the state using the state update function. During associated data loading, the control bits a and b are set to 1, i.e, $a = b = 1$ for the first $l_d + 256$ clocks. Therefore, the feedback function f_I given in Table 3.1 will be used for the first l_d+256 steps of the associated data loading process with the input vector $A_1 = (d_0, \cdots, d_{l_d-1}||\pi_{A0}, \pi_{A1}, \cdots, \pi_{A255})$. For the remaining 256 iterations, control bit a is set to zero and b is set to one. That is,

the feedback function f_A given in Table 3.1 is used for the last 256 clocks of the associated data loading process with the input vector $A_2 = (\pi_{A256}, \cdots, \pi_{A511})$. Note that even if there is no associated data the cipher will still need to run for 512 clocks to process the padding bits.

3.2.3 Encryption

The encryption procedure takes the plaintext, P, as input and computes the output ciphertext, C. During the encryption phase, the control bits a and b are set to one and zero, respectively. Therefore, the nonlinear update function f_E given in Table 3.1 is used during the encryption phase. Figure 3.3 illustrates the encryption process of ACORN.

The l_p-bit plaintext message $P = p_0 p_1 \cdots p_{l_p-1}$ is loaded into the register stages using the state update function. In the state update function, the plaintext message P will be used as the input message M during the encryption phase. The output bits Z are used as keystream bits and the ciphertext, C is computed by XOR-ing these bits with the input message bits, P.

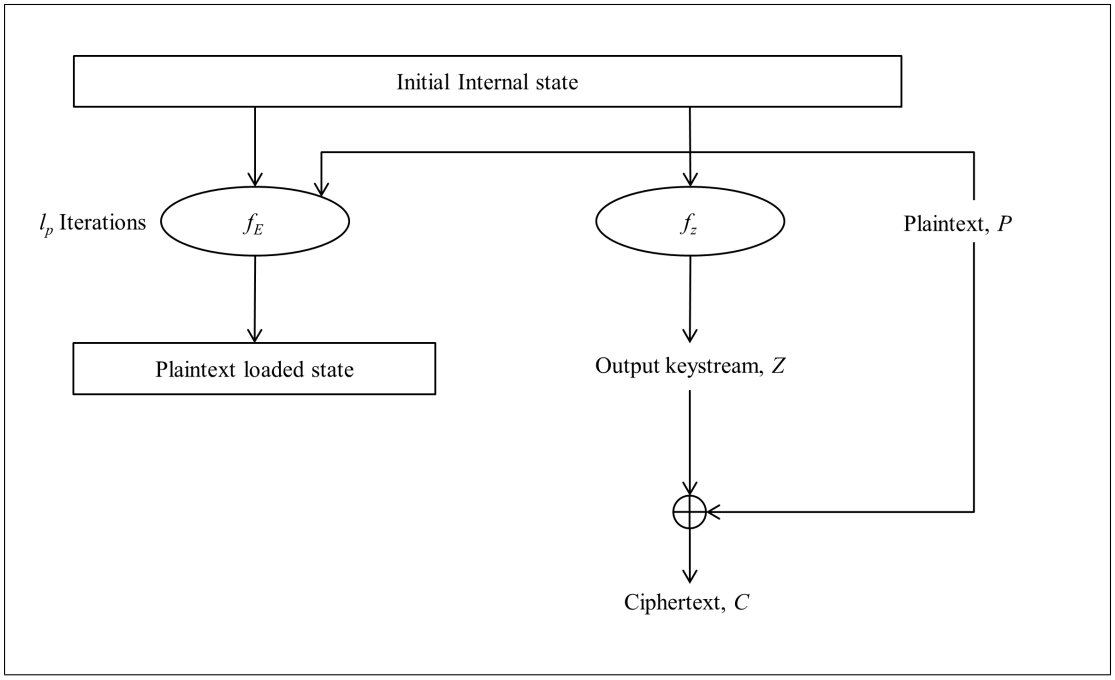

Figure 3.3: Encryption

3.2.4 Tag Generation

After all the plaintext has been encrypted, the cipher goes through some finalization steps to generate the authentication tag. During this procedure, it takes a 1024 bit input vector π_τ, consisting of one bit with value 1, followed by 1023 bits of value 0.

$$\begin{aligned}\pi_\tau &= (\pi_{\tau_0}, \pi_{\tau_1}, \cdots, \pi_{\tau_{1023}}) \\ &= (1, 0, 0, 0, \cdots, 0)\end{aligned} \tag{3.10}$$

This input vector is XOR-ed with the feedback bits to update the cipher. The cipher uses different feedback functions at different steps of this phase. For the first 256 clocks, the feedback function f_E given in Table 3.1 is used in the state update function. For the next 256 clocks, both of the control bits are set to zero and the function f_T given in Table 3.1 is used. For the final 512 clocks, the cipher is run with the feedback function f_I and for the last l_{tag} number of iterations the keystream bits Z are computed and used as the tag, where $64 \leq l_{tag} \leq 128$ is the length of the tag. The tag generation process for ACORN is shown in Figure 3.4.

3.2.5 Decryption and Tag Verification

To carry out the decryption, the cipher first performs the initialization process to obtain the initial state. During the actual decryption phase, the keystream generator generates a keystream bit Z^t at each clock, t. This bit is XOR-ed with the ciphertext bit C^t at that time instant to obtain the decrypted plaintext bit. The decrypted plaintext bit is then fed into the keystream generator as the input message M. This process is iterated till all the ciphertext bits are processed. After processing all the ciphertext bits the tag is generated at the receiver, following the same procedure as mentioned above. Finally for the verification, the tag generated at the receiver is matched with the received tag from the sender. Clearly, the verification succeeds if the plaintext sent by the sender is the same as the decrypted plaintext at the receiver.

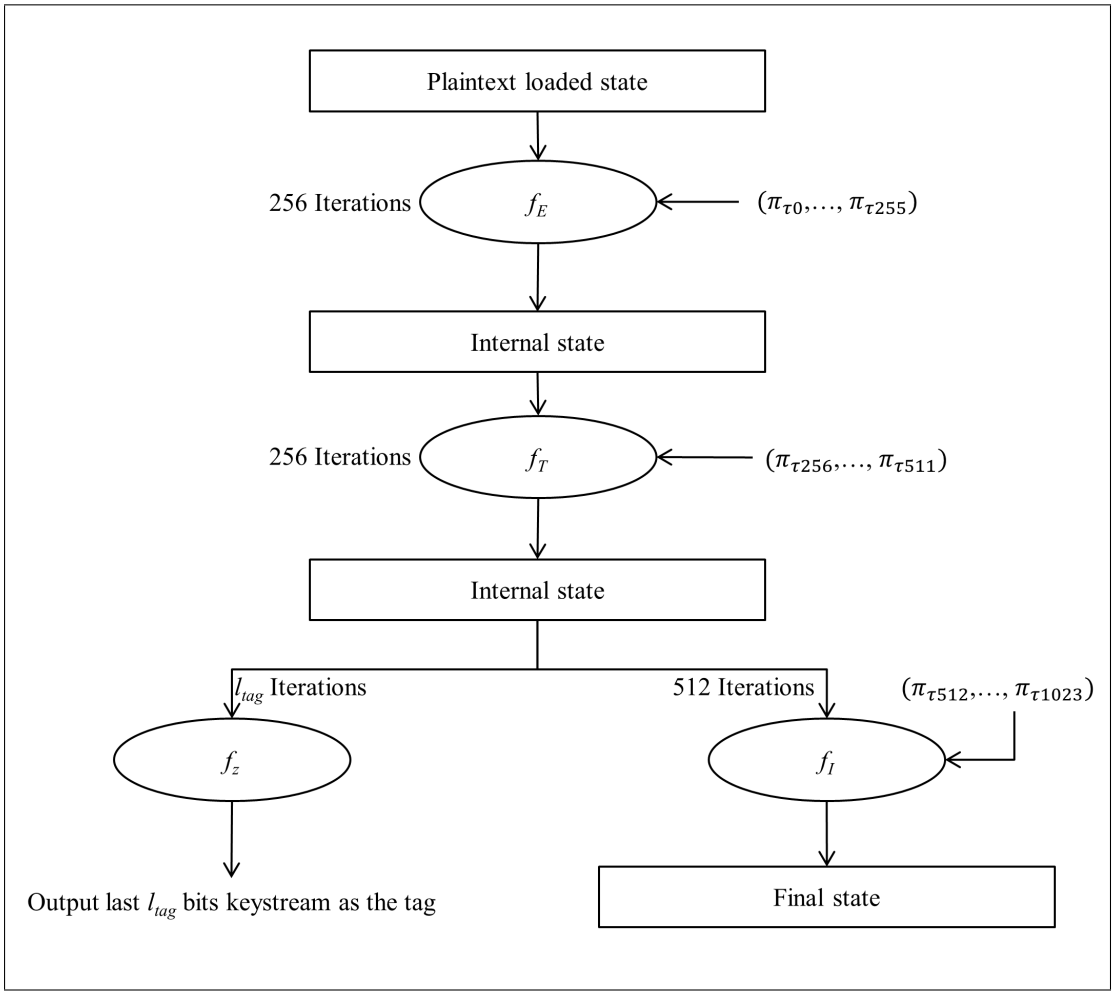

Figure 3.4: Tag Generation

3.2.6 Changes in ACORNv2

The initialization phases in ACORNv1 and ACORNv2 differ with respect to the feedback function used during the different rounds of the cipher, and also in the selection of the padding bits. The following changes are included in the second round submission ACORNv2.

3.2.6.1 Changes in the Initialization Phase

In ACORNv2, the number of steps in the key and initialization vector loading phase, associated data processing phase are changed from 1536, $l_d + 512$ to 1792, $l_d + 256$, respectively. That is, the input vector I in ACORNv2 consists of 1792 bits, including the 128-bit key, K, the 128-bit initialization vector, V, and 1536 bits of padding, π_I, as shown in Figure 3.5.

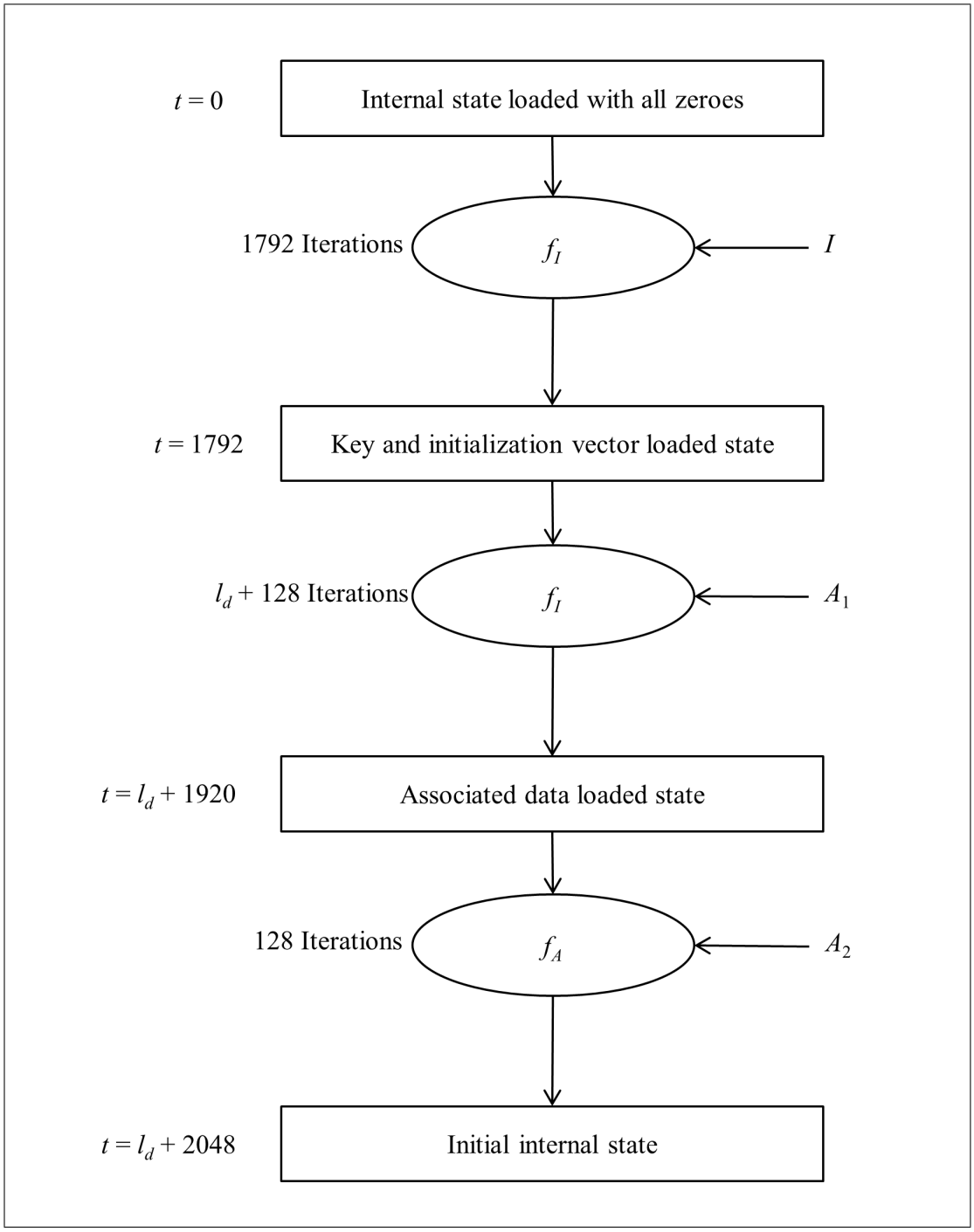

Figure 3.5: Initialization Phase of ACORNv2

For ACORNv1, the padding bits are constant whereas in ACORNv2 the padding bits are derived from the key bits. The method for incorporating the key bits in the padding bits of the initialization phase of ACORNv2 are defined

in Equation 3.11.

$$I^{t+256} = \begin{cases} k_{0 \bmod 128} \oplus 1 & \text{for } t = 0 \\ k_{t \bmod 128} & \text{for } t = 1, \cdots, 1535 \end{cases} \tag{3.11}$$

The input vector A in ACORNv2 consists of $l_d + 256$ bits, where l_d is the length of the associated data and the remaining 256 bits are padding bits π_A. The first padding bit is set to one and the remaining of the padding bits are set to zeroes, i.e, $\pi_A = 1, 0, 0, 0, \cdots, 0$. The cipher is run for $l_d + 256$ clocks to update the state using the state update function. During associated data loading, the control bits a and b are set to 1, i.e, $a = b = 1$ for the first $l_d + 128$ clocks. Therefore, feedback function f_I given in Table 3.1 will be used for the first $l_d + 128$ steps of the associated data loading process with the input vector $A_1 = (d_0, \cdots, d_{l_d-1}||\pi_{A0}, \pi_{A1}, \cdots, \pi_{A127})$. For the remaining 128 iterations, control bit a is set to zero and b is set to one. That is, the feedback function f_A given in Table 3.1 is used for the last 128 clocks of the associated data loading process with the input vector $A_2 = (\pi_{A128}, \cdots, \pi_{A255})$. Note that even if there is no associated data the cipher will still need to run for 256 clocks to process the padding bits. Figure 3.5 illustrates the initialization phase of ACORNv2.

Note that the tweaks made in ACORNv2 do not change the total number of iterations in the initialization phase of the two versions. Both versions of ACORN have an initialization phase of $l_d + 2048$ rounds.

3.2.6.2 Changes in the Tag Generation Phase

ACORNv2 also differs in the varying feedback functions used at specific times of the tag generation phase. For the first 128 clocks of ACORNv2, the feedback function f_E given in Table 3.1 is used in the state update function. For the next 128 clocks, both of the control bits are set to zero and the function f_T given in Table 3.1 is used. For the final 768 clocks, the cipher is run with the feedback function f_I and for the last l_{tag} number of iterations the keystream bits Z are computed and used as the tag. The changes in the varying feedback functions of ACORNv2 are illustrated in Figure 3.6.

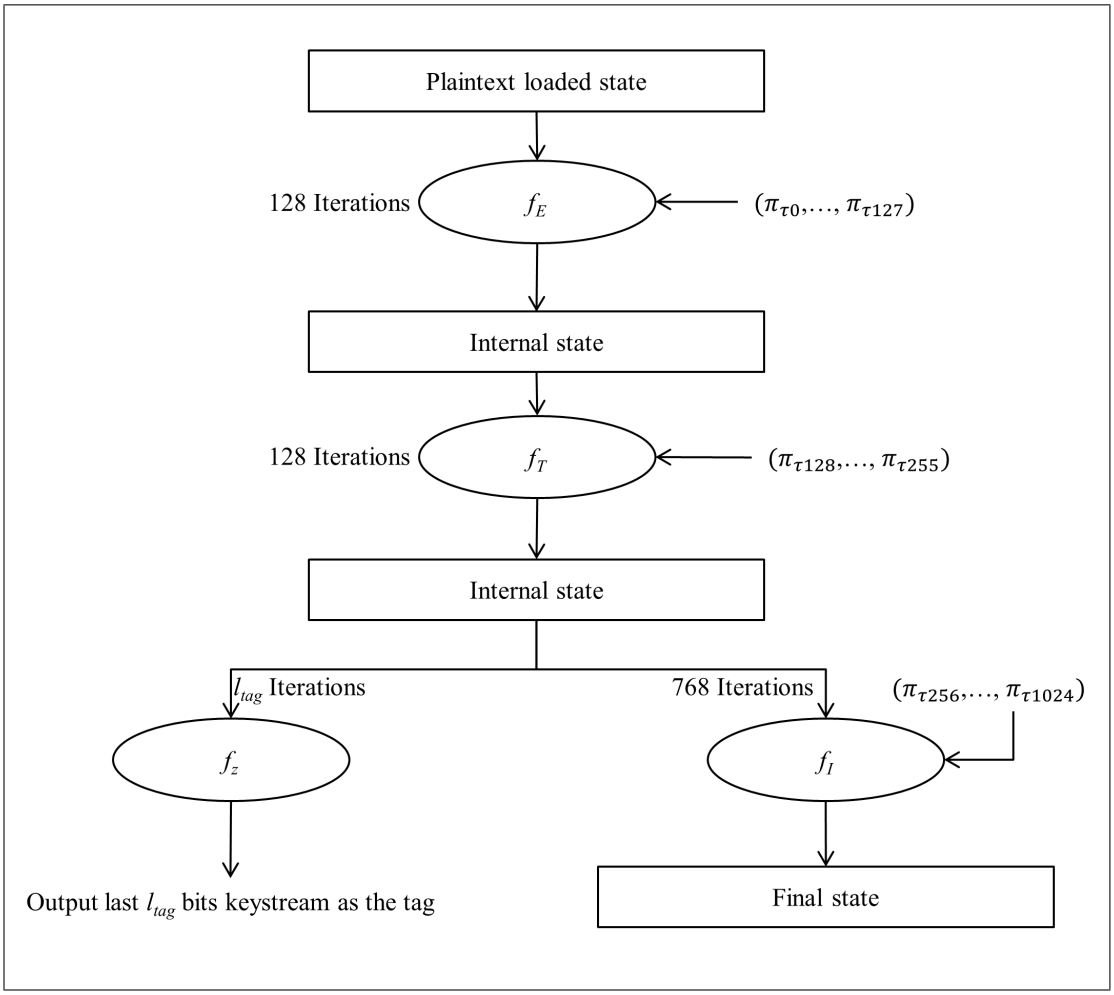

Figure 3.6: Tag Generation of ACORNv2

3.2.7 Changes in ACORNv3

ACORNv3 differs from ACORNv2 in its component function. In particular, changes are made in the output function and feedback function of ACORNv3. These changes are described below.

3.2.7.1 Changes in the Output Function

The output function of ACORNv3 has been tweaked by including the XOR of the choice function (defined in Equation 3.5) in addition to the existing terms (of Equation 3.1). Including the choice function in Equation 3.1, Wu's version of the output bit Y_{v3}^t at time t is defined as:

$$Y_{v3}^t = s_{12}^t \oplus s_{154}^t \oplus maj(s_{61}^t, s_{193}^t, s_{235}^t) \oplus ch(s_{230}^t, s_{111}^t, s_{66}^t) \tag{3.12}$$

Similar to ACORNv1, Wu's version of the output function of ACORNv3 also does not highlight the intermediate updates of register stage s_{61}, s_{154}, s_{193} and s_{230}. We substitute the updated values of stages s_{61}, s_{154} and s_{193} into Equation 3.12 and also use the definition of the majority function and choice function to obtain the following output function for ACORNv3:

$$\begin{aligned} Z^t_{v3} = {} & s^t_{12} \oplus s^t_{154} \oplus s^t_{111} \oplus s^t_{107} \oplus s^t_{61}s^t_{193} \oplus s^t_{61}s^t_{160} \oplus s^t_{61}s^t_{154} \oplus s^t_{23}s^t_{193} \oplus s^t_{23}s^t_{160} \\ & \oplus s^t_{23}s^t_{154} \oplus s^t_{0}s^t_{193} \oplus s^t_{0}s^t_{160} \oplus s^t_{0}s^t_{154} \oplus s^t_{193}s^t_{235} \oplus s^t_{160}s^t_{235} \oplus s^t_{154}s^t_{235} \oplus s^t_{61}s^t_{235} \\ & \oplus s^t_{23}s^t_{235} \oplus s^t_{0}s^t_{235} \oplus s^t_{230}s^t_{111} \oplus s^t_{196}s^t_{111} \oplus s^t_{193}s^t_{111} \oplus s^t_{230}s^t_{66} \oplus s^t_{196}s^t_{66} \\ & \oplus s^t_{193}s^t_{66} \oplus s^t_{66} \end{aligned} \tag{3.13}$$

In total, the output function Z^t_{v3} of ACORNv3 will take input from thirteen register stages: s^t_0, s^t_{12}, s^t_{23}, s^t_{61}, s^t_{66}, s^t_{107}, s^t_{111}, s^t_{154}, s^t_{160}, s^t_{193}, s^t_{196}, s^t_{230} and s^t_{235}.

3.2.7.2 Changes in the Feedback Function

The tweak made in the feedback function of ACORNv3 excludes the XOR of the choice function (defined in Equation 3.5) from this function. Removing the choice function in Equation 3.4, Wu's version of the feedback function $f'_{v3}(S^t, a^t, b^t)$ for ACORNv3 is defined as:

$$f'_{v3} = s^t_0 \oplus \overline{s^t_{107}} \oplus maj(s^t_{244}, s^t_{23}, s^t_{160}) \oplus (a^t s^t_{196}) \oplus (b^t Y^t) \tag{3.14}$$

where $\overline{s^t_{107}}$ denote the complement value of the content of register stage s^t_{107}.

Wu's version of the feedback function given in Equation 3.14 implicitly incorporates the intermediate updates of the register stages s_{61}, s_{107}, s_{154}, s_{193} and s_{230}. After substituting the updated values of these register stages and using algebraic expressions for the majority function and output function, we can express the generic form of the feedback function as follows:

$$\begin{aligned} f_{v3} = {} & 1 \oplus s^t_0 \oplus s^t_{107} \oplus s^t_{61} \oplus s^t_{66} \oplus s^t_{244}s^t_{23} \oplus s^t_{23}s^t_{160} \oplus s^t_{160}s^t_{244} \oplus a^t s^t_{196} \oplus b^t \big(s^t_{12} \\ & \oplus s^t_{154} \oplus s^t_{111} \oplus s_{107} \oplus s^t_{61}s^t_{193} \oplus s_{61}s^t_{160} \oplus s^t_{61}s^t_{154} \oplus s^t_{23}s^t_{193} \oplus s^t_{23}s^t_{160} \oplus s^t_{23}s^t_{154} \\ & \oplus s^t_0 s^t_{193} \oplus s^t_0 s^t_{160} \oplus s^t_0 s^t_{154} \oplus s^t_{193}s^t_{235} \oplus s^t_{160}s^t_{235} \oplus s^t_{154}s^t_{235} \oplus s^t_{61}s^t_{235} \oplus s^t_{23}s^t_{235} \\ & \oplus s^t_0 s^t_{235} \oplus s^t_{230}s^t_{111} \oplus s^t_{196}s^t_{111} \oplus s^t_{193}s^t_{111} \oplus s^t_{230}s^t_{66} \oplus s^t_{196}s^t_{66} \oplus s^t_{193}s^t_{66} \oplus s^t_{66}\big) \end{aligned} \tag{3.15}$$

Equation 3.15 is quadratic for given constant values of a and b. We fix the values of a and b to specify four possible feedback functions. We use the notation $f_{T_{v3}}$, $f_{A_{v3}}$, $f_{E_{v3}}$ and $f_{I_{v3}}$ to denote the specific feedback functions for the cases: $(a, b) = (0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$, respectively. These four feedback functions are presented explicitly in Table 3.2.

Table 3.2: Feedback Functions for Different Operation Phases of ACORNv3

a	b	f	Feedback Function
0	0	$f_{T_{v3}}$	$1 \oplus s_0^t \oplus s_{61}^t \oplus s_{66}^t \oplus s_{107}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right)$
0	1	$f_{A_{v3}}$	$1 \oplus s_0^t \oplus s_{12}^t \oplus s_{61}^t \oplus s_{111}^t \oplus s_{154}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right)$ $\oplus \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t\right) \left(s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right) \oplus s_{235}^t \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t \oplus s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right)$ $\oplus \left(s_{230}^t \oplus s_{196}^t \oplus s_{193}^t\right) \left(s_{111}^t \oplus s_{66}^t\right)$
1	0	$f_{E_{v3}}$	$1 \oplus s_0^t \oplus s_{61}^t \oplus s_{66}^t \oplus s_{107}^t \oplus s_{196}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right)$
1	1	$f_{I_{v3}}$	$1 \oplus s_0^t \oplus s_{12}^t \oplus s_{61}^t \oplus s_{111}^t \oplus s_{154}^t \oplus s_{196}^t \oplus s_{23}^t s_{244}^t \oplus s_{160}^t \left(s_{23}^t \oplus s_{244}^t\right) \oplus \left(s_{193}^t \oplus s_{196}^t \oplus s_{230}^t\right) \left(s_{66}^t \oplus s_{111}^t\right)$ $\oplus \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t\right) \left(s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right) \oplus s_{235}^t \left(s_0^t \oplus s_{23}^t \oplus s_{61}^t \oplus s_{154}^t \oplus s_{160}^t \oplus s_{193}^t\right)$ $\oplus \left(s_{230}^t \oplus s_{196}^t \oplus s_{193}^t\right) \left(s_{111}^t \oplus s_{66}^t\right)$

3.3 Existing Analysis of ACORN

In this section we discuss the existing public analysis of ACORN. Existing work on ACORN includes the analysis of slid pairs, key recovery or state recovery attack under the nonce reuse scenario, SAT based cryptanalysis, cube attacks, fault attacks and finding linear approximations. These are briefly discussed below.

3.3.1 Slid Pairs in ACORN

Liu and Lin [122] analyse the existence of slid pairs in ACORNv1, i.e., key-IV pairs that generate the same state, up to a difference of 37 clocks. Their analysis of slid pair properties for ACORN shows that there always exist two distinct key-initialization vector pairs which generate slid pairs for ACORN. However, the analysis also reveals that without knowing the exact key it is not feasible to compute the exact relations for such slid pairs.

Based on the existence of the slid pairs, they also explore possible state recovery attacks using guess-and-determine and differential algebraic techniques. The attacks described in their paper have worse complexity than exhaustive key search.

3.3.2 Nonce Reuse/ Decryption Misuse Key Recovery Attack on ACORN

Chaigneau et al. [123] show that a key recovery attack can be applied to ACORNv1 under the nonce-reuse and decryption-misuse settings. Their analysis can recover the internal state under the nonce-reuse scenario with a relatively low data complexity. According to their analysis, to recover the internal state with a complexity lower than the exhaustive search an adversary needs to encrypt three or more plaintext messages using the same key and initialization vector. For ACORNv1, state recovery also means the secret key recovery, since the adversary can reverse back to the loading state from the recovered state. However, as the designer specifically prohibits the nonce-reuse and decryption-misuse settings, this does not threaten the security claim of ACORNv1.

3.3.3 SAT based Cryptanalysis of ACORN

Lafitte et al. [124] explored the SAT based cryptanalysis of ACORNv1 and ACORNv2, aiming to achieve state recovery, key recovery, state collisions and forgery attacks. Their state recovery attack requires the knowledge of more than 260 bits of internal states. Their analysis of SAT based key recovery claims that for a given internal state an adversary can recover the secret key with probability 1, for both versions ACORNv1 and ACORNv2. This however does not threaten the security claim of ACORN since the state recovery attack requires the knowledge of more than 260 of the internal state bits, which is higher than the exhaustive search on a 128-bit key.

Lafitte et al. [124] also generalise the results of our work described in Section 3.4 to find state collisions in ACORN. This includes finding collisions for several internal states which lead to the same tag value. Similar to the work presented in this thesis, their finding on state collisions and state collisions based forgery also requires the knowledge of the secret key/ internal state.

Later, Dwivedi et al. [125] also provided some observations on the SAT based cryptanalysis of ACORN. Their analysis of SAT based state recovery requires the knowledge of 170 internal state bits. From these analyses, it seems that SAT based cryptanalysis does not threaten the security claim of ACORN.

3.3.4 Cube Attack on ACORN

Todo et al. [126] developed a cube attack on ACORN based on the division property (a technique to find higher order differential characteristics). This analysis considers the underlying cryptographic polynomial as a non-blackbox polynomial, whereas the original cube attack is performed by considering the underlying cryptographic polynomial as a blackbox polynomial. This cube attack on ACORN (with a cube size of 64) can recover the superpoly of a cube up to 704 rounds of the initialization phase with a complexity of 2^{122}.

3.3.5 Fault Attack on ACORN

Dey et al. [127] proposed a single bit fault based key recovery attack on the first two versions: ACORNv1 and ACORNv2. Their proposed attack is based on hard fault injection in the internal state and works under the nonce-respecting scenario. This requires to inject only a single bit fault at a random location of the internal state. The best result described in this attack claims to recover the secret key with a complexity of $2^{55.85}$.

3.3.6 Linear Relations between Message and Ciphertext Bits

Roy et al. [128] analysed the probabilistic linear relations among the input message and output ciphertext bits of ACORN. The analysis claims there exist linear relations between the message and ciphertext which holds with a probability greater than half.

Roy et al. [128] also described a key recovery attack on ACORN by generating and solving a system of quadratic equations. This analysis is based on the observation that the degree of the output polynomial of ACORN does not increase, if the contents of last register stage s_{292} remains zero for all the clockings of the encryption phase. Based on this assumption, adversary can generate and solve the generated system of equations, which requires a complexity of about 2^{40}. The authors stated that the contents of last register stage s_{292} can be controlled with the external input M; however, without knowing the exact value of the feedback function this will not be possible to achieve in practice. On the other hand if the adversary requires to guess the values of the feedback function

for all of the generated equations, then the complexity of the attack goes far beyond the exhaustive search on the key space of ACORN.

Jiao et al. [129] also presented a state recovery attack on ACORN to assess the security margin of the cipher. Their proposed method focuses on finding linear approximations of the output function of ACORN. Their observation shows that the majority function used in ACORN can be linearly approximated with a high probability.

Jiao et al. [129] also described a guess and determine state recovery attack on ACORN. This attack uses the linear approximation of the output function of ACORN with a combination of some guessed bits in the internal state. In particular, their attack obtains 265 linear approximation equations and guessed 28 bits of the internal state. With this, their analysis requires 2^{157} tests to find the right solution for the state recovery.

3.4 State Convergence and Collisions in ACORN

In this section we make observations on the ACORNv1 design, related to the possibility of state convergence and state collision in the internal state. Our motivation for analysing this in ACORNv1 is to find different inputs which generate the same tag, and hence could be used to facilitate forgery attacks.

We consider state convergence for the situation in which the keystream generator has no external input or in which the external input is fixed (an attacker therefore has no opportunity to influence the values of M). Convergence occurs when two or more distinct states at a given time are mapped into the same state after α iterations, for some $\alpha > 0$. In the operation of a general cipher, state convergence can occur during any phase, i.e., initialization, encryption or the tag generation, if the state update function is not one-to-one.

State collisions occur when two different sets of inputs produce identical internal states at some point of operation of the cipher. Here, the input combination implies different possibilities for key, initialization vector and external input message combination. State collisions may be exploited in a forgery attack or used in secret key recovery attacks.

Both state convergence and state collision result in identical internal states at some point in the operation of a cipher. The difference is that state convergence occurs when the cipher runs autonomously (or with fixed input) whereas state

collisions can be engineered by manipulating the external inputs to the cipher. ACORN does not have any external variable input messages during the tag generation phase, so state collisions cannot be forced in this phase of the cipher operation. However, state collisions might be obtained in either the initialization or encryption phase for different combinations of key, initialization vector and input message. This will then lead to the formation of identical message authentication tags. That is, if two sets of inputs K, V, M and K', V', M' result in colliding states, then the generated tag after the final phase will be the same, given that the rest of the input bits are the same once the colliding states are obtained.

A forgery attack can be applied to the ACORN message authentication algorithm if two input combinations which result in the same tag value can be identified. This is particularly useful for the scenario where $K = K'$, $V = V'$ but $M \neq M'$. That is, for a given key and initialization vector, an attacker can find two distinct input messages which will result in the same tag. A malicious sender can manipulate either the associated data or the plaintext messages to obtain the collision of the tag and therefore can frame a forged message that is accepted as legitimate.

3.4.1 Convergence of Two Different States

We explore the possibility of state convergence in ACORN by considering the effects of the state update function (Equation 3.7 and Figure 3.1) on the register contents. Note firstly that the contents of stages s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230} and s_{289} are the only values which do not appear directly in the updated state. Therefore two states which differ in the contents of any other stage cannot clock to the same state at the next time instant. We therefore focus our search on sets of states which differ only in the contents of (some or all of) the above stages.

Consider two distinct states S and S' from such a set. From Equation 3.7, it is clear that the updated versions of these states will differ in at least one of the stages s_{60}, s_{106}, s_{153}, s_{192}, s_{229} and s_{288} unless the values in all of the stages s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230} and s_{289} in S' are the complements of the corresponding values in S. However, even in this case, state convergence only occurs if both states lead to the same value of stage s_{292} at the next time step, that is, if the feedback function $f(S^t, a^t, b^t)$ gives the same output for both states.

Now ACORN uses different feedback functions f_I, f_A, f_E and f_T at various

Table 3.3: Different Inputs Resulting in A State Collision

State	Contents of register stages	Input
S	s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230}, s_{289}	M
S'	$\overline{s_0}$, $\overline{s_{61}}$, $\overline{s_{107}}$, $\overline{s_{154}}$, $\overline{s_{193}}$, $\overline{s_{230}}$, $\overline{s_{289}}$	$\overline{M}$

points during the different operation phases of the cipher, but in all cases the register stages s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230} and s_{289} occur in an odd number of linear terms and occur in quadratic terms only as canceling pairs (e.g., $s_{230}s_{111} \oplus s_{193}s_{111}$). Therefore the sum of quadratic terms is the same for both S and S' but the sums of linear terms differ, so the combined output of the feedback function (and hence the updated value of s_{292}) differs between these two states. Hence the state update function is one-to-one and state convergence is not possible during any of the operation phases of ACORN.

3.4.2 Collision of States from Different Inputs

During the initialization and encryption phases, external input messages are fed into the internal state of the keystream generator. During the initialization phase the inputs to the keystream generator are the key, initialization vector and associated data, whereas during the encryption phase the external input is the plaintext. In this section we discuss the possibility of obtaining state collisions by exploiting these external messages during the initialization and encryption phase of ACORN.

In Section 3.4.1 we noted that two ACORN states which differ only in the contents of register stages s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230} and s_{289}, will have identical register stages in the next clock for all stages except stage s_{292}. Note that the content of register stage s_{292} can be influenced at certain times by the external input M. We can therefore force a state collision by choosing the appropriate external input. For example, consider two different states S and S', which differ only in the register stages shown in Table 3.3.

Here, $\overline{s_i}$ corresponds to the complement of s_i, and $\overline{M}$ denote the complement value of input M. Notice that the two states described in Table 3.3 will collide in the next clock, if the next input bit M is complemented for the corresponding time instant. So, there will be $2^7 = 128$ possible combinations of values for the seven register stages shown in Table 3.3. We can group these into 64 complementary pairs. For any combination of values in these stages and either choice

for the external message bit, a collision can be obtained by complementing all of these stages and also complementing the external input message bit. In practice, this means if we take any state and any value for the next bit of input data, that combination will collide with the state that is identical apart from having the contents of register stages s_0, s_{61}, s_{107}, s_{154}, s_{193}, s_{230} and s_{289} complemented and also using the complementary value for the next bit of input data.

Now consider the different phases of ACORN to determine when we can obtain a state collision with chosen set of external input data. At the beginning, the 128-bit secret key and 128-bit public initialization vector are the input to the cipher for the key and initialization vector loading process of the initialization phase. The cipher will be clocked 256 times to load this key and initialization vector into the register stages. To manipulate the state collision with the selected input value, i.e., using chosen key and initialization vector for this instance, two states need to have complementary values in the above mentioned seven register stages. However, this will not happen after 256 clocks since the key-initialization vector mapping will reach up to register stage s_{37} and the rest of the register stages $s_0, \cdots, s_{36}$ still contain zero values. To force the collision, the two states need to have complementary values in s_0 which is not the case. Therefore, collisions can not be forced during the key and initialization vector loading phase by the above mentioned process.

During the associated data loading process in the initialization phase, the feedback function f_I is used to load the l_d bits of associated data into the register stages of the cipher. If $l_d + 256 > 293$ then the input to the cipher in bits is greater than the total internal state size. Therefore, if the associated data length is more than 37 bits, i.e, $l_d > 37$, then there must exist state collisions. At the beginning of the associated data loading process, the internal state consists of the key and initialization vector loaded state and the input to the cipher is the publicly known associated data. The key, initialization vector and associated data combination will be mapped into all the 293 register stages when there are more than 37 bits of associated data. Alternatively, if we consider a fixed secret key while varying both the initialization vector and associated data to obtain a state collision, then we need to have the associated data length, $l_d > 165$, to map the initialization vector and the associated data into all the 293 register stages.. At this point any two states which are identical with the exception of values in these specific seven register stages (which are complemented) will collide at

the next time step if they are fed with complementary values of associated data in the corresponding time instant. Therefore, a collision can be forced during the associated data loading process. Similar comments apply to the encryption phase of the cipher where one can obtain a state collision by manipulating the plaintext messages.

3.4.3 Finding State Collisions in ACORN

In this section, we describe the procedure for finding state collisions in ACORN. We also discuss the feasibility of this approach.

To find the state collisions, a set of quadratic equations is defined, relating the contents of two ACORN states formed from two different input messages. We can solve this system of equations to obtain pairs of distinct input messages which generate the colliding states.

3.4.3.1 Equation Generation

Let S^t denote an internal state of ACORN at time t with an input message M. Our goal is to construct another input message M' for an internal state S'^t in such a way that $M \neq M'$ and it results in identical state after processing the input messages. The input messages M and M' can represent either the associated data or the plaintext.

We first define the contents of the internal state S^t with 293 binary variables, i.e., $S^t = (s_0^t, \cdots, s_{292}^t)$. Similarly, the contents of the internal state $S'^t = (s_0'^t, \cdots, s_{292}'^t)$. At this stage, we are not assuming that $S^t = S'^t$. To find the colliding states, we construct a set of equations by relating the contents of S^{t+293} and S'^{t+293}, formed from the input message M and M', respectively. For an input message of length 293 bits, processing the message results in changes in all the register stages.

While the input message is being loaded, the degree of the generated equation system increases because of the nonlinear feedback function. To keep the degree of the nonlinear feedback function in quadratic form we apply the relabeling technique [143] at each iteration of the message loading. Using the relabelling technique, the quadratic feedback function is replaced with a new variable at each time step. Therefore after 293 bit message processing the relabelling technique will provide 293 equations, and the output function will provide another 293

equations. In total, this process introduces 586 new variables and 586 new equations in the equation system. Also, 586 more variables are required to represent the input messages to the internal state S^t and S'^t.

After loading the 293-bit message, the two ACORN internal states are defined as: $S^{t+293} = (s_0^{t+293}, \cdots, s_{292}^{t+293})$ and $S'^{t+293} = (s_0'^{t+293}, \cdots, s_{292}'^{t+293})$. We obtain 293 equations which equate the content of each register stage in S^{t+293} and S'^{t+293} except for register stages numbered 0, 61, 107, 154, 193, 230 and 289. Equations for these remaining seven pairs of register stages are generated in a way that the corresponding register stages in S^{t+293} and S'^{t+293} contain complementary values. Thus after 293 clocks of message loading phase, the equations comparing the internal state S^{t+293} and S'^{t+293} are given by Equation 3.16.

$$s_i^t \oplus s_i'^t = \begin{cases} 1 & i = 0, 61, 107, 154, 193, 230, 289 \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

where $0 \leq i \leq 292$. In total there are 879 equations with 1758 variables. Obtaining solutions for M and M' by solving these system of equations means we can obtain all possible combinations of states and input messages which differ only by complementing the specific register stages. If a solution for this system of equations can be obtained, then a collision can be forced by choosing complementary values for M and M' at the 294th clock of the associated data loading phase or plaintext loading phase.

3.4.3.2 Solving the Equations

In this section we discuss how to solve the generated system of equations to obtain inputs which demonstrate state collision in ACORN. Experiments to obtain collision examples are performed using Sage version 6.4.1 [144] on a standard 3.4GHz Intel core i7 PC with 16GB memory.

The generated equation system as described in the previous section is underdefined and the number of equations in the system is quite large. A large portion of the variables are due to the unknowns in the internal state S^t and S'^t. So, we reduced the size of the system of equations by fixing the contents in S^t and S'^t instead of defining them with unknown variables. More precisely the assumption here is that the internal states S^t and S'^t are known. The key and initialization vector loaded state are assumed to be known if associated data is used as the input message to generate the collision. The initial state is

assumed to be known if plaintext is used as the input message to generate the collision. Also, we assume that the input message M for state S^t is known. Only the input message M' for state S'^t is represented with 293 unknown variables. These assumptions reduce the size of the equation system to 586 equations with 586 variables. These equations are solved using Gröbner basis methods [145]. Specifically an implementation of Buchberger's algorithm in Sage [144] was used to compute the set of solutions. It first performs a monomial ordering and then compares two polynomial from the equation system, and eliminates the leading terms by applying appropriate coefficients. This process is continued until an univariate polynomial equation is generated, followed by solving the generated equation. This process is repeated for the entire equation system. Solving this equation system gives a set of solutions for the unknown variables that constitute the bits of the input message M'. Note that solutions may not necessary exist or be unique. In our application, this is verified experimentally to determine whether solutions can be obtained to achieve state collisions in ACORN.

3.4.4 Demonstrating State Collisions in ACORN

We performed experiments using the techniques described in Section 3.4.3 to demonstrate state collisions in ACORN. The details of these experimental results are described below.

3.4.4.1 Collisions for Different Associated Data Sets

This section demonstrates state collisions in ACORN for two distinct inputs of associated data. We used the technique described in the previous section to find two different sets of associated data which generate identical associated data loaded states.

In ACORN, associated data are fed into the key and initialization vector loaded state after $t = 1536$ iterations of the key and initialization vector loading phase. Let the key and initialization vector loaded states S^{1536} and S'^{1536} be formed using the key and initialization vector pair (K, V) and (K', V'), respectively. Suppose first that these two states S^{1536} and S'^{1536} are identical, i.e., $K = K'$ and $V = V'$. At this point, associated data D and D' are fed into S^{1536} and S'^{1536}, respectively. At each iteration of the associated data loading, a set of equations is generated relating the input associated data D and D' with the contents of the corresponding internal state. To keep the size of the equation system

Table 3.4: Example of Collision for Same K, V, with Different D but Dame P

K	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D	**ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 3f**
P	81 01 01 01 01 01 01 01 01 81
C	a6 46 3a 55 73 f6 13 bb dd 7f
τ	c7 a8 06 c1 5a d1 40 50 62 59 7b 47 63 51 18 57
K'	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V'	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D'	**fe ff ff ff 19 35 bd 53 37 b8 eb 8c cd c5 bc a0 57 6b 21 fd dd 82 47 71 f9 46 f5 b6 21 72 7d fe 84 2c 9a 4a 1a**
P'	81 01 01 01 01 01 01 01 01 81
C'	a6 46 3a 55 73 f6 13 bb dd 7f
τ'	c7 a8 06 c1 5a d1 40 50 62 59 7b 47 63 51 18 57

small, we assume that the key and IV loaded states S^{1536} and S'^{1536} are known. For a chosen associated data D, calculations are then performed to obtain the associated data D' generating state S^{1829} and S'^{1829}, respectively, which differ only in the register stages numbered 0, 61, 107, 154, 193, 230 and 289. A state collision is then forced at the 294th clock by choosing complementary values of the input associated data bits D and D' for that time instant. This gives two identical associated data loaded state $S^{1830} = S'^{1830}$, for two distinct associated data sets D and D', respectively.

Table 3.4 gives an example of a pair of input data sets which leads to a collision for the same key and initialization vector pair with different associated data. All tabulated values are in hexadecimal notation. For all the examples presented in this section, the inputs are processed for each byte of data and fed into the keystream generator starting from the least significant bit of each data byte.

As shown in Table 3.4, except for the associated data all the other inputs are same, i.e., $K = K'$, $V = V'$ and $P = P'$. The input associated data D in the above example is changed to D' which results in a collided associated data loaded state after 294 iterations. Since the inputs to the states are kept the same once the identical associated data loaded states are obtained, it will result in the same ciphertext $C = C'$ and tag $\tau = \tau'$ after processing the rest of the inputs.

The same procedure can be also applied to the case of different key and initialization vector loaded state, i.e., $K \neq K'$ and $V \neq V'$, and therefore $S^{1536} \neq S'^{1536}$. Table 3.5 gives an example where $S^{1536} \neq S'^{1536}$, having two different associated data inputs D and D', respectively, which generate two identical associated data loaded states.

The above mentioned examples show that given the secret key, public initialization vector and at least 294 bits (37 bytes) of input associated data, a state collision can be forced against the authenticated encryption cipher ACORN. In

Table 3.5: Example of Collision for Different K, V and D with Same P

K	**12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1**
V	**ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff**
D	**ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 3f**
P	81 01 01 01 01 01 01 01 01 81
C	a6 46 3a 55 73 f6 13 bb dd 7f
τ	c7 a8 06 c1 5a d1 40 50 62 59 7b 47 63 51 18 57
K'	**10 20 36 40 9a bc de f0 10 35 59 78 9a be dd f0**
V'	**f0 f1 f0 f1 f0 f1 f0 f1 f0 f1 f0 f1 f0 f1 f0 f1**
D'	**f2 f1 c5 2f a3 cc 25 e5 d4 74 be 3b 54 8f d1 30 9d e2 6e ce e7 81 c9 ab 8f e3 db 0f c4 89 da ea 11 84 c0 28 18**
P'	81 01 01 01 01 01 01 01 01 81
C'	a6 46 3a 55 73 f6 13 bb dd 7f
τ'	c7 a8 06 c1 5a d1 40 50 62 59 7b 47 63 51 18 57

practice a malicious sender who knows the secret key can find two distinct associated data strings which generate the same tag and therefore can break the integrity component of the cipher. The associated data usually consists of information which is publicly available such as source address and destination address. In this case, the sender can dispute the validity of the source or destination of the message by replacing the original associated data with the forged data. A similar forgery on ACORN was later reported by Lafitte et al. [124]. We note that this forgery attack does not refute the security claim of the designer; however, it is important in developing an understanding of the internal workings of ACORN as a stepping stone to developing attacks with unknown state.

Note that earlier we stated that there must exist state collision if the associated data length is more than 37 bits. However for our experiments we have used 294 bits of associated data to find the collision. This is because to find the collision after 37 bits of associated data loading, we need to manipulate the key and initialization vector with the 37 bits of associated data. In that case, the experiments need to go through 1536 clocks of key and initialization vector loading phase which is infeasible. Therefore, we have used 294 bits of associated data to obtain state collisions in ACORN.

3.4.4.2 Collisions for Different Plaintexts

In this section we demonstrate collisions in the ACORN internal state for two distinct plaintext messages. The same technique used to obtain the collision during the associated data loading process can be also applied in the encryption phase to find a pair of input plaintext messages which result in identical states. The resultant ciphertexts will be different for this case but the generated tag after the final phase will be the same given that rest of the input plaintexts (if any) are the same once the colliding plaintext loaded states are obtained.

Plaintext bits are fed into the initial state of the keystream generator after

Table 3.6: Example of Collision for Same K, V and D with Different P

K	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 3f
P	**51 75 65 65 6e 73 6c 61 6e 64 20 55 6e 69 76 65 72 73 69 74 79 20 6f 66 20 54 65 63 68 6e 6f 6c 6f 67 79 21 21**
C	**76 32 5e 31 1c 84 7e 9b e1 9a 9e ac 7f c9 01 fd 10 8c d7 88 ea 16 f3 72 bc 6d cb 9f bc 1e 96 58 02 bb 11 9a 51**
τ	b5 83 b2 04 73 a5 35 e0 b4 df 98 1b 1f 40 08 9a
K'	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V'	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D'	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 3f
P'	**50 75 65 65 88 b9 2c 49 a6 23 f0 bb 7c 31 35 3d 85 8c 30 c5 f2 5d c5 d3 e9 49 cd c8 9a 4b 06 43 42 5e 20 3b 05**
C'	**77 32 5e 31 fa 4e 3e b7 29 dd 4e 52 6d 90 42 a5 e4 4e 86 6a 45 6b d1 b6 56 04 e3 a2 32 ba 9f 61 73 82 46 a1 75**
τ'	b5 83 b2 04 73 a5 35 e0 b4 df 98 1b 1f 40 08 9a

$t = l_d + 2048$ iterations of initialization phase. The initial states S^{l_d+2048} and S'^{l_d+2048} are formed using the key, initialization vector and associated data pair (K, V, D) and (K', V', D'), respectively. Suppose firstly that these two initial states S^{l_d+2048} and S'^{l_d+2048} are identical, i.e., $K = K'$, $V = V'$ and $D = D'$. Plaintexts P and P' are fed to the initial state S^{l_d+2048} and S'^{l_d+2048}, respectively. At each iteration of the plaintext loading, equations are generated relating the input plaintext P and P' with the contents of the corresponding states. To reduce the size of the equation system, we assume that the initial states S^{l_d+2048} and S'^{l_d+2048} are known. For a chosen input plaintext, P, calculations are then performed to determine the other plaintext, P' which will lead to the colliding states. To obtain collisions in the internal state the plaintext needs to be at least 294 bits since we need to map the plaintext inputs into all the internal state bits of the keystream generator. For the first 293 iterations of the encryption phase the input plaintexts are used to obtain two different states S^{l_d+2341} and S'^{l_d+2341} which differ only in the register stages numbered 0, 61, 107, 154, 193, 230 and 289. The collision is forced at the 294th iteration by having complementary values of input plaintexts P and P' for that time instant. This gives us two identical plaintext loaded states $S^{l_d+2342} = S'^{l_d+2342}$, for two distinct plaintext messages P and P', respectively.

Table 3.6 shows an example of obtaining state collision for two different input of plaintexts, P and P', with the same key $K = K'$, initialization vector $V = V'$ and associated data $D = D'$. As shown in the example, the resultant ciphertexts C and C' are different because the input plaintexts P and P' are different, however the tags τ and τ' generated after the final phase are the same, since no additional plaintexts are fed into the input after the colliding plaintext loaded states are obtained.

For the example shown in Table 3.6, the two initial state pairs of ACORN are the same before the encryption phase. This is because for both pairs of

Table 3.7: Example of Collision for Same K, V with Different D and P

K	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D	**ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 3f**
P	**51 75 65 65 6e 73 6c 61 6e 64 20 55 6e 69 76 65 72 73 69 74 79 20 6f 66 20 54 65 63 68 6e 6f 6c 6f 67 79 21 21**
C	**76 32 5e 31 1c 84 7e 9b e1 9a 9e ac 7f c9 01 fd 10 8c d7 88 ea 16 f3 72 bc 6d cb 9f bc 1e 96 58 02 bb 11 9a 51**
τ	b5 83 b2 04 73 a5 35 e0 b4 df 98 1b 1f 40 08 9a
K'	12 24 36 48 9a bc de f0 10 35 59 78 9b bc de f1
V'	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
D'	**aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa 2a**
P'	**62 af ff 7d fd 60 5b 3c f5 d7 73 72 50 8a 3e 10 a1 09 3f 05 61 a2 b7 a9 2e aa 56 90 63 8f 23 d0 7b ac ce d3 1a**
C'	**52 c5 0b 94 04 05 b2 06 39 0a 63 20 a7 49 5e 9c 60 9e 39 8c 06 b5 1d cf 18 5c 75 d1 4c c7 47 59 b5 84 64 a8 6a**
τ'	b5 83 b2 04 73 a5 35 e0 b4 df 98 1b 1f 40 08 9a

internal state S^{l_d+2048} and S'^{l_d+2048}, we considered the same key, initialization vector and associated data input. Note that a collision can be still obtained in the encryption phase by using appropriate plaintext inputs even if any of these inputs are different, e.g., $K \neq K'$, $V \neq V'$ or $D \neq D'$, during the initialization phase. Table 3.7 shows such an example where the initial states are different, i.e., $S^{l_d+2048} \neq S'^{l_d+2048}$ and two collided states are obtained by manipulating the plaintexts.

As shown in Table 3.7, the key and initialization vector are the same for both states, i.e., $K = K'$ and $V = V'$. On the other hand both the associated data and the plaintext are different for both states S^t and S'^t, i.e., $D \neq D'$ and $P \neq P'$, and the state collision is forced by manipulating the plaintext. This means anyone with knowledge of the secret key, e.g., a malicious sender, can forge both the associated data and the plaintext. For this instance, the changes in the associated data can be made arbitrarily, whereas changes in the plaintext are determined by the equations that must be satisfied in order to obtain a collision. In this case, a malicious sender can apply a selective forgery attack on the associated data as he/she can select the associated data of his/her own choice.

3.4.4.3 Collisions for More than 294 Clocks

In this section we briefly discuss state collisions in ACORN if we manipulate more than 294 bits of input message. The examples shown in the previous sections obtain state collisions after exactly 294 iterations of the associated data or plaintext loading phase. Experiments were also performed to obtain a collision by manipulating more than 294 bits of input messages.

As the number of iterations were increased the time to obtain the solutions increases. This is because when we increase the number of clocks there are more free variables in the equation system and the time complexity is expected to

grow exponentially. However, the number of alternative inputs that can be used to obtain a collision doubles with the increase of each clock. Table 3.8 lists the required CPU time and memory for solving the equation system with 294 or more iterations of input message loading.

Table 3.8: Required Resources to Find A Solution for the Input Message with More than 294 Clocks

Number of Clocks	Number of Equations	CPU Time (Seconds)	Memory (KB)
294	586	3.57	127568
295	587	3.85	130872
296	588	4.19	140792
297	589	4.44	144828
298	590	4.68	145528
299	591	5.35	150748
300	592	5.58	152916
301	593	6.01	152792
302	594	6.44	152940
303	595	6.64	152828
304	596	6.89	152988
305	597	7.14	153396
306	598	7.40	153560
307	599	7.88	153744
308	600	8.21	153880
309	601	9.34	154196
310	602	11.18	167968

Even with a limited computational environment, experiments were able to be performed up to 310 iterations of input message loading. The program runs out of allocated memory for experiments with 311 or more iterations.

3.4.4.4 Collisions for Unknown Internal State

In this section we discuss the feasibility of finding state collisions in ACORN when the internal state of the keystream generator is unknown. To do this, we defined the key and initialization vector loaded state S^{1536} and S'^{1536} in terms of unknown variables. For a chosen associated data, D, experiments are then conducted to determine the other associated data, D' which will generate two colliding associated data loaded states. However, it was found that the program

Table 3.9: Required Resources for Solving the Equation System for Different Number of Unknown Variable in the Internal State of ACORN (with Full Relabeling)

Unknowns in the state	CPU Time (Seconds)	Memory (KB)	Number of equations	Number of variables	Max. degree
1	7.60	131348	879	880	2
2	7.80	139392	879	881	2
3	8.90	152948	879	882	2
4	16.22	320700	879	883	2
5	78.08	1585012	879	884	2

becomes very slow and runs out of computational resources when all the key and initialization vector loaded state bits are defined as variables.

So, instead of defining all the key and initialization vector loaded state bits as variables, we tried a guess and determine approach by defining some of the state bits as variable while keeping the remaining bits fixed. We started with defining one variable bit in the internal state of S^{1536} and S'^{1536}, while the remaining 292 bits are kept as fixed. Successive experiments were performed by increasing the number of unknown variables in the key and initialization vector loaded state. For this experiment, we have also applied the relabeling technique at each iteration of the associated data loading into the states S^{1536} and S'^{1536}. It was found that the experiment could be performed while there are less than 6 unknown variables in the key and initialization vector loaded state. For these attempts the required CPU time, memory, size of the equation system and maximum degree of the generated equations are given in Table 3.9.

It can be seen that the complexity of solving the equation system is increasing as the number of unknown variable increases. The experiment seems to run out of memory when there are more than 5 unknown variables in the key and initialization vector loaded state. Table 3.9 shows that there is a significant increase in the required CPU time and memory when the number of unknown variables are increased to 5. This may be due to the limitations of the software. As given in Table 3.9, obtaining a solution with 5 unknown variables will require an exhaustive search to guess the contents of the other 288 register stages with a complexity of 2^{288}. The total complexity of the attack will be $2^{288} \times$ complexity of solving the equation system. This is worse than the exhaustive search on the key and therefore not a feasible approach.

Table 3.10: Required Resources for Solving the Equation System for Different Number of Unknown Variable in the Internal State of ACORN (with Partial Relabeling)

Unknowns in the state	CPU Time (Seconds)	Memory (KB)	Number of equations	Number of variables	Max. degree
1	3.46	128232	586	587	2
2	3.61	127732	586	588	2
3	4.20	131544	586	589	2
4	9.37	181712	586	590	3
5	16.38	251044	586	591	5
6	488.07	923560	586	592	5

Similar experiments were also conducted to determine the feasibility of the above-mentioned experiment when the relabeling technique is not applied to the loading of the associated data D into the internal state S^{1536}. In this case, relabeling technique is only applied while the associated data D' are being loaded to the internal state S'^t. For these attempts the required CPU time, memory, size of the equation system and maximum degree of the generated equations are given in Table 3.10.

The experiments were able to perform when there are less than 7 unknown variables in the internal state. Thus this will require an exhaustive search of 2^{286} over the internal state. The total complexity of this attack will be $2^{286} \times$ complexity of solving the equation system. The size of the generated equation system shown in Table 3.10 is small compared to the one given in Table 3.9. This is because relabeling is applied only when loading the associated data D' for the experiment results given in Table 3.10. However, this comes with the trade-off of an increase in the degree of the generated equation system, as indicated in Table 3.10. These comments also apply to the case for obtaining state collision by manipulating the plaintexts during the encryption phase of ACORN.

Experiments were also performed by representing the key bits as unknown variables, rather than defining the internal state bits as variables. For this case the experiment needs to go through a large 1536 iterations of key and initialization vector loading phase. Therefore, if there are too many unknown variables then the degree of the generated equations will be high and can reach up to the maximum degree.

On the other hand, relabeling 1536 iterations will keep the degree of the gen-

erated equations to quadratic, but this will introduce a large number of equations and variables in the system. Instead, we approach this without applying the relabeling during these 1536 iterations. One unknown variable is defined in the key space while the remaining are kept fixed. The experiment gives a solution when there is only one unknown variable in the key space. This is because the system of equation remains quadratic when there is only one unknown variable in the key. However, for more than one unknown variable the degree of the generated equations goes beyond quadratic and may reach up to the maximum degree. This makes it infeasible to obtain a solution when there is more than one unknown variable in the key space. It would be nice to have estimates of the complexity for a range of smaller guessing components and larger unknowns, but given the data in the Table 3.9 and 3.10 increases rapidly it is difficult to extrapolate beyond the table limits with any reasonable degree of confidence.

3.4.5 Summary on the Analysis of State Convergence and State Collisions in ACORN

We have identified a weakness in the state update function which result in collisions in the internal state of ACORNv1. We identified the phases during operation of the cipher where inputs to the state can be manipulated to obtain state collisions. Our analysis shows that collision of the internal states can be forced by selecting complementary values in the external input messages if there exist two ACORN states which differ only in the values contained in seven particular register stages. The input bits that are manipulated can be either associated data bits or plaintext bits.

Experimental results show that for a chosen key and initialization vector, it is trivial to find two distinct messages which result in two ACORN states which differ only in the stages mentioned above. Given these states, a collision can be easily forced in the internal state of ACORN.

In our experiments, we assumed that the internal state or the key is known. This is not necessarily true for an external attacker. However, the sender has access to the key and allowing the sender to create collisions is not a desirable property either. For ACORN, the sender can find a collision for any given key, initialization vector and input message combination, thereby compromising the integrity component of the cipher. A malicious sender can manipulate either the associated data or the plaintext messages to obtain the collision of the tag and

therefore can frame a forged message which will be accepted as legitimate.

We have also conducted experiments to determine the feasibility of finding the state collisions when the key or the internal state of the keystream generator is unknown. It seems that the software Sage runs out of computational resources when the entire internal state is considered unknown. Experiments are also conducted to determine the feasibility of finding the state collision when only part of the internal states are known. We were able to perform the experiments when there are less than seven unknowns in the internal state of the keystream generator. This requires to guess a large portion of the internal states and does not produce a better result than the exhaustive search.

Note that Lafitte et al. [124] also analysed the state collisions in ACORNv1 and ACORNv2. Their experiments partly build on the work reported here and generalise the state collision results provided in this section. Our experimental result shows state collision in ACORN for two distinct internal states. Lafitte et al. [124] shows that this can be extended to finding collisions for more than two states. Both of these analyses require the knowledge of the internal states of ACORN.

3.5 Cube Attack on ACORN

This section provides a description of the application of the cube attack to the authenticated encryption stream cipher ACORNv1. The attack can be performed either in the initialization phase, encryption phase or in the decryption phase. ACORN does not take any external input during the tag generation phase and therefore a cube attack is not applicable in this phase. In the following, we discuss the applicability of the cube attack to the initialization and encryption phases of ACORN.

3.5.1 Cube Attack during the Initialization Phase

In the initialization phase the key, initialization vector and associated data are loaded in to the internal state of ACORN. In general, the cube can be selected either from the input key, the initialization vector or the input associated data set. However, an adversary needs to have the ability to manipulate the key bits if the cube bits are chosen from the input key. On the other hand, the attack scenario falls under the nonce-reuse scenario when the cube bits are chosen from the

associated data set. This is since multiple associated data will be authenticated using the same key and initialization vector, if we choose to select the cube bits from the associated data set. The designer of ACORN does not claim any security when the same initialization vector is used with the same key to encrypt or authenticate multiple sets of data.

We consider the scenario where the cube is chosen from the initialization vector set. This requires the preprocessing phase to identify suitable cube bits chosen from the input initialization vector, which generate linear equations in terms of the key bits. Each of the linear equations is computed by summing the output function of ACORN for all the possible values of the corresponding cube. Therefore in the online phase, an adversary first needs to compute the right hand side of these equations for the corresponding cube. This follows the chosen initialization vector model, where an adversary encrypts the plaintext with the key and chosen initialization vectors (varying the cube bits obtained from the preprocessing phase) and sums the output bits over n-dimensional Boolean cube to compute the right hand side of the corresponding equation. The secret key can be recovered by solving these equations if sufficient number of equations are generated. The attack requires an output bit from the ACORN output function for $n_c \times 2^{l_c}$ chosen initialization vectors where n_c and l_c represent the total number and the length of the cubes, respectively. So, the complexity of finding the right hand side of the linear equations during the online phase of the attack is no more than $n_c \times 2^{l_c}$. This will be followed by solving the equations using Gaussian elimination, which will require approximately n_c^3 operations when $n_c = 128$. Thus the total complexity of the attack is about $n_c \times 2^{l_c} + n_c^3$. If the equations generated are insufficient, i.e., $n_c < 128$, an adversary can achieve partial key recovery by solving the equations and the rest of the key bits can be found by exhaustive search.

3.5.2 Cube Attack during the Encryption Phase

In the encryption phase, plaintext bits are loaded into the internal state of ACORN. Therefore plaintext can be considered as the public variables in this phase and cubes can be chosen from the input plaintext set. In this case, during the preprocessing phase an adversary manipulates the plaintext bits to generate linear equations in terms of the state bits. The linear equations are computed by summing the output function of ACORN for all the possible values of the

corresponding cube. In the online phase, the adversary first needs to find the right hand side of these equations. This is a chosen plaintext attack, where an adversary encrypts the chosen plaintext (varying the cube bits obtained from the preprocessing phase) with the same key and initialization vector and sums the output bits over n-dimensional Boolean cube to compute the right hand side of the corresponding equation. Finally the initial state of the cipher can be recovered by solving the generated equations if sufficient equations are generated, i.e., $n_c = 293$. The attack requires the output bit from the ACORN output function for $n_c \times 2^{l_c}$ chosen plaintext vectors. Following the similar computation as shown in the previous section, the total attack complexity of the state recovery attack requires about $n_c \times 2^{l_c} + n_c^3$ operations.

Unlike the initialization phase, the encryption phase of ACORN does not need to go through a large number of rounds before producing the output bits, so the degree of the output polynomial is expected to be low if an adversary searches for the cube bits from the plaintext variables. Note that the cube attack is more effective on low degree polynomials since a lower degree polynomial will require a smaller cube size. For ACORN an adversary can manipulate the plaintext bits to generate linear equation from the 58th round of the encryption phase when the first plaintext bit p_0 reaches to the output keystream generation function f_z. The degree of the output polynomial of ACORN at the 58th round of encryption phase is only 3. Therefore, at that point of the encryption phase an adversary needs a cube size of 2 at most. This makes it trivial to find linear relations in terms of the state bits.

Note that this attack falls under the nonce-reuse scenario. This means an adversary chooses different set of cubes from the plaintext variables and evaluates the output function of ACORN with a fixed key and initialization vector. As mentioned earlier, the designer of ACORN does not claim any security when the same initialization vector is used with the same key to encrypt or authenticate multiple sets of data.

3.5.3 Applying Cube Attack on ACORN

We conducted experiments to analyse the feasibility of the cube attack on different phases of ACORN. Our experimental analysis of the cube attack on ACORN is conducted using Sage version 6.4.1 [144] on a standard 3.4GHz Intel Core i7 PC with 16GB memory. Experiments are performed on both the initialization

and encryption phases of ACORN to identify suitable cubes. The following sections discuss the application of the cube attack to these two different phases of ACORN.

3.5.3.1 Cube Attack using the Initialization Vector

The initialization of ACORN has a large number of rounds: $l_d + 2048$ rounds. This has a minimum value of 2048 when there are no associated data inputs, i.e., $l_d = 0$. The degree of the output function of ACORN grows with the increase in the number of rounds and is expected to be quite high when the full 2048 rounds are used. Therefore, the size of the cube is also expected to be high if we choose to find a cube after 2048 rounds. This requires a significant amount of computational time. We therefore tested the cube attack on reduced round versions of ACORN.

For a reduced round version of ACORNv1, with an initialization phase of 500 rounds, we have a total of 21 linear equations in terms of the secret key bits. These equations are derived during the preprocessing phase of the cube attack. For this 8000 random cubes of size 2 were tested; however, none of these cubes passed the linearity test. So we increased the cube size and checked if there exist linear equations when the cube size is increased. No suitable cubes were found for a cube size of 3 and 4 after searching over 1000 random cubes. For a cube size of 5, 1000 randomly chosen cubes were tested among which 20 passed the linearity test, but for most of these cubes the linear coefficient of the secret variable was found to be 0, i.e., the cube summation results only in a constant. Only 3 cubes $\{v_{120}, v_{124}, v_{93}, v_7, v_{63}\}$, $\{v_{31}, v_{124}, v_{115}, v_{18}, v_{122}\}$, $\{v_{39}, v_{51}, v_{124}, v_{76}, v_{115}\}$ were found which give linear coefficients in terms of the secret variable. These cubes were obtained by running the preprocessing phase for about a week.

Note that there are possibly more cubes of size 5 after 500 initialization rounds, however to find more cubes we need to increase the cube search space. Instead of increasing the search space, we used the following method to obtain new cubes from the randomly chosen cubes. For a given randomly chosen cube, increase both the cube indices and the number of rounds by one. Choose the new cube as a valid one if it satisfies the linearity test. This reduces the time complexity in the preprocessing phase since the adversary does not need to search for a suitable random cube from a total possible search space of $\binom{128}{5}$. With this technique, we were able to find total 12 cubes of size 5 which gives

12 linear equations in terms of the secret key bits. Further experiments are performed by choosing the cubes from a smaller subset of the previously found cubes, i.e., the cubes of size 5 were selected from the subset of the cube indices $\{120, 124, 93, 7, 63, 31, 115, 18, 122, 39, 51, 76\}$. This technique gave us two more new cubes: $\{v_{39}, v_{93}, v_{31}, v_{124}, v_{122}\}$, $\{v_{120}, v_{51}, v_{124}, v_7, v_{63}\}$ at round 500 and an additional cube at round 503: $\{v_{42}, v_{125}, v_{21}, v_{127}, v_{118}\}$. Using these cubes, combined with the previously mentioned technique, we were able to find 9 more cubes which result in distinct linear equations. In total 21 linear equations are found after the 500 initialization round of ACORNv1. Therefore for ACORNv1 with 500 initialization rounds, adversary can guess 107 key bits and the remaining 21 bits can be found by solving these linear equations. In this case, the complexity of the attack is dominated by the exhaustive search. In the following we discuss about reducing the dominance of the exhaustive search by reducing the initialization round further.

Observing the linear equations generated for round 500, we have found that the equations consist only of the first 99 variables of the key. Therefore, an adversary needs 99 independent linear equations to find out these 99 key bits. To reduce the dominance of the exhaustive search we have further examined on generating more linear relations by reducing the number of rounds and the cube indices. The new cube is considered valid if it still satisfies the linearity test. The experiment is repeated by reducing the number of rounds and the cube indices, till 99 or more cubes are found. These equations were found after reducing the round of ACORN initialization phase from 500 to 477. Examples of some of these cubes resulting in linear equations are listed in Table 3.11. Please refer to the Appendix A.1 for the full list of equations.

During the online phase of the attack, an adversary first needs to find the right hand side of these linear equations. An adversary encrypts the plaintext with the key and chosen initialization vectors (varying the cube bits) and sums the respective output bits over the n-dimensional Boolean cube to compute the right hand side of the corresponding equation. This requires a total $99 \times 2^5 \approx 2^{11.6}$ chosen initialization vectors. Therefore, an adversary can expect to recover the key bits with complexity less than exhaustive search, if the initialization phase of ACORN is reduced to 477.

We implemented the attack to verify the online phase of the cube attack on the 477 initialization round version of ACORNv1. We started by computing

Table 3.11: Example of Linear Equations Obtained for ACORN with 477 Initialization Rounds

Cube Indexes	Round	Linear Equation
$16, 28, 101, 53, 92$	477	$k_3 \oplus k_8 \oplus k_{15} \oplus k_{17} \oplus k_{19} \oplus k_{23} \oplus k_{25} \oplus k_{28} \oplus k_{29} \oplus k_{62}$
$\vdots$	$\vdots$	$\vdots$
$42, 54, 127, 79, 118$	503	$k_1 \oplus k_2 \oplus k_7 \oplus k_9 \oplus k_{11} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{29} \oplus k_{34} \oplus k_{41} \oplus k_{43} \oplus k_{45} \oplus k_{49} \oplus k_{51} \oplus k_{54} \oplus k_{55} \oplus k_{88} \oplus 1$
$19, 73, 11, 104, 102$	480	$k_1 \oplus k_{16} \oplus k_{18} \oplus k_{21} \oplus k_{22} \oplus k_{55}$
$\vdots$	$\vdots$	$\vdots$
$42, 96, 34, 127, 125$	503	$k_0 \oplus k_2 \oplus k_4 \oplus k_5 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{19} \oplus k_{24} \oplus k_{39} \oplus k_{41} \oplus k_{44} \oplus k_{45} \oplus k_{78} \oplus 1$
$13, 106, 97, 0, 104$	482	$k_1 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{14} \oplus k_{18} \oplus k_{21} \oplus k_{25} \oplus k_{29} \oplus k_{30} \oplus k_{31} \oplus k_{32} \oplus k_{35} \oplus k_{38} \oplus k_{40} \oplus k_{43} \oplus k_{44} \oplus k_{68} \oplus k_{77}$
$\vdots$	$\vdots$	$\vdots$
$34, 127, 118, 21, 125$	503	$k_0 \oplus k_3 \oplus k_4 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{22} \oplus k_{23} \oplus k_{24} \oplus k_{25} \oplus k_{28} \oplus k_{30} \oplus k_{31} \oplus k_{32} \oplus k_{35} \oplus k_{39} \oplus k_{42} \oplus k_{46} \oplus k_{50} \oplus k_{51} \oplus k_{52} \oplus k_{53} \oplus k_{56} \oplus k_{59} \oplus k_{61} \oplus k_{64} \oplus k_{65} \oplus k_{89} \oplus k_{98} \oplus 1$
$113, 117, 86, 0, 56$	493	$k_5 \oplus k_7 \oplus k_{10} \oplus k_{11} \oplus k_{44} \oplus 1$
$\vdots$	$\vdots$	$\vdots$
$123, 127, 96, 10, 66$	503	$k_0 \oplus k_{15} \oplus k_{17} \oplus k_{20} \oplus k_{21} \oplus k_{54}$
$113, 44, 117, 0, 56$	493	$k_5 \oplus k_7 \oplus k_8 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{27} \oplus k_{32} \oplus k_{44} \oplus k_{47} \oplus k_{49} \oplus k_{52} \oplus k_{53} \oplus k_{86}$
$\vdots$	$\vdots$	$\vdots$
$123, 54, 127, 10, 66$	503	$k_4 \oplus k_5 \oplus k_8 \oplus k_9 \oplus k_{15} \oplus k_{17} \oplus k_{18} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{26} \oplus k_{28} \oplus k_{29} \oplus k_{30} \oplus k_{37} \oplus k_{42} \oplus k_{54} \oplus k_{57} \oplus k_{59} \oplus k_{62} \oplus k_{63} \oplus k_{96}$
$42, 125, 21, 127, 118$	503	$k_2 \oplus k_3 \oplus k_8 \oplus k_{10} \oplus k_{12} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{30} \oplus k_{35} \oplus k_{42} \oplus k_{44} \oplus k_{46} \oplus k_{50} \oplus k_{52} \oplus k_{55} \oplus k_{56} \oplus k_{89} \oplus 1$

the right hand side of each equation by summing the output bits for all the

possible values of the corresponding cube. This requires access to $2^{11.6}$ specific output bits. The equations are then solved once the right hand side of all the equations are computed. We found that the solution was not unique for some of the key bits. Some of the solutions for the secret key bits are found to be dependent on the key bits: $k_{91}, k_{94}, k_{95}, k_{96}, k_{97}, k_{98}$. This is because some of the computed equations are linearly dependent. Guessing these six key bits correctly results in a unique solution which provides the key bits for $k_0, \cdots, k_{90}, k_{92}, k_{93}$. We also need to guess the rest of the 29 key bits to recover the whole key. Therefore, the attack require to guess total $29 + 6 = 35$ of the key bits. The linear equations can be solved using row reduction and back substitution, and the resulting solutions must then be checked with each of the 2^{29} possibilities for the remaining key bits. So, in practice the total attack complexity is about $2^{11.6} + 93^3 + 6 \times 93 \times 2^6 + 2^6 \times 2^{29} \approx 2^{35}$.

We have also performed the experiments for the reduced round (477 rounds) variant of ACORNv2. For this experiment, we tested the same cubes that were found for ACORNv2 (as above). Interestingly we noticed that the same cube sets result in linear equations for ACORNv2 as well. This was also verified by running the online phase of the attack to recover the key bits.

3.5.3.2 Cube Attack using Plaintext

This section discusses the application of the cube attack to the encryption phase of ACORN. Unlike the initialization phase, the degree of the output polynomial during the encryption phase is expected to be comparatively low. Therefore cube attack using the plaintext can be applied to the full version of ACORN. In the following, we describe the cube attack to the encryption phase of ACORN.

For searching suitable cubes we first represented the internal state, S of ACORN symbolically and checked the output function Z while loading each bit of plaintext P. At each round of the plaintext loading we looked into the symbolic output function to determine whether it becomes linear when differentiating it with respect to a subset of plaintext variables. Any such set of plaintext, if it exists, is equivalent to the set of cubes in the cube attack. For the first 57 round of the plaintext loading phase no such set was found because the plaintext bit does not reach the output function till the 58th round. At the 58th round of the plaintext loading, we found that differentiating the output equation Z^{58} with respect to plaintext p_0 results in a linear equation. That also means that if we

do a cube attack numerically, then summing the output polynomial Z^{58} over the cube p_0 will result a linear equation. However, when running the experiments symbolically, the software runs out of resource after a small number of rounds. The symbolic computation with Sage was able to successfully compute the output function until 148th round of the plaintext loading phase. This method gave us 103 linear equations with 293 unknowns. This is not sufficient to reduce the complexity below the exhaustive search of 2^{128}.

To find more cubes that produce linear equations we performed the experiment numerically. We started with the cubes $\{p_0\}$, $\{p_0, p_7\}$ and $\{p_0, p_{21}\}$ for round 58, 107 and 121, respectively, which are found using the symbolic computation. We then used the following method to obtain a new cube: Increase the cube indices and the number of rounds by one and choose the new cube as a valid one if it satisfies the linearity test. 100 linearity tests were performed for each of the cubes. With this technique, we have obtained 245 linear equations with 293 variables. Among the 245 cubes for these equations, 42 are of size 1 and the rest are of size 2. Example of some of these cubes resulting in linear superpolys are listed in Table 3.12. For the whole list of equations, please refer to the Appendix A.2.

In the online phase of the attack an adversary needs to compute the right hand side of these equations and then solve the equations to recover the state bits. An adversary first encrypts the chosen plaintexts (varying the cube bits) with the same key and initialization vector and sums the respective output bits over the n-dimensional Boolean cube to compute the right hand side of the corresponding equation. This requires about $42 \times 2^1 + 203 \times 2^2 \approx 2^{9.81}$ chosen plaintext bits to find the right hand side of these 245 equations.

To verify the online phase of the attack, we implemented the attack by solving these linear equations. We first computed the right hand side of each equation by summing the output bits for all the possible values of the corresponding cube. This requires an adversary to have access to $2^{9.81}$ specific output bits. We then represented these linear equations by a matrix and found that the rank of the matrix is 234. Applying Gaussian elimination to the underdetermined system yields a row reduced matrix that can be used with each guess of the 59 undetermined variables to calculate the remaining 234 variables. This process enables us to recover all the state bits. Therefore in practice the total attack complexity is about $2^{9.81} + 234^3 + 59 \times 234 \times 2^{59} \approx 2^{72.8}$.

Table 3.12: Example of Linear Equations Obtained for ACORN by Choosing the Cube Set from the Plaintext Bits

Cube Indexes	Round	Linear Equation
0	58	$s_{73} \oplus s_{78} \oplus s_{119} \oplus s_{214} \oplus s_{217} \oplus s_{251}$
⋮	⋮	⋮
41	99	$s_{68} \oplus s_{78} \oplus s_{113} \oplus s_{114} \oplus s_{117} \oplus s_{119} \oplus s_{160} \oplus s_{218} \oplus s_{224} \oplus s_{233} \oplus s_{238} \oplus s_{255} \oplus s_{258} \oplus s_{292}$
0, 7	107	$s_{11} \oplus s_{34} \oplus s_{72} \oplus s_{170} \oplus s_{176} \oplus s_{209}$
⋮	⋮	⋮
83, 90	190	$s_{18} \oplus s_{33} \oplus s_{41} \oplus s_{63} \oplus s_{71} \oplus s_{73} \oplus s_{76} \oplus s_{79} \oplus s_{94} \oplus s_{108} \oplus s_{109} \oplus s_{112} \oplus s_{114} \oplus s_{117} \oplus s_{154} \oplus s_{155} \oplus s_{160} \oplus s_{175} \oplus s_{181} \oplus s_{187} \oplus s_{193} \oplus s_{214} \oplus s_{216} \oplus s_{218} \oplus s_{219} \oplus s_{222} \oplus s_{224} \oplus s_{225} \oplus s_{226} \oplus s_{233} \oplus s_{238} \oplus s_{253} \oplus s_{255} \oplus s_{258} \oplus s_{259} \oplus s_{292}$
0, 21	121	$s_{83} \oplus s_{88} \oplus s_{127} \oplus s_{129} \oplus s_{131} \oplus s_{174}$
⋮	⋮	⋮
118, 139	239	$s_{63} \oplus s_{83} \oplus s_{107} \oplus s_{109} \oplus s_{115} \oplus s_{119} \oplus s_{122} \oplus s_{124} \oplus s_{151} \oplus s_{153} \oplus s_{158} \oplus s_{159} \oplus s_{160} \oplus s_{162} \oplus s_{165} \oplus s_{168} \oplus s_{169} \oplus s_{175} \oplus s_{179} \oplus s_{183} \oplus s_{187} \oplus s_{193} \oplus s_{198} \oplus s_{200} \oplus s_{201} \oplus s_{204} \oplus s_{206} \oplus s_{211} \oplus s_{213} \oplus s_{215} \oplus s_{218} \oplus s_{219} \oplus s_{222} \oplus s_{224} \oplus s_{225} \oplus s_{226} \oplus s_{233} \oplus s_{238} \oplus s_{245} \oplus s_{247} \oplus s_{249} \oplus s_{253} \oplus s_{255} \oplus s_{258} \oplus s_{259} \oplus s_{292}$

3.5.4 Summary of Cube Attacks on ACORN

We applied cube attack to the reduced round versions of ACORN. That is, the number of rounds of initialization is reduced from the specified 2048 rounds. Our analysis shows that cube attack can recover the secret key with a complexity of 2^{35} when the number of initialization rounds is reduced to 477. We have also tested and verified the attack by recovering the actual key bits for a randomly chosen key. The attack can be possibly extended to higher number of initialization rounds, but it will require a larger cube size which requires a search over very large cube spaces. It is difficult to evaluate the performance of the cube attack for larger versions of ACORN without knowing the suitable choices of the cube. Due to the high time complexity of searching larger cubes, our experiments were conducted only for smaller cube sizes.

Table 3.13: Summary of Cube Attacks on ACORN

Algorithm	Initialization Rounds	Attack Type	Attack Model	Complexity
ACORNv1	$l_d + 477$	Key Recovery	Chosen IV	2^{35}
ACORNv1	$l_d + 2048$	State Recovery	Chosen Plaintext	$2^{72.8}$

Also, note that the cubes identified for the 477 initialization rounds are only of size 5, whereas the degree of the output function after 477 round is expected to be much higher than 5. This suggests that the key and the initialization vector are not mixed properly yet after the 477 rounds; however, it would be interesting to check if this behaviour continues for higher rounds of the initialization phase as well.

We have also shown that it is trivial to recover the state bits of the full version of ACORN with complexity less than exhaustive search, if the same key and initialization vector is used to encrypt or authenticate multiple sets of input plaintext. This does not threaten the security of ACORN if it is used as the designers suggested.

An overall summary of the cube attacks applied to ACORNv1 in this thesis is tabulated in Table 3.13. As illustrated in Table 3.13, cube attacks on ACORN can be applied to recover either the secret key or the internal state of ACORNv1. Table 3.13 illustrates that the key recovery attack on ACORNv1 falls under the chosen initialization vector model and requires a complexity of 2^{35} for a reduced initialization round of 477 iterations.

Also as tabulated in Table 3.13, the state recovery attack on ACORNv1 falls under the chosen plaintext model and can be applied to the full version of the cipher with a complexity of $2^{72.8}$. Note that the state recovery attack on ACORNv1 tabulated in Table 3.13 requires to encrypt multiple sets of plaintext with the same key and initialization vector, and therefore falls under the nonce reuse scenario. Compared to this, the best known nonce-reuse attack [123] on ACORNv1 can recover the internal state with a complexity of 2^{25} when six plaintexts are encrypted with the same key and initialization vector.

After publication of this work, a variant of the cube attack was also applied by Todo et al. [126] to ACORNv3. These authors show the theoretical complexity to recover a single superpoly of a cube for 704 initialization rounds. Note also that a verified superpoly recovery attack is claimed by Todo et al. [126] for up to 517 initialization rounds of ACORN with practical complexity. On the other

Table 3.14: Comparison of Cube Attacks on ACORN to Recover the Superpoly

Algorithm	Initialization Rounds	Cube Size	Complexity	Verified Attack?
ACORNv1	$l_d + 503$	5	2^5	Yes
ACORNv3	$l_d + 517$	11	2^{20}[126]	Yes
ACORNv3	$l_d + 704$	64	2^{122}[126]	No

hand, our attack is a verified key recovery attack for 477 initialization rounds, which can work up to 503 initialization rounds when the goal is a superpoly recovery.

A comparison of the attacks by Todo et al. and the cube attacks presented in this chapter is tabulated in Table 3.14. Table 3.14 illustrates that the theoretical results on cube attacks by Todo et al. improves the verified cube attacks presented in this chapter by 201 rounds, at the expense of greatly increased complexity. Note also that these results of cube attacks by Todo et al.[126] on 704 initialization rounds of ACORN are theoretical and have not been verified experimentally. In relation to a verified superpoly recovery, the cube attack by Todo et al. improves upon our attack by 14 rounds at the expense of an increased but still practical complexity.

The cube attack applied to ACORN in this thesis works by considering the underlying algorithm as a blackbox polynomial, whereas the attack developed by Todo et al.[126] considers the underlying algorithm as a non-blackbox polynomial. Thus, our attack requires experimental analysis and can not evaluate cube sizes when the complexity reaches beyond the experimental range. It is therefore difficult to validate the cube size and the complexity of our attack for a larger number of rounds when comparing it to the attack by Todo et al. [126].

3.6 Forgery Attacks on ACORN

This section provides our observations on the fault based forgery attacks on ACORN. The fault based forgery attack described here is a bit flip forgery attack.

3.6.1 Fault based Forgery Attack on ACORN

This section discusses a fault based forgery attack on ACORN. The goal of the attack is to modify the input message by flipping specific message bits and have

the modified message accepted as legitimate by the receiver.

A fault based forgery attack on the authenticated encryption cipher CLOC and SILC was introduced by Roy et al. [146]. Following this, in the CAESAR Google discussion forum it was pointed out by Iwata et al. [147] that any authenticated encryption scheme can be forged using faulty encryption queries. This requires an adversary to inject faults in the inputs submitted to the encryption oracle and then use the output of the encryption oracle with the faulty inputs to continue with the forgery.

We describe here a similar fault injected forgery on ACORN, however the faults are injected into the internal state instead of the inputs. The generic fault based forgery attack by Iwata et al. [147] applies a specific fault in the message before the message is loaded in to the device. Their attack process requires the attacker to enquire about the faulty ciphertext and the faulty tag for a faulty (modified) message. However, in our attack, the faults are applied in the encryption device after the message is loaded. This attack may be more practical in some applications where the attacker is able to access the encryption device, rather than requiring the attacker to intercept and alter messages being sent to the device. In the following, we discuss the particular details of such a fault injection based forgery for ACORN.

In the associated data processing and encryption phases of ACORN, the input associated data and the plaintext are loaded into the internal state of the cipher. Therefore, any changes in the input associated data or the plaintext will affect the internal state. An adversary can mount a forgery attack by injecting faults into the internal state of ACORN to reflect the changes made in the associated data or in the plaintext. For the following we use the term input message M to represent either the associated data or the plaintext.

Suppose the adversary wants to modify the t^{th} bit of the input message block M^t. Let M'^t denote the modified input message block, where the modification is bit flip in the t^{th} bit of the original input message block M^t. We observe that the input message block M^t is XOR-ed with the feedback function of ACORN and the output of this XOR is loaded in the last register stage s_{292}. Thus XOR-ing M'^t with the feedback function of ACORN will flip the contents of register stage s^t_{292}.

To perform the forgery attack, at time instant t adversary applies bit flipping faults to the register stage s^t_{292} of the sender's side. This is equivalent to flipping

the t^{th} bit of the original input message block M^t. Let C'^t and τ' denote the ciphertext and the tag generated for the fault induced version of the state. An adversary can simply intercept and change the ciphertext or the associated data (depending on whether the fault was applied in encryption or associated data processing phase), and send it with the faulty tag τ' to perform the attack. The adversary does not need to apply any faults at the receiver's side.

In the decryption and tag verification phase, the receiver will XOR the received/recovered message block with the feedback function of ACORN and the output of this XOR is loaded in the register stage s^t_{292}. The received/recovered message will be the same as the modified input message M'^t. This is equivalent to applying the bit flipping faults in the register stage s^t_{292} at time t.

Following this, the finalization process of ACORN at the receiver side generates the tag τ''. Clearly, the tag τ'' generated at the receiver is the same as the tag τ' sent by the sender. Therefore the faulty ciphertext C'^t and faulty tag τ' will be accepted as legitimate at the receiver. The total number of faults required for the forgery attack is equal to the number of bits flipped in the original message.

3.6.1.1 Attack Algorithm for Fault Based Forgery on ACORN

The steps involved in the fault based forgery attack on ACORN are outlined in Algorithm 3.2.

Algorithm 3.2 Algorithm for Fault Based Forgery Attack on ACORN

1: At time t, insert bit flipping fault in register stage s^t_{292} of the sender device.
2: Continue the finalization process at the sender and observe the generated tag τ'^t.
3: At time t, modify the associated data D^t to D'^t, if the goal is associated data modification. Alternatively, modify the ciphertext C^t to C'^t.
4: Send the modified associated data D'^t or the modified ciphertext C'^t, with the tag τ'.

3.7 Summary on the Security Analysis of ACORN

This chapter examined the feasibility of different attack scenarios on the authenticated encryption cipher ACORN. Particularly we analysed the cube attack, state collision attack and forgery attack on ACORN.

We analysed state convergence and state collisions in ACORNv1. The analysis shows that the state update function of ACORN is one-to-one and state convergence is not possible during any operation phase of the cipher. On the other hand, the analysis of state collisions in ACORN reveals that collisions in the internal states can be forced by manipulating the external inputs of ACORN. Experimental results show that it it trivial to find distinct messages which can force a state collision in the internal state of ACORN. This is applicable to all the versions of ACORN. Note that the adversary requires the knowledge of the internal states of ACORN to find these collisions.

We also investigated the cube attacks on ACORNv1 and ACORNv2. The attack is applied to a reduced version where the initialization phase is reduced to 477 rounds. The cube attack can recover the secret key with a complexity of 2^{35} for the reduced version of ACORNv1. This attack has been tested and verified by recovering the secret key of ACORN for a randomly generated key. In addition, we demonstrated that this attack also applies to ACORNv2.

We have also shown that cube attack can recover the initial state of ACORN with a complexity less than the exhaustive search. In particular, the cube attack requires a complexity of about $2^{72.8}$ to recover the 293 bits of internal state. This however requires to encrypt multiple sets of plaintext with the same key and initialization vector. The designer of ACORN does not claim any security when the same key and initialization vector are used to encrypt multiple inputs.

Finally we illustrated a fault based forgery attack on ACORN. The fault based forgery attack uses bit flipping fault injections in the internal state of ACORN to construct a state collision in the internal state. This state collision is then used to construct the tag forgery for two different inputs of ACORN. This type of fault based forgery attack is trivial when adversary has access to the implementation of the algorithm in the sender's device. This attack requires the introduction of a single bit fault in the ACORN state for a single bit flip in the input message. As pointed out by Iwata et al. [147], this type of forgery attack is applicable to other CAESAR candidates.

3.7.1 Security Impact

This section compares the applicability of different attack methods on ACORN. Table 3.15 provides an overall comparison of different attacks applied to ACORN.

As illustrated in Table 3.15, the cube attack works only for a reduced version

Table 3.15: Comparison of Different Attack Methods on ACORN

Attack Method	Attack Type	Complexity	Assumption
Cube Attack	Key Recovery	Complexity: 2^{35}.	Reduced initialization phase.
Cube Attack	State Recovery	Complexity: $2^{72.8}$.	Nonce-reuse.
State Collision	Forgery	Negligible	Known internal state.
Fault Attack	Forgery	Negligible	Bit flipping faults

of ACORN. The best cube attack described in this chapter can work up to l_d+477 rounds of the initialization phase of ACORN, while the full version of ACORN has l_d+2048 rounds of initialization phase. Therefore, we conclude that ACORN has a large security margin against key recovery using cube attacks.

Table 3.15 also shows that cube attack can recover the internal state of the full version of ACORN with a complexity less than the exhaustive search. This attack however requires the assumption of nonce-reuse scenario for which the designer of ACORN does not claim any security.

Table 3.15 shows that an adversary can apply a state collision based existential forgery attack on ACORN when the internal state is known. The assumption of known internal state is not necessarily true for an external attacker. However, the sender has knowledge of the internal state of ACORN, and allowing the sender to create state collisions is not a desirable property. A malicious sender can manipulate the external inputs of ACORN to obtain such collision based forgery attacks on ACORN.

Also, Table 3.15 shows that fault based universal forgery attack is trivial for ACORN with a reasonable number of faults. This type of fault attack is applicable to other CAESAR candidates and it is argued in the CAESAR Google group that the goal of the fault attack is key recovery rather than to construct a forgery attack [147]. Alternatively, it is also argued in the same Google group that fault based forgery attack compromises the integrity component of the authenticated encryption algorithm and compromising one of the components is sufficient to justify the relevancy of fault based forgery attacks. We think the latter argument is reasonable since it can break one of the components of the AE cipher. The fault based forgery attack seems to be a powerful attack on the integrity component of ACORN and it is important to develop necessary countermeasures to prevent this type of attack.

Chapter 4

Analysis of Tiaoxin-346

This chapter investigates the security of the authenticated encryption stream cipher Tiaoxin-346 [32, 33]. Tiaoxin-346 is one of the third-round candidates in the CAESAR [25] competition. The investigation includes the state cycle analysis, and the applicability of cube attacks and fault attacks on Tiaoxin-346. This chapter also discusses some structural similarties in Tiaoxin-346 and AEGIS. The results presented in Section 4.6 of this chapter have been published in the Proceedings of the Australasian Computer Science Week Multiconference 2018 (ACSW 2018) [39].

Tiaoxin-346 [32, 33] is an authenticated encryption stream cipher design, similar to a word based nonlinear feedback shift register based stream cipher. It is word based, with a word size of 128 bits. The AES round function is used as a component in the nonlinear state update function of Tiaoxin-346. The cipher is intended to provide confidentiality and integrity assurance for the input message.

Tiaoxin-346 has two versions: Tiaoxin-346v1 [32] and Tiaoxin-346v2 [33]. Both versions have the same structure and same phases of operation. Additionally, different use cases of the cipher are described in Tiaoxin-346v2. For the rest of the chapter, we use Tiaoxin-346 to refer to both of these versions.

This chapter is organised as follows. Section 4.1 describes the notations and operations used in this chapter. Section 4.2 provides a detailed description of Tiaoxin-346. Section 4.3 describes the existing cryptanalysis of Tiaoxin-346. Section 4.4 provides our observations on the state cycle analysis of Tiaoxin-346. Section 4.5 describes our application of a cube attack on Tiaoxin-346. Section 4.6

describes our application of a fault based attack used for forgery and for key recovery. Section 4.7 discusses the similarities in the construction techniques of Tiaoxin-346 and another CAESAR stream cipher candidate, AEGIS. This section also demonstrate that, due to the structural similarities, AEGIS may be vulnerable to similar attacks to those we applied to Tiaoxin-346. Finally, Section 4.8 provides an overall summary for this chapter.

4.1 Notations

The notations used in this chapter are as follows.

- Word: A sequence of 16 bytes (128 bits).
- Block: A sequence of two words.
- $K = k_0 k_1 \cdots k_{127}$: denotes the 128-bit secret key.
- $V = v_0 v_1 \cdots v_{127}$: denotes the 128-bit public initialization vector.
- Z_0: A constant `0x428a2f98d728ae227137449123ef65cd` in hexadecimal.
- Z_1: A constant `0xb5c0fbcfec4d3b2fe9b5dba58189dbbc` in hexadecimal.
- $Tr(X)$: One AES round transformation applied to the word X without XOR-ing the subkey.
- T_s: A state composed of s words .
- T_s^t: A state T_s at time t.
- $T_s^t[i]$: Denotes the i^{th} word of state T_s at time t, where $i \in \{0, \cdots, s-1\}$.
- M_s^t: External input to the state T_s at time t.
- $D^t = D_0^t || D_1^t$: One block of input associated data at time t. Each block of associated data consists of two words D_0^t and D_1^t.
- $P^t = P_0^t || P_1^t$: One block of input plaintext at time t. Each block of plaintext consists of two words P_0^t and P_1^t.
- $C^t = C_0^t || C_1^t$: One block of output ciphertext at time t. Each block of ciphertext consists of two words C_0^t and C_1^t.

- τ: A 128-bit authentication tag.
- $msglen$: Length of the plaintext in bits where $0 \leq msglen < 2^{128} - 1$.
- $adlen$: Length of the associated data in bits where $0 \leq adlen < 2^{128} - 1$.
- $X \oplus Y$: Bitwise XOR of words X and Y.
- $X \otimes Y$: Bitwise AND of words X and Y

4.2 Description of Tiaoxin-346

Tiaoxin-346 uses a 128-bit key K and a 128-bit initialization vector V. The input plaintext P is of arbitrary length, $msglen$, where $0 \leq msglen \leq 2^{128} - 1$. Confidentiality is achieved by encrypting the input plaintext P to form ciphertext C. The length of the ciphertext C is the same as the length of the plaintext P. Integrity assurance is achieved by providing an authentication tag τ of length 128 bits. Tiaoxin-346 also provides an integrity assurance service for the associated data D which does not require confidentiality. Tiaoxin-346 supports associated data of arbitrary length, $adlen$, where $0 \leq adlen \leq 2^{128} - 1$.

4.2.1 Structure of Tiaoxin-346

The structure of Tiaoxin-346 is similar to feedback shift register based stream ciphers. Figure 4.1 shows the structure and the state update process for Tiaoxin-346 in diagrammatic form. As shown in Figure 4.1, the state has three components T_3, T_4 and T_6 consisting of three, four and six 128-bit register stages, respectively. This gives the cipher a total internal state size of $13 \times 128 = 1664$bits.

In each round, for each component (except for the first two stages in each of component) the state words are updated by shifting. That is, state words $T_3[2]$, $T_4[2]$, $T_4[3]$, $T_6[2]$, $T_6[3]$, $T_6[4]$, $T_6[5]$ are updated by shifting the contents from the previous word of the corresponding component. The other state words $T_3[0]$, $T_3[1]$, $T_4[0]$, $T_4[1]$, $T_6[0]$, $T_6[1]$ are updated nonlinearly, by applying the AES based transformation function $Tr(X)$ to some of the state words and XOR-ing the transformed value with inputs including state words and external inputs.

Tiaoxin-346 uses the state update function $Update(T_3^t, T_4^t, T_6^t, M_3^t, M_4^t, M_6^t)$ to update the internal state. Here, M_3, M_4 and M_6 are the external inputs to the state T_3, T_4 and T_6, respectively. Depending on the operational phase of the

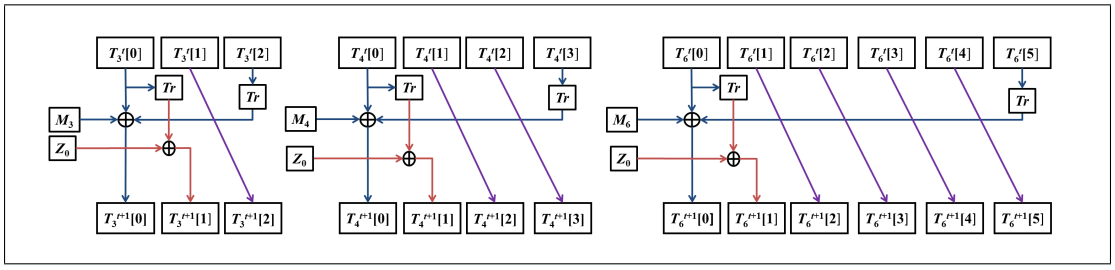

Figure 4.1: Tiaoxin-346 State Update

cipher, these external inputs are defined by the constants or the associated data or the plaintext. The state of Tiaoxin-346 at time $t+1$ is defined as:

$$T_s^{t+1}[i] = \begin{cases} Tr(T_s^t[s-1]) \oplus T_s^t[0] \oplus M_s^t & \text{for } i = 0 \\ Tr(T_s^t[0]) \oplus Z_0 & \text{for } i = 1 \\ T_s^t[i-1] & \text{otherwise} \end{cases} \tag{4.1}$$

where $0 \leq i \leq s-1$ and $s \in \{3, 4, 6\}$.

4.2.1.1 AES Round Function

Tiaoxin-346 uses the AES round transformation function [2] $Tr(X)$ as the non-linear part of its state update function. For completeness we describe the AES round transformation function shown in Figure 4.1.

The round transformation function $Tr(X)$ used in Tiaoxin-346 is the same as one round of AES without XOR-ing the sub key. This is defined as:

$Tr(X) = \text{MixColumns}(\text{ShiftRows}(\text{SubBytes}(X)))$

The input to the AES round function is ordered as a 4×4 matrix (referred as AES state), where each of the matrix positions contains one byte of the input. In each round, SubBytes, MixColumns and ShiftRows operations are applied to the input bytes. These operations work in the same procedure as described in the original description of AES. These are described below:

***SubBytes*:** The SubBytes operation involves sixteen individual byte substitutions. The input bytes are substituted using the Rijndael/AES S-box. The S-box is the only source of nonlinearity in the AES transformation function.

***ShiftRows*:** The ShiftRows operation is applied to the rows of the AES state obtained after the SubBytes operation of each round. The rows of the state are cyclically shifted based on some specific values. In particular row zero, row one, row two and row three are shifted by 0, 1, 2, and 3 positions, respectively.

***MixColumns*:** The MixColumns operation is applied on the rows of the AES state obtained after the ShiftRows operation of each round. This is a linear transformation applied to each column of the AES state. This operation simply multiplies the AES state with a predefined constant matrix.

4.2.2 Phases of Operation

Operations performed in Tiaoxin-346 can be divided into five phases. These are:

1. Initialization
2. Associated data loading
3. Encryption
4. Tag generation
5. Decryption & tag verification

During the initialization and tag generation phase the external inputs M_3, M_4 and M_6 are defined in terms of the constant values Z_0 and Z_1. For the encryption and associated data loading phase these external inputs are defined in terms of the input plaintext P^t and associated data D^t, respectively.

4.2.2.1 Initialization

During the initialization phase, as shown in Figure 4.2, the three components T_3, T_4 and T_6 are each loaded with the key, initialization vector and some constant values in a specific loaded state format. The first two words for each component are loaded with the 128-bit key; that is, $T_3[0] = T_3[1] = T_4[0] = T_4[1] = T_6[0] = T_6[1] = K$. The third state word in each component is loaded with the initialization vector; that is, $T_3[2] = T_4[2] = T_6[2] = V$. The constants Z_0 and Z_1 are loaded in the state elements $T_4[3]$ and $T_6[3]$, respectively. The remaining two words in component T_6 are loaded with zero; that is, $T_6[4] = T_6[5] = 0$.

The external inputs to T_3, T_4 and T_6, denoted M_3, M_4 and M_6, are set to the constants $M_3 = Z_0$, $M_4 = Z_1$ and $M_6 = Z_0$, respectively.

The state of Tiaoxin-346 is updated using $Update(T_3^t, T_4^t, T_6^t, Z_0, Z_1, Z_0)$ for 15 iterations without producing any output. At the end of the initialization phase, the internal state of the three components together forms the initial state of Tiaoxin-346.

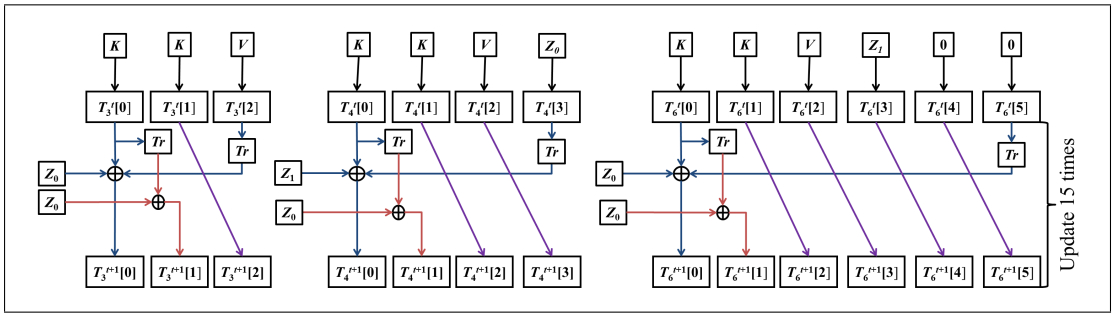

Figure 4.2: Tiaoxin-346 Initialization Procedure

4.2.2.2 Associated Data Loading

At the beginning of the associated data loading phase, the three components T_3, T_4 and T_6 have been loaded with the initial state. The associated data D is divided into l_d blocks: $D^1, \cdots, D^{l_d}$. At each round one block of the associated data is processed. Each block of associated data is composed of two words: $D^t = D_0^t || D_1^t$, where D_0^t, D_1^t represents the two words of the corresponding block.

During the associated data processing phase, the external inputs M_3, M_4 and M_6 are constructed using the associated data words D_0^t, D_1^t. These are defined as: $M_3^t = D_0^t$, $M_4^t = D_1^t$ and $M_6^t = D_0^t \oplus D_1^t$.

The internal state of Tiaoxin-346 is updated using the state update function $Update(T_3^t, T_4^t, T_6^t, D_0^t, D_1^t, D_0^t \oplus D_1^t)$ for l_d rounds, with each round processing one associated data block. During these l_d rounds, no outputs are produced.

4.2.2.3 Encryption

The encryption phase begins after the associated data processing phase is completed. The three components T_3, T_4 and T_6 have been loaded with the associated data at the beginning of the encryption phase. Figure 4.3 shows the diagrammatic form of the encryption process of Tiaoxin-346. As shown in Figure 4.3, the plaintext P is divided into blocks which are successively loaded into the internal state of each component. Additionally, the ciphertext blocks C are computed after loading each plaintext block. Suppose the plaintext P is divided into l_p blocks. Each of these plaintext blocks is composed of two words: $P^t = P_0^t || P_1^t$.

During the encryption phase the external inputs M_3, M_4 and M_6 are constructed using the plaintext words P_0^t, P_1^t. These are defined as: $M_3^t = P_0^t$, $M_4^t = P_1^t$ and $M_6^t = P_0^t \oplus P_1^t$.

At each round, the computation of the ciphertext block corresponding to each

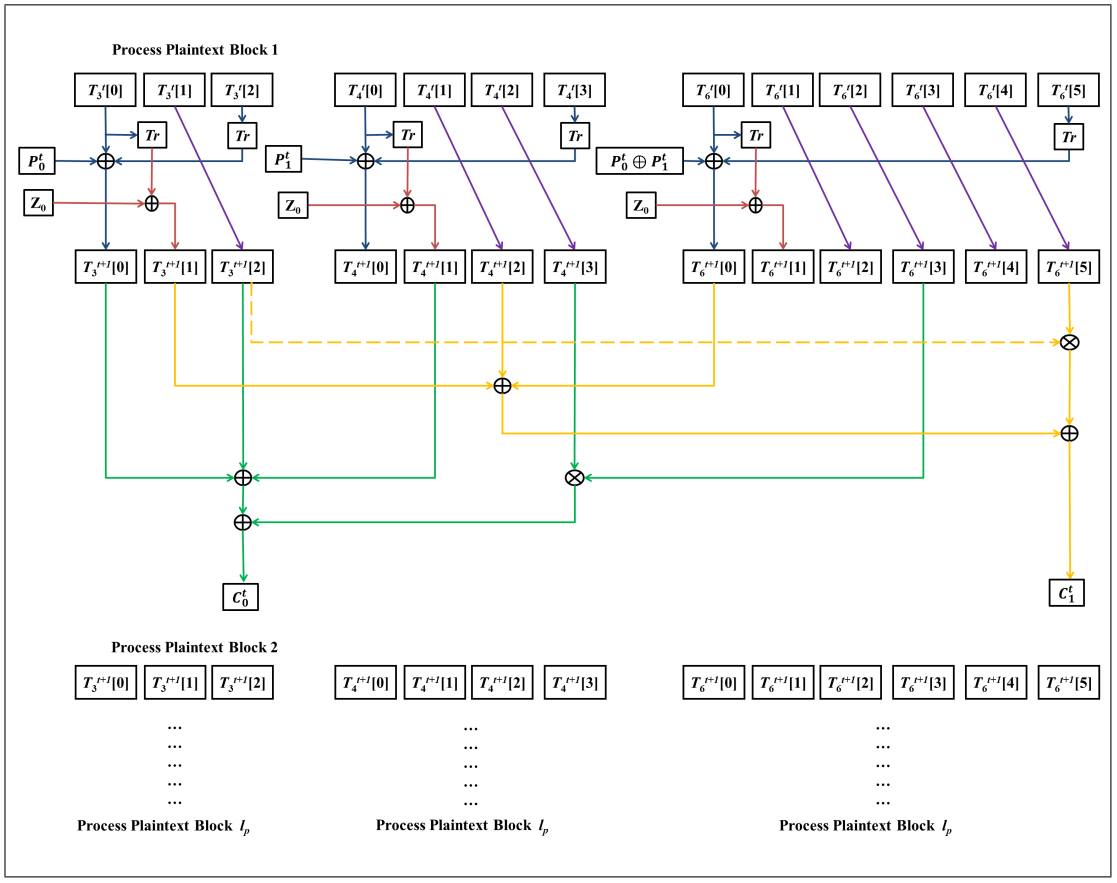

Figure 4.3: Tiaoxin-346 Encryption Procedure

plaintext block takes input from five specific words of the internal state. The internal state of Tiaoxin-346 is updated using $Update(T_3^t, T_4^t, T_6^t, P_0^t, P_1^t, P_0^t \oplus P_1^t)$ for l_p rounds, with each round processing one plaintext block. Each ciphertext block C^t is composed of two words; that is, $C^t = C_0^t || C_1^t$. At each round the ciphertext words of each block are computed as below:

$$C_0^t = T_3^{t+1}[0] \oplus T_3^{t+1}[2] \oplus T_4^{t+1}[1] \oplus (T_6^{t+1}[3] \otimes T_4^{t+1}[3]) \tag{4.2}$$

$$C_1^t = T_6^{t+1}[0] \oplus T_4^{t+1}[2] \oplus T_3^{t+1}[1] \oplus (T_6^{t+1}[5] \otimes T_3^{t+1}[2]) \tag{4.3}$$

4.2.2.4 Tag Generation

After all of the plaintext has been encrypted, the internal state of Tiaxon-346 is updated with some finalization steps to generate the authentication tag. During these finalization steps the state is updated 21 times without producing any output. For the first finalization round, the external inputs M_3, M_4 and M_6 are set to $adlen$, $msglen$ and $adlen \oplus msglen$, respectively. For the remaining 20

updates, the external inputs M_3, M_4 and M_6 are set to the constants Z_1, Z_0 and Z_1, respectively. After this, the tag τ is computed by XOR-ing the contents of all of the register stages. That is

$$\tau = T_3[0] \oplus T_3[1] \oplus T_3[2] \oplus T_4[0] \oplus T_4[1] \oplus T_4[2] \oplus T_4[3] \oplus T_6[0] \oplus T_6[1] \oplus T_6[2] \oplus T_6[3] \oplus T_6[4] \oplus T_6[5] \quad (4.4)$$

4.2.2.5 Decryption & Tag Verification

To carry out the decryption, first the initialization process is performed to obtain the initial state. The associated data processing phase is also performed if the length of associated data $adlen > 0$.

At each round of the decryption phase, one block of the ciphertext is decrypted as shown in Figure 4.4. During the processing of each ciphertext block, the internal state of Tiaoxin-346 is first updated using the state update function $Update(T_3^t, T_4^t, T_6^t, 0, 0, 0)$. Following this, the ciphertext block C^t is decrypted as illustrated in Equation 4.5 and Equation 4.6, recovering the two plaintext words $P^t = P_0^t || P_1^t$.

$$P_0^t = C_0^t \oplus T_3^{t+1}[0] \oplus T_3^{t+1}[2] \oplus T_4^{t+1}[1] \oplus (T_6^{t+1}[3] \otimes T_4^{t+1}[3]) \quad (4.5)$$

$$P_1^t = C_1^t \oplus T_6^{t+1}[0] \oplus T_4^{t+1}[2] \oplus T_3^{t+1}[1] \oplus (T_6^{t+1}[5] \otimes T_3^{t+1}[2]) \oplus P_0^t \quad (4.6)$$

The recovered plaintext block P^t is then XOR-ed with some specific state words. In particular, P_0^t, P_1^t and $P_0^t \oplus P_1^t$ are XOR-ed with the state words $T_3[0]$, $T_4[0]$ and $T_6[0]$, respectively. This process is continued l_c times to process all the ciphertext blocks, where l_c is the number of ciphertext blocks.

After processing all the ciphertext blocks, the tag is generated following the procedure discussed in Section 4.2.2.4. Finally for the tag verification, the received tag is compared with the tag generated after the decryption process. If the received tag is not the same as the tag computed after the decryption process then the tag verification fails, and the ciphertext and computed tag should not be released. Otherwise the tag verification process is considered successful.

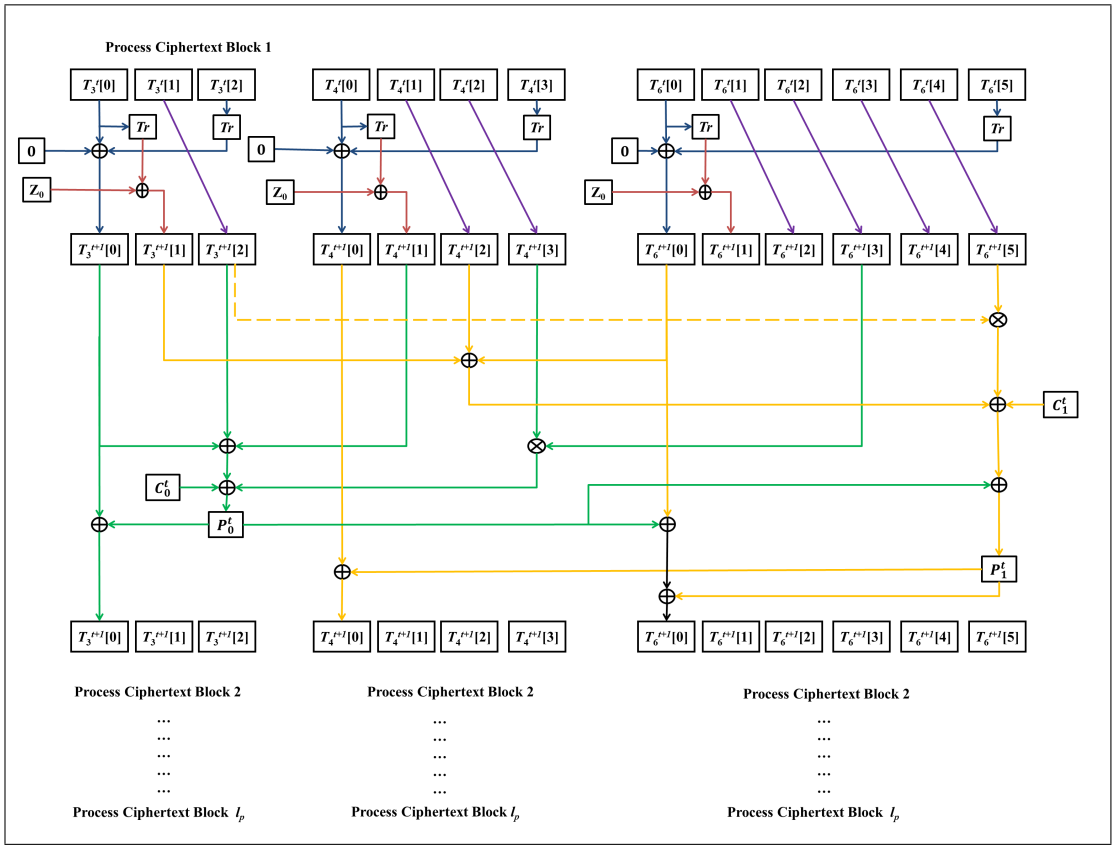

Figure 4.4: Tiaoxin-346 Decryption Procedure

4.3 Existing Analysis of Tiaoxin-346

There is little public analysis of Tiaoxin-346; to date, the only published analysis on Tiaoxin-346 is a differential fault analysis. This attack was presented by Dey et al. [38] in 2016, and requires 384 bit-flipping faults to be introduced in the state component $T_6[5]$ of Tiaoxin-346, to recover the 384-bit state of component T_3. The state of Tiaoxin-346 is invertible; that is, recovering the contents of any state T_3, T_4 or T_6 leads to recovery of the secret key. Therefore, the above attack implies the recovery of the 128-bit secret key with the 384 bit flipping faults.

This is a differential fault attack, which means the adversary needs to encrypt two sets of data using the same key and nonce (initialization vector). For one set the encryption is fault free, and for the other a fault is induced. The adversary observes both the correct and faulty ciphertexts. This falls under the nonce-reuse scenario, which the Tiaoxin-346 designer prohibits. This work strengthens the designer claim by showing that Tiaoxin-346 is not nonce misuse-resistant.

4.4 State Cycle Analysis of Tiaoxin-346

In this section, we analyse the state cycles for Tiaoxin-346. In particular we investigate the existence of short cycles in the internal state of Tiaoxin-346. A short cycle in the internal state means that the keystream generated will be repeated after a small number of steps. In such a case, multiple plaintexts may be encrypted with the same keystream which may be used by an adversary to apply a ciphertext only attack [148]. To avoid such an attack the length of the non-repeated keystream should be at least equal to the maximum message length supported by the cipher.

The loaded state of Tiaoxin-346 is formed by combining a 128-bit key with a 128-bit initialization vector. This gives a total of 2^{256} possible loaded states. Due to the high computational time it is infeasible to explore all of the 2^{256} loaded states. Hence we created a toy version of Tiaoxin-346 by reducing the size of the words in each component, and examined the cycle structure for the toy. The structure of the toy version is the same as the original version. The toy version for our analysis is described below.

- Each of the state words is reduced from 128 to 8 bits.
- The AES round transformation is replaced by a lookup table, i.e., each 8-bit input is replaced by an 8-bit output by using the corresponding values of the lookup table. We have used the AES S-box as our lookup table.
- Two randomly chosen 8-bit numbers are generated to replace the constant values Z_0 and Z_1.
- The key and initialization vector size are reduced to 8 bits. Thus we need to explore only 2^{16} loaded states.
- The remaining operations and procedure are the same as Tiaoxin-346.

Consider the three components T_3, T_4 and T_6 of the toy version of Tiaoxin-346 as a finite state machine (FSM) with internal state sizes of 24 bits, 32 bits and 48 bits, respectively. Let the state update function be the state transition function for this FSM. As the state size is finite, repeated application of the state transition function will eventually reach a point when the current state is one that has been visited previously. The number of transitions between visits to a such point is the period/cycle of the FSM.

A component has a maximum cycle if all possible states are visited before revisiting a state. For the toy version of Tiaoxin-346, components T_3, T_4 and T_6 can have maximum cycles of 2^{24}, 2^{32} and 2^{48}, respectively. Combining these three components the entire internal state of this toy version of Tiaoxin-346 can have maximum cycle no greater than 2^{104}.

We investigated the number of distinct cycles for each of the three components. All the three components of the cipher are updated independently. Therefore the state cycles of each of these components can be investigated individually. Section 4.4.1 discusses the experimental set-up and observations from the experiments on this toy version.

4.4.1 Experimental Procedure for State Cycle Analysis

From Section 4.2.2.1, observe that each of the components has a particular loaded format. The first three words of each component in a loaded state contain the key and initialization vector. The remaining words (if any) are set to predefined constants.

The toy version of Tiaoxin-346 has $2^8 = 256$ possible keys and $2^8 = 256$ possible initialization vectors. Therefore there are 2^{16} possible loaded states for each component. Let (K_l, V_l) denote the key and initialization vector pair, which is used with fixed constants to load the internal state of each component. Repeated application of the state update function will eventually return to the loaded state containing (K_l, V_l). This represents the completion of one cycle for the corresponding component. In our experiments, we aim to identify the cycle length for the loaded state formed by K_l and V_l.

We also aim to identify other loaded states which appear in the same cycle as the loaded state with the key and initialization vector pair (K_l, V_l). Let K_i and V_j denote the key and initialization vector pairs for $i \in \{0, \cdots, 255\}, j \in \{0, \cdots, 255\}$ and $(K_i, V_j) \neq (K_l, V_l)$, forming the rest of the possible loaded states. Within the cycle of the loaded state formed by K_l and V_l, we search for any intermediate states formed by K_i and V_j, which are in the loaded state format of the corresponding component. In our experiments, we also identify the distance between these loaded states.

In particular, we search for pairs of loaded states which have distance less than or equal to 15. If any such pair of loaded states exists, they will appear within the initialization phase of the cipher and will generate the same keystream up

Algorithm 4.1 Algorithm for Analysis of State Cycles

1: **Inputs:** Loaded state of a component formed by (K_l, V_l)
2: **Output:** Cycle length, intermediate state appearing within the cycle formed by (K_i, V_j) which are in the format of the loaded state, distance of these intermediate loaded states
3: Generate all the 2^{16} possible key-initialization vector pairs and store them in list L
4: Generate two random constants for Z_0 and Z_1
5: **if** List L is not empty **then**
6: Select a key-initialization vector pair (K_l, V_l) from list L
7: Perform the state update
8: **while** Current state is not equal to the loaded state formed by (K_l, V_l) **do**
9: **if** Current state formed by (K_i, V_j) are in the loaded state format **then**
10: Identify the current loaded state
11: Identify the distance from the previous loaded state
12: Delete (K_i, V_j) from L
13: **end if**
14: **end while**
15: Output the length of the cycle
16: Output the number of intermediate states which are in the loaded state format
17: **end if**

to a clock difference. These are called slid pairs. In our experiments, we identify these slid pairs appearing in each individual component.

All possible intermediate loaded states formed by (K_i, V_j) may not appear in the cycle formed by (K_l, V_l). We repeat the above experiments by choosing a different loaded state formed by key and initialization vector pairs (K_i, V_j) which have not been visited in any previous cycles of our experiments. This procedure is continued until all of the 2^{16} possible loaded state were visited. This will help to determine if there are any loaded states with a short cycle. Algorithm 4.1 provides a pseudo-code for the experimental procedure described above.

4.4.1.1 Observations on the State Cycle of T_3

We started the experiment by loading the state words of T_3 with the key and initialization vector set to $(0, 0)$. Two random 8 bits values were generated as the constant values for Z_0 and Z_1. The format for the loaded state of component T_3 requires the same values in the first two words. In our experiments we looked for such states which were in the cycle of a particular input set.

Table 4.1: Experimental Results for Component T_3

Loaded State K, K, V	Cycle Length	No. of Loaded States	Average Distance	Loaded States with Distance ≤ 15
(0, 0, 0)	4442236	17204	258	979
(0, 0, 1)	7996342	31343	255	1821
(0, 0, 4)	376440	1478	255	83
(0, 0, 6)	1231504	4855	254	296
(0, 0, 9)	2240162	8703	257	462
(0, 0, 91)	384193	1517	253	95
(1, 1, 12)	77292	329	235	21
(2, 2, 86)	15173	52	292	2
(7, 7, 41)	7281	32	228	0
(15, 15, 180)	4497	15	300	1
(31, 31, 24)	850	2	425	2
(41, 41, 221)	885	4	221	0
(150, 150, 74)	119	1	119	0
(252, 252, 15)	214	1	214	0
	16777188	65536		3762

Experimental analysis for the component T_3 of the toy version of Tiaoxin-346 shows that about 2^{14} different loaded states appear in the same cycle as the loaded state with the key and initialization vector pair $(0, 0)$. The distances between these loaded states were variable. The average distance between these loaded states was 258 steps. The minimum and maximum distance between these loaded states in this cycle were 1 and 3231 steps, respectively.

For the loaded state with the key and initialization vector pair $(0, 0)$, the starting state for component T_3 was repeated after about 2^{22} iterations. This indicates the the cycle length for this loaded state is approximately 2^{22}, whereas the maximum cycle is 2^{24}.

Note that there are total 2^{16} possible loaded states for this toy version, while only 2^{14} of them have been visited in the cycle formed by the key and initialization vector pair (0, 0). This indicates that not all the possible loaded states appeared in this cycle. We selected a loaded state that did not appear in the previous cycle and repeated the experiment. We continued with this approach until all 2^{16} loaded states had been visited. This revealed 14 distinct state cycles containing loaded states. Table 4.1 provides a summary of the experimental results for component T_3.

As shown in Table 4.1, all 2^{16} possible loaded states appeared within the cycles obtained from 14 distinct loaded states. These 14 distinct cycles covered $16,777,188$ possible states, whereas there were in total $2^{24} = 16,777,216$ possible

states for component T_3. This indicates that there are some internal states (28 for this experiment) which never appeared (impossible states) within the cycles of any of the loaded states for component T_3. These impossible states also have short cycles; however, the particular format of the loaded state ensures the avoidance of these states.

Experimental results on cycles for component T_3 also indicate that there are a number of loaded states which have cycles that are much shorter than the maximum possible length. For example, Table 4.1 shows that component T_3 has 436 loaded states with a state cycle less than 2^{16}, whereas the maximum possible state cycle length is 2^{24}. The experimental result also shows that a loaded state appears, on average, after every 258 cycles.

We also found $3,762$ loaded states for which another loaded state appeared within the initialization round of this toy version of Tiaoxin, i.e., the distance between these loaded states was less than or equal to 15. These were slid pairs for component T_3 of this toy version which generate identical states up to a clock difference during the initialization phase.

Note that the above experiment was conducted for two randomly chosen constants $Z_0 = 70$ and $Z_1 = 63$. We have performed experiments for several other randomly chosen constants. The constant value appears to affect the cycle structure of the states of Tiaoxin-346, and hence the number of distinct cycles in which loaded states appear. For each of these experiments we found that there were some loaded states which lie on cycles with a much shorter length. Our experiments also indicated that there are some states which never appear within any cycle containing loaded states.

4.4.1.2 Observations on the State Cycle of T_4

We performed similar experiments for component T_4 of the toy version. We started the experiment by loading the state words of T_4 by setting the key and initialization vector to $(0, 0)$. For this experiment we used the same randomly chosen constant constant values Z_0 and Z_1, which were used for the experiments on component T_3. The format for the loaded state of component T_4 requires the same values in the first two words of this component, and the constant Z_0 in the last word of component T_4. After loading, the state update function was applied repeatedly until the loaded state recurred, to determine other loaded states which are in the cycle formed by a particular input set. Other loaded states on this

cycle were noted as was the distance between them.

For the loaded state with the key and initialization vector pair $(0, 0)$, the starting state for T_4 was repeated after about $2^{30.5}$ iterations. That is, the the cycle length for this loaded state is about $2^{30.5}$, whereas the maximum possible cycle length is 2^{32}.

Analysis for the component T_4 of the toy version of Tiaoxin-346 shows that about $2^{14.5}$ different loaded states appeared in the same cycle as the loaded state with key and initialization vector pair $(0, 0)$. The distances between these loaded state was variable. The average distance between these loaded states was 6944 steps.

Note that there are 2^{16} possible loaded states for this toy version of Tiaoxin-346, but only $2^{14.5}$ of them had been visited in the cycle formed by the key and initialization vector pair (0, 0). This indicates that not all the possible loaded states appeared in this cycle. We selected a loaded state that did not appear in the previous cycle and repeated the experiment. We continued with this approach until all 2^{16} loaded states had been visited. Table 4.2 provides a summary of the experimental results for component T_4.

As shown in Table 4.2, all 2^{16} possible loaded states appeared within 15 distinct cycles. These 15 distinct cycles covered $4,294,923,769$ states, however; the total number of possible states for component T_4 was $2^{32} = 4,294,967,296$. This indicates that there are some internal states (43,527 states for this experiment) which never appeared within a cycle containing a loaded state. These impossible states also had short cycles; however, the particular format of the loaded state ensures that these states were avoided during the initialization phase. Experimental analysis for component T_4 shows that there are a number of loaded states which have cycles that are much shorter than the maximum possible length.

Also note that 21 loaded states were found for which another loaded state appears within the initialization round of Tiaoxin-346 toy version. That is, the distance between two loaded states were less than or equal to 15. These are slid pairs for component T_4 of this toy version which generated identical states up to a clock difference during the initialization phase. Recall that for the experiment on component T_3 of the toy version, $3,762$ slid pairs were found during the initialization phase. However, the slid pairs found for component T_4 did not have any common pairs with those found for component T_3.

Experiments were also performed to determine the state cycle of component

Table 4.2: Experimental Results for Component T_4

Loaded State (K, K, V, Z_0)	Cycle Length	No. of Loaded States	Average Distance	Loaded States with Distance ≤ 15
(0, 0, 0, 70)	1475144246	22714	64944	6
(0, 0, 1, 70)	1112274121	16729	66481	3
(0, 0, 3, 70)	28147327	375	75304	1
(0, 0, 4, 70)	1526994855	23382	65306	8
(0, 0, 31, 70)	4949946	72	67736	0
(0, 0, 33, 70)	128610	2	60004	1
(0, 0, 39, 70)	97579588	1505	64808	1
(0, 0, 49, 70)	18188496	290	63075	0
(1, 1, 104, 70)	280267	6	43534	1
(1, 1, 196, 70)	18737023	277	67494	0
(4, 4, 138, 70)	1465480	22	69507	0
(4, 4, 175, 70)	10258254	149	68603	0
(19, 19, 136, 70)	390683	6	76362	0
(21, 21, 130, 70)	353614	6	51497	0
(92, 92, 30, 70)	31259	1	31259	0
	4294923769	65536		21

T_4 with loaded states that resulted in a comparatively short cycle for component T_3. None of these loaded state results in a relatively shorter state cycle in component T_4.

4.4.1.3 Observations on the State Cycle of T_6

The format for the loaded state of component T_6 requires the same values in the first two words and the constants Z_1, 0 and 0 in the last three words of this component, respectively. Again experiments were performed to search for the existence of loaded states, which are in a particular input cycle. For this experiment, we used the same random constants which were used for the experiments on components T_3 and T_4.

It seems that the experiments for the state cycle analysis of T_6 requires significant time. We could not finish the experiment after running for a few weeks in the desktop computing environment. This is because state T_6 of the toy version of Tiaoxin-346 has a maximum possible cycle of 2^{48}, which seems to be too much computation in our experimental environment.

4.4.2 Summary of the State Cycle Analysis

We investigated the cycles of the internal states of each of the three components of the toy version of Tiaoxin-346, individually. For components T_3 and T_4 the largest cycles containing the loaded states were found to be of length 2^{23} and $2^{30.5}$, respectively. This shows that neither of these components achieves the maximum possible cycle lengths of 2^{24} and 2^{32}, respectively. For component T_6, we could not finish the experiment because of the high time complexity. It is likely that component T_6 will not reach to the maximum cycle.

In our investigation, we found some loaded states for components T_3 and T_4 of the toy version of Tiaoxin-346 which resulted in much shorter cycles than the maximum possible cycle. It is likely that component T_6 will also have some loaded states which have much shorter cycles.

Experimental results indicate that the loaded states resulting in relatively short cycles in component T_3 do not result in any relatively short cycles in component T_4. If the loaded state were resulting in the same short cycle for two components or if they have a small least common multiple, then that is the period for these two components, when combined. Otherwise, the period of these two components can be determined by finding the least common multiple of the periods of each individual components. A similar comment applies for combining these cycles with those for the component T_6.

We did not find any loaded states which resulted in a relatively short cycle for all the three components. Therefore, a shorter cycle in any specific component might not result in a shorter cycle for the entire state of the toy version of Tiaoxin-346. That is, the cycle for the entire state of Tiaoxin-346 is obtained by finding least common multiple of the cycles for each individual component. Experimental results on the toy version of Tiaoxin-346 internal state did not result in any relatively shorter cycle which is smaller than the maximum supported plaintext length. It seems shorter cycle in any individual component does not pose a threat to the cycle of the entire state of Tiaoxin-346 toy version.

The structure of the toy version and the original version of Tiaoxin-346 is the same. We can not infer the exact results for Tiaoxin-346 from the toy version, but given the similarity in the structures, it is reasonable to assume that the larger components will demonstrate similar properties as of the toy version. Based on the analysis of the toy version it is likely that the original version of Tiaoxin-346 will not have any comparatively short cycles which are smaller than its maximum

supported message length. If there are no such short cycles, then the keystream sequence produced by Tiaoxin-346 will not be repetitive and provide resistance against attacks such as automated ciphertext only attack [148].

4.5 Cube Attack on Tiaoxin-346

This section provides a description of the application of a cube attack on the authenticated encryption stream cipher Tiaoxin-346. The attack can be performed either in the initialization phase, associated data processing phase or the encryption phase. In this section we discuss the applicability of cube attacks on the initialization and encryption phases of Tiaoxin-346.

Recall that Tiaoxin-346 uses the AES round function as a part of its state update. Dinur and Shamir's work [149] claims that a cube of size 27 is required when the input state goes through two AES rounds. This makes it difficult to analyse the cube attack on the full round version of Tiaoxin-346. We conducted experiments to analyse the feasibility of cube attacks on reduced round version of Tiaoxin-346.

4.5.1 Cube Attack during the Initialization Phase

In the initialization phase, the key and initialization vector are loaded in to the internal state of Tiaoxin-346. In general, the cube can be selected either from the input key or the initialization vector. However, an adversary needs to have the ability to manipulate the key which is not a very realistic assumption, if the cube bits are chosen from the input key.

We consider the scenario where the cube is chosen from the initialization vector set. This requires the preprocessing phase to identify suitable cube bits chosen from the input initialization vector, which generate linear equations in terms of the key bits. Each of the linear equations is computed by summing the output function of Tiaoxin-346 for all the possible values of the corresponding cube. Therefore in the online phase, an adversary first needs to compute the value of these equations for the corresponding cube. This follows the chosen plaintext model, where an adversary encrypts the plaintext with the key and chosen initialization vectors (varying the cube bits obtained from the preprocessing phase) and sums the output bits over n-dimensional Boolean cube to compute the value of the corresponding equation. The secret key can be recovered by

Table 4.3: Example of Cubes Obtained for 4 Round Tiaoxin-346

Cube Indices	Output Index	Cube Size
46, 126, 6	34	3
121, 41, 81	57	3
86, 46, 6	31	3
59, 67, 19	52	3
26, 114, 74	44	3
118, 30, 38	29	3
22, 110, 70	30	3

solving these equations if sufficient number of linearly independent equations are generated.

4.5.1.1 Application of the Cube Attack

We applied the cube attack to a reduced round version of Tiaoxin-346, where the number of initialization rounds is reduced from 15 to 4. We performed the preprocessing phase by conducting experiments to find cube variables from the initialization vector bits. We started our experiment with a cube of size 2. We selected over 5000 random cubes, where each of them is evaluated with 100 linearity tests. However, none of those cubes passed the linearity test. We then increased the cube size and repeated the test. We selected 5000 random cubes for the selected cube size and tested each cube using 100 linearity tests. For cubes of size 3, we found seven cubes which passed the linearity test; however, none of these cubes resulted in a non-constant linear superpoly.

In our experiment, we tested up to cube size of 20. We did not find any cubes resulting in non-constant superpolys; but for the cube size 3 to 20, we found cubes which result in constant superpolys. These cubes can be used as a distinguisher for 4 round Tiaoxin-346. Some examples of cubes of size three are listed in Table 4.3. The cube indices in Table 4.3refers to the corresponding bit positions of the initialization vector, whereas the output index refer to the corresponding bit position of the first output block of Tiaoxin-346.

In the online phase of the attack, an adversary can select any of the cubes from Table 4.3 and observe the sum of the output bits over all the possible values of that cube. This sum will be equal to the constant determined in the preprocessing phase; which the adversary can use to distinguish the output of Tiaoxin-346 from the output of a random function.

The smallest size of the cube obtained for the reduced round Tiaoxin is 3. Therefore for cube size three, the adversary needs to observe specific output bit for 2^3 chosen initialization vectors. The best cubes for 4 round Tiaoxin can distinguish the cipher output from random with a complexity of 2^3.

4.5.2 Cube Attack during the Encryption Phase

This section provides our observations on the application of cube attacks on Tiaoxin-346 encryption phase. These observations work under the assumption that an attacker can manipulate the state words and select the cube bits from the internal state variables. Note that a similar independent work for state recovery (based on differential fault injection) was introduced by Dey et al [38]. In the following, we describe the state contents recovery for component T_3 under the assumption that adversary can manipulate the state bits.

As described in Section 4.2.2.3, at each iteration of the encryption phase the output ciphertext block $(C_0^t||C_1^t)$ can be computed using Equation 4.2 and Equation 4.3. Each ciphertext block consist of two words of length 128 bits. Let $C_{0,i}^t$, $C_{1,i}^t$ and $T_{s,i}^t$ denote the i^{th} bit of ciphertext word C_0^t, C_1^t and state word T_s, respectively, where $0 \leq i < 128$. Following Equation 4.2 and Equation 4.3 for the ciphertext block C^t composed of C_0^t and C_1^t, the i^{th} bit of the ciphertext can be represented as

$$C_{0,i}^t = T_{3,i}^{t+1}[0] \oplus T_{3,i}^{t+1}[2] \oplus T_{4,i}^{t+1}[1] \oplus (T_{6,i}^{t+1}[3] \otimes T_{4,i}^{t+1}[3]) \tag{4.7}$$

$$C_{1,i}^t = T_{6,i}^{t+1}[0] \oplus T_{4,i}^{t+1}[2] \oplus T_{3,i}^{t+1}[1] \oplus (T_{6,i}^{t+1}[5] \otimes T_{3,i}^{t+1}[2]) \tag{4.8}$$

We observe that differentiating Equation 4.8 with respect to $T_{6,i}^{t+1}[5]$ results in a linear equation. This means $T_{6,i}^{t+1}[5]$ can be used as a cube bit to obtain a linear equation. Therefore summing the i^{th} bit of output ciphertext bit $C_{1,i}^t$ over all the possible value of cube $T_{6,i}^{t+1}[5]$ will result in a linear equation which reveals the i^{th} bit of state word $T_{3,i}^{t+1}[2]$. Repeating this process for $i = 0, \cdots, 127$ will reveal the content of state word $T_3^{t+1}[2]$. We can extend the same procedure to recover the state word $T_3^{t+2}[2]$ and $T_3^{t+3}[2]$. Note that $T_3^{t+2}[2] = T_3^{t+1}[1]$. Also $T_3^{t+3}[2] = T_3^{t+2}[1]$, which can be used to recover the state word $T_3^{t+1}[0]$. This allows to recover all the contents of component T_3 at time $t+1$. Recovering the internal state of component T_3 of Tiaoxin leads to the recovery of the secret key by inverting the internal states up to the beginning of the initialization phase.

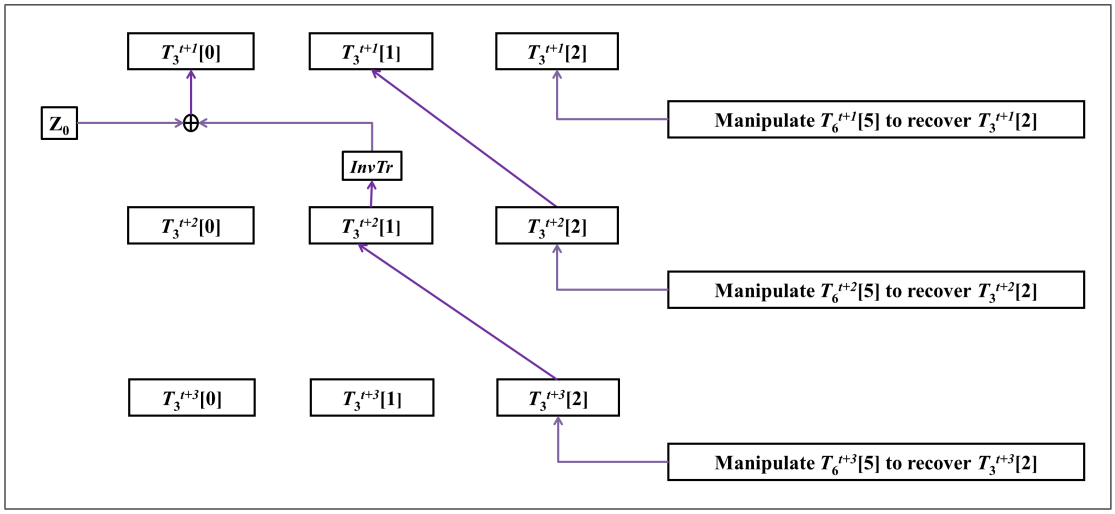

Figure 4.5: Recovering the Contents of T_3 by Manipulating the Contents of State Word $T_6[3]$

This process is shown in figure 4.5. A similar process can be also applied to Equation 4.7 by differentiating it with respect to $T_{6,i}^{t+1}[3]$.

4.5.3 Summary of Cube Attacks on Tiaoxin-346

We applied the cube attack to reduced versions of Tiaoxin-346, where the modification is a reduction in the number of rounds of the initialization phase from 15 to 4 rounds. For the reduced version of Tiaoxin-346, we did not find any cubes resulting in linear equation. This is due to the high algebraic degree of the AES round function used in the Tiaoxin-346 state update function.

For the reduced round version of Tiaoxin-346 many cubes were found which result in a constant superpoly. These cubes can be used to distinguish the output of this reduced version of Tiaoxin-346 from randomly generated output. The complexity of constructing a distinguisher for 4-round Tiaoxin-346 is 2^3.

The best cubes identified for reduced version of Tiaoxin-346 are of size 3; while the actual degree of the output equation after 4 rounds of Tiaoxin-346 is much higher than 3. This means after 4 rounds of the initialization phase there exist comparatively lower degree monomials in the output equation which do not appear together with any other monomials of that equation. This indicates that the key and initialization vectors are not mixed properly at this point of the initialization phase.

For more than four rounds of the initialization of Tiaoxin-346 we continued

the experiment to test cubes up to size 20, but we did not find any suitable cube which generates a linear superpoly. It is possible that there may exist some large sized cubes which can be applied to more than four rounds of the initialization of Tiaoxin-346, with a complexity less than that of exhaustive key search. However, for our attack the cubes are identified experimentally and thus it is difficult to extrapolate the complexity for larger cube size beyond the experimental range.

We have also provided some observation on the application of cube attack during the encryption phase of Tiaoxin-346. The observation made here requires the capability to inject differences in the internal state of Tiaoxin-346. These differences can be injected using multi-byte faults. This also requires an adversary to encrypt multiple sets of input using the same key and initialization vector, for which the designer of Tiaoxin-346 does not claim any security. A similar observation was also presented by Dey et. al. [38].

Constructing a distinguisher for the 4-round initialization of Tiaoxin-346 is the best result obtained using the cube attack. To date this is the best result obtained on a reduced version of Tiaoxin-346. Based on our analysis, Tiaoxin-346 is currently secure against the cube attack if the full initialization is performed.

4.6 Fault Attack on Tiaoxin-346

A fault attack [102] is a type of side-channel attack which works on the physical implementation of a cryptographic algorithm. The most common type of fault attack is the differential fault attack (DFA). In the differential fault attack, the adversary introduces some error(s) in the underlying implementation of the cryptographic primitive, and then tries to recover the secret key of the cryptosystem by observing the original and faulty output. In this section we describe two different attacks based on fault insertion in the internal state of Tiaoxin-346.

The first technique described in Section 4.6.1 uses the bit-flipping fault model to apply a forgery attack on Tiaoxin-346. This forgery attack only requires access to the physical implementation of the algorithm at the sender's side.

The second technique described in Section 4.6.2 uses the differential fault injection in the internal state to recover the contents of component T_6 in Tiaoxin-346. Recall that the components in Tiaoxin-346 are independent of each other; that is, recovering the contents of any component leads to the recovery of the secret key by inverting the internal states up to the beginning of the initialization

phase. Thus, this state recovery of component T_6 can be used to recover the secret key of Tiaoxin-346. This differential fault attack can work with the random fault model. It requires an adversary to observe both the original and the faulty ciphertext computed using the same key and initialization vector. Therefore, this attack works under the nonce-misuse scenario.

4.6.1 Fault based Forgery Attack on Tiaoxin-346

This section discusses a fault based forgery attack on Tiaoxin-346. The goal of the attack is to modify the input message by flipping specific message bit(s) and to have the modified message accepted as legitimate at the receiver side. To have the modified message accepted, the adversary needs to apply bit flipping faults in the internal state of the sender's device. The attack requires access to the physical implementation of the algorithm at the sender's side.

As discussed in Chapter 3, a fault based forgery attack on the authenticated encryption ciphers CLOC and SILC was introduced by Roy et al. [146]. In the CAESAR Google discussion forum it was pointed out by Iwata et al. [147] that any authenticated encryption scheme can be forged using faulty encryption queries. This requires an adversary to inject faults in the inputs submitted to the encryption oracle and then use the output of the encryption oracle with the faulty inputs to continue with the forgery. We describe here a similar fault injected forgery on Tiaoxin-346; however, the faults are injected into the internal state instead of the inputs.

The generic fault based forgery attack by Iwata et al. [147] applies a specific fault in the message before the message is loaded in to the device. Their attack requires the attacker to enquire about the faulty ciphertext and the faulty tag for a faulty (modified) message. However, in our attack the faults are applied in the encryption device after the message is loaded. This attack may be more practical in some applications where the attacker is able to access the encryption device, rather than requiring the attacker to intercept and alter messages being sent to the device. The forgery attack presented here is the first proposal of such kind. The particular details of this attack on Tiaoxin-346 are discussed below.

In the associated data processing and encryption phases of Tiaoxin-346, the input associated data and the plaintext are loaded into specific state components. Therefore, any changes in the input associated data or the plaintext will affect the respective state component. An adversary can apply a forgery attack by

injecting faults into the state components of Tiaoxin-346 to reflect the intended changes in the associated data or in the plaintext.

We use the term M to represent either the associated data or the plaintext. Suppose the message block M^t is to be sent and the adversary wants the received message block M'^t to be modified such that $M'^t_0 = M^t_0 \oplus e_i$, where e_i is the word which has a single 1 in the i^{th} bit position. Note that the input message word M^t_0 is processed at the sender's side by XOR-ing it with the contents of $T_3[0]$ and $T_6[0]$. To ensure that the correct tag for the modified message is calculated at the receiver's side, the i^{th} bit of $T^{t+1}_3[0]$ and $T^{t+1}_6[0]$ must therefore be modified at the sender's side by introducing bit-flipping faults at these locations. The faulty state words will then be $T'^{t+1}_3[0] = T^{t+1}_3[0] \oplus e_i$ and $T'^{t+1}_6[0] = T^{t+1}_6[0] \oplus e_i$. Note that these changes are equivalent to the internal state values that would occur if the i^{th} bit of the original input message word M^t_0 had been flipped and no faults were applied.

If M^t is a plaintext block, flipping $T^{t+1}_3[0]$ and $T^{t+1}_6[0]$ will also affect both of the ciphertext words C^t_0 and C^t_1, since $T^{t+1}_3[0]$ and $T^{t+1}_6[0]$ are both used for computation of these ciphertext words. Applying the Tiaoxin-346 output function given in Equation 4.2 and Equation 4.3, the faulty ciphertext words C'^t_0 and C'^t_1 can be written as:

$$C'^t_0 = T'^{t+1}_3[0] \oplus T^{t+1}_3[2] \oplus T^{t+1}_4[1] \oplus (T^{t+1}_6[3] \otimes T^{t+1}_4[3]) \tag{4.9}$$

$$C'^t_1 = T'^{t+1}_6[0] \oplus T^{t+1}_4[2] \oplus T^{t+1}_3[1] \oplus (T^{t+1}_6[5] \otimes T^{t+1}_3[2]) \tag{4.10}$$

Let $C'^t = C'^t_0 || C'^t_1$ and τ' denote the ciphertext and the tag generated from the faulty state. The faulty ciphertext C'^t and faulty tag τ' are sent to the receiver. In the decryption and tag verification phase, the receiver will XOR the received/recovered message block with the contents of $T_3[0]$ and $T_6[0]$. The recovered plaintext at the receiver is computed as shown in Equation 4.11 and Equation 4.12, respectively.

$$\begin{aligned} P'^t_0 &= C'^t_0 \oplus T^{t+1}_3[0] \oplus T^{t+1}_3[2] \oplus T^{t+1}_4[1] \oplus (T^{t+1}_6[3] \otimes T^{t+1}_4[3]) \\ &= P^t_0 \oplus e_i \end{aligned} \tag{4.11}$$

$$\begin{aligned} P'^t_1 &= C'^t_1 \oplus T^{t+1}_6[0] \oplus T^{t+1}_4[2] \oplus T^{t+1}_3[1] \oplus (T^{t+1}_6[5] \otimes T^{t+1}_3[2]) \oplus P'^t_0 \\ &= P^t_1 \oplus P^t_0 \oplus e_i \oplus P^t_0 \oplus e_i \\ &= P^t_1 \end{aligned} \tag{4.12}$$

Equation 4.12 shows that the recovered plaintext word $P_1'^t$ is as the sender intended; however, Equation 4.11 shows a complementation in the i^{th} bit of the recovered plaintext word $P_0'^t$. This recovered message is then XOR-ed with the contents of register stages $T_3^{t+1}[0]$ and $T_6^{t+1}[0]$ at the receiver's device. Therefore the modified plaintext word $P_0'^t$ will be introduced into the state, resulting in complementation of the i^{th} bit of register stage $T_3[0]$ and $T_6[0]$. The state contents are now equivalent to the state that resulted after applying the bit-flipping faults at the sender side.

Once the ciphertext is processed, the finalization process of Tiaoxin-346 at the receiver side generates the tag τ''. Clearly, the tag τ'' generated at the receiver device is the same as the tag τ' sent by the sender, since both are computed from the same state contents.

Alternatively, if M^t is an associated data block (which will not be encrypted), then in addition to inserting the faults in the sender device, the adversary also needs to intercept the transmitted message, modify the i^{th} bit of the associated data block, and send this modified associated data along with the faulty tag τ'.

In the decryption and tag verification phase, the receiver will XOR the received modified associated data block with the contents of $T_3[0]$ and $T_6[0]$. As previously, once the modified associated data word is introduced into the state, the i^{th} bit of register stages $T_3[0]$ and $T_6[0]$ are complemented. The state contents are now equivalent to the state that resulted after applying the bit-flipping faults at the sender side, and the tag τ'' generated at the receiver device is the same as the tag τ' sent by the sender.

Therefore in both cases the modified message and the faulty tag τ' will be accepted as legitimate by the receiver. The total number of faults required for this forgery attack are twice the number of bits flipped in the original message. Note that the adversary does not need to apply any faults at the receiver's side.

A similar attack applies if the adversary wants to modify bit(s) in the input message word M_1^t. In that case, the modified bit(s) will affect the contents of $T_4[0]$ and $T_6[0]$, and therefore the locations of the bit-flipping faults need to be changed accordingly.

4.6.1.1 Attack Algorithm for Fault Based Forgery on Tiaoxin-346

The steps involved in the application of our fault based forgery attack on Tiaoxin-346 are outlined in Algorithm 4.2.

Algorithm 4.2 Algorithm for Fault Based Forgery Attack on Tiaoxin-346

1: Insert bit-flipping faults in the i^{th} bit of state element $T_3^{t+1}[0]$ or $T_4^{t+1}[0]$ at the sender device. The fault is injected in $T_3^{t+1}[0]$ if the goal is to modify M_0^t. Alternatively, the fault is injected in $T_4^{t+1}[0]$ if the goal is to modify M_1^t.
2: Insert bit-flipping faults in the i^{th} bit of state element $T_6^{t+1}[0]$ at the sender device.
3: If the goal is associated data modification, then during the transmission complement the i^{th} bit of the associated data from D^t to D'^t.

The attacker's actions all occur before the message is received. If the remainder of the processing is carried out as usual, the modified message M'^t will be accepted by the receiver.

4.6.2 Fault based Key Recovery Attack on Tiaoxin-346

The fault based key recovery attack described in this section can be applied during the encryption phase of Tiaoxin-346. This attack requires an adversary to observe multiple faulty and fault free ciphertext pairs which are encrypted using the same key and initialization vector.

We start our fault based key recovery attack during the first round of the encryption phase of Tiaoxin-346. Suppose T_3^t, T_4^t and T_6^t denote the internal state obtained after the initialization and associated data loading phases of Tiaoxin-346 are completed. Recall from Section 4.2.2.3, Tiaoxin-346 outputs two ciphertext words during each round of the encryption phase. These ciphertext words C_0^t and C_1^t are computed as shown in Equation 4.2 and Equation 4.3.

From Equation 4.2 and Equation 4.3, observe that both of these functions have a common input: the state word $T_3^{t+1}[2]$ is XOR-ed in the computation of C_0^t (as shown in Equation 4.2), and AND-ed with the state word $T_6^{t+1}[5]$ in the computation of C_1^t (as shown in Equation 4.3). This can be exploited by an adversary; inserting faults in $T_3^{t+1}[2]$ permits recovery of the internal state word $T_6^{t+1}[5]$. We first show how this can be done with bit-flipping faults and then relax our assumption to achieve the same outcome using random faults. This is a more realistic assumption for fault attacks than a bit-flipping model, since it requires less precise control of the fault outcome by the attacker.

Using bit-flipping faults. Suppose an adversary introduces fault e in state word $T_3^{t+1}[2]$ which complements all of its contents. That is, $e = 111...111$ and the faulty state word $T_3'^{t+1}[2]$ is defined as $T_3'^{t+1}[2] = T_3^{t+1}[2] \oplus e = \overline{T_3^{t+1}[2]}$, where

$\overline{T_3^{t+1}[2]}$ denotes the complement of $T_3^{t+1}[2]$. The fault free and faulty ciphertext words C_1^t and $C_1'^t$ are defined in Equation 4.3 and Equation 4.13.

$$C_1^t = T_6^{t+1}[0] \oplus T_4^{t+1}[2] \oplus T_3^{t+1}[1] \oplus (T_6^{t+1}[5] \otimes T_3^{t+1}[2]) \tag{4.3}$$

$$C_1'^t = T_6^{t+1}[0] \oplus T_4^{t+1}[2] \oplus T_3^{t+1}[1] \oplus (T_6^{t+1}[5] \otimes \overline{T_3^{t+1}[2]}) \tag{4.13}$$

In this case, XOR-ing Equation 4.3 and Equation 4.13 allows the adversary to recover the contents of state word $T_6^{t+1}[5]$, as shown in Equation 4.14.

$$\begin{aligned} C_1^t \oplus C_1'^t &= (T_6^{t+1}[5] \otimes T_3^{t+1}[2]) \oplus (T_6^{t+1}[5] \otimes \overline{T_3^{t+1}[2]}) \\ &= T_6^{t+1}[5] \otimes (T_3^{t+1}[2] \oplus \overline{T_3^{t+1}[2]}) \\ &= T_6^{t+1}[5] \end{aligned} \tag{4.14}$$

In Equation 4.14, the adversary has access to the ciphertext pair $(C_1^t, C_1'^t)$. Thus with the bit-flipping faults in $T_3^{t+1}[2]$ an adversary can recover the contents of $T_6^{t+1}[5]$. Repeating the bit flipping six times (for $T_3^{t+1}[2]$, $T_3^{t+2}[2]$, $T_3^{t+3}[2]$, $T_3^{t+4}[2]$, $T_3^{t+5}[2]$, and $T_3^{t+6}[2]$) and making use of the fact that $T_6^{t+1}[i] = T_6^t[i-1] =$ for $i = 2, \cdots, 5$, the whole contents of T_6^{t+1} can be recovered. The attack described by Dey et.al. [38] used this idea to recover the contents of the smaller component T_3. This requires fewer faults than recovering the contents of T_6. However, we can adapt our attack targeting T_6 to use random faults rather than bit-flipping. This adaptation cannot be applied to T_3.

Using random faults. We describe an adaptation of the above technique which recovers the internal state word $T_6^{t+1}[5]$ with random faults; this is a more realistic assumption than the stringent requirements on the attacker's capabilities for bit-flipping fault attacks. Now, suppose the fault e is a randomly generated value and the adversary does not have any knowledge of the specific fault value. Let the random fault e affect the contents of $T_3^{t+1}[2]$ such that the faulty state word $T_3'^{t+1}[2] = T_3^{t+1}[2] \oplus e$. In this case, the fault free and faulty ciphertext words C_0^t and $C_0'^t$ are shown in Equation 4.2 and Equation 4.15.

$$C_0^t = T_3^{t+1}[0] \oplus T_3^{t+1}[2] \oplus T_4^{t+1}[1] \oplus (T_6^{t+1}[3] \otimes T_4^{t+1}[3]) \tag{4.2}$$

$$C_0'^t = T_3^{t+1}[0] \oplus T_3'^{t+1}[2] \oplus T_4^{t+1}[1] \oplus (T_6^{t+1}[3] \otimes T_4^{t+1}[3]) \tag{4.15}$$

We observe that the adversary can obtain the value of the random error e by

simply XOR-ing Equation 4.2 and Equation 4.15, as shown in Equation 4.16.

$$\begin{aligned} C_0^t \oplus C_0'^t &= T_3^{t+1}[2] \oplus T_3'^{t+1}[2] \\ &= T_3^{t+1}[2] \oplus T_3^{t+1}[2] \oplus e \\ &= e \end{aligned} \tag{4.16}$$

In Equation 4.16, the adversary has access to the ciphertext pair $(C_0^t, C_0'^t)$. Thus from the faulty and fault free ciphertext alone, the adversary can obtain the value of the random error introduced in state word $T_3^{t+1}[2]$.

We also observe that replacing $\overline{T_3^{t+1}[2]}$ by $T_3^{t+1}[2] \oplus e$, Equation 4.14 becomes

$$C_1^t \oplus C_1'^t = T_6^{t+1}[5] \otimes e \tag{4.17}$$

This allows us to use the random fault value e to find the value of bits in $T_6^{t+1}[5]$. Note that for bit positions in e with value 0, the bits in the corresponding position in $T_3^{t+1}[2]$ and $T_3'^{t+1}[2]$ will have the same value, and for bit positions in e with value 1, the bits in the corresponding position in $T_3^{t+1}[2]$ and $T_3'^{t+1}[2]$ will be complementary.

Unlike the bit-flipping fault attack described by Dey et. al.[38], our approach using random faults does not necessarily recover all of the bits in $T_6^{t+1}[5]$ with a single fault. A single random fault recovers only the bits in $T_6^{t+1}[5]$ where the corresponding bits of e have a value of one. Thus we need to perform the random fault injection multiple times until all the bits in $T_6^{t+1}[5]$ are recovered.

The same process can be used to recover the state words $T_6^{t+2}[5]$, $T_6^{t+3}[5]$, $T_6^{t+4}[5]$, $T_6^{t+5}[5]$, and $T_6^{t+6}[5]$. As outlined for the bit-flipping case, this is equivalent to recovering all of the contents of T_6^{t+1}. The process for recovering the contents of T_6 with these random faults is shown in Figure 4.6.

Recall that the state component of Tiaoxin-346 is invertible; that is, recovering the contents of any state component T_3, T_4 or T_6 leads to the recovery of the secret key. Therefore following the recovery of T_6, the secret key can be recovered by inverting the internal state T_6^{t+1} up to the beginning of the initialization phase. Then the key can be obtained from $T_6^{t+1}[0]$; similarly, the initialization vector can be obtained from $T_6^{t+1}[3]$; since this value is public, this provides a confirmation of a successful attack.

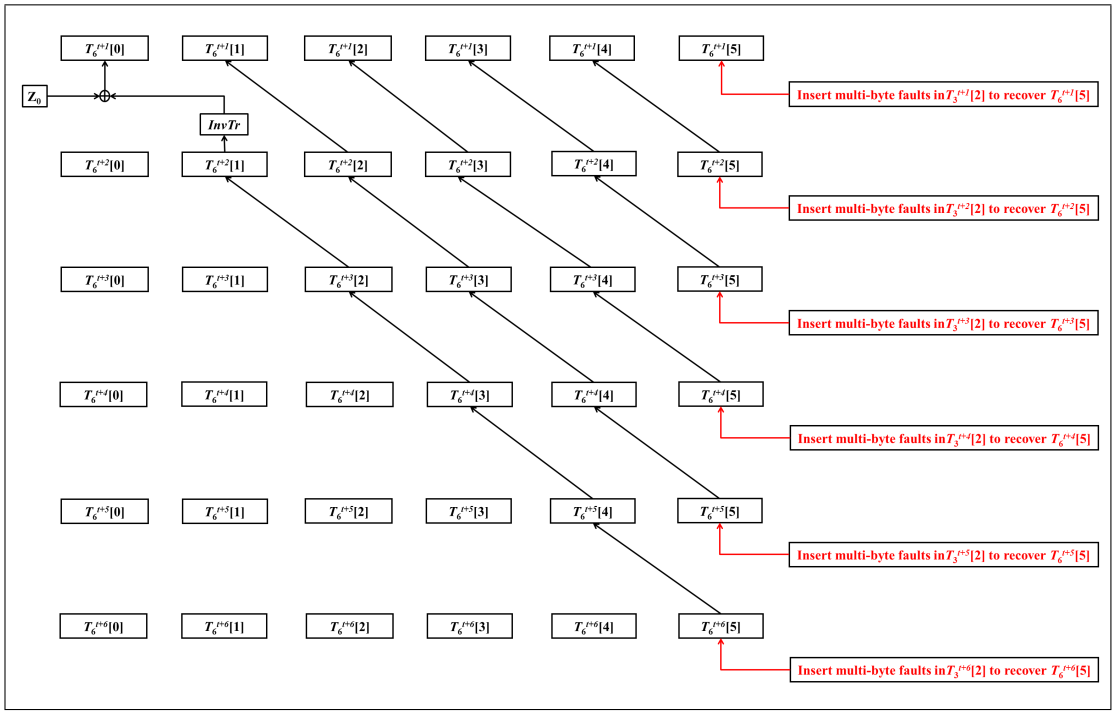

Figure 4.6: Recovering the Contents of T_6 by Injecting Random Faults in T_3

4.6.2.1 Attack Algorithm

The steps of our fault based key recovery attack on Tiaoxin-346 using random fault injection are outlined in Algorithm 4.3.

Algorithm 4.3 Algorithm for Fault Based Key Recovery Attack on Tiaoxin-346

1: Load key and initialization vector and perform the initialization phase. Let T_s^t denote the initial state. Proceed to compute the i^{th} block of fault free ciphertext.
2: Repeat Step 1 but insert a random multi-byte fault e in state word $T_3^{t+i}[2]$. Proceed to compute the i^{th} block of faulty ciphertext.
3: Observe the faulty and fault free ciphertext and apply Equation 4.16 to recover the value of the random fault e.
4: For any bits in the random fault e equal to one, observe the values in the corresponding bit positions in the faulty and fault free ciphertext and apply Equation 4.17 to recover the corresponding bits of $T_6^{t+i}[5]$.
5: Repeat steps 2 to 4 until all of the bits in $T_6^{t+i}[5]$ are recovered.
6: Repeat steps 2 to 5 for $i = 1, \cdots, 6$. This will recover the entire state of component T_6 at time $t+1$.
7: Invert the internal state T_6 up to the beginning of the initialization phase and recover the secret key from $T_6[0]$.

4.6.2.2 Experimental Results

Experiments were conducted to analyse the feasibility of the fault attack described in Section 4.6.2. The experiments were performed as computer simulations using Python 3.6 on a standard desktop computer. The faulty 128-bit words were generated using the Python built-in random number function.

We first investigated the success rate for recovering the state word $T_6^{t+1}[5]$ using multiple random faults. The investigation considered multi-byte fault model, that is the error e affected multiple bytes of state word $T_3^{t+1}[2]$. We observed the average number of bits recovered for an increasing number of faults. For a given number of faults, the random fault attack was performed 10,000 times. Table 4.4 presents the average number of bits recovered for different numbers of faults. Each trial was considered a success if at least the average number of bits was recovered. The success rate for recovering this average number of bits is also recorded in Table 4.4.

Table 4.4: Average Number of Recovered Bits

Number of Faults	Average Number of Bits Recovered	Success Rate
1	64	53.71%
2	96	54.61%
3	111	65.86%
4	120	59.57%
5	124	63.45%
6	126	67.63%
7	126	91.88%
8	127	91.07%
9	127	97.49%
10	127	99.32%

As shown in Table 4.4, with seven or more faults an adversary can recover 126 or more of the 128 bits of $T_6^{t+1}[5]$, with a success rate higher than 90%. With ten faults, 127 bits can be recovered, with a success rate of 99.32%. The adversary needs to guess the remaining bits to recover the entire state word $T_6^{t+1}[5]$.

We also observed that the success rate can be increased to a higher value with comparatively lesser faults, if the number of recovered bits is reduced in comparison to the average number of bits recovered. In that case, an adversary needs to guess a larger proportion of the state word $T_6^{t+1}[5]$. This is the trade-off between the number of recovered bits and the required number of faults.

Table 4.5: Success Rate for Recovering All the Bits of $T_6^{t+1}[5]$

Number of Faults	Success Rate
1	0%
2	0%
3	0%
4	0.04%
5	1.73%
6	12.81%
7	37.24%
8	61.72%
9	78.30%
10	87.64%
11	93.69%
12	96.78%
13	98.61%
14	99.16%

Now consider an attack to be successful only if all of the 128 bits of the state word $T_6^{t+1}[5]$ are recovered. A similar experiment was conducted, with 10,000 trials for each given number of random faults. Table 4.5 presents the result of this, giving the success rate of recovering all the bits of $T_6^{t+1}[5]$, for an increasing number of random multi-byte faults, from 1 up to 10.

From Table 4.5 it is clear that the probability of recovering all 128 bits of $T_6^{t+1}[5]$ increased as the number of random faults was increased. With less than four faults, the success rate was about zero for this experiment. The success rate increased from less than 2% to over 78% as the number of faults was increased from 5 to 9. For ten or more faults the success rate was over 85%. The experiments were performed for an increasing number of faults, until a success rate over 99% was achieved. As shown in Table 4.5, 14 multi-byte faults were required to achieve 99% success rate for recovering all 128 bits of $T_6^{t+1}[5]$.

Finally, we conducted experiments to determine the number of bits (less than 128) it was possible to recover with a 99% success rate for specific numbers of random faults. In this case, the adversary can perform the random fault attack to partially recover the state word $T_6^{t+1}[5]$, and then guess the remaining bits. To perform this experiment, we began by selecting the average number of bits recovered from Table 4.4, and then iteratively reduced the number of bits recovered by one until a 99% success rate was achieved. The success rate for

Table 4.6: Success Rate for Partial Recovery of $T_6^{t+1}[5]$ with Different Number of Faults

Number of Faults	Number of Bits Recovered	Success Rate
4	119	72.3%
	118	82.4%
	117	89.4%
	116	93.9%
	115	96.9%
	114	98.8%
	113	99.3%
5	123	78.3%
	122	89.1%
	121	94.9%
	120	98.2%
	119	99.2%
6	125	85.2%
	124	94.6%
	123	98.7%
	122	99.7%
7	125	98.3%
	124	99.7%
8	126	98.5%
	125	99.8%

recovering partial state words of $T_6^{t+1}[5]$ is tabulated in Table 4.6.

From Table 4.6, we observe that a 99% success rate can be achieved with a much smaller number of faults when the number of recovered bits is reduced by two to seven bits (depending on the number of faults) from the average number of recovered bits. For instance, the number of required faults is reduced from fourteen to seven, when the number of recovered bits is reduced from 128 bits to 124 bits. In other words, an adversary can recover 124 bits of state $T_6^{t+1}[5]$ with over 99% success rate by introducing seven faults, and the remaining four bits can be guessed with a guessing complexity of 2^4.

Recall that the adversary needs to continue the same process five more times to recover the state word $T_6^{t+2}[5]$, $T_6^{t+3}[5]$, $T_6^{t+4}[5]$, $T_6^{t+5}[5]$ and $T_6^{t+6}[5]$. This process is required to recover the entire contents of T_6^{t+1}.

Table 4.7 shows the complexity of guessing as the total number of required faults is varied, to recover the entire contents of T_6^{t+1} with a success rate of 99%. As shown in Table 4.7, the complexity of guessing remains practical with 48,

Table 4.7: Total Number of Faults Required to Recover the Contents of Component T_6^{t+1}

Total Number of faults	Complexity of Guessing
24	2^{90}
30	2^{54}
36	2^{36}
42	2^{24}
48	2^{18}

42 or 36 faults. The complexity of guessing increases significantly and becomes infeasible for any further reduction in the number of faults.

4.6.3 Summary of Fault Attacks on Tiaoxin-346

We described two different fault based attacks on Tiaoxin-346. The first attack is a forgery attack which works under the bit flipping fault model. The second type of fault attack is a key recovery attack which works under random fault model.

The fault based forgery attack is simple and trivial, which works by flipping specific bits in the message and then introducing bit flipping faults in the corresponding bit(s) of relevant state elements. The faulty ciphertext and faulty tag can be used to get a modified message accepted as legitimate. This attack works because the changes in the input message affect the internal state bits, and these are reflected in the state elements by introducing faults in the corresponding bits where the changes are made. This is true for all of the CAESAR cipher proposals which XOR the input message with the contents of the internal state [147].

There is some argument in the cryptographic community on the goals of an attack based on fault injection. Some researchers argued in the CAESAR discussion forum that the goal of the fault attack should be secret key recovery of the underlying algorithm [147], others contend that fault based forgery is relevant [147] because it compromises one of the security goals of an authenticated encryption algorithm, i.e., compromising the integrity component. The latter position is reasonable, and it is important to have physical protection against fault injection based forgery on Tiaoxin-346, since this can breach the integrity component of the cipher.

We also described a random fault based key recovery attack on Tiaoxin-346. The fault based key recovery attack uses a random multi-byte fault model. The

Table 4.8: Comparison of Our Approach with Existing Approach

Reference	No. of Faulty Encryptions	Fault Model	Complexity
Our Work	36	Random	2^{36}
[38]	3	Bit Flipping	1

attack can recover the secret key of Tiaoxin-346 with practical complexity using 36 random faults. The attack complexity for applying the attack with 36 faults is about 2^{36}. The attack complexity can be further reduced by increasing the number of faults. For instance, as shown in Table 4.7 the attack complexity can be reduced to 2^{24} and 2^{18} for 42 and 48 faults, respectively. Note that this is a differential fault attack which requires observing multiple ciphertexts computed over the same key and initialization vector. Thus, this falls under the nonce-reuse scenario. The practical nature of our attack confirms the importance of adhering to this restriction.

As discussed earlier in Section 4.3, Dey et al. [38] also described a differential fault based key recovery attack on Tiaoxin-346. Table 4.8 compares the work of Dey et al. [38] with our approach of differential fault attack on Tiaoxin-346. As shown in Table 4.8 our technique requires a much larger number of faults and a comparatively larger complexity when compared to the attack described by Dey et al. [38]. However, our described technique works under a random fault model, whereas the other attack works under a bit-flipping fault model. The assumption of the random fault model is more practical, as the inserted fault e can be any 128-bit value, whereas in the bit-flipping fault model, the fault e must be $e = 111...1$. This is a serious restriction. The random fault model has been shown to be feasible in actual hardware; whereas the bit-flipping fault model is largely theoretical [103].

We did not perform our experiments on the hardware implementation of Tiaoxin-346, but instead used computer simulations. However, other researchers [103, 151] have demonstrated that it is feasible to apply this random fault model in the hardware implementation of an algorithm. Therefore our fault attack should be practical for the key recovery of Tiaoxin-346.

Note that both of these fault based key recovery approach proposed in this chapter and by Dey et. al. [38] are differential fault attacks, therefore these attacks require the observation of multiple ciphertexts computed over the same key and initialization vector. This falls under the nonce-reuse scenario, which is

prohibited by the designer of Tiaoxin-346. To the best of our knowledge, there are no other nonce-reuse based attacks on Tiaoxin-346.

Also note that both the fault injection based key recovery technique proposed in this chapter and by Dey et. al. [38] target only a single component of the cipher. This is possible because the state update of the components is independent (no mixing between the contents of the individual components), and at the beginning of the initialization phase the entire key is loaded in each of the component. These attacks would be more difficult to perform if the key was distributed across the entire state during the initialization phase.

4.7 Similarities in Tiaoxin-346 and AEGIS

Recall from Chapter 2 that AEGIS is also a third round authenticated encryption stream cipher candidate in the CAESAR cryptographic competition. Due to the time constraints AEGIS has not been extensively analysed in this thesis. However, we note that there are some structural similarity between AEGIS and Tiaoxin-346. The designer also stated that the design of Tiaoxin-346 has been inspired from the design of AEGIS. In this section we discuss the construction of one of the AEGIS variants and the similarity that it has with Tiaoxin-346.

4.7.1 Structure of AEGIS

AEGIS has three different variants namely AEGIS-128, AEGIS-256 and AEGIS-128L. The structure of these variants are similar; however, the state sizes are different. Hence, the key and initialization vector loading procedure, number of initialization rounds, number of finalization rounds, state update function and the output functions are different for these variants. Here we discuss AEGIS-128L which processes two ciphertext words at each time instant, similar to Tiaoxin-346. This is not the case for AEGIS-128 and AEGIS-256 which only process one ciphertext word at each time instant.

At time instant t, the AEGIS-128L consists of eight 128-bit register stages s_0^t, s_1^t, s_2^t, s_3^t, s_4^t, s_5^t, s_6^t and s_7^t, respectively. Let S^t denote the $8 \times 128 = 1024$ bit internal state of the cipher. Figure 4.7 shows the state update process of AEGIS-128L in diagrammatic form.

The internal state of AEGIS-128L is updated at each time instant using a nonlinear state update function StateUpdate128L(S^t, M_0^t, M_1^t). The nonlinear

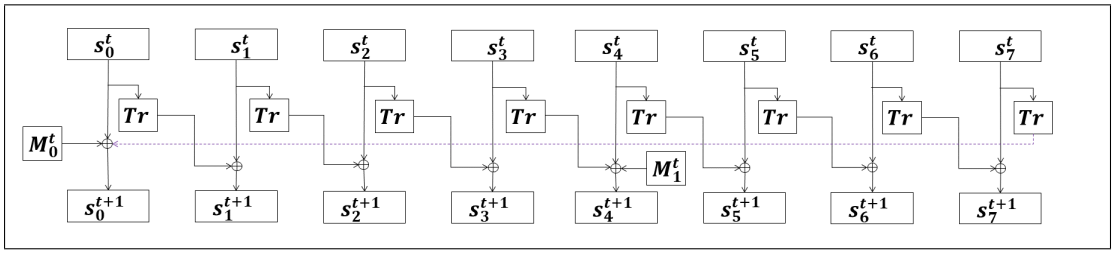

Figure 4.7: AEGIS-128L State Update

update is provided using the transformation function $Tr(x)$, applied to the contents of each register stage. The round transformation function $Tr(X)$ used in AEGIS is the same as the one used in Tiaoxin-346, which performs one AES round transformation without XOR-ing the sub key. This is defined as:

$Tr(X) = MixColumns(ShiftRows(SubBytes(X)))$

Figure 4.7 shows that the state update function of AEGIS-128L has two external inputs M_0^t and M_1^t. The state of AEGIS-128L at time $t+1$ is defined as:

$$s_i^{t+1} = \begin{cases} Tr(s_7^t) \oplus s_0^t \oplus M_0^t & \text{for } i = 0 \\ Tr(s_3^t) \oplus s_4^t \oplus M_1^t & \text{for } i = 4 \\ Tr(s_{i-1}^t) \oplus s_i^t & \text{otherwise} \end{cases} \tag{4.18}$$

where $0 \leq i \leq 7$.

Note here that both of the state update functions of Tiaoxin-346 and AEGIS-128L are using one round of AES transformation to update the state contents; however, in Tiaoxin-346 this transformation function $Tr(X)$ is not applied to all the register stages, whereas in AEGIS-128L (and all of the other variants of AEGIS) it is applied to all the register stages. Comparing the two ciphers we observe that Tiaoxin-346 uses the AES transformation function six times for one round of state update, while AEGIS-128L uses it eight times.

4.7.2 Phases of Operation in AEGIS-128L

AEGIS-128L has five different operation phases. These are:

1. Initialization
2. Associated data processing
3. Encryption
4. Finalization

4.7.2.1 Initialization

During the initialization phase of AEGIS-128L the 128-bit key, the 128-bit initialization vector and two specific constants are loaded into the internal state in a particular loading format. The internal state is then updated 10 times using the nonlinear state update function StateUpdate128L(S^t, M_0^t, M_1^t). For these 10 rounds of update the external input M_0 and M_1 are set to the 128-bit initialization vector V and 128-bit key K, respectively. No outputs are produced during these 10 rounds of initialization phase.

4.7.2.2 Associated Data Processing

AEGIS-128L performs the associated data processing phase after the initialization phase. The associated data D is divided into l_d blocks: $D^1, \cdots, D^{l_d}$. At each round one block of the associated data is processed. Each block of associated data is composed of two words: $D^t = D_0^t || D_1^t$, where D_0^t, D_1^t represents the two words of the corresponding block. The internal state of AEGIS-128L is updated using the state update function StateUpdate128L(S^t, D_0^t, D_1^t) for l_d rounds. During these l_d rounds no outputs are produced. Also the associated data processing phase is skipped when $l_d = 0$.

4.7.2.3 Encryption

AEGIS-128L performs the encryption phase following the associated data processing phase. Each plaintext block P is 256 bits and composed of two plaintext words: $P^t = P_0^t || P_1^t$, where P_0^t, P_1^t represents the two words of the corresponding block for that time instant t. A plaintext block P is processed at each time instant t. This process is continued for l_p rounds with each round processing one block of plaintext P^t to produce the corresponding ciphertext block $C^t = C_0^t || C_1^t$, where l_p represents the number of input plaintext blocks and C_0^t, C_1^t represents two output ciphertext words. At first the keystream words z_0^t and z_1^t are computed as:

$$z_0^t = s_1^t \oplus s_6^t \oplus (s_2^t \otimes s_3^t) \tag{4.19}$$

$$z_1^t = s_2^t \oplus s_5^t \oplus (s_6^t \otimes s_7^t) \tag{4.20}$$

As shown in Equation 4.19 and 4.20 the computation of the keystream words z_0^t and z_1^t takes inputs from four specific register stages of AEGIS-128L. Following

the keystream computation, the ciphertext words are computed by XOR-ing the plaintext words with the corresponding keystream words and defined as:

$$\begin{aligned} C_0^t &= P_0^t \oplus z_0^t \\ &= P_0^t \oplus s_1^t \oplus s_6^t \oplus (s_2^t \otimes s_3^t) \qquad (4.21) \\ C_1^t &= P_1^t \oplus z_1^t \\ &= P_1^t \oplus s_2^t \oplus s_5^t \oplus (s_6^t \otimes s_7^t) \qquad (4.22) \end{aligned}$$

Following the ciphertext computation, the internal state of AEGIS-128L is updated using the state update function StateUpdate128L(S^t, P_0^t, P_1^t). That is the input plaintext words P_0^t and P_1^t are used as the external input during the encryption phase of AEGIS -128L.

4.7.2.4 Finalization

During the finalization process the internal state is updated using $M_0^t = M_1^t = s_2^t \oplus (adlen||msglen)$ for 7 rounds. During these seven rounds of update no output is produced. Following these updates the authentication tag τ is computed by XOR-ing the contents of the register stages $s_0, \cdots, s_6$.

4.7.3 Is AEGIS Vulnerable to Similar Attacks to those Applied to Tiaoxin-346?

We observe that during the encryption phase, the plaintext is XOR-ed with specific register stages of AEGIS-128L. This is similar to Tiaoxin-346 encryption phase and the other ciphers discussed in this thesis. This means an adversary can perform a fault based forgery attack similar to the one for Tiaoxin-346, as discussed in Section 4.6.1. This attack is straightforward, and similar to the other fault based forgery attack discussed in this thesis. Therefore the details of this attack are not discussed here.

We also observe that other than the structural similarities discussed in Section 4.7.1, Tiaoxin-346 and AEGIS-128L also have similarities in the construction of the ciphertext. Comparing the ciphertext word computation discussed in Section 4.2.2.3 and Section 4.7.2.3, we notice that both Tiaoxin-346 and AEGIS-128L process two ciphertext words at each round. Also the ciphertext computation function for both of these ciphers have similarities in their structure. That

is, in both of the ciphertext computation there is one AND operation involved between two specific register stages, along with the XOR operation of few other specific register stages. Due to these similarities, it is possible to perform a fault based state recovery attack in AEGIS-128L, similar to the attack on Tiaoxin-346 described in Section 4.6.2.

Note that AEGIS-128L takes external input from the secret key K during the 10 updates of the initialization phase; therefore, the initial state of AEGIS-128L is not invertible without the knowledge of the secret key. This means that, unlike for the case of Tiaoxin-346, an initial state recovery of AEGIS-128L does not necessarily recover the secret key of the cipher. The details of the fault based state recovery attack on AEGIS-128L are discussed below.

4.7.3.1 Fault based State Recovery Attack on AEGIS-128L

The fault based state recovery attack on AEGIS-128L is applied during the encryption phase of the cipher. This attack requires an adversary to observe multiple faulty and fault free ciphertext pairs which are computed using the same key and initialization vector.

We start our fault based state recovery attack during the first round of the encryption phase of AEGIS-128L. Suppose $S^t = s_0^t, \cdots, s_7^t$ denote the internal state obtained after the initialization and associated data loading phases of AEGIS-128L are completed. Recall from Section 4.7.2.3, AEGIS-128L outputs two ciphertext words during each round of the encryption phase. These ciphertext words C_0^t and C_1^t are computed as shown in Equation 4.21 and Equation 4.22.

From Equation 4.21 and Equation 4.22, observe that both of these functions have some common inputs: the contents of register stage s_6^t is XOR-ed in the computation of C_0^t (as shown in Equation 4.21), and AND-ed with the contents of register stage s_7^t in the computation of C_1^t (as shown in Equation 4.22). This can be exploited by an adversary; inserting faults in s_6^t permits recovery of the contents of register stage s_7^t.

Similar observation shows that contents of register stage s_2^t is also used in both of the ciphertext computation function described in Equation 4.21 and 4.22: the contents of register stage s_2^t is XOR-ed in the computation of C_1^t (as shown in Equation 4.22), and AND-ed with the contents of register stage s_3^t in the computation of C_0^t (as shown in Equation 4.21). This can be exploited by an adversary; inserting faults in s_2^t permits recovery of the contents of register

stage s_3^t.

We first discuss this state recovery process with bit-flipping faults, proposed by Dey et. al. [38], and then propose our technique to relax this assumption to achieve the same outcome using random faults. This is a more realistic assumption for fault attacks than a bit-flipping model, since it requires less precise control of the fault outcome by the attacker.

State recovery of AEGIS-128L using bit-flipping faults. Suppose an adversary introduces fault e in register stage s_2^t which complements all of its contents. That is, $e = 111...111$ and the faulty contents of register stage $s_2'^t$ is defined as $s_2'^t = s_2^t \oplus e = \overline{s_2^t}$, where $\overline{s_2^t}$ denotes the complement of s_2^t.

The fault free and faulty ciphertext words C_0^t and $C_0'^t$ of AEGIS-128L are defined in Equation 4.21 and Equation 4.23.

$$C_0^t = P_0^t \oplus s_1^t \oplus s_6^t \oplus (s_2^t \otimes s_3^t) \tag{4.21}$$

$$C_0'^t = P_0^t \oplus s_1^t \oplus s_6^t \oplus (\overline{s_2^t} \otimes s_3^t) \tag{4.23}$$

In this case, XOR-ing Equation 4.21 and Equation 4.23 allows the adversary to recover the contents of register stage s_3^t, as shown in Equation 4.24.

$$\begin{aligned} C_0^t \oplus C_0'^t &= (s_2^t \otimes s_3^t) \oplus (\overline{s_2^t} \otimes s_3^t) \\ &= s_3^t \oplus (s_2^t \oplus \overline{s_2^t}) \\ &= s_3^t \end{aligned} \tag{4.24}$$

In Equation 4.24, the adversary has access to the ciphertext pair $(C_0^t, C_0'^t)$. Thus with 128 bit-flipping faults in s_2^t, an adversary can recover the contents of register stage s_3^t. If the plaintext P_0^t is different in Equation 4.21 and Equation 4.23, then the adversary requires access to the corresponding keystream word, that is, the attack will fall under a known plaintext model.

Now, suppose the adversary introduces fault e in register stage s_3^t which complements all of its contents. That is, $e = 111...111$ and the faulty contents of register stage $s_3'^t$ is defined as $s_3'^t = s_3^t \oplus e = \overline{s_3^t}$, where $\overline{s_3^t}$ denotes the complement of s_3^t. With the introduction of such bit-flipping faults in s_3^t, XOR-ing the faulty and fault free ciphertext words C_0^t and $C_0'^t$ recovers the content of register stage s_2^t. Thus with the introduction of $2 \times 128 = 256$ bit-flipping faults in register stages s_2^t and s_3^t an adversary can recover the contents of register stages s_3^t and s_2^t, respectively. This is possible due to the AND operation between the two

register stages s_2^t and s_3^t in the computation of ciphertext word C_0^t.

Notice also that in Equation 4.22 the ciphertext word C_1^t computation performs an AND operation between register stages s_6^t and s_7^t. Therefore, using similar techniques as discussed above an adversary can introduce $2 \times 128 = 256$ bit-flipping faults in register stages s_6^t and s_7^t to recover the contents of register stages s_7^t and s_6^t, respectively.

Following this, the adversary can substitute the recovered value of s_2^t, s_3^t, s_6^t and s_7^t in Equation 4.19 and 4.20 to recover the value of s_1^t and s_5^t, respectively. To recover the value of these two register stages, an adversary needs to have the knowledge of the keystream words z_0^t and z_1^t. That is, the adversary now has recovered s_1^t, s_2^t, s_3^t, s_5^t, s_6^t and s_7^t.

The adversary can then compute the value of the contents of register stages s_2^{t+1}, s_3^{t+1}, s_6^{t+1} and s_7^{t+1} using the relations shown in Equation 4.25, 4.26, 4.27 and 4.28, respectively.

$$s_2^{t+1} = s_2^t \oplus Tr(s_1^t) \tag{4.25}$$

$$s_3^{t+1} = s_3^t \oplus Tr(s_2^t) \tag{4.26}$$

$$s_6^{t+1} = s_6^t \oplus Tr(s_5^t) \tag{4.27}$$

$$s_7^{t+1} = s_7^t \oplus Tr(s_6^t) \tag{4.28}$$

Now, following the keystream generation function as described in Equation 4.19 and 4.20, the keystream words of AGEIS-128L at time $t+1$ are shown in Equation 4.29 and 4.30.

$$z_0^{t+1} = s_1^{t+1} \oplus s_6^{t+1} \oplus (s_2^{t+1} \otimes s_3^{t+1}) \tag{4.29}$$

$$z_1^{t+1} = s_2^{t+1} \oplus s_5^{t+1} \oplus (s_6^{t+1} \otimes s_7^{t+1}) \tag{4.30}$$

Substituting the values of s_2^{t+1}, s_3^{t+1}, s_6^{t+1} and s_7^{t+1} in Equation 4.29 and 4.30, the adversary can recover the value of s_1^{t+1} and s_5^{t+1}, respectively. To recover the value of these two register stages, an adversary needs to have the knowledge of the keystream words z_0^{t+1} and z_1^{t+1}. Finally, the adversary can recover the contents of register stages s_0^t and s_4^t by substituting the values of s_1^{t+1} and s_1^t in

Equation 4.31, and the values of s_5^{t+1} and s_5^t in Equation 4.32, respectively.

$$\begin{aligned} s_1^{t+1} &= s_1^t \oplus Tr(s_0^t) \\ \Rightarrow s_0^t &= InvTr(s_1^{t+1} \oplus s_1^t) \end{aligned} \tag{4.31}$$

$$\begin{aligned} s_5^{t+1} &= s_5^t \oplus Tr(s_4^t) \\ \Rightarrow s_4^t &= InvTr(s_5^{t+1} \oplus s_5^t) \end{aligned} \tag{4.32}$$

where $InvTr$ represents the inverse operation of one AES round transformation.

That is, with the specific $4 \times 128 = 512$ bit-flipping faults in register stages s_2^t, s_3^t, s_6^t and s_7^t the adversary can recover the entire state $S^t = s_0^t, \cdots, s_7^t$ of AEGIS-128L. Considering multi-byte faults this attack will require 4 faults.

The attack described by Dey et.al. [38] on AEGIS-128L used this idea to recover the state S^t of AEGIS-128L using bit-flipping faults. However, we can adapt our attack targeting to recover the state of AEGIS-128L, using random faults rather than bit-flipping. Our proposed technique uses the similar idea as the one introduced for Tiaoxin-346 .

State recovery of AEGIS-128L using random faults. We describe an adaptation of the above technique to recover the internal state S^t with random faults; this is a more realistic assumption than the stringent requirements on the attacker's capabilities for bit-flipping fault attacks. Now, suppose the fault e is a randomly generated value and the adversary does not have any knowledge of the specific fault value. Let the random fault e affect the contents of register stage s_2^t such that the faulty state word $s_2'^t = s_2^t \oplus e$. In this case, the fault free and faulty ciphertext words C_1^t and $C_1'^t$ are shown in Equation 4.22 and Equation 4.33.

$$C_1^t = P_1^t \oplus s_2^t \oplus s_5^t \oplus (s_6^t \otimes s_7^t) \tag{4.22}$$

$$C_1'^t = P_1^t \oplus s_2'^t \oplus s_5^t \oplus (s_6^t \otimes s_7^t) \tag{4.33}$$

We observe that the adversary can obtain the value of the random error e by simply XOR-ing Equation 4.22 and Equation 4.33, as shown in Equation 4.34.

$$\begin{aligned} C_1^t \oplus C_1'^t &= s_2^t \oplus s_2'^t \\ &= s_2^t \oplus s_2^t \oplus e \\ &= e \end{aligned} \tag{4.34}$$

In Equation 4.34, the adversary needs to have access to the the fault free and faulty ciphertext pair $(C_1^t, C_1'^t)$. Given access to these ciphertext pairs and assuming that the plaintext pair P_1^t are the same in both Equation 4.22 and Equation 4.33, the adversary can obtain the value of the random error e introduced in register stage s_2^t. If the plaintext P_1^t is different in Equation 4.22 and Equation 4.33, then the adversary requires access to the corresponding keystream word, that is, the attack will fall under a known plaintext model.

We note that the random fault in the register stage s_2^t also affects the ciphertext word C_0^t. Replacing $\overline{s_2^t}$ by $s_2^t \oplus e$, Equation 4.24 becomes

$$C_0^t \oplus C_0'^t = s_3^t \otimes e \tag{4.35}$$

This allows us to use the random fault value e to find the value of bits in s_3^t. Note that for bit positions in e with value 0, the bits in the corresponding position in s_2^t and $s_2'^t$ will have the same value, and for bit positions in e with value 1, the bits in the corresponding position in s_2^t and $s_2'^t$ will be complementary.

Unlike the bit-flipping fault attack described by Dey et. al.[38], our approach using random faults does not necessarily recover all of the bits in the register stage s_3^t with a single fault. A single random fault recovers only the bits in s_3^t where the corresponding bits of e have a value of one. Thus we need to perform the random fault injection multiple times until all the bits in s_3^t are recovered.

The random fault e in the register stage s_2^t can be used to recover the contents of register stage s_3^t because, s_2^t is XOR-ed in the computation of ciphertext word C_1^t and also AND-ed with s_3^t in the computation of ciphertext word C_0^t. We also observe similar use of the contents of register stage s_6^t. In particular, s_6^t is XOR-ed in the computation of ciphertext word C_0^t and also AND-ed with s_7^t in the computation of ciphertext word C_1^t. Thus, using similar random faults e in the register stage s_6^t, the adversary can recover the contents of register stage s_7^t.

Similarly, in the successive time instant $t+1$, separate application of the random faults e in register stages s_2^{t+1} and s_6^{t+1} allows an adversary to recover the contents of register stages s_3^{t+1} and s_7^{t+1}, respectively. Substituting the values of s_3^{t+1} and s_3^t in Equation 4.36 and the values of s_7^{t+1} and s_7^t in Equation 4.37,

the adversary can recover the contents of s_2^t and s_6^t, respectively.

$$\begin{aligned} s_3^{t+1} &= s_3^t \oplus Tr(s_2^t) \\ \Rightarrow s_2^t &= InvTr(s_3^{t+1} \oplus s_3^t) \end{aligned} \tag{4.36}$$

$$\begin{aligned} s_7^{t+1} &= s_7^t \oplus Tr(s_6^t) \\ \Rightarrow s_6^t &= InvTr(s_7^{t+1} \oplus s_7^t) \end{aligned} \tag{4.37}$$

Now, the adversary has recovered the contents of s_2^t, s_3^t, s_6^t and s_7^t. Substituting the value of s_2^t, s_3^t, s_6^t and s_7^t in Equation 4.19 and Equation 4.20 the adversary can recover the contents of register stages s_1^t and s_5^t. Finally, using Equation 4.31 and Equation 4.32 the adversary can recover the contents of register stages s_0^t and s_4^t. That is, now an adversary can recover the entire state $S^t = s_0^t, \cdots, s_7^t$ of AEGIS-128L using the above mentioned random faults.

4.7.4 Remarks on AEGIS

The discussion on AEGIS-128L presented in this chapter demonstrates the similarity in the structure, and in the output function of the cipher with Tiaoxin-346. We demonstrated that, due to the similarity in the output function and in the number of output ciphertext words produced by AEGIS-128L and Tiaoxin-346, random fault based state recovery can be performed in AEGIS-128L, as simmilar to the one of Tiaoxin-346. This relaxes the stringent requirement of the bit-flipping fault based state recovery of AEGIS-128L described by Dey et. al. [38], by using our random fault based technique.

Note that the random fault based state recovery may not be applicable for the other two variants AEGIS-128 and AEGIS-256, as they output only one ciphertext word at each time instant. Therefore, it is not clear if it is possible to recover the value of the random fault, if applied to these two variants. Nevertheless, it is worth to look into a more extensive analysis of these variants along with a more detailed analysis of AEGIS-128L. Due to the time constraints we have not provided an in-depth analysis of different variants of AEGIS. In future, we plan to extend our investigation on other variants of AEGIS.

4.8 Summary on the Analysis of Tiaoxin-346

This chapter examined the feasibility of different attack scenarios on the authenticated encryption cipher Tiaoxin-346. In particular, the state cycle was examined for possible weaknesses, and the application of cube attacks and fault based forgery and key recovery attacks on Tiaoxin-346 were investigated.

Analysis of the state cycle for a toy version of Tiaoxin-346 shows that for components T_3 and T_4, some loaded states result in shorter cycles in the individual components. This is also likely to be true for component T_6. However, we did not find any loaded states that have a relatively short cycle for all the components. From the experiment, we also did not find cycles for the components which have a small least common multiple. Thus, the cycle for the entire state, obtained as the least common multiple of the cycle lengths for all three individual components, will be large. The result for the toy version of Tiaoxin-346 does not have any relatively short cycles in its state which may cause security threats. Therefore it is expected that Tiaoxin-346 does not have any relatively short cycles, since it has the same structure as the toy version.

The application of the cube attacks on Tiaoxin-346 was explored. The attack was applied to a reduced version where the number of initialization rounds is reduced from 15 to 4. Based on the cube attack, we did not find any linear equations which can be used to recover the secret key for this reduced version of Tiaoxin-346. This is due to the AES round function used in Tiaoxin-346 state update function, which results in a high algebraic degree in its output function.

For the reduced version of Tiaoxin-346, we found cubes which result in constant linear superpolys. These cubes can be used to distinguish the output of 4-round Tiaoxin-346 from the output of a randomly generated function. This distinguishing attack has a complexity of 2^3.

We also have shown that the idea of cube attack can be used during the encryption phase of Tiaoxin-346 to recover the contents of any component. This attack however requires the ability of manipulating the state bits of Tiaoxin-346. Additionally, this attack works under the nonce-reuse scenario. The designer of Tiaoxin-346 does not claim any security under this assumption.

The best result obtained for cube attack on Tiaoxin-346 is the construction of a 4-round distinguisher. Currently, the cube attack does not seem to threaten the security of Tiaoxin-346 if the full initialization phase is performed.

We illustrated a fault based forgery attack on Tiaoxin-346. The attack uses

Table 4.9: Comparison of Different Attack Techniques on Tiaoxin-346

Attack Method	Attack Type	Initialization Rounds	Complexity
Cube Attack	Distinguishing	4	2^3
Fault Attack	Forgery	15	1
Fault Attack	Key Recovery	15	2^{36}

bit flipping fault injections in the internal state of Tiaoxin-346 to construct the tag forgery for two different inputs of Tiaoxin-346. This type of fault based forgery attack works when adversary has access to the implementation of the algorithm in the sender's device. This attack requires introducing two bit faults in the Tiaoxin-346 state for a single bit flip in the input message.

We described a differential fault based key recovery attack on Tiaoxin-346. The attack can recover the secret key of Tiaoxin-346 with 36 or more faults, with a practical complexity. Compared to the existing work [38] this application of fault based key recovery attacks requires a larger number of faults, but the technique works under random fault model whereas the other attack is under bit flipping fault model. The assumption of the random fault model is more practical than the bit flipping fault model, since the adversary does not need to have control over the actual fault values. Thus the random fault model can be considered much more practical compared to the bit flipping fault model [103].

Finally, we identified structural similarities in the construction techniques of Tiaoxin-346 and AEGIS-128L. We demonstrated that AEGIS may be vulnerable to similar attacks to those we applied to Tiaoxin-346. In particular, an improved fault based state recovery attack was demonstrated on AEGIS-128L, where the improvement in our attack is the fault model, which used random faults instead of bit-flipping faults proposed by Dey et. al. [38].

4.8.1 Security Impact

This section compares the applicability of different attack methods on Tiaoxin-346. Table 4.9 provides an overall comparison of the different attacks applied to Tiaoxin-346.

As shown in Table 4.9, the cube attack works only for a reduced version of Tiaoxin-346. The best cube attack described in this chapter can work up to four rounds of the initialization phase of Tiaoxin-346, while the full version of Tiaoxin-346 has 15 rounds of initialization phase. It seems that Tiaoxin-346 has

a large security margin against cube attacks.

Table 4.9 shows that fault based universal forgery attack can be applied to Tiaoxin-346 with a reasonable number of faults. As discussed in the CAESAR cryptographic forum [147], this type of fault attack is applicable to other CAESAR candidates and it is argued there that the goal of the fault attack is key recovery rather than to construct a forgery attack. Alternatively, it is also argued in the same forum [147] that fault based forgery attack compromises the integrity component of the authenticated encryption algorithm and compromising one of the components is sufficient to justify the relevancy of fault based forgery attacks. We think the latter argument is reasonable. Also, note that this fault based forgery attack works under a bit flipping fault model. The attack will be more practical if this can be extended under the random fault model. Note that the fault based forgery attack on Tiaoxin-346 is not a differential attack, and hence the designer restrictions on nonce reuse do not prohibit this.

Finally, we can see from Table 4.9 that fault based key recovery attack with a practical complexity can be applied to Tiaoxin-346 using the random fault model, which is more practical than the bit-flipping fault model. However, this attack works under nonce-reuse scenario, for which the designer of Tiaoxin-346 does not claim any security.

Analysing Table 4.9, we see that cube attack does not threaten the security of Tiaoxin-346. Note that fault based attack can be used in either a forgery or key recovery attack on Tiaoxin-346; these attacks however are applied to the implementation of Tiaoxin-346, rather than the algorithm itself. Provided the implementation is tamper resistant and nonce reuse is prevented, Tiaoxin-346 appears to be a secure cipher.

Chapter 5

Analysis of MORUS

This chapter investigates the security of the authenticated encryption cipher MORUS [34]. The investigation includes the applicability of the cube attack, fault attack, rotational cryptanalysis and forgery attack on MORUS. The results presented in Section 5.4 of this chapter have been published in the Proceedings of the 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (IEEE TrustCom-17) [41]. The results presented in Sections 5.5 and 5.7.2 of this chapter have been published in Cryptography, 2(1) [42].

MORUS is a family of authenticated encryption stream cipher algorithms, a third-round candidate in the CAESAR Authenticated Encryption (AE) competition [25]. There are three variants: MORUS-640-128, MORUS-1280-128 and MORUS-1280-256; where the first number represents the state size and the latter one represents the key size. The cipher is intended to provide confidentiality and integrity assurance for the input data.

The MORUS family of stream ciphers supports a key K of either 128-bits or 256-bits. The initialization vector V is 128-bits for all the variants of MORUS. It takes an input plaintext P of arbitrary length. Confidentiality is achieved by encrypting the plaintext P by XOR-ing with the output keystream generated by the cipher to obtain the ciphertext C. The cipher also takes associated data D of arbitrary length as input. The associated data is not encrypted. MORUS provides integrity assurance for both the plaintext P and associated data D. This is done by injecting the plaintext and associated data into the internal state of

the cipher and computing a tag τ in terms of the internal state.

The rest of the chapter is organized as follows. Section 5.1 describes the notations and operations used in this chapter. Section 5.2 provides a detailed description on the working principle of MORUS. The existing work on MORUS is described in Section 5.3. Section 5.4 illustrates the applicability of the cube attack on MORUS. Section 5.5 describes fault based key recovery attacks on MORUS. Section 5.6 describes our observations and analysis on the rotational cryptanalysis of MORUS. Section 5.7 describes our observations on the application of forgery attacks to MORUS. Finally, Section 5.8 provides an overall summary on the analysis of MORUS presented in this chapter.

5.1 Notations and Operations

- $\oplus$: Bit-wise XOR operation.
- $\otimes$: Bit-wise AND operation.
- Word: A sequence of 32 bits or 64 bits, for MORUS-640 and MORUS-1280, respectively.
- Block: A sequence of 128 bits or 256 bits, for MORUS-640 and MORUS-1280, respectively.
- $Rotl\_xxx\_yy(x, b)$: Divide a xxx-bit block x into 4 yy-bit words and rotate each word to the left by b bits.
- $K = k_0 k_1 ... k_{l_{k-1}}$: The secret key of size l_k bits.
- V: The initialization vector of size 128 bits.
- $const_0$: A 128-bit constant `0x000101020305080d1522375990e97962` in hexadecimal format.
- $const_1$: A 128-bit constant `0xdb3d18556dc22ff12011314273b528dd` in hexadecimal format.
- M^t: The external input to the state at step t.
- P^t: The input plaintext block at step t.
- D^t: The input associated data block at step t.

- Z^t: The output keystream block at step t.
- C^t: The output ciphertext block at step t.
- $msglen$: Length of the plaintext in bits where $0 \leq msglen < 2^{64}$.
- $adlen$: Length of the associated data in bits where $0 \leq adlen < 2^{64}$.
- l_p: Number of input plaintext blocks.
- l_d: Number of input associated data blocks.
- τ: Authentication tag.
- S^t: The internal state at step t.
- S_j^t: The internal state at the j^{th} round of step t.
- $S_{j,k}^t$: k^{th} element of state S_j^t.
- $\overleftarrow{X}^r$: Rotation of the input X to the left by r bits.
- X_{rotl_b}: $Rotl\_xxx\_yy(x, b)$ operation applied to X. xxx and yy assumed to be known from context.

5.2 Description of MORUS

MORUS [34] has 5 state elements $S_{0,0}, \cdots, S_{0,4}$. Each element is a register of length either 128 bits or 256 bits, for MORUS-640 and MORUS-1280, respectively. This gives a total internal state size of 640 and 1280 bits, for MORUS-640 and MORUS-1280, respectively. Operations performed in MORUS can be divided into five phases:

1. Initialization
2. Processing associated data
3. Encryption
4. Finalization
5. Decryption and tag verification

Table 5.1: Rotation Constants Used in MORUS

	MORUS-640	MORUS-1280
b_0	5	13
b_1	31	46
b_2	7	38
b_3	22	7
b_4	13	4
w_0	32	64
w_1	64	128
w_2	96	192
w_3	64	128
w_4	32	64

Note that there are two versions of the MORUS family of authenticated encryption cipher: MORUSv1 [34] and MORUSv2 [35]. The two versions differ only in the finalization phase. The general description provided in this chapter is based on MORUSv2 [35]. The operations performed in different phases of MORUS are based on several component functions. These are:

1. Keystream Generation Function
2. State Update Function
3. Combining Function

Figure 5.1 shows the different component functions of MORUS in generic form.

5.2.1 State Update Function

One of the main component function of MORUS is the state update function $Update(S^t, M^t)$. As shown in Figure 5.1, at each step t of the state update function there are 5 rounds with similar operations. In each round, two of the state elements $S^t_{j,k}$ are updated. The operations in the state update function include AND, XOR and rotation operation. MORUS uses the bitwise left rotation $\overleftarrow{X}^{w_i}$ which is a simple rotation of the input X to the left by w_i bits, where $0 \leq i \leq 4$. It also uses the $Rotl\_xxx\_yy(x, b_i)$ operation which divides a xxx-bit block input x into 4 yy-bit words and rotates each word to the left by b_i bits, where $0 \leq i \leq 4$. The rotation constants b_i and w_i are listed in Table 5.1.

The state update function takes input from the internal state and external input M. Depending on the phase the cipher is in, the external input M can be:

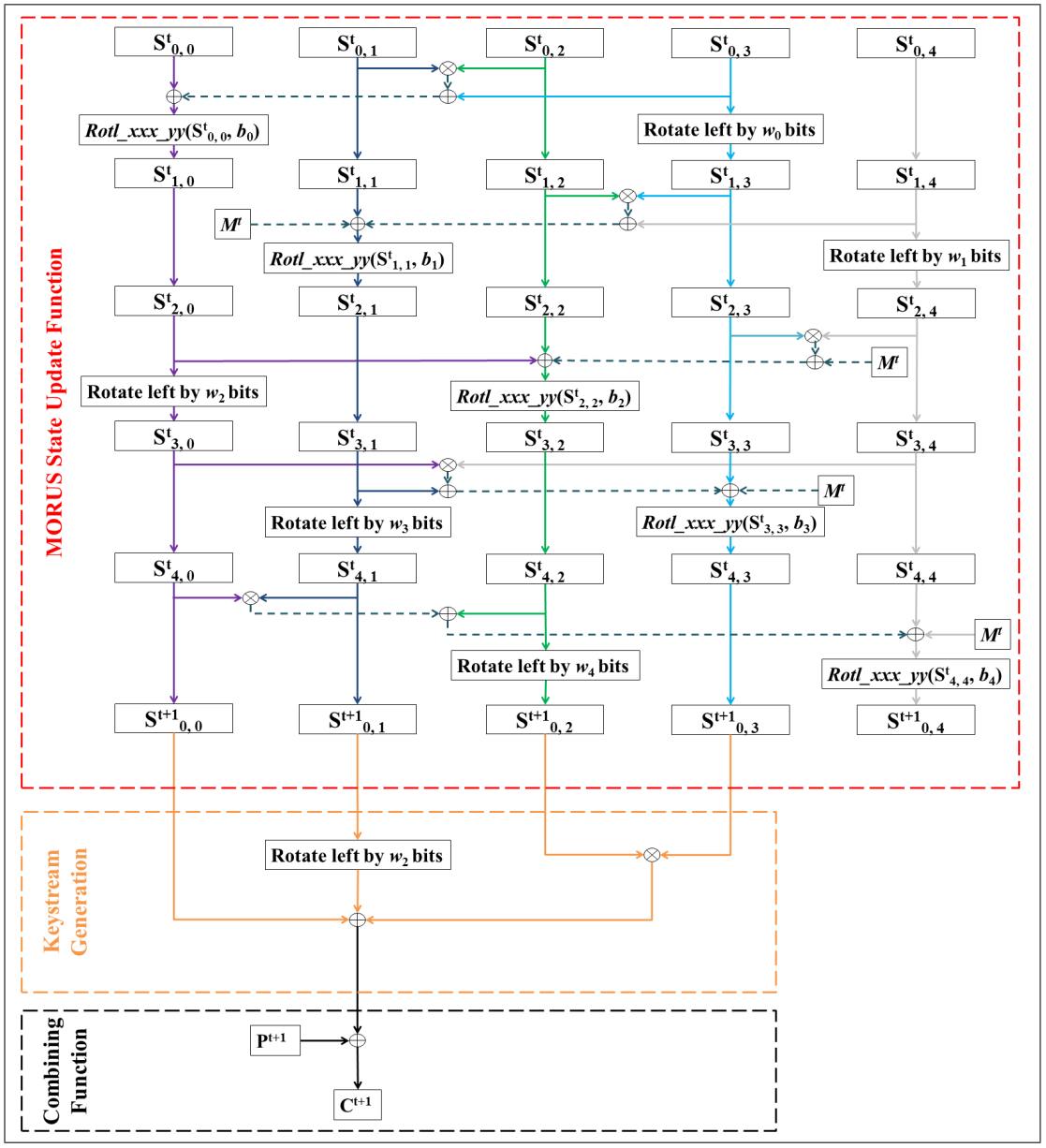

Figure 5.1: Generic Diagram of MORUS

all zero bits, the associated data, the plaintext, or a representation of the length of the associated data and plaintext.

During the initialization phase the external input M^t is set to zero. For the encryption and associated data loading phases the external inputs M^t are defined in terms of the plaintext P^t and associated data D^t, respectively. For the finalization phase the external input M^t is defined in terms of the length of the plaintext and associated data. Different phases of MORUS-640 and MORUS-1280 are described in Sections 5.2.4 and 5.2.5.

5.2.2 Keystream Generation Function

The keystream generation function is used during the encryption phase of MORUS. It outputs a keystream block at each step of the encryption phase. This is computed as a function of the first four state elements $S_{0,0}, \cdots, S_{0,3}$ of MORUS. The keystream generation function is defined as:

$$Z^t = S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{w_2} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \tag{5.1}$$

In the keystream generation function w_2 is the left rotation constant, set to 96 bits or 192 bits for MORUS-640 and MORUS-1280, respectively.

5.2.3 Combining Function

The combining function used during the encryption phase of MORUS is a simple XOR operation. That is, the ciphertext block is computed by XOR-ing the input plaintext block and the keystream block.

5.2.4 MORUS-640

MORUS-640 has 5 state elements, $S_0, \cdots, S_4$ where each element is a 128 bit register. The block size and word size for MORUS-640 are 128 bits and 32 bits, respectively. At each step it processes a 128-bit input data block, i.e., plaintext or associated data.

5.2.4.1 Initialization

At the beginning of the initialization phase the state elements of MORUS-640 are loaded with the key, initialization vector and two 128-bit constant $const_0$ and $const_1$. Particularly, the state elements $S^{t=0}_{0,0}$, $S^{t=0}_{0,1}$, $S^{t=0}_{0,2}$, $S^{t=0}_{0,3}$, $S^{t=0}_{0,4}$ are loaded with the 128-bit initialization vector V, 128-bit key K, sequence of 128 bits of value 1, $const_0$ and $const_1$, respectively. After this, the state update function $Update(S^t, M^t)$ of MORUS is applied 16 times. During these updates the external input M^t is set to zero and the cipher does not produce any output. After these 16 updates the content of state element $S^{t=16}_{0,1}$ is XORed with the 128-bit key. The state value at the end of this process is the initial state of MORUS-640.

5.2.4.2 Processing associated data

At the beginning of the associated data processing phase, the internal state of MORUS-640 is the initial state. For MORUS-640, the input associated data is divided and processed in blocks of size 128 bits. If the last associated data block is not a full block, then it is padded with zeroes to create a 128-bit block. At each step the cipher takes an input associated data block D^t and uses this as the external input M^t to update the state of MORUS-640. This process is continued l_d times where l_d is the number of associated data blocks. Note that during this process the cipher does not produce any output. This phase is not performed if the length of the associated data is zero.

5.2.4.3 Encryption

After completing the associated data processing phase, MORUS-640 processes the input plaintext. Similar to the associated data processing, the plaintext is also divided and processed in blocks of size 128 bits. If the last plaintext block is not a full block, then it is padded with zeroes to create a 128-bit block. At each step t of the encryption phase MORUS-640 computes a 128-bit output keystream block using Equation 5.1. This output keystream block Z^t is XOR-ed with the plaintext block P^t to compute the respective ciphertext block. The ciphertext computation is shown in Equation 5.2.

$$\begin{aligned} C^t &= P^t \oplus Z^t \\ &= P^t \oplus S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{96} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \end{aligned} \tag{5.2}$$

After computing the ciphertext block, the input plaintext block P^t is used as the external input M^t to update the state of MORUS-640. This process is continued until all of the plaintext blocks are processed, i.e., l_p times where l_p is the number of plaintext blocks.

The difference between the associated data processing and encryption phase is that for each plaintext block a corresponding ciphertext block is computed first before loading the input in to the internal state. Similar to the associated data loading phase, no operations will be performed if the length of the input plaintext is zero.

5.2.4.4 Finalization

After processing all of the input plaintext, the cipher goes through the finalization phase to generate the authentication tag. The first step of the finalization phase is updating state element $S^t_{4,0}$ by XOR-ing its content with the content of state element $S^t_{0,0}$, i.e., $S^t_{4,0} = S^t_{4,0} \oplus S^t_{0,0}$. The internal state is then updated 10 times using the state update function $Update(S^t, M^t)$ of MORUS. The external input M^t is set as $adlen||msglen$ for these 10 iterations. During these 10 updates the cipher does not produce any output. Finally, the tag τ is computed by using the Equation 5.3.

$$\tau = S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{96} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \tag{5.3}$$

5.2.4.5 Decryption and Tag Verification

To carry out the decryption, first the initialization process is performed to obtain the initial state. The associated data processing phase is also performed as described in Section 5.2.4.2, if the length of associated data $adlen > 0$.

At each step t of the decryption process the 128-bit output keystream block is computed using Equation 5.1. This output keystream block Z^t is XOR-ed with the ciphertext block C^t to compute the corresponding decrypted plaintext block. The recovered plaintext block computation is shown in Equation 5.4.

$$\begin{aligned} P^t &= C^t \oplus Z^t \\ &= C^t \oplus S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{96} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \end{aligned} \tag{5.4}$$

At each step, the recovered plaintext block P^t is used as the external input M^t to update the state. This process is continued l_c times to process all the ciphertext blocks, where l_c is the number of ciphertext blocks.

After processing all the ciphertext blocks, the tag is generated following the procedure discussed in Section 5.2.4.4. Finally, for the tag verification, the received tag is compared with the tag generated after the decryption process. If the received tag is not the same as the tag computed after the decryption process then the tag verification fails, and the ciphertext and computed tag should not be released. Otherwise the tag verification process is considered successful.

5.2.5 MORUS-1280

MORUS-1280 has 5 state elements, $S_0, \cdots, S_4$. Each element is a 256 bit register. The block size and word size for MORUS-1280 are 256 bits and 64 bits, respectively. At each step it processes a 256-bit input data, i.e., plaintext or associated data. MORUS-1280 supports either a 128-bit key or a 256-bit key. The size of the initialization vector for MORUS-1280 is 128-bit.

5.2.5.1 Initialization

Similar to MORUS-640, in the initialization phase the state elements of MORUS-1280 are loaded with the key, initialization vector and some constant values. Let K_{128} and K_{256} represent the 128-bit and 256-bit key of MORUS-1280. Let $K_I = K_{128}||K_{128}$ when the key size is 128-bit. Alternatively $K_I = K_{256}$ when the key size is 256 bits.

The state elements $S_{0,0}^{t=0}$, $S_{0,1}^{t=0}$, $S_{0,2}^{t=0}$, $S_{0,3}^{t=0}$, $S_{0,4}^{t=0}$ are loaded with $V||0^{128}$, K_I, 1^{256}, 0^{256} and $const_0||const_1$, respectively. The cipher is then updated 16 times without producing any output keystream bits. The external input M^t is set to zero during these updates. After these 16 updates the contents of state element $S_{0,1}^{t=16}$ is XORed with K_I. The state value formed after this process is the initial state of MORUS-1280.

5.2.5.2 Processing associated data

For MORUS-1280 the processing of associated data handling is similar to MORUS-640, except that the input associated data is divided and processed in blocks of size 256 bits for this instance. If the last associated data block is not a full block, then it is padded with zeroes to create a 256-bit block. At each step, the cipher takes an input associated data block D^t and uses this as the external input M^t to update the state of MORUS-640. This process is continued l_d times where l_d is the number of associated data blocks. Note that during this process the cipher does not produce any output. This phase is not performed if the length of the associated data is zero.

5.2.5.3 Encryption

Similar to the associated data processing phase of MORUS-1280, the plaintext is divided and processed in blocks of size 256 bits. If the last plaintext block is

not a full block, then it is padded with zeroes to create a 256-bit block.

At each step t of the encryption phase MORUS-1280 computes a 256-bit output keystream block using Equation 5.1. This output keystream block Z^t is XOR-ed with the plaintext block P^t to compute the respective ciphertext block. The ciphertext computation is shown in Equation 5.5.

$$\begin{aligned} C^t &= P^t \oplus Z^t \\ &= S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{192} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \end{aligned} \tag{5.5}$$

After computing the ciphertext block, the input plaintext block P^t is used as the external input M^t to update the state of MORUS-1280. This process is continued until all the plaintext blocks are processed, i.e., until l_p times where l_p is the number of plaintext blocks. No operations will be performed if the length of the input plaintext is zero.

5.2.5.4 Finalization

After processing all of the input plaintext, the cipher goes through the finalization phase to generate the authentication tag. The first step of the finalization phase is updating state element $S^t_{4,0}$ by XOR-ing its content with the content of state element $S^t_{0,0}$, i.e., $S^t_{4,0} = S^t_{4,0} \oplus S^t_{0,0}$. The internal state is then updated 10 times using the state update function $Update(S^t, M^t)$ of MORUS. The external input M^t is set as $adlen||msglen$ for these 10 iterations. During these 10 updates the cipher does not produce any output. Finally, the tag τ is computed by using Equation 5.6.

$$\tau = S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{192} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \tag{5.6}$$

5.2.5.5 Decryption and Tag Verification

To start the decryption phase, first the initialization and the associated data processing phases are performed following the same process used for encryption.

Following this, at each step t of the decryption process the 256-bit output keystream block is computed using Equation 5.1. This output keystream block Z^t is XOR-ed with the ciphertext block C^t to compute the corresponding decrypted plaintext block. The decrypted plaintext block computation is shown in

Equation 5.7.

$$\begin{aligned} P^t &= C^t \oplus Z^t \\ &= C^t \oplus S^t_{0,0} \oplus \overleftarrow{S^t_{0,1}}^{192} \oplus (S^t_{0,2} \otimes S^t_{0,3}) \end{aligned} \tag{5.7}$$

Similar to MORUS-640, the recovered plaintext block P^t is used as the external input M^t to update the state. This process is continued until all the ciphertext blocks are processed.

After processing all the ciphertext blocks, the tag is generated following the same procedure as illustrated in Section 5.2.5.4. Finally for the tag verification, the received tag is matched with the tag generated after the decryption process.

5.3 Existing Analysis of MORUS

There is little public analysis of MORUS. In this section we briefly discuss the existing cryptanalysis of MORUS.

5.3.1 Distinguishing Attack on MORUS

Mileva et al. [135] described the existence of distinguisher for MORUS-640 which can analogously be used for MORUS-1280. Their analysis shows that for a given ciphertext C_1 computed on a message (D_1, P), an adversary can predict 125 bits of a different ciphertext C_2 computed over message (D_2, P), where D_1 and D_2 differ in one bit. However, these distinguishers are obtained when message (D_1, P) and message (D_2, P) are processed using the same key and initialization vector. This falls under the nonce-reuse scenario for which the designer of MORUS does not claim any security.

5.3.2 State Collisions and Forgery Attack on MORUS

Mileva et al. [135] also analysed the state update function of MORUS, reporting collisions in the internal state. To obtain the collision, an adversary needs to be capable of injecting specific differences in both the internal state and the external input. The designer of MORUS claims that this type of collision can not be achieved in practice because the adversary can only manipulate and inject differences in the external input. However, we would like to point out that an

adversary may consider the fault attack which allows injection of differences into the internal state.

Mileva et al. [135] also analysed the tag forgery attack by producing the state collision described above. The analysis is performed by generating and solving a system of equations which satisfies the state collision criterion. However, they reported no solutions were found apart from the trivial one where the difference between the input messages was zero.

5.3.3 Rotational and Differential Cryptanalysis of MORUS

Dwivedi et al. [40] applied the differential and rotational cryptanalysis on a reduced version of MORUS. The proposed key recovery approach first performs a pre-computation phase which needs to generate differential or rotational characteristics for a 1-bit internal state difference or 1-bit internal state rotation, respectively. In the online phase of their attack, the adversary guesses the differential/rotational counterpart of the key and accepts the key bit if it matches the characteristics generated during the pre-computation phase.

Their application of differential cryptanalysis on MORUS-1280-256 with 18 initialization rounds (3.6 steps) can recover the 256-bit key with a complexity of 2^{253}. They have also described a rotational attack on MORUS-1280-256 with 8 initialization rounds (1.6 steps) which can recover the key with a complexity of 2^{251}. Both of these attack complexities are almost the same as the exhaustive search attack and can not be extended to the full version of MORUS.

5.3.4 SAT Based State Recovery

Dwivedi et al. [125] analysed the complexity of SAT based state recovery for MORUS-640. Their approach based on SAT solver requires a complexity of 2^{370}. They concluded that the SAT based approach does not work for MORUS because it requires much more time than the exhaustive search of the key.

5.4 Cube Attack on MORUS

This section discusses the applicability of the cube attack [93] to different phases of MORUS. Recall that the key and initialization vector are loaded into the elements of the internal state during the initialization phase. Following this,

the associated data and plaintext are injected into the internal state during the associated data loading and encryption phases, respectively. Therefore the output polynomial of MORUS is expected to contain variables from the key, initialization vector, associated data and plaintext; since the output keystream of MORUS is computed in terms of the internal state. As a result, cube attacks on MORUS can be performed either in the initialization phase, the associated data loading phase, the encryption phase or in the decryption phase.

If the attack is performed during the initialization phase, then the cube bits are chosen from the input initialization vector. The goal of the attack is to recover the secret key of the cipher if applied to the initialization phase.

If the attack is performed during the associated data processing phase, then the cube bits are chosen from the input associated data, while if the attack is performed during the encryption phase then the cube bits are chosen from the input plaintext. The goal of the attack is state recovery if applied to either the associated data loading phase or the encryption phase. The state recovery may then be used to reveal the secret key K.

Note that a cube attack on either the associated data loading phase or the encryption phase requires the attacker to choose the cube bits from the associated data and the plaintext bits, respectively. Therefore, the sum over all the possible values of the cube chosen from the associated data/plaintext needs to be calculated. This means that cube attack during these two phases requires authentication/encryption of multiple sets of associated data/plaintext using the same key and initialization vector. This falls under the nonce-reuse scenario, for which the designer of MORUS does not claim any security. Therefore we do not consider these further; instead, we focus our analysis on the initialization phase of MORUS.

5.4.1 Cube Attack on the Initialization Phase

In the initialization phase the key and initialization vector are loaded into the internal state of MORUS. Therefore the cube bits can be selected from either the key bits or the initialization vector bits. However, an adversary needs to have the capability to manipulate the secret key when the cube bits are chosen from the key. So, we consider the more feasible scenario where the cube bits are chosen from the initialization vector. The attack follows the chosen initialization vector model. The goal of the cube attack applied to the initialization phase of

Table 5.2: Estimated Degree Accumulation of MORUS-640 State Contents and the Output Function

	Degree						
Step	S_0	S_1	S_2	S_3	S_4	Output	Attack Complexity
0	0	1	0	0	0	1	$2^0 \times 2^7$
1	1	1	1	1	2	2	$2^1 \times 2^7$
2	2	2	3	4	4	7	$2^6 \times 2^7$
3	5	7	8	9	12	17	$2^{16} \times 2^7$
4	15	17	21	27	32	48	$2^{47} \times 2^7$
5	38	48	59	70	86	128	$2^{127} \times 2^7$

MORUS is to recover its secret key.

The application of a cube attack to the initialization phase of MORUS requires finding appropriate cube bits in the initialization vector which generate linear superpolys in terms of the key bits. This is done in the preprocessing phase of the attack following the steps mentioned in Section 2.4.3.2. For a 128-bit secret key of MORUS, an adversary requires to find 128 independent linear superpolys.

In the online phase of the attack, an adversary needs to evaluate the output function for all of the cubes computed in the preprocessing phase. This evaluation in the online phase determines the value of the corresponding linear superpolys following the steps in Section 2.4.3.2.

5.4.1.1 Estimated Complexity Analysis of Cube Attack

In this section we discuss the estimated complexity growth for the cube attack as the number of steps in the initialization phase of MORUS increases. The size of the cube is closely related to the degree d of the output function. For an output polynomial of degree d, the corresponding cube size is at most $d-1$. Choosing a cube size of degree $d-1$ guarantees that a linear superpoly will be found. For a degree d polynomial, we can avoid the linearity test of the superpoly if the chosen cube is of size $d-1$.

The degree of MORUS state contents can be estimated by defining the key bits as variables and then computing the degree of the updated state contents after each round of operations. For a linear update involving the rotation of the state contents, we estimated that the degree of the updated contents will remain the same. On the other hand consider a linear update involving a XOR operation of

Table 5.3: Estimated Degree Accumulation of MORUS-1280 State Contents and the Output Function

	Degree						
Step	S_0	S_1	S_2	S_3	S_4	Output	Attack Complexity
0	0	1	0	0	0	1	$2^0 \times 2^8$
1	1	1	1	1	2	2	$2^1 \times 2^8$
2	2	2	3	4	4	7	$2^6 \times 2^8$
3	5	7	8	9	12	17	$2^{16} \times 2^8$
4	15	17	21	27	32	48	$2^{47} \times 2^8$
5	38	48	59	70	86	128	$2^{127} \times 2^8$
6	107	129	156	193	236	256	$2^{255} \times 2^8$

the contents of two state elements. In this case, the larger valued degree between these two state elements is estimated as the degree of the updated state contents. For a nonlinear update involving an AND operation of the contents of two state elements, the updated state content's degree is estimated as a summation of the degree for the contents of the respective state elements.

Table 5.2 shows the estimated degree accumulation for the MORUS-640 state contents and the output function. The degree of the output function grows significantly with the increase in the number of steps. This is because of the AND operation involved in each round of every step. There are five AND operations involved in each step. In each round the state update is expected to double the degree of the contents because of the application of the AND operation. The output polynomial is represented in terms of the state contents and therefore the degree of the output polynomial is also expected to grow more than double at each step. At this rate the degree of the output polynomial of MORUS-640 is expected to reach the maximum possible degree of 128 after only five steps of initialization phase. Table 5.2 also shows the increase in the complexity of cube attack with the increases in the number of steps. It seems that the cube attack would be infeasible just after few steps of initialization phase if the cube sizes are chosen as $d - 1$.

Table 5.3 shows the estimated maximum degree accumulation for the MORUS-1280 state contents and the output function. Similar to MORUS-640, the degree of the output function grows significantly with the increase in the number of steps. This is because of the AND operation involved in each round of every step. At this rate the degree of the output polynomial of MORUS-1280 is expected to reach the maximum possible degree of 256 after only six steps of initialization

Table 5.4: Total Possible Search Spaces for Different Cube Sizes of MORUS

Cube Size	Search Space	Exhaustive Search Complexity
1	128	$2^7 \times 2^1$
2	8128	$2^{12.99} \times 2^2$
3	341376	$2^{18.38} \times 2^3$
4	10668000	$2^{23.35} \times 2^4$
5	264566400	$2^{27.98} \times 2^5$
6	5423611200	$2^{32.34} \times 2^6$
7	94525795200	$2^{36.46} \times 2^7$
8	1429702652400	$2^{40.38} \times 2^8$
⋮	⋮	⋮

phase. Table 5.3 also shows the increase in the complexity of cube attack with the increases in the number of steps. It seems that the cube attack would be infeasible just after few steps of initialization phase if the cube sizes are chosen as $d-1$.

Note that for a polynomial of degree d, there may exist other cubes which are of size less than $d-1$. This happens if the initialization vector is not mixed well with the secret key. This behaviour is expected to occur at least in the first couple of steps of the initialization phase. In our analysis we investigate the existence of such lower dimension cubes. Searching for these lower dimension cubes involves testing cubes of different sizes.

To perform cube attack on the initialization phase of MORUS, we selected cube bits from the 128-bit initialization vector. For a cube of size n, there are $\binom{128}{n}$ possible cube choices. Table 5.4 shows the increase in the size of the corresponding search space with the increase in the cube sizes.

It is evident from Table 5.4 that the search space increases significantly with the increase in the cube sizes, and it will not be possible to exhaustively test all the possible cubes over the whole search space.

5.4.2 Applying Cube Attack on MORUS

We conducted experiments to analyse the feasibility of the cube attack on the initialization phase of MORUS. Our analysis is performed using Sage 6.4.1 [144] and Python 3.6, on a standard 3.4 GHz Intel Core i7 PC with 16 GB memory.

In all variants of MORUS, the initialization phase has 16 steps. Using Sage we tried to generate the output equations of MORUS symbolically to determine its

degree. We found that after two initialization steps the degree of the symbolical representation of MORUS output equation is the same as the theoretical analysis tabulated in Table 5.2. However, Sage ran out of memory when we tried to generate the equations symbolically for more than two steps of the initialization process.

As discussed in Section 5.4.1.1, the degree d of the output equation is expected to reach the maximum for the full version of MORUS. This means the expected cube size $d-1$ is also very high for the full version of MORUS. Evaluating cubes directly related to the degree requires high computational time for the full step version of MORUS. So, we modified the MORUS design by considering fewer steps for the state update function during the initialization phase.

Also, as illustrated in Table 5.4 it is not possible to exhaustively test all the possible cubes over the whole search space for a reasonable cube size. Therefore in our experiment we tested the existence of lower dimension cubes and the cubes are chosen randomly instead of searching over all possible cube choices. The chances of finding a cube with linear superpoly may increase with the increase in the numbers of cube tested; however, this will also increase the time complexity of the preprocessing phase.

We conducted experiments on the reduced version of MORUS-640 and MORUS-1280-128, to find the existence of lower dimension cubes. In our experiments, the length of the associated data is set to zero (to prevent degree accumulation) since MORUS does not produce any output during the associated data processing phase.

5.4.2.1 Attack Algorithm

The steps for finding lower dimension cubes in the preprocessing phase of our experiments follows the similar steps as outlined in Section 2.4.3.2 of Chapter 2, except that we started the experiment with a cube size of two instead of a random cube size. The online phase of our experiment follows the same steps as outlined in Section 2.4.3.2 of Chapter 2.

Finding New Cubes Using Existing Cube In this section we discuss the technique for finding new cubes using the existing ones, found using the random search technique illustrated in Section 2.4.3.2 of Chapter 2. To determine new cubes resulting in linear superpolys, we can increase/decrease the cube indices

and the respective output index by one. The new cube is valid if it satisfies the linearity test and if the resultant linear superpoly is not a constant. When increasing the cube indices, the process is continued until the upper limit is reached for either any of the cube indices or the output index, i.e., any of the cube indices or the output index reaches the value 127 or 255 for MORUS-640 and MORUS-1280, respectively. Similarly, when decreasing the cube indices, the process is continued until the lower limit is reached for either any of the cube indices or the output index, i.e., any of the cube indices or the output index reaches the value 0.

5.4.2.2 Applying Cube Attack on MORUS-640

We applied the cube attack to MORUS-640 with an initialization phase of 4 steps. The preprocessing phase of the attack was performed following the steps mentioned in Section 5.4.2.1. To search for the cubes, we started with a cube size of 2. We tested over 20,000 random cubes of size 2; however, none of these random cubes passed the linearity test. We then increased the cube size to 3 and searched over 20,000 random cubes. Each cube was tested using 50 linearity tests. We found 2344 linear superpolys for cube size 3, among which only 192 are non-constant. Note that 103 of these 192 equations consists of only a single variable.

Further experiments reveals that a lot of these 192 cubes failed the linearity test when the number of tests are increased from 50 to 100. This indicates that 50 linearity tests are not sufficient. We expect that the superpolys found with the 100 linearity tests will be linear with high probability. We focused our experiments on the specific 103 cubes mentioned above and obtained only 34 cubes where the resultant superpolys passed 100 linearity tests. These cubes are listed in Table 5.5.

Recall that at each step MORUS-640 encrypts a plaintext block of 128-bits. Therefore at each step we can observe 128 output bits. In our experiment we considered the first output block. In Table 5.5, the output indices refer to the corresponding bits of the first output block. From Table 5.5 we observe that some of these cubes result in the same linear superpolys. Analysing Table 5.5 we identified that there are 31 linearly independent superpolys.

We used the technique described in Section 5.4.2.1 for finding new cubes using the cubes listed in Table 5.5. Using this technique we obtained more than

Table 5.5: Example of Linear Superpolys Obtained for MORUS-640 with 4 Steps of Initialization Phase

Cube Indices	Output Index	Superpoly
49, 13, 110	20	k_4
27, 107, 61	17	k_{18}
108, 106, 51	58	k_{18}
120, 115, 109	41	$k_{19} \oplus 1$
7, 6, 33	83	k_{44}
53, 35, 29	26	k_{44}
7, 13, 0	108	k_{45}
102, 83, 22	123	$k_{60} \oplus 1$
107, 104, 58	95	$k_{64} \oplus 1$
69, 21, 9	59	$k_{52} \oplus 1$
85, 10, 124	65	$k_{34} \oplus 1$
58, 68, 40	31	k_{49}
12, 91, 60	50	$k_{51} \oplus 1$
7, 104, 125	60	$k_{87} \oplus 1$
102, 119, 56	93	$k_{94} \oplus 1$
1, 66, 19	103	$k_{104} \oplus 1$
82, 106, 3	58	k_{120}
76, 36, 51	24	$k_{89} \oplus 1$
25, 38, 27	114	$k_{102} \oplus 1$
72, 15, 44	75	$k_{35} \oplus 1$
87, 63, 112	7	$k_{69} \oplus 1$
4, 23, 63	123	$k_{69} \oplus 1$
32, 46, 70	5	k_{55}
93, 5, 59	10	$k_{114} \oplus 1$
29, 39, 106	79	$k_{16} \oplus 1$
16, 122, 91	15	k_{42}
53, 59, 118	0	$k_{65} \oplus 1$
66, 20, 44	104	k_{82}
20, 71, 25	109	$k_{97} \oplus 1$
42, 120, 81	30	k_{119}
28, 31, 46	106	k_{84}
65, 41, 79	38	$k_{88} \oplus 1$
21, 125, 46	69	$k_3 \oplus 1$
57, 118, 21	28	k_{12}

300 linear superpolys in terms of the key bits. The majority of these equations consist only of a single variable. The linear superpolys obtained through this method cover all of the key bits except k_{22}, k_{53} and k_{118}. An adversary needs

to guess these three secret key bits and solve the equation system in the online phase to recover the rest of the secret key bits. A list of 125 linear independent superpolys chosen from these 300 linear superpolys is provided in Appendix B.1.

In the online phase of the cube attack on the reduced version of MORUS-640, an adversary first needs to find the value of the linear superpolys. From the preprocessing phase, we can select 125 independent linear superpolys with 125 variables. This requires an adversary to observe $2^{9.97}$ output ciphertext/keystream bits for $125 \times 2^3 \approx 2^{9.97}$ chosen initialization vectors. Also note that most of these linear superpolys consist of only a single variable of the secret key. Only eight of the superpolys contains two variables. Thus, the complexity of solving these linear system of equation is negligible.

We implemented the online phase of the cube attack to verify the correctness of the linear superpolys. We started with a randomly generated 128-bit secret key. We then computed the values of all the linear superpolys by summing the output keystream bits for all the possible values (varying the corresponding initialization vector bits) of the respective cubes. The rest of the initialization vector bits are set to zero. To compute the value of the linear superpolys, we accessed $2^{9.97}$ keystream bits of the first output block constructed over $2^{9.97}$ chosen initialization vectors. We then reconstructed and solved the linear equations. This correctly recovers 125 of the secret key bits. Note that three of the secret key bits do not appear in any of the linear superpolys found in our experiment. We need to guess these three secret key bits to recover the whole key. So the total attack complexity of the online phase is about $2^{9.97} + 2^3 \approx 2^{9.98}$.

5.4.2.3 Applying Cube Attack on MORUS-1280-128

We applied the cube attack to MORUS-1280-128 with an initialization phase of 4 steps. In the preprocessing phase we conducted experiments to find cube variables from the initialization vector bits by randomly selecting cubes of different sizes and testing them for linearity. We started with a cube size of 2 and tested 20,000 random cubes on MORUS-1280-128. Each of these cubes were tested for 100 linearity tests using a randomly selected output bit of the first ciphertext block.

Unlike MORUS-640, we found cubes of size 2 which resulted in linear superpolys for the reduced version of MORUS-1280-128. We found 3947 cubes where each cube passed at least 100 linearity tests. However most of these cubes re-

Table 5.6: Example of Linear Superpolys Obtained for MORUS-1280-128 with 4 Steps of Initialization Phase

Cube Indices	Output Index	Superpoly
7, 68	171	$k_{66} \oplus 1$
75, 64	158	k_{31}
79, 8	159	k_6
101, 117	127	$k_{58} \oplus 1$
66, 67	13	k_7
95, 20	1	$k_{62} \oplus 1$
53, 42	72	$k_{73} \oplus 1$
19, 75	183	k_{114}
78, 62	218	$k_{64} \oplus 1$
49, 48	251	$k_{117} \oplus 1$
89, 2	53	k_{27}
96, 40	42	k_{106}
0, 56	37	$k_{97} \oplus 1$

sulted in a constant. We found only 13 linear superpoly which are non-constant. Cubes resulting in non-constant linear superpolys are listed in Table 5.6.

Recall that at each step MORUS-1280-128 encrypts the plaintext blocks of 256-bits. Therefore at each step we can observe 256 bits of the first output keystream/ciphertext block. In Table 5.6, the output indices refer to the corresponding bits of the first output block.

We used the technique described in Section 5.4.2.1 for finding new cubes using these 13 cubes listed in Table 5.6. Using this technique we obtained 408 linear superpolys in terms of 128 of the key bits. These linear superpolys cover all the 128 secret key bits of MORUS-1280-128. An adversary can easily select 128 independent linear superpolys covering all the secret key bits and solve those in the online phase of the attack to recover the secret key. A list of 128 linear independent superpolys (chosen from these 408 linear superpolys) covering all the secret key bits is presented in Appendix B.2.

The cubes found for the reduced version of MORUS-640 and MORUS-1280-128 are of size 3 and 2, respectively. This indicates MORUS-1280 has a slower diffusion compared to MORUS-640. This is due to the differences in loading format of the internal state. In MORUS-1280, one of the state elements is loaded with all zero values which may have resulted in the comparatively slower diffusion.

To determine the value of the linear superpolys, the online phase of cube at-

tack on MORUS-1280-128 requires an adversary to observe 2^9 ciphertext/keystream bits for $128 \times 2^2 = 2^9$ chosen initialization vectors. Note that all the linear superpolys obtained for MORUS-1280-128 consist of only a single variable of the secret key. So the complexity of solving these equations are negligible. The complexity of the total attack is 2^9.

To verify the correctness of the linear superpolys, we implemented the online phase of the cube attack on the 4 step initialization version of MORUS-1280-128. We started by selecting a randomly generated key. We then computed the value of all the linear superpolys by summing the output keystream bits for all the possible values (varying the corresponding initialization vector bits) of the respective cubes. The rest of the initialization vector bits were set to zero. To compute the sum over all the cube bits, we accessed the respective output keystream bits of the first output block for 2^9 chosen initialization vectors. This requires access to 2^9 keystream bits. Following this we were able to recover all of the 128 secret key bits.

5.4.2.4 Applying Cube Attack on MORUS-1280-256

We applied the cube attack to MORUS-1280-256 with an initialization phase of 4 steps. Note that the cubes found for MORUS-1280-128 can also be used for recovering 128 bits of the 256 bits key of MORUS-1280-256. We need to find 128 more equations to recover the rest of the key of MORUS-1280-256.

We therefore performed the preprocessing phase by conducting experiments to find cube variables from the initialization vector bits. The preprocessing phase of the attack is performed following the steps mentioned in Section 5.4.2.1. We started the experiments with cube size 2 and tested 50,000 random cubes. Each of these cubes were tested for 100 linearity tests using a randomly selected output bit of the first ciphertext block.

We found cubes of size 2 which resulted in linear superpolys for the 4-step version of MORUS-1280-256. Among the tested 50,000 random cubes, we found 40,034 of them passed the 100 linearity tests. However, most of these cubes resulted in a linear superpoly which only consists of a constant. We found only 18 linear superpolys which are non-constant. These cubes are listed in Table 5.7.

With the random search technique, we obtained only 18 linear superpolys. This is not sufficient to recover the 256 bit key of MORUS-1280-256. So, we used the technique described in Section 5.4.2.1 for finding new cubes using the

Table 5.7: Example of Linear Superpolys Obtained for MORUS-1280-256 with 4 Steps of Initialization Phase

Cube Indices	Output Index	Superpoly
6, 29	10	k_{72}
123, 117	23	$k_{148} \oplus 1$
102, 22	242	$k_{88} \oplus 1$
107, 106	53	k_{47}
34, 35	237	k_{231}
126, 115	145	k_{146}
1, 81	221	k_{67}
62, 46	27	k_{60}
11, 102	91	k_{133}
66, 5	169	$k_{64} \oplus 1$
51, 107	53	$k_{117} \oplus 1$
92, 44	132	$k_{187} \oplus k_{90}$
106, 101	198	$k_6 \oplus 1$
96, 21	2	$k_{191} \oplus 1$
79, 80	26	k_{20}
73, 2	153	$k_0 \oplus k_{97}$
77, 66	160	k_{161}
101, 117	127	$k_{58} \oplus 1$

cubes listed in Table 5.7. Using this technique we obtained 697 non-constant linear superpolys. These linear superpolys cover all the secret key bits except $k_{192}, k_{250}, k_{251}, k_{252}, k_{253}, k_{254}, k_{255}$. An adversary needs to guess these seven secret key bits and solve the equation system of 249 linear superpolys in the online phase to recover the rest of the key bits. A list of 249 linear independent superpolys chosen from these 697 linear superpolys is presented in Appendix B.3.

In the online phase, an adversary first needs to find the value of the linear superpolys. From the preprocessing phase, adversary can select 249 independent linear superpolys with 249 variables of the secret key. This requires an adversary to observe $2^{9.96}$ output ciphertext/keystream bits for $249 \times 2^2 \approx 2^{9.96}$ chosen initialization vectors. Also note that most of these linear superpolys consist of only a single variable of the secret key. Thus, the complexity of solving these linear system of equation is negligible.

To verify the correctness of the linear superpolys, we implemented the online phase of the cube attack on the 4 step initialization version of MORUS-1280-256. We started the online phase by selecting a randomly generated key and

then computed the value of all the linear superpolys by summing the output keystream bits for all the possible values (varying the corresponding initialization vector bits) of the respective cubes. For this we accessed the respective output bits of the first output block for $2^{9.96}$ chosen initialization vectors. This required to access $2^{9.96}$ keystream bits. We then reconstructed and solved the linear equation system, which correctly recovered the 249 of the secret key bits.

Recall that seven of the secret key bits do not appear in any of the linear superpolys found in our experiment. These seven key bits need to be guessed to recover the entire key. Therefore, the total complexity of the online phase of the attack on MORUS-1280-256 is about $2^{9.96} + 2^7 \approx 2^{10.13}$.

5.4.2.5 Applying Cube Testers on MORUS-1280

We conducted experiments searching for cubes on modified versions of MORUS with five or more initialization steps. In the preprocessing phase of these experiments, we did not find any cubes which resulted in non-constant superpolys; however, for MORUS-1280 with an initialization phase of five steps, we found cubes of size 9 which always resulted in cube sum being equal to a constant.

This suggests that we can use these cubes of size 9 as distinguishers for MORUS-1280 when the initialization steps are reduced to five. These cubes are listed in Table 5.8. These cubes passed at least 100 linearity tests.

In the online phase, an adversary can evaluate any of the cubes listed in Table 5.8. If the adversary is given access to 2^9 ciphertext/keystream for 2^9 chosen initialization vectors, they can sum those ciphertext/keystream bits and use the sum to distinguish the output of the cipher from a randomly generated output. The complexity of distinguishing this modified version of MORUS-1280 is 2^9.

New distinguishers can be obtained using these existing cubes following the technique described in Section 5.4.2.1. Experimental analysis shows that with this technique we can find more cubes of size 9, resulting in distinguishers for five steps initialization version of MORUS-1280.

To obtain cubes of size 8 or less we extended our experiments by removing one or more of the cube variables from the cubes listed in Table 5.8. Evaluating the linearity of the superpolys for such cubes, we found some valid cubes of size 8. None of these cubes resulted in a non-constant superpoly. Therefore, these new cubes can only be used as a distinguisher for the reduced version of

Table 5.8: Cubes Obtained for MORUS-1280 Resulting in Distinguishers with 5 Steps of Initialization Phase

Cube Indices	Output Index
39, 30, 26, 110, 77, 56, 28, 70, 32	219
61, 29, 32, 46, 103, 115, 116, 26, 28	219
88, 25, 56, 39, 45, 70, 16, 13, 94	236
49, 109, 53, 78, 114, 127, 68, 59, 93	244
13, 111, 1, 12, 43, 28, 26, 120, 5	163
22, 78, 100, 116, 111, 94, 25, 103, 31	216
50, 41, 110, 24, 82, 10, 43, 11, 91	130
46, 20, 24, 108, 116, 121, 100, 52, 42	133
99, 16, 25, 80, 61, 64, 21, 19, 110	216
98, 118, 101, 22, 53, 75, 89, 117, 110	205
50, 84, 113, 46, 80, 38, 118, 76, 32	227
106, 109, 68, 9, 94, 114, 86, 25, 19	104
68, 70, 56, 98, 91, 52, 109, 53, 95	196
16, 20, 106, 42, 35, 32, 10, 7, 70	189
8, 45, 103, 91, 90, 94, 29, 32, 22	218
48, 96, 79, 76, 122, 25, 53, 127, 94	216
53, 38, 94, 96, 85, 84, 123, 17, 58	19
81, 74, 82, 25, 42, 121, 15, 4, 32	183
44, 76, 42, 17, 23, 127, 80, 115, 75	15
106, 122, 20, 54, 120, 58, 75, 55, 45	205
65, 97, 90, 92, 108, 93, 38, 87, 74	243
11, 17, 35, 55, 6, 33, 38, 83, 114	230

MORUS-1280. We did not find any cubes of size 7 or less for reduced version of MORUS-1280. The best cubes obtained for MORUS-1280 with five initialization steps distinguishes the cipher output from random with complexity 2^8.

5.4.3 Summary of Cube Attacks on MORUS

We applied the cube attack to reduced versions of MORUS-640, MORUS-1280-128 and MORUS-1280-256, where the modification is a reduced initialization phase of 4 steps. The cube attack can successfully recover the key for reduced version of MORUS-640, MORUS 1280-128 and MORUS-1280-256 with an approximated complexity of 2^{10}, 2^9 and 2^{10}, respectively.

The cubes identified for reduced version of MORUS-640, MORUS-1280-128 and MORUS-1280-256 are of size 3, 2 and 2, respectively; while the actual degree of the output equation after 4 steps for these variants of MORUS is much higher

Table 5.9: Comparison of Cube Attacks on Different Variants of MORUS

Algorithm	No. of Initialization Steps	Attack Type	Cube Size	Complexity
MORUS-640	4	Key Recovery	3	2^{10}
MORUS-1280-128	4	Key Recovery	2	2^{9}
MORUS-1280-256	4	Key Recovery	2	2^{10}
MORUS-640	4	Distinguisher	3	2^{3}
MORUS-1280	5	Distinguisher	8	2^{8}

than 3. This means that after 4 steps of the initialization phase there exist comparatively lower degree monomials in the output equation which do not appear together with any other monomials of that equation. This indicates that the key and initialization vectors are not mixed properly at this point of the initialization phase.

We also observed that the cubes obtained for the reduced version of MORUS-1280 are of smaller size compared to the ones obtained for MORUS-640. This indicates MORUS-1280 has a slower rate of diffusion compared to MORUS-640. This happens due to the differences in loading format of the internal state. In MORUS-1280, one of the state elements is loaded with all zero values which may have resulted in the comparatively slower diffusion.

We searched also for cubes with a higher number of steps in the initialization phase. We did not find any cubes resulting in non-constant linear superpolys when the number of initialization steps is more than 4. However, for MORUS-1280 with 5 initialization steps, we obtained cubes which result in constant. These cubes can be used as distinguishers for MORUS-1280 with 5 steps of initialization. The complexity of this attack 2^8. Table 5.9 illustrates the feasibility of cube attack applied to different variants of MORUS.

Table 5.9 illustrates that our application of cube attack on MORUS-640 works only for 4 steps of the initialization phase, whereas for MORUS-1280 we were able to extend this up to 5 steps of the initialization phase. This is due to the comparatively slower diffusion rate of MORUS-1280. Also as illustrated in Table 5.9, for MORUS-1280 the cubes resulting in a distinguisher for the 5 steps initialization phase are of size 8; while the cubes for 4 steps initialization phase are of size 2. This shows that the size of the cube is increased at least by an amount of six. Even with the increase of this amount, we were only able to

Table 5.10: Comparison of Cube Attack with Other Existing Key Recovery Attacks on MORUS-1280-256

Number of Initialization Steps	Attack Type	Complexity
1.6	Rotational Attack [40]	2^{253}
3.6	Differential Attack [40]	2^{251}
4	Cube Attack	$2^{10.13}$

construct a distinguisher. This indicates that the increases in the cube size are significant with the increase in the number of initialization steps and therefore the complexity of the attack also grows significantly with the increase in the the number of initialization steps.

The results show that key recovery with the cube attack performs better than the rotational and differential attacks [40] applied on reduced versions of MORUS. Table 5.10 compares the performance of the cube attack against the existing rotational and differential key recovery attack on MORUS-1280. The comparison is done in terms of the number of initialization steps that the attack can be applied to and also the required complexity of the attack. In Table 5.10, we only compare the cube attack on MORUS-1280-256 since the key recovery attacks by Dwivedi et al. [40] was demonstrated only for this variant.

As illustrated in Table 5.10, the rotational and differential cryptanalysis approaches described by Dwivedi et al. [40] are applied on 1.6 steps and 3.6 steps of MORUS-1280-256, respectively; whereas cube attack is applied to 4 steps of MORUS-1280-256. Also, we can see from the Table 5.10 that the complexity of the rotational and differential cryptanalysis is only slightly less than the exhaustive search. Compared to that, the cube attack approach demonstrated on the reduced version of MORUS can recover the secret key with practical complexity.

It is hard to estimate the size of the cubes for higher number of initialization steps in MORUS without knowing the exact algebraic normal form of the output polynomial. We can choose cube size based on the estimated algebraic degree. However, the estimated growth in the degree is significant for each step of the state update; thus the complexity of the attack is expected to grow exponentially. This is illustrated in Table 5.2.

As indicated in Table 5.2, the degree of the output polynomial of MORUS-640 is expected to reach the maximum possible degree of 128 for only five steps of the initialization phase. Also, Table 5.2 indicates that the complexity of the attack is about 2^{134} after five steps of the initialization phase of MORUS-640. This is worse

than the exhaustive search on the key space of MORUS-640. Similarly, Table 5.3 indicates that the degree of the output polynomial of MORUS-1280 is expected to reach the maximum possible degree of 256 after six steps of the initialization phase. Table 5.3 illustrates that the complexity of the cube attack is about 2^{263} for six steps of the initialization phase of MORUS-1280, which is worse than the exhaustive search. Therefore, applying the cube attack by selecting the cube sizes based on the degree of the output function will be impractical just after a few steps of initialization phase. Currently, the cube attack does not threaten the security of MORUS if the full initialization phase is performed.

5.5 Fault Attacks on MORUS

A fault attack [102] is a type of side-channel attack which works on the physical implementation of a cryptographic algorithm. The basic idea of this attack requires an adversary to introduce some error(s) in the underlying implementation of the cryptographic primitive. The adversary then tries to recover the secret key of the cryptosystem by observing the original and faulty output. This section provides our analysis of the application of fault attacks to MORUS. We applied two different types of fault based key recovery attack on MORUS. The first fault attack described is a permanent fault attack, whereas the second fault attack is a transient fault attack.

The fault attacks described in this section are based on the observation that the key is directly XOR-ed with the internal state element $S_{0,1}$ during the last step of the initialization phase. The associated data processing phase is performed after the initialization phase, and is followed by the encryption phase. During the associated data phase no keystream bits are generated. However, if there is no associated data processing phase, i.e., associated data length is zero, then the key will be XOR-ed in the first block of keystream. This is because the content of state element $S_{0,1}$ is used in the keystream generation function. An adversary can introduce faults to eliminate the effect of the rest of the inputs in the keystream generation function and observe the first block of keystream to recover the key. This works under the assumption that there is no associated data processing phase and therefore all the fault attacks described below are based on this assumption.

On the other hand, if the associated data processing phase is performed,

then the key bits will get mixed with the contents of other state element before appearing in the output keystream. In this case, the fault attack will not work since the key bits are not directly XOR-ed in the output keystream generation function.

5.5.1 Permanent Fault Attack on MORUS-640

For MORUS-640 the initialization phase is updated 16 times to mix the 128-bit key and 128-bit initialization vector in to the 640 bits internal state. We observe that after these 16 operations the 128-bit key is directly XOR-ed to update the contents of state element $S_{0,1}^{t=16}$. Also note that the state element $S_{0,1}$ is used in the computation of the output keystream generated at time $t = 16$ (see Equation 5.2). MORUS-640 processes the input plaintext after completing the initialization and associated data processing phase. During the associated data processing phase no output keystream is generated. Output keystream bits are generated only during the encryption phase. So if the associated data processing phase is skipped, i.e., there is no input associated data, then the key bits will be directly XOR-ed in the output keystream function while processing the first plaintext block. When there is no associated data processing phase, Equation 5.2 for the first plaintext block can be represented as:

$$C^{16} = P^{16} \oplus S_{0,0}^{16} \oplus \overleftarrow{S_{0,1}^{16} \oplus K}^{96} \oplus (S_{0,2}^{16} \otimes S_{0,3}^{16}) \tag{5.8}$$

In Equation 5.8, if we can remove the effect of $S_{0,0}^{t=16}$, $S_{0,1}^{t=16}$, $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ then the ciphertext $C^{t=16}$ will just be the XOR of the plaintext $P^{t=16}$ and the key K. In such a scenario one can recover the key bits under a known plaintext model. To remove the effect of the above mentioned state elements an adversary can introduce a permanent fault to set the contents of some specific registers to zero, i.e., destroying the registers. The locations that we target for our fault attack are shown in Figure 5.2 and discussed in detail below.

We observe that during the 16th step (last update step) of the initialization phase, destroying the last 64 registers (i.e., register 64 to 127) of state element $S_{3,1}^{t=15}$ will result in all zero bits in the state element $S_{4,1}^{t=15}$. This is due to the fact that at the fourth round of each step, the contents of $S_{3,1}$ are rotated left by 64 bits. The left rotation will fetch the zero valued contents of the last 64 bits (faulty bits valued zero) in to the first 64 register bits of state element $S_{4,1}$.

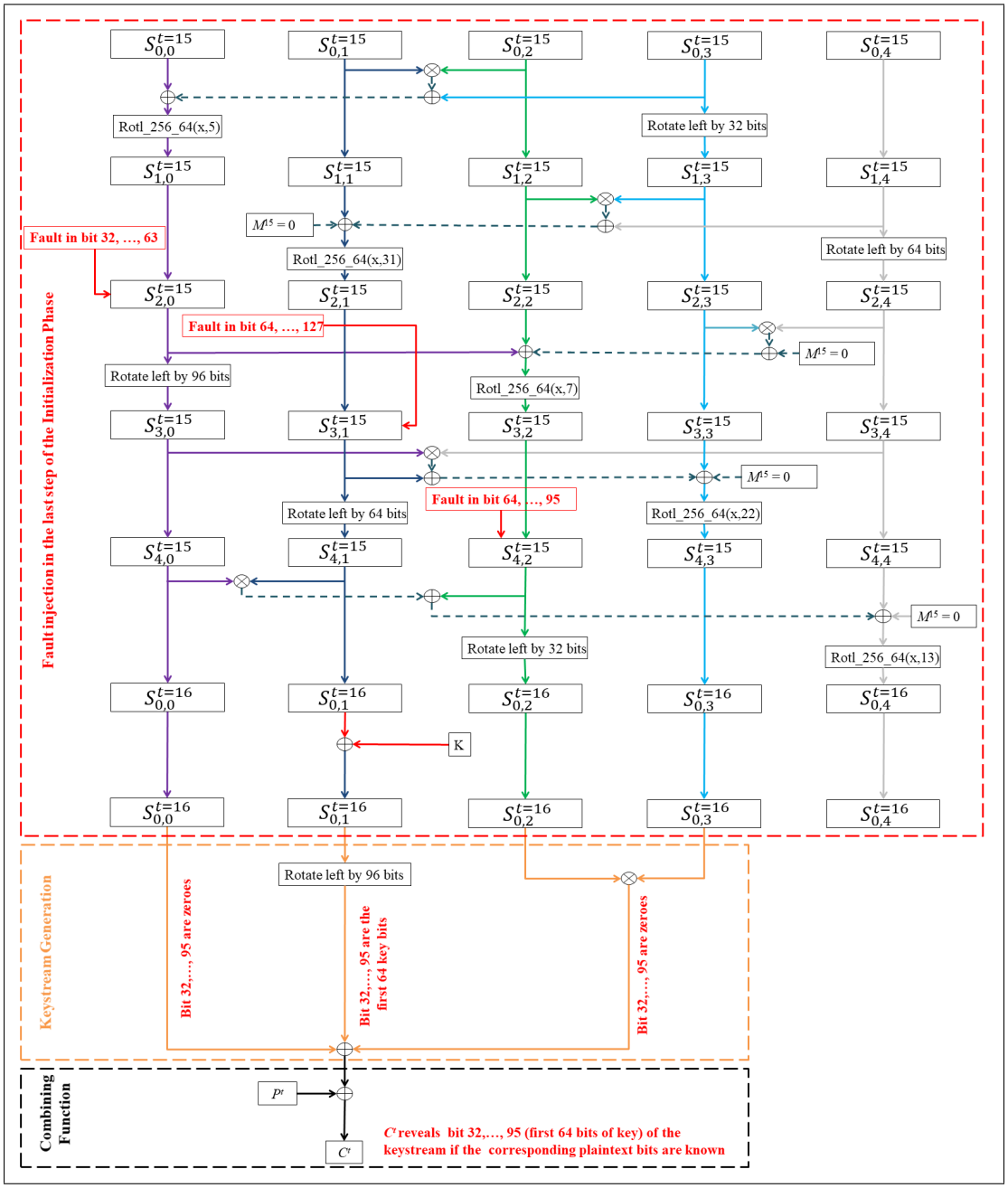

Figure 5.2: Inducing Permanent Fault at the Last Step of the Initialization Phase of MORUS-640

Since the last 64 registers are permanently set to zero, the rotation will not have any effect on these bits. After the left rotation there are no changes in the state element $S_{4,1}^{t=15}$ for that step, i.e., $S_{4,1}^{t=15} = S_{0,1}^{t=16}$. As mentioned earlier, the key bits are XOR-ed to the state element $S_{0,1}^{t=16}$ after the 16 times update of initialization phase. As a result, at the end of the initialization phase the first 64 bits of state

element $S_{0,1}^{t=16}$ will contain the first 64 bits of the key bits. Due to the permanent fault, XOR-ing of the last 64 bits of the key will have no effect on this state element. Note that in the encryption phase, the combining function rotates the contents of this state element by 96 bits. Thus the first 64 key bits will actually be XOR-ed at the bit position $32, \cdots, 95$ during the ciphertext computation. One can recover the first 64 bits of the corresponding key by eliminating the effect of the bit $32, \cdots, 95$ of state element $S_{0,0}^{t=16}$, $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ from the first ciphertext block. To remove the effect of these bits one can follow a similar technique which was used for state element $S_{0,1}^{t=16}$.

To eliminate the effect of bits $32, \cdots, 95$ of state element $S_{0,0}^{t=16}$ in the keystream computation, one can permanently set bits $32, \cdots, 63$ of state element $S_{2,0}^{t=15}$ to zero (i.e., destroy the register bits 32 to 63). At the third round of each step, contents of state element $S_{2,0}$ goes through a left rotation of 96 bits. Therefore introducing 32 bits of permanent fault in the state element $S_{2,0}^{t=15}$ will result in zero in bits $32, \cdots, 95$ of state element $S_{3,0}^{t=15}$. The left rotation will fetch the zero valued contents of the bit $32, \cdots, 63$ (faulty bits) in to the register bits $64, \cdots, 95$ of state element $S_{3,0}$. The left rotation will not have any effect on the register bits $32, \cdots, 63$ due to the permanent fault. As a result bits $32, \cdots, 95$ of state element $S_{0,0}^{t=16}$ will be set to zero, since there are no changes in state element $S_{3,0}$ after the third round of each step.

Also in Equation 5.8 observe that $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ are multiplied bitwise and used in the ciphertext computation. Therefore setting bits $32, \cdots, 95$ of either of these two state elements to zero will eliminate the effect of both in the keystream computation. To introduce zero valued contents in bits $32, \cdots, 95$ of state element $S_{0,2}^{t=16}$, one can permanently set bits $64, \cdots, 95$ of state element $S_{4,2}^{t=15}$ to zero, i.e., destroy the corresponding registers. At the fifth round of each step, contents of state element $S_{4,2}$ goes through a left rotation of 32 bits. Therefore introducing 32 bits of permanent fault in the state element $S_{4,2}^{t=15}$ will result in zero in bits $32, \cdots, 95$ of state element $S_{0,2}^{t=16}$. The left rotation will fetch the zero valued contents of state bits $64, \cdots, 95$ (faulty bits) in to the register bits $32, \cdots, 63$ of state element $S_{0,2}$. The left rotation will not have any effect on the register bits $64, \cdots, 95$ due to the permanent fault. As a result bits $32, \cdots, 95$ of state element $S_{0,2}^{t=16}$ will be set to zero, which will eliminate the effect of state $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ in the keystream computation.

In summary, an adversary using this technique must introduce a total of 128

bits of permanent faults in the 640-bit internal state of MORUS-640 to recover the first 64 bits of the key. One can recover the remaining 64 key bits by exhaustive search, which will require a complexity of about 2^{64}. Note that it is not possible to recover the full key of MORUS-640 using only permanent faults, as the faults applied to $S_{3,1}^{t=15}$ will result in permanent zero values in 64 bits of $S_{0,1}^{t=16}$, even after the key K has been XOR-ed with this state word. In particular, the attack described above is unable to recover bits 64 to 127 of the key, as these bits of $S_{0,1}^{t=16}$ have been permanently set to zero.

5.5.1.1 Attack Algorithm

The steps involved in recovering the secret key of MORUS-640 with the permanent fault are outlined in Algorithm 5.1.

Algorithm 5.1 Algorithm for Permanent Fault Based Key Recovery Attack on MORUS-640

1: Load key and initialization vector and continue the initialization phase.
2: Insert permanent set to zero faults in bit 32 to 63 of state element $S_{2,0}^{t=15}$.
3: Insert permanent set to zero faults in bit 64 to 127 of state element $S_{3,1}^{t=15}$.
4: Insert permanent set to zero faults in bit 64 to 95 of state element $S_{4,2}^{t=15}$.
5: Complete the initialization phase and construct the first block of the keystream.
6: Output bit 32 to 95 of the keystream as the first 64 bits of the secret key.

5.5.1.2 Reducing the Number of Faults

The combining function of MORUS-640 as described in Equation 5.8 has three XOR operations and an AND operation. During the processing of the first input plaintext block, the AND operation in Equation 5.8 takes input from the state elements $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$. The output of the AND operation is heavily biased towards zero, so one can remove the output of the AND operation from Equation 5.8 with a high probability. Equation 5.8 can be rewritten as

$$C^{16} = P^{16} \oplus S_{0,0}^{16} \oplus \overleftarrow{S_{0,1}^{16}}^{96} \oplus K \tag{5.9}$$

In this case, to recover the first 64 bits of the key one needs to induce 96 bits of permanent fault at the last step of the initialization phase. The output of

the first 64 bits of the AND operation can be cancelled with a probability of $(3/4)^{64} \approx 2^{-26.56}$.

5.5.2 Permanent Fault Attack on MORUS-1280

We observe that at the 16th step (last update step) of the initialization phase, similar to MORUS-640, MORUS-1280 also XORs the contents of state element $S_{0,1}^{t=16}$ with the key bits. For the 128-bit key K_{128}, state element $S_{0,1}^{t=16}$ is updated by XOR-ing its contents with $K_I = K_{128}||K_{128}$. For the 256-bit key state element $S_{0,1}^{t=16}$ is updated by XOR-ing its contents with $K_I = K_{256}$.

Also note that the contents of state element $S_{0,1}$ are used in the combining function at each time instant t (see Equation 5.5). Similar to the technique used for MORUS-640, if the associated data processing phase is skipped, i.e., there is no input associated data, then the key bits will be directly XOR-ed in the output keystream function while processing the first plaintext block. When there is no associated data processing phase, Equation 5.5 for the first plaintext block can be represented as:

$$C^{16} = P^{16} \oplus S_{0,0}^{16} \oplus \overleftarrow{S_{0,1}^{16} \oplus K_I}^{192} \oplus (S_{0,2}^{16} \otimes S_{0,3}^{16}) \tag{5.10}$$

In Equation 5.10, one needs to eliminate the effect of $S_{0,0}^{t=16}$, $S_{0,1}^{t=16}$, $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$, to recover the key K_I under a known plaintext model. We discuss a similar technique that was used for MORUS-640, to remove the effect of these state elements by introducing faults in the internal states of MORUS-1280. To remove the effect of the above mentioned state elements an adversary may introduce permanent faults to set the contents of some specific register to zero, i.e., destroying the register. The locations that we target for our fault attack are shown in Figure 5.3 and discussed in detail below.

During the 16th step (last update step) of the initialization phase, destroying the last 128 registers (i.e., register 128 to 255) of state element $S_{3,1}^{t=15}$ will result in all zero bits in the state element $S_{4,1}^{t=15}$. This is due to the fact that at the fourth round of each step, the contents of $S_{3,1}$ are rotated left by 128 bits. The left rotation will fetch the zero valued contents of the last 128 bits(faulty bits valued zero) in to the first 128 register bits of state element $S_{4,1}$. Since the last 128 registers are permanently set to zero, the rotation will not have any effect on these bits. After the left rotation there are no changes in the state element

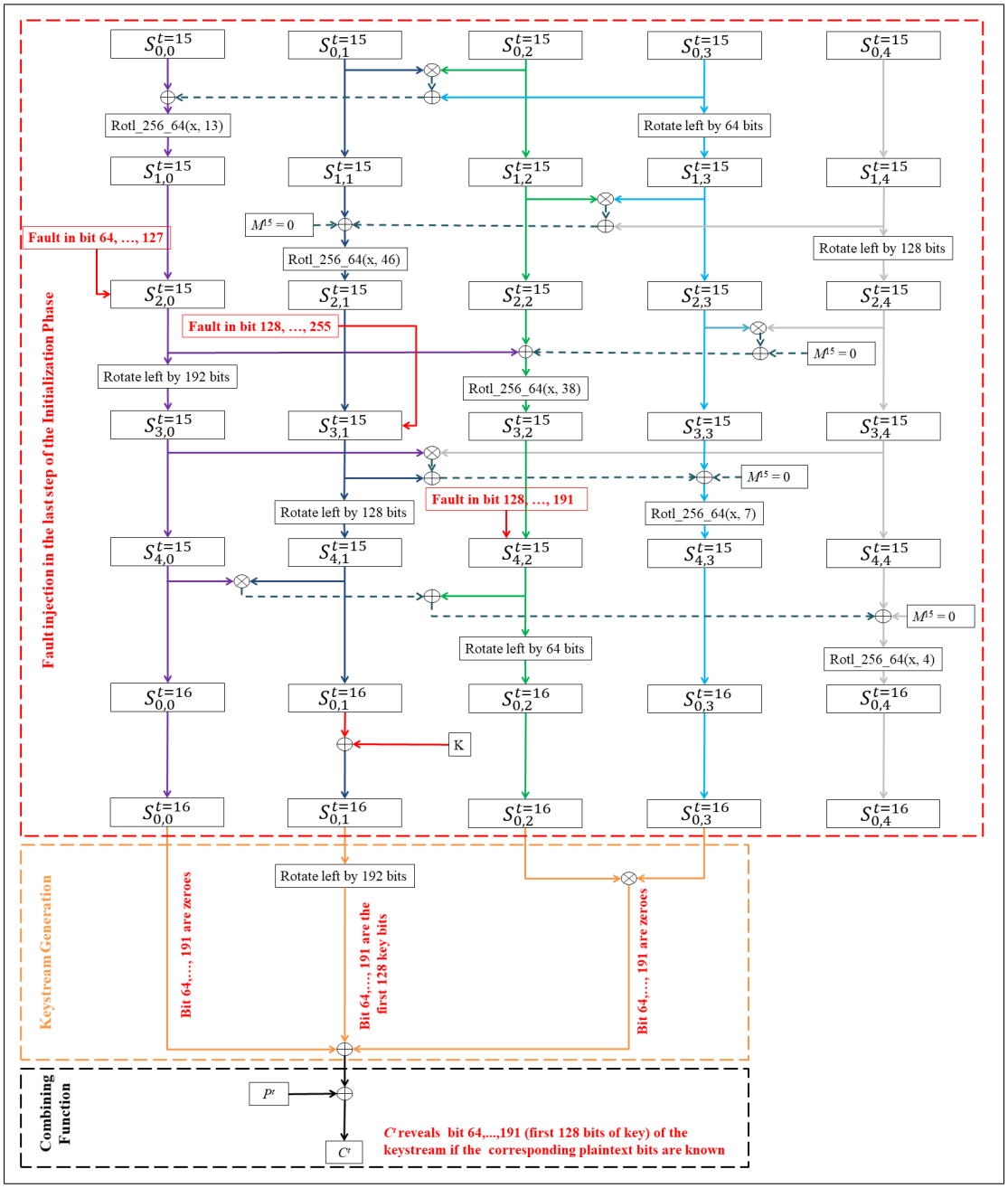

Figure 5.3: Inducing Permanent Fault at the Last Step of the Initialization Phase of MORUS-1280

$S_{4,1}^{t=15}$ for that step, i.e., $S_{4,1}^{t=15} = S_{0,1}^{t=16}$.

Since K_I is XOR-ed with the contents of this state element after these 16 steps, the first 128 bits of state element $S_{0,1}^{t=16}$ will contain the first 128 bits of K_I. Due to the permanent fault, XOR-ing of the last 128 bits of the K_I will have no effect on these bits.

Note that in the encryption phase, the combining function rotates the contents of this state element by 192 bits. So the first 128 bits of K_I will be involved in bits $64, \cdots, 191$ of the output of combining function. Thus one can recover the first 128 bits of the K_I by eliminating the effect of the bits $64, \cdots, 191$ of state element $S_{0,0}^{t=16}$, $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ from the first ciphertext block. To remove the effect of these bits one can follow a similar technique which was used for state element $S_{0,1}^{t=16}$.

To eliminate the effect of bits $64, \cdots, 191$ of state element $S_{0,0}^{t=16}$ in the keystream computation, one can permanently set bits $64, \cdots, 127$ of state element $S_{2,0}^{t=15}$ to zero (i.e., destroy the register bits 64 to 127). At the third round of each step, contents of state element $S_{2,0}$ goes through a left rotation of 192 bits. Therefore introducing 64 bits of permanent fault in these register of state element $S_{2,0}^{t=15}$ will result in zero bits in bits $64, \cdots, 191$ of state element $S_{3,0}^{t=15}$. The left rotation will fetch the zero valued contents of bits $64, \cdots, 127$ (faulty bits) in to the register bits $128, \cdots, 191$ of state element $S_{3,0}$. The left rotation will not have any effect on the register bits $64, \cdots, 127$ due to the permanent fault. As a result bits $64, \cdots, 191$ of state element $S_{0,0}^{t=16}$ will be set to zero, since there are no changes in state element $S_{3,0}$ after the third round of each step.

Also, in Equation 5.10, $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ are multiplied bitwise and used in the combining function. Therefore setting bits $64, \cdots, 191$ of either of these two state elements to zero will eliminate the effect of both in the keystream computation. To introduce zero valued contents in bits $64, \cdots, 191$ of state element $S_{0,2}^{t=16}$, one can permanently set bit $128, \cdots, 191$ of state element $S_{4,2}^{t=15}$ to zero, i.e., destroy the corresponding registers. At the fifth round of each step, contents of state element $S_{4,2}$ goes through a left rotation of 64 bits. Therefore introducing 64 bits of permanent fault in the state element $S_{4,2}^{t=15}$ will result in zero bits in bits $64, \cdots, 191$ of state element $S_{0,2}^{t=16}$. The left rotation will fetch the zero valued contents of state bits $128, \cdots, 191$ (faulty bits) in to the register bits $64, \cdots, 127$ of state element $S_{0,2}$. The left rotation will not have any effect on the register bits $128, \cdots, 191$ due to the permanent fault. As a result bits $64, \cdots, 191$ of state element $S_{0,2}^{t=16}$ will be set to zero, which will eliminate the effect of state $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$ in the combining function.

By introducing such faults, an adversary can easily recover the first 128 bits of the secret key under a known-plaintext scenario. Recall that this is the entire key when the key size is 128-bit, while it represents the first 128 bits of the key

when the key size is 256-bit.

An adversary using this technique must introduce a total 256 bits of permanent fault in the 1280 bits internal state of MORUS-1280. An adversary recovers the whole key when the key size is 128-bit, whereas an adversary recovers the first 128 bits of the key when the key size is 256-bit. An adversary can recover the remaining bits of the key for MORUS-1280-256 by exhaustive search, which will require a complexity of about 2^{128}.

Note that similar to MORUS-640 it is not possible to recover the full key of MORUS-1280-256 using only permanent faults, as the faults applied to $S_{3,1}^{t=15}$ will result in permanent zero values in 128 bits of $S_{0,1}^{t=16}$, even after the key K has been XOR-ed with this state word. In particular, the attack described above is unable to recover bits 128 to 255 of the key, as these bits of $S_{0,1}^{t=16}$ have been permanently set to zero.

5.5.2.1 Attack Algorithm

The steps involved in recovering the secret key of MORUS-1280 with the permanent fault are outlined in Algorithm 5.2.

Algorithm 5.2 Algorithm for Permanent Fault Based Key Recovery Attack on MORUS-1280

1: Load key and initialization vector and continue the initialization phase.
2: Insert permanent set to zero faults in bit 64 to 127 of state element $S_{2,0}^{t=15}$.
3: Insert permanent set to zero faults in bit 128 to 255 of state element $S_{3,1}^{t=15}$.
4: Insert permanent set to zero faults in bit 128 to 191 of state element $S_{4,2}^{t=15}$.
5: Complete the initialization phase and construct the first block of the keystream.
6: Output bit 64 to 191 of the keystream as the first 128 bits of the secret key.

5.5.2.2 Reducing the Number of Faults

The combining function of MORUS-1280 as described in Equation 5.10 also has three XOR operations and an AND operation. During the processing of the first input plaintext block, the AND operation in Equation 5.10 take input from the state element $S_{0,2}^{t=16}$ and $S_{0,3}^{t=16}$. The output of the AND operation is heavily biased towards zero, so one can remove the output of the AND operation from

Equation 5.10 with a high probability. Equation 5.10 can be rewritten as

$$C^{16} = P^{16} \oplus S^{16}_{0,0} \oplus \overleftarrow{S^{16}_{0,1} \oplus K_I}^{192} \tag{5.11}$$

In this case, to recover the first 128 bits of the key one needs to induce 192 bits of permanent fault at the last step of the initialization phase. The output of the corresponding 128 bits of the AND operation can be cancelled with a probability of $(3/4)^{128} \approx 2^{-53.1}$.

5.5.3 Transient Fault Attacks on MORUS

The observation from the above sections can also be extended by taking an approach that induces transient faults in the MORUS states to recover the secret key. In the last step of the initialization phase for MORUS, the key bits are directly XOR-ed to update the contents of state element $S^{t=16}_{0,1}$. Also, the contents of state element $S^{t=16}_{0,1}$ are used in the keystream generation function. After the initialization phase, MORUS performs the associated data processing phase which does not output any keystream bits. Following this, the encryption phase is performed where keystream block is generated at each step. Therefore, the key bits will be XOR-ed in the first block of the keystream generation function if there is no associated data processing phase. We consider this scenario when there is no associated data.

The keystream generation function also takes input from the contents of $S^{t=16}_{0,0}$, $S^{t=16}_{0,2}$ and $S^{t=16}_{0,3}$. Thus adversary needs to eliminate the effect of these bits in the keystream generation function. An adversary can introduce transient set to zero faults at specific times and locations to eliminate the effect of the contents from these state elements. Following this adversary can observe the corresponding bit in the first keystream block to recover one bit of the secret key .

The contents of state elements $S^{t=16}_{0,2}$ and $S^{t=16}_{0,3}$ are multiplied bitwise and used in the keystream generation function. Therefore, setting any of the bits of these two state elements to zero (with the transient set to zero fault) will set the output of the bitwise multiplication for the corresponding bits to zero.

In particular, the adversary needs to introduce three bits of faults to recover a single bit of the secret key. To recover the jth bit of the secret key an adversary can introduce a one bit transient fault on the jth register of state element $S^{t=15}_{4,1}$ and $((l_s - w_2 + j) \mod l_s)$th register of state elements $S^{t=15}_{4,0}$ and $S^{t=15}_{4,3}$, where

l_s is the block size of 128-bit or 256-bit for MORUS-640 and MORUS-1280, respectively. The transient fault introduced here is a set to zero fault which temporarily resets the corresponding bits to zero. With the introduction of these three bit faults in state elements $S_{4,0}^{t=15}, S_{4,1}^{t=15}, S_{4,3}^{t=15}$, an adversary can recover the jth bit of the secret key. At each time step, an adversary needs only three faults in specific locations, but depending on the key size this process must be repeated 128 or 256 times in order to reveal the whole key.

Consider the scenario where an adversary can induce such faults to different locations of the initial states, which are computed from the same key but with different initialization vectors. For MORUS-640, an adversary can select 128 different initial states computed from the same key but different initialization vectors, and for each initial state induce the faults in 3 specific location of state elements $S_{4,0}^{t=15}, S_{4,1}^{t=15}, S_{4,3}^{t=15}$. With this the adversary can recover the whole secret key of MORUS-640-128. This will require $128 \times 3 = 384$ bits of transient faults.

For MORUS-1280-128, an adversary can select 128 different initial states computed from the same key but different initialization vectors, and for each initial states induce the transient set to zero faults in 3 specific location of state elements $S_{4,0}^{t=15}, S_{4,1}^{t=15}, S_{4,3}^{t=15}$. With this the adversary can recover all the secret key of MORUS-1280-128. This will also require $128 \times 3 = 384$ bits of transient faults. Similarly, for MORUS-1280-256, this attack will require $256 \times 3 = 768$ bits of transient faults to recover the 256-bit secret key of MORUS-1280-256.

5.5.3.1 Attack Algorithm

The steps for the transient fault based key recovery attack are outlined in Algorithm 5.3.

Algorithm 5.3 Algorithm for Transient Fault Based Key Recovery Attack

1: Repeat steps 2 to 5 for $j = 0, \ldots, l_k - 1$.
2: Load the key and a fresh initialization vector, and continue the initialization phase.
3: Insert transient set to zero fault in register j of state element $S_{4,1}^{t=15}$.
4: Insert transient set to zero faults in register $((l_s - w_2 + j) \mod l_s)$ of state elements $S_{4,0}^{t=15}$ and $S_{4,3}^{t=15}$.
5: After the initialization, construct the first block of the keystream.
6: Output keystream bit $((l_s - w_2 + j) \mod l_s)$ as the jth bit of the secret key.

5.5.4 Summary of Fault Attacks on MORUS

We described two fault attacks on different variants of MORUS. The first type of fault attack uses permanent set to zero fault injection into the internal state of MORUS. The second fault attack described in this chapter uses temporary set to zero fault injection into the internal state of MORUS. The results from these fault attacks are tabulated in Table 5.11.

Table 5.11: Summary of Fault Attacks on MORUS

	Key Size	Type of Fault	Number of Faults	Recovered Key Bits
MORUS-640	128	Permanent	128	64
MORUS-640	128	Transient	384	128
MORUS-1280-128	128	Permanent	256	128
MORUS-1280-128	128	Transient	384	128
MORUS-1280-256	256	Permanent	256	128
MORUS-1280-256	256	Transient	768	256

The fault model for these attacks requires fault injections at a specific time and location. This can be achieved using optical fault injection as adopted in several other research papers [103, 127]. The trade-off between the permanent fault and the transient fault is in the number of faults and the number of key bits recovered. With permanent faults, the adversary is able to recover at least half of the secret key, but it requires to introduce less number of faults. For the transient fault, an adversary can recover the entire key; however, this requires to up to three times as many faults compared to the permanent fault attacks.

Generally speaking, permanent fault may be easier to apply since it can be obtained just by destroying the memory cells or cutting wires. The adversary can simply do this given that they have access to the physical implementation of the algorithm. For the transient faults, adversary needs access to the physical implementation of the algorithm as well as additional equipments to apply the attack.

The cost associated with the type of fault varies depending on the fault injection technique [103]. The permanent fault injection can be achieved by simply cutting a wire [150] or by using sophisticated technology such as using laser beam as adopted in [127]. Injecting the faults with laser beam will require a higher cost but possibly provides more accuracy [103]. For the transient fault, adversary can use optical fault injections. This model can work with low budget

equipments such as simple flashgun or laser pointer to set or reset bit(s) of SRAM in a microcontroller [151].

The adversary needs to have access to the physical implementation of the algorithm to apply these fault attacks. Also note that both of these transient and permanent fault attacks described in this chapter are based on the assumption that there is no associated data. These attacks are not feasible otherwise.

The fault model described here requires an adversary to use faults which set specific bits of the registers to zero, i.e., set to zero fault. These results are theoretical, and we did not perform any experiments since the results are straightforward given that the faults can be applied in the implementation of the algorithm. However, we note that the fault models for our proposed key recovery attacks have been shown to work in hardware [151], and also adopted in several other research papers [127, 152].

5.6 Rotational Cryptanalysis of MORUS

Rotational cryptanalysis investigates the propagation of rotational pairs in the outputs of a cryptographic scheme for any given rotational input pairs. This was applied [94] to ciphers composed of three operations: addition, rotation and XOR (ARX). MORUS is an ARX like cipher, except that it uses bitwise multiplication instead of the addition operation. Therefore we considered investigating the application of rotational cryptanalysis on MORUS, since it has similar ties to the ARX ciphers.

This section discusses the applicability of rotational cryptanalysis to MORUS. We investigated the basic rotational properties of different operations used in the MORUS state update function. XOR-ing of constants also plays an important role in the analysis of rotational pairs. Our investigation also includes the analysis of rotation invariant bits in the constants used for MORUS. We then investigated the rotational properties of MORUS state contents using the above mentioned operations. The goal of these investigations is to construct a distinguisher for MORUS, if the rotational properties are preserved in the input and the output.

5.6.1 Rotational Properties of Operations used in MORUS

The operations in the state update function of MORUS include AND, XOR and rotation operation. It also uses the $Rotl\_xxx\_yy(x, b)$ operation which divides

a xxx-bit block input x into 4 yy-bit words and rotates each word to the left by b bits.

Rotational properties of bitwise XOR and rotation operation were investigated by Khovratovich and Nikolić [94]. In the following, we review these properties. We also analyse the properties of the $Rotl\_xxx\_yy(x, b)$ operation, which is not explored in any previous literature.

5.6.1.1 Rotational Properties of XOR and Rotation Operation

Khovratovich and Nikolić stated that XOR operation preserves rotational pairs [94]. That is for an r-bit rotation, Equation 5.12 is true. Their work stated that Equation 5.12 is true, but did not include the proof of this statement. For completeness, we provide a proof for this.

Theorem 5.1. *Bitwise XOR operation applied to a binary string X preserves the rotational pairs for any arbitrary rotation distance r. That is*

$$\overleftarrow{X}^r \oplus \overleftarrow{Y}^r = \overleftarrow{X \oplus Y}^r \tag{5.12}$$

Proof. Suppose $X = [X_L|_i X_R]$ denotes the notation for n-bit string $X = x_{n-1} \cdots x_0$, where $X_L = x_{n-1} \cdots x_i$ and $X_R = x_{i-1} \cdots x_0$. Similarly we define the string $Y = [Y_L|_i Y_R]$ for n-bit string $Y = y_{n-1} \cdots y_0$, where $Y_L = y_{n-1} \cdots y_i$ and $Y_R = y_{i-1} \cdots y_0$. Then for a r-bit arbitrary left rotation with $0 \leq r \leq n$, we can represent X and Y as $X = [X_L|_{n-r} X_R]$ and $Y = [Y_L|_{n-r} Y_R]$, respectively. We can write

$$\begin{aligned}
\overleftarrow{X}^r \oplus \overleftarrow{Y}^r &= \overleftarrow{[X_L|_{n-r} X_R]}^r \oplus \overleftarrow{[Y_L|_{n-r} Y_R]}^r \\
&= [X_R|_r X_L] \oplus [Y_R|_r Y_L] \\
&= [(X_R \oplus Y_R)|_r X_L \oplus Y_L]
\end{aligned}$$

and

$$\begin{aligned}
\overleftarrow{X \oplus Y}^r &= \overleftarrow{[X_L|_{n-r} X_R] \oplus [Y_L|_{n-r} Y_R]}^r \\
&= \overleftarrow{[(X_L \oplus Y_L)|_{n-r} (X_R \oplus Y_R)]}^r \\
&= [(X_R \oplus Y_R)|_r X_L \oplus Y_L]
\end{aligned}$$

Therefore, $\overleftarrow{X}^r \oplus \overleftarrow{Y}^r = \overleftarrow{X \oplus Y}^r$. □

Their analysis show that the bitwise rotation operation also preserves the rotational pairs. Therefore, XOR and rotation operation in the state update function of MORUS does not break the symmetry in the rotational pairs.

5.6.1.2 Rotational Properties of AND Operation

Similar to the XOR operation, it is easy to prove that the bitwise AND operation always preserve the rotational pairs. That is, Equation 5.13 is true for any rotation distance r.

Theorem 5.2. *Bitwise AND operation applied to a binary string X preserves the rotational pairs for any arbitrary rotation distance r. That is*

$$\overleftarrow{X}^r \otimes \overleftarrow{Y}^r = \overleftarrow{X \otimes Y}^r \tag{5.13}$$

Proof. Suppose $X = [X_L|_i X_R]$ denotes the notation for n-bit string $X = x_{n-1} \cdots x_0$, where $X_L = x_{n-1} \cdots x_i$ and $X_R = x_{i-1} \cdots x_0$. Similarly we define the string $Y = [Y_L|_i Y_R]$ for n-bit string $Y = y_{n-1} \cdots y_0$, where $Y_L = y_{n-1} \cdots y_i$ and $Y_R = y_{i-1} \cdots y_0$. Then for a r-bit arbitrary left rotation with $0 \leq r \leq n$, we can represent X and Y as $X = [X_L|_{n-r} X_R]$ and $Y = [Y_L|_{n-r} Y_R]$, respectively. We can write

$$\begin{aligned}
\overleftarrow{X}^r \otimes \overleftarrow{Y}^r &= \overleftarrow{[X_L|_{n-r} X_R]}^r \otimes \overleftarrow{[Y_L|_{n-r} Y_R]}^r \\
&= [X_R|_r X_L] \otimes [Y_R|_r Y_L] \\
&= [X_R \otimes Y_R|_r X_L \otimes Y_L]
\end{aligned}$$

and

$$\begin{aligned}
\overleftarrow{X \otimes Y}^r &= \overleftarrow{[X_L|_{n-r} X_R] \otimes [Y_L|_{n-r} Y_R]}^r \\
&= \overleftarrow{[(X_L \otimes Y_L)|_{n-r} (X_R \otimes Y_R)]}^r \\
&= [(X_R \otimes Y_R)|_r X_L \otimes Y_L]
\end{aligned}$$

Therefore, $\overleftarrow{X}^r \otimes \overleftarrow{Y}^r = \overleftarrow{X \otimes Y}^r$. □

Therefore, bitwise AND operation in the state update function of MORUS does not break the symmetry in the rotational pairs.

5.6.1.3 Rotational Properties of $Rotl\_xxx\_yy(x, b)$ Operation

MORUS uses the $Rotl\_xxx\_yy(x, b)$ operation which is a composition of two operations. This includes dividing the input word into four sub-words of equal length and then applying bitwise left rotation operation to these sub-words of MORUS. This operation can be considered as a bit-wise permutation. In general bit-wise permutation does not preserve the rotational pairs. This is illustrated below with a simple example.

Let $X = x_0x_1x_2x_3x_4x_5x_6x_7$ be a sequence of 8 bits. The application of $Rotl\_8\_2y(x, 1)$ operation on this 8-bit sequence X results in:

$$X_{rotl_1} = Rotl\_8\_2(X, 1) = x_1x_0x_3x_2x_5x_4x_7x_6$$

Following this, the application of a one-bit left rotation applied to X_{rotl_1} results in:

$$\overleftarrow{X_{rotl_1}}^{1} = x_0x_3x_2x_5x_4x_7x_6x_1$$

Now, consider the alternative where first the one bit left rotation is applied to the 8 bit sequence X, followed by the application of $Rotl\_8\_2(x, 1)$ operation on the rotated sequence. The one bit left rotation applied to the sequence X results in

$$\overleftarrow{X}^{1} = x_1x_2x_3x_4x_5x_6x_7x_0$$

Following that, application of $Rotl\_xxx\_yy(x, 1)$ operation on the rotated sequence $\overleftarrow{X}^{1}$ results in:

$$\overleftarrow{X}^{1}_{rotl_1} = x_2x_1x_4x_3x_6x_5x_0x_7$$

Clearly $\overleftarrow{X_{rotl}}^{1} \neq \overleftarrow{X}^{1}_{rotl}$ in general, and therefore a rotational pair is not guaranteed to be preserved.

Conditions for Preserving a Rotational Pair under Rotl_xxx_yy(x, b)
We observe that the operation $Rotl\_xxx\_yy(x, b)$ preserves the rotational pair, if the distance r of the left rotation applied to X is equal to a multiple of the sub-word length yy. For the above example the sub-word length is two. Applying two bit left rotation to the sequence X results in:

$$\overleftarrow{X}^{2} = x_2x_3x_4x_5x_6x_7x_0x_1$$

Application of $Rotl\_xxx\_yy(x,1)$ operation on the rotated sequence $\overleftarrow{X}^2$ results in:

$$\overleftarrow{X}^2_{rotl_1} = x_3x_2x_5x_4x_7x_6x_1x_0$$

Alternatively, application of two bit left rotation applied to X_{rotl_1} results in:

$$\overleftarrow{\overleftarrow{X_{rotl_1}}}^2 = x_3x_2x_5x_4x_7x_6x_1x_0$$

As shown in the example, $\overleftarrow{X}^2_{rotl} = \overleftarrow{\overleftarrow{X_{rotl}}}^2$ when the distance of the rotation applied is equal to the sub-word size.

Experimental verification also confirms that the rotational pairs are preserved when the distance of the left rotation applied to X is equal to a multiple of the sub-word length yy. Also, the rotational pair $(X, \overleftarrow{X}^r)$ will be preserved if the sequence X is rotation invariant for any arbitrary rotation distance.

We define the following theorem which identifies the condition for preserving rotation pairs when the $Rotl\_xxx\_yy(x,b)$ operation is applied.

Theorem 5.3. *The $Rotl\_xxx\_yy(x,b)$ operation applied to a binary string X preserves the rotational pairs if the distance of the rotation r applied is equal to a multiple of the sub-word size or if the input X is rotation invariant. That is*

$$\overleftarrow{X}^r_{rotl_b} = \overleftarrow{\overleftarrow{X_{rotl_b}}}^r \tag{5.14}$$

if r is a multiple of the sub-word size.

Proof. Suppose $X = X_1|X_2|X_3|X_4$ denotes a n-bit string $X = x_{n-1}\cdots x_0$, where $X_1 = x_{n-1}\cdots x_{n-\frac{n}{4}}$, $X_2 = x_{n-\frac{n}{4}-1}\cdots x_{n-\frac{2n}{4}}$, $X_3 = x_{n-\frac{2n}{4}-1}\cdots x_{n-\frac{3n}{4}}$ and $X_4 = x_{n-\frac{3n}{4}-1}\cdots x_0$. Let $Y = [Y_L|_iY_R]$ denote the notation for n-bit string $Y = y_{n-1}\cdots y_0$, where $Y_L = y_{n-1}\cdots y_i$ and $Y_R = y_{i-1}\cdots y_0$. Let b be the rotation distance applied in the $Rotl\_xxx\_yy(x,b)$ operation. Then $X_{rotl_b} = (X_1|X_2|X_3|X_4)_{rotl_b}$ can be represented as $X_{rotl_b} = (\overleftarrow{X_1}^b|\overleftarrow{X_2}^b|\overleftarrow{X_3}^b|\overleftarrow{X_4}^b)$.

The length of the sub-word is $n/4$ for a n bit string. So we require to show that the rotation distance $r = n/4,\ 2n/4,\ 3n/4, n$ on the $Rotl\_xxx\_yy(x,b)$ operation will preserve the rotational pairs.

For a r-bit left rotation with $r = n/4$, we can represent the left hand side of

Equation 5.14 as:

$$\begin{aligned}
\overleftarrow{X}^{r=n/4}_{rotl} &= \overleftarrow{(X_1|X_2|X_3|X_4)_{rotl_b}}^{r=n/4} \\
&= (X_2|X_3|X_4|X_1)_{rotl_b} \\
&= (\overleftarrow{X_2}^b|\overleftarrow{X_3}^b|\overleftarrow{X_4}^b|\overleftarrow{X_1}^b) \\
&= (\overleftarrow{[X_{2L}|_{n/4-b}X_{2R}]}^b|\overleftarrow{[X_{3L}|_{n/4-b}X_{3R}]}^b|\overleftarrow{[X_{4L}|_{n/4-b}X_{4R}]}^b|\overleftarrow{[X_{1L}|_{n/4-b}X_{1R}]}^b) \\
&= ([X_{2R}|_bX_{2L}]||[X_{3R}|_bX_{3L}]||[X_{4R}|_bX_{4L}]||[X_{1R}|_bX_{1L}])
\end{aligned}$$

For a r-bit left rotation with $r = n/4$, we can represent the right hand side of Equation 5.14 as:

$$\begin{aligned}
\overleftarrow{X_{rotl_b}}^{r=n/4} &= \overleftarrow{(X_1|X_2|X_3|X_4)_{rotl_b}}^{r=n/4} \\
&= (\overleftarrow{X_1}^b|\overleftarrow{X_2}^b|\overleftarrow{X_3}^b|\overleftarrow{X_4}^b)^{\overleftarrow{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{r=n/4}} \\
&= (\overleftarrow{[X_{1L}|_{n/4-b}X_{1R}]}^b|\overleftarrow{[X_{2L}|_{n/4-b}X_{2R}]}^b|\overleftarrow{[X_{3L}|_{n/4-b}X_{3R}]}^b|\overleftarrow{[X_{4L}|_{n/4-b}X_{4R}]}^b) \\
&= \overleftarrow{([X_{1R}|_bX_{1L}]||[X_{2R}|_bX_{2L}]||[X_{3R}|_bX_{3L}]||[X_{4R}|_bX_{4L}])}^{r=n/4} \\
&= ([X_{2R}|_bX_{2L}]||[X_{3R}|_bX_{3L}]||[X_{4R}|_bX_{4L}]||[X_{1R}|_bX_{1L}])
\end{aligned}$$

Therefore

$$\overleftarrow{X}^r_{rotl_b} = \overleftarrow{X_{rotl_b}}^r$$

when $r = n/4$. Similarly, we can prove that Theorem 5.3 is true for a rotation distance $r = 2n/4, 3n/4, n$.

Additionally, if the input sequence X is rotation invariant for arbitrary rotation distance r, then application of $Rotl\_xxx\_yy(x, b)$ to the input X will not have any effect on it. Therefore, the operation $Rotl\_xxx\_yy(x, b)$ will preserve the rotational pairs if the input X is rotation invariant. □

In MORUS-640, the sub-word size is 32 bits. So to maintain the rotational pairs in the $Rotl\_xxx\_yy(x, b)$ operation of MORUS-640, the distance of the rotations applied should be a multiple of 32. Analogously, the sub-word size is 64 bits for MORUS-1280. Therefore, the distance of the rotations applied needs to be a multiple of 64 to preserve the rotational pair in the $Rotl\_xxx\_yy(x, b)$ operation of MORUS-1280.

5.6.2 Rotational Properties of the Constants in MORUS

MORUS uses two constants $const_0$ and $const_1$ at the beginning of the initialization phase, which are used to load some of the state elements. These constants are XOR-ed with the contents of specific state elements as a part of the state update process. Khovratovich and Nikolić show that addition/ XOR of a constant breaks the symmetry in rotational relations, if the constant is not rotation invariant [94]. Therefore, XOR-ing of constant plays an important role in the rotational cryptanalysis. In the following, we discuss the rotational properties of the constants used in MORUS.

5.6.2.1 Rotational Properties of MORUS-640 Constants

We have analysed the MORUS-640 constants when r-bit rotation is applied to it. We investigated the number of bits in the constant which are rotation invariant for the r-bit rotation. MORUS-640 applies the $Rotl\_128\_32(x, b)$ operation in its state update function. According to Theorem 5.3, the $Rotl\_128\_32(x, b)$ preserves the rotational pairs if the rotation distance is a multiple of 32, i.e., 32, 64 or 96. Therefore for the analysis of MORUS-640 constants, we set the rotation distance r as a multiple of 32.

With the above conditions, the constant also needs to be 32 bits, 64 bits, 96 bits or 128 bits rotation invariant depending on the distance of the rotations applied. If the constant is 32-bit rotation invariant (i.e., the constant can be divided into 4 small sub-word of 32 bits where each of the sub-word has same content), then there are 2^{32} rotation invariant constants. For a 64 bits rotation invariant constant, there are 2^{64} rotation invariant constants.

We investigated the rotation invariant bits in the constants for any distance of the rotation which are multiple of $r = 32$. For any multiples of $r = 32$ the constants used in MORUS-640 are not rotation invariant among all the bits; except for the trivial one $r = 128, 256, \cdots$ which is basically the same constant. In the following, we discuss the number of bits that remain rotation invariant for different rotation distances set based on above criteria.

In MORUS-640, $const_0$ and $const_1$ are used to load the state element $S_{0,3}$ and $S_{0,4}$. We examined the number of rotation invariant bits in these constants for any rotation distance which is a multiple of 32. The rotated constant $\overleftarrow{const0}^{r}$ has 74, 60 and 74 rotation invariant bits for 32, 64, and 96 bit rotation distance, respectively. Similarly, the rotated constant $\overleftarrow{const1}^{r}$ has 64, 62 and 64 rotation

invariant bits for 32, 64, and 96 bit rotation distance, respectively. This means XOR-ing of these constants with the contents of any state element inverts about half of the bits in the corresponding state element compared to the result of XOR-ing with the rotated version of the constant.

Also, in the first step of the initialization phase, $\overleftarrow{const0}^{32} \oplus const1$ is used to update $S_{1,1,}$. The rotated version of this constant $\overleftarrow{\overleftarrow{const0}^{32} \oplus const1}^{r}$ has 62, 74 and 62 rotation invariant bits for 32, 64, and 96 bit rotation distance, respectively.

At the first step of initialization phase, constant $(\overleftarrow{const0}^{32} \otimes \overleftarrow{const1}^{64}) \oplus Rotl\_128\_32(const0, 5)$ is used in the update of the state element $S_{2,2}$. The rotated version of this constant $\overleftarrow{(\overleftarrow{const0}^{32} \otimes \overleftarrow{const1}^{64}) \oplus Rotl\_128\_32(const0, 5)}^{r}$ has 72, 64 and 72 rotation invariant bits for rotation distance 32, 64, and 96 bits, respectively.

5.6.2.2 Rotational Properties of MORUS-1280 Constants

We investigated the constants used in MORUS-1280, when r-bit rotation is applied to it. The investigation explores the number of rotation invariant bits in the constant for rotation distance r. MORUS-1280 applies the $Rotl\_256\_64(x, b)$ operation in its state update function. According to Theorem 5.3, the operation $Rotl\_256\_64(x, b)$ preserves the rotational pairs if the rotation distance is a multiple of 64, i.e., 64, 96, 192 or 256. Therefore for the analysis of MORUS-1280 constants, we set the rotation distance r as a multiple of 64.

With the above conditions, the constant also needs to be 64 bits, 128 bits, 192 bits or 256 bits rotation invariant depending on the distance of rotations applied. For MORUS-1280, if the constant is 64-bit rotation invariant (i.e., the constant can be divided into 4 small sub-word of 64 bits where each of the sub-word has same content), then there are 2^{64} rotation invariant constants. For a 128 bits rotation invariant constant, there are 2^{128} rotation invariant constants which will preserve the rotational pairs for MORUS-1280.

We investigated the rotation invariant bits in the constants for any distances of rotation which are multiples of $r = 64$. For any multiples of $r = 64$ the constant $const0||const1$ used in MORUS-1280 is not rotation invariant among all the bits; except for the trivial one $r = 256, 512, \cdots$ which is basically the same constant. In the following, we discuss the number of bits that remain rotation invariant for different rotation distances set based on above criteria.

In MORUS-1280, $const0||const1$ is used to load the state element $S_{0,4}$. The

rotated version of this constant $\overleftarrow{const0||const1}^{r}$ has 138, 114 and 138 rotation invariant bits for rotation distance 64, 128 and 192 bits, respectively. This means that XOR-ing this constant $const0||const1$ with the contents of the state element $S_{0,4}$ will preserve the rotational pairs in 138 bits when the rotation distance in the input is set to 64 bits.

The constant used to initialize state element $S_{0,2}$ consists of all 1. This constant is rotation invariant for any arbitrary rotation distance. Similarly, the constant used to initialize state element $S_{0,3}$ consists of all 0. This constant is also rotation invariant for any arbitrary rotation distance.

5.6.3 Rotational Properties of MORUS State Contents

This section investigates the rotational properties in the MORUS state contents. The contents of MORUS state elements are updated using the state update function $Update(S^t, M^t)$. Rotational properties of the operations and constants used in the state update function $Update(S^t, M^t)$ are discussed in Section 5.6.1 and Section 5.6.2. In our investigation we used these properties to determine the propagation of rotational pairs in MORUS state contents.

We first investigated the probability of preserving the rotational pairs in the MORUS state elements after one step of the initialization phase. We then extended our analysis for more than one step of the initialization phase.

We started our experiment by defining the key K and initialization vector V in terms of variables We then generated equations in terms of K and V to represent the state contents of n-step MORUS. We then generated the equations in terms of rotated versions of the key $\overleftarrow{K}^{r}$ and initialization vector $\overleftarrow{V}^{r}$ to represent the state contents of n-step MORUS. Finally, we investigated the rotational pair in the state elements $(S, \overleftarrow{S}^{r})$ to determine whether they preserve the rotational pair. Preserving the rotational pair in the state elements means that the rotational pairs in the output keystream is also preserved.

If the rotational pair is preserved in the output pair, then adversary can observe the output pair and use this to distinguish the MORUS output from a randomly generated output. This can be considered as a distinguisher under the related key-IV model.

5.6.3.1 Rotational Properties of MORUS-640 State Contents

MORUS-640 uses the operations XOR, AND, bitwise left rotation and the operation $Rotl\_128\_32(x, b)$ in its state update function. As illustrated in Theorem 5.1 and Theorem 5.2, bitwise XOR and AND operation will preserve the rotational pair for any arbitrary number of rotations. However, for the $Rotl\_128\_32(x, b)$ operation as described in Theorem 5.3, the rotation distance in the input needs to be a multiple of 32, i.e., 32, 64 or 92 bits for MORUS-640, to preserve rotational pairs in the output. So in our analysis of MORUS-640 state contents, the distance of the rotation r is set to 32. We conducted our analysis with the rotation distance of 32 bits since it preserves the rotational pairs in maximum number of bits after performing the XOR operation with the constant $const_0$ (see Section 5.6.2.1).

Rotational Pairs in MORUS-640 with 1-step Initialization Phase For a 32 bit rotated input, rotational pairs are preserved in 74 bits of the state element $S_{0,0}$ with probability 1. Rotational pairs are preserved in the rest of the 54 bits of the state element $S_{0,0}$ with probability 0, i.e., the corresponding bits are inverted. This is because of the the XOR of $const0$ with the contents of state element $S_{0,0}$, at the beginning of the initialization phase. As shown in Section 5.6.2.1 for $r = 32$ bit rotation distance, $const0$ has 74 bits which are rotation invariant. Thus this introduces 54 inverted bits.

For state element $S_{0,1}$, rotational pairs are preserved in 62 bits with probability 1. Rotational pairs are preserved in the rest of the 66 bits of the state element $S_{0,1}$ with probability 0, i.e., the corresponding bits are inverted. Note that $\overleftarrow{const0}^{32} \oplus const1$ is XOR-ed with the contents of state element $S_{1,1}$. As shown in Section 5.6.2.1, constant $\overleftarrow{\overleftarrow{const0}^{32} \oplus const1}^{r=32}$ has 62 rotation invariant bits. Thus this introduces $128 - 62 = 66$ inversion in the resulting rotational pair.

Rotational pairs are preserved in 72 bits of the state element $S_{0,2}$ with probability 1. Rotational pairs are preserved in the remaining 56 bits of the state element $S_{0,2}$ with probability 0, i.e., the corresponding bits are inverted. We also observe that the constant $\overleftarrow{(\overleftarrow{const0}^{32} \otimes \overleftarrow{const1}^{64}) \oplus Rotl\_128\_32(const0, 5)}^{32}$ is XOR-ed with the contents of $S_{0,2}$. As illustrated in Section 5.6.2.1, constant $\overleftarrow{(\overleftarrow{const0}^{32} \otimes \overleftarrow{const1}^{64}) \oplus Rotl\_128\_32(const0, 5)}^{32}$ has 72 rotation invariant bits.

Table 5.12: Probability of Preserving Rotational Pairs after One Step of the Initialization of MORUS-640

State Element	$p = 1$	$p = 0.5$	$p = 0$
$S_{0,0}$	74	0	54
$S_{0,1}$	62	0	66
$S_{0,2}$	72	0	56
$S_{0,3}$	36	64	28
$S_{0,4}$	14	94	20
	258	158	224

Thus this introduces $128 - 72 = 56$ inversion in the resulting rotational pair.

Rotational pairs are preserved with probability 1, 0.5 and 0 in 36 bits, 64 bits and 28 bits of $S_{0,3}$, respectively. Finally, rotational pairs are preserved with probability 1, 0.5 and 0 in 14 bits, 94 bits and 20 bits of $S_{0,4}$, respectively.

Table 5.12 summarizes the probability of preserving rotational pairs in the state elements of 1-step MORUS-640. As shown in Table 5.12, after one step there are 482 known differences in rotational pairs based on 32-bit rotations of the state elements of MORUS. Therefore, observing these known differences in the specific output bit of a rotational pair, an adversary can distinguish the keystream of 1-step MORUS-640 from a randomly generated output.

Rotational pairs in MORUS-640 with Initialization Phase beyond One Step We extended our experiments to determine the probability of preserving rotational pairs in more than one step of the initialization phase of MORUS-640. For two steps of the initialization phase, we found there are only 21 known differences in rotational pairs of the state contents of MORUS-640. In particular, we found only 14 bits and 7 bits are preserved in state element S_0 and S_1, respectively, with a probability of 1 or 0. This is not sufficient to determine a distinguisher because there are no known differences in the rest of the state elements.

For more than two steps of the initialization phase, Sage fails to generate the equations. This is due to the fact that the equations get very complicated after two steps of the initialization phase and the software Sage runs out of memory to perform the necessary computations.

As indicated above after the two steps of the initialization phase there are not enough rotational pairs in the state elements to construct a distinguisher and

such situation becomes worse with more steps. Therefore, it is unlikely for an adversary to construct a distinguisher for MORUS-640 based on observing the rotational pairs.

5.6.3.2 Rotational Properties of MORUS-1280 State Contents

MORUS-1280 uses the operations XOR, AND, bitwise left rotation and the $Rotl\_256\_64(x, b)$ operation in its state update function. As illustrated in Theorem 5.1 and Theorem 5.2, bitwise XOR and AND operation will preserve the rotational pair for any arbitrary number of rotations. However, for the $Rotl\_256\_64(x, b)$ operation as described in Theorem 5.3, the rotation distance in the input needs to be a multiple of 64, i.e., 64, 128 or 192 bits for MORUS-1280, to preserve rotational pairs in the output. So in our analysis of MORUS-1280 state contents, the distance of the rotation r is set to 64. We conducted our analysis with the rotation distance of 64 bits since it preserves the rotational pairs in maximum number of bits after performing the XOR operation with the constant $const_0$ (see Section 5.6.2.2).

Rotational Pairs in MORUS-1280 with 1-step Initialization Phase Rotational pairs are preserved in all of the 256 bits of the state element $S_{0,0}$ with probability 1. This is because the constant XOR-ed with the contents of this state element is all zeroes and so rotation invariant for arbitrary rotation distance r.

Rotational pairs are preserved in 138 bits of the state element $S_{0,1}$ with probability 1. Rotational pairs are preserved in the remaining 118 bits of the state element $S_{0,1}$ with probability 0, i.e., the corresponding bits are inverted. Observe that $const0||const1$ is XOR-ed with the contents of the state element $S_{0,1}$. As described in Section 5.6.2.2, $\overleftarrow{const0||const1}^{64}$ has 138 rotation invariant bits. Thus XOR-ing of this constant introduces 118 bits of inversion in the rotational pair.

Rotational pairs are preserved in all of the 256 bits of the state element $S_{0,2}$ with probability 1. This is because the constant XOR-ed with the contents of this state element is all $const0 \otimes 0^{128}$, e.g., all zeroes, and so rotation invariant for arbitrary rotation distance r.

Rotational pairs are preserved with probability 1, 0.5 and 0 in 73 bits, 118 bits and 65 bits of $S_{0,3}$, respectively. Rotational pairs are preserved with probability 1, 0.5 and 0 in 78 bits, 118 bits and 60 bits of $S_{0,4}$, respectively.

Table 5.13 summarizes the probability of preserving rotational pairs in the state elements of 1-step MORUS-1280. We found that rotational pairs are preserved with probability 1 in 801 bits of the 1-step MORUS-1280. After one step of the initialization phase, there are only 236 bits which are unknown with probability 0.5. Therefore, observing these known differences in the specific output bit of a rotational pair, an adversary can distinguish the keystream of 1-step MORUS-1280 from a randomly generated output.

Table 5.13: Probability of Preserving Rotational Pairs after One Step of the Initialization of MORUS-1280

State Element	$p = 1$	$p = 0.5$	$p = 0$
$S_{0,0}$	256	0	0
$S_{0,1}$	138	0	118
$S_{0,2}$	256	0	0
$S_{0,3}$	73	118	65
$S_{0,4}$	78	118	60

Rotational pairs in MORUS-1280 with Initialization Phase beyond one step We extended our experiments to determine the probability of preserving rotational pairs in more than one step of the initialization phase of MORUS-640. For two steps of the initialization phase, we found only 30 bits and 17 bits are preserved in state element S_0 and S_1, respectively, with a probability of 1 or 0. This is not sufficient to determine a distinguisher because there are no known differences in the rest of the state elements.

For more than two steps of the initialization phase, Sage fails to generate the equations for MORUS-1280. This is due to the fact that the equations get very complicated after two steps of the initialization phase and the software Sage runs out of memory.

As indicated above after the two steps of the initialization phase there are not enough rotational pairs in the state elements to construct a distinguisher and such situation becomes worse with more number of steps. Therefore, it is unlikely for an adversary to construct a distinguisher for the full version of MORUS-1280 based on observing the rotational pairs.

5.6.4 Summary of the Rotational Cryptanalysis on MORUS

We investigated the feasibility of rotational cryptanalysis on different variants of MORUS. Our investigation shows that all the operations used in MORUS preserve the rotational pairs when the rotation distance is set to a multiple of 32 or 64 for MORUS-640 and MORUS-1280, respectively. We have also verified that an adversary can build a distinguisher for the full version of MORUS if rotation invariant constants are used in the state update function of MORUS. However, the constants used in MORUS are not rotation invariant which makes it infeasible to build the distinguisher for more than one step.

Due to the non-invariant constants used in the state update function of MORUS, we found that rotational cryptanalysis can distinguish the MORUS output for only one step of the initialization phase. For more than one step of the initialization phase, the probability of preserving rotational pair becomes 0.5 in most of the bits. This makes it infeasible to apply the distinguisher for more than one round.

Note that rotational attack on MORUS was also investigated by Dwivedi et al. [40]. Their application of rotational cryptanalysis tried to recover the key of MORUS. They have developed a key recovery attack based on the rotational cryptanalysis. Their approach requires slightly less complexity than the exhaustive search for the key recovery of MORUS-1280-256. On the other hand, we used the rotational cryptanalysis to build distinguisher for MORUS. As indicated above, our approach of rotational cryptanalysis can build the distinguisher for just one step of the initialization phase of MORUS. Both of these works shows that the rotational cryptanalysis is not feasible to mount an attack on MORUS.

We would also like to point out that the work by Dwivedi et al. [40] suggests that all the operations except for the XOR-ing of constant preserves the rotational pairs in MORUS for any arbitrary number of rotation distance. However, according to our analysis this is not always true. We note that according to Theorem 5.3, the $Rotl\_xxx\_yy(x, b)$ operation does not preserve rotational pairs in all the bits for arbitrary rotation distance, instead it preserves the rotational pairs in all the bits when the rotation distance is a multiple of the sub-word length yy. This does not affect their cryptanalysis since it uses known rotational characteristics in specific bits, which are computed in a pre-computation phase.

5.7 Forgery Attack on MORUS

This section provides our analysis and observation on the finalization phase of MORUS. We first discuss the necessary conditions and procedures to apply a forgery attack on MORUS by deleting specific message blocks. This can be considered as a bit deletion forgery attack. We also introduce a fault based forgery attack on MORUS. The fault based forgery attack introduced here is similar to a bit flip forgery attack.

5.7.1 Forgery Attack Using Block Deletion

From the finalization phase of MORUS, we observe that the same state update function and output function is used for both the keystream generation and the tag generation phases. In the initial version of MORUSv1 two different functions were used for the keystream generation and tag generation. This was changed in MORUSv2 to reduce the hardware cost.

Note that the main difference between the encryption and the finalization phase of MORUSv2 is the update of state element $S_{0,4}$ at the beginning of the finalization phase. Also a block including the total length of the associated data and the plaintext is used as the external input during the finalization phase. This is intended to prevent an attacker from using part of the keystream as the authentication tag by reducing the input plaintext/ciphertext size.

In the following we discuss the conditions when the keystream in the encryption phase reveals the tag for a plaintext P' which is generated by reducing the size of the original plaintext P, i.e., deleting blocks from P.

5.7.1.1 Conditions for the Forgery Attack Using Block Deletion

Let P and P' denote two input messages with length $msglen$ and $msglen'$, respectively where $msglen' = msglen - 10$. Let the plaintext message P consists of at least $l_p = 11$ blocks. The contents of the last 10 blocks of the plaintext $P_{l_p-10}, \cdots, P_{l_p-1}$ are set as $adlen||msglen'$. The rest of the plaintext blocks, i.e., $P_0, \cdots, P_{l_p-11}$ need to satisfy the criteria such that state element $S_{0,0}$ consists of all zero bits after processing these plaintext blocks.

Note that among the total 2^{640} states of MORUS-640, there are 2^{512} such states for which the content of $S_{0,0}$ is all zero bits. Similarly, for MORUS-1280

there are 2^{1024} states among 2^{1280} for which the content of $S_{0,0}$ is all zero bits. Thus, the probability of this forgery attack is 2^{-128} and 2^{-256}, respectively.

Also, observe that at each step of the encryption phase, plaintext blocks are XOR-ed into the state elements $S_{0,1}, S_{0,2}, S_{0,3}, S_{0,4}$; however, they are not directly injected into the state element $S_{0,0}$. Also, state element $S_{0,0}$ takes input from the state element $S_{0,1}, S_{0,2}, S_{0,3}$. So, an adversary can manipulate the contents of $S_{0,1}, S_{0,2}, S_{0,3}$ using the external input; however, without knowing the exact contents of the state element $S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3}$ adversary can not set $S_{0,0}$ to zero.

Attack Description An adversary can mount a forgery attack if the above conditions are satisfied. Adversary selects a new plaintext P' consisting of $l'_p = l_p - 10$ blocks. This new plaintext P' is selected by deleting the last 10 blocks from original plaintext P, i.e., plaintext P' consist of blocks $P'_0 = P_0, \cdots, P'_{l'_p-1} = P_{l_p-11}$.

To perform the attack, adversary first observes the keystream block Z_{l_p-1} for the original plaintext P. In the encryption phase the plaintext blocks $P_0, \cdots, P_{l_p-1}$ are encrypted by XOR-ing these block with the corresponding keystream block $Z_0, \cdots, Z_{l_p-1}$. Therefore adversary can find the keystream Z_{l_p-1} by XOR-ing the corresponding plaintext and ciphertext block. The adversary then sends the modified plaintext P' with the tag $\tau' = Z_{l_p-1}$.

In the decryption and tag verification of the modified plaintext P', the receiver will first recover the plaintext P' and fed it into the internal state. This process will set the contents of state element $S_{0,0}$ to zero after processing the modified plaintext P', if the above conditions are satisfied.

For the tag verification phase, the receiver then generates the tag τ'' at the receiver side. At the beginning of this tag generation, state element $S_{4,0}$ is XORed with the contents of state element $S_{0,0}$, i.e., $S_{4,0} = S_{4,0} \oplus S_{0,0}$. However, this update will not have any effect on $S_{4,0}$, given that the contents of $S_{0,0}$ were set to zero after processing the recovered plaintext P'.

Following this, the finalization process at the receiver side will update the state for 10 steps with the external input $adlen||msglen'$. Finally, the tag at the tag τ'' at the receiver side will be generated using the tag generation function. Recall that $adlen||msglen'$ was also set as the contents of the the last 10 blocks for plaintext P. Also note that for MORUSv2, the tag generation function is the same as the keystream generation function. Therefore, the tag τ'' generated at the receiver will be equal to the received tag τ' for the plaintext P'. The receiver

accepts the modified plaintext P' as a legitimate message since the received tag τ' and generated tag τ'' are equal. Figure 5.4 provide a detailed illustration of this forgery attack.

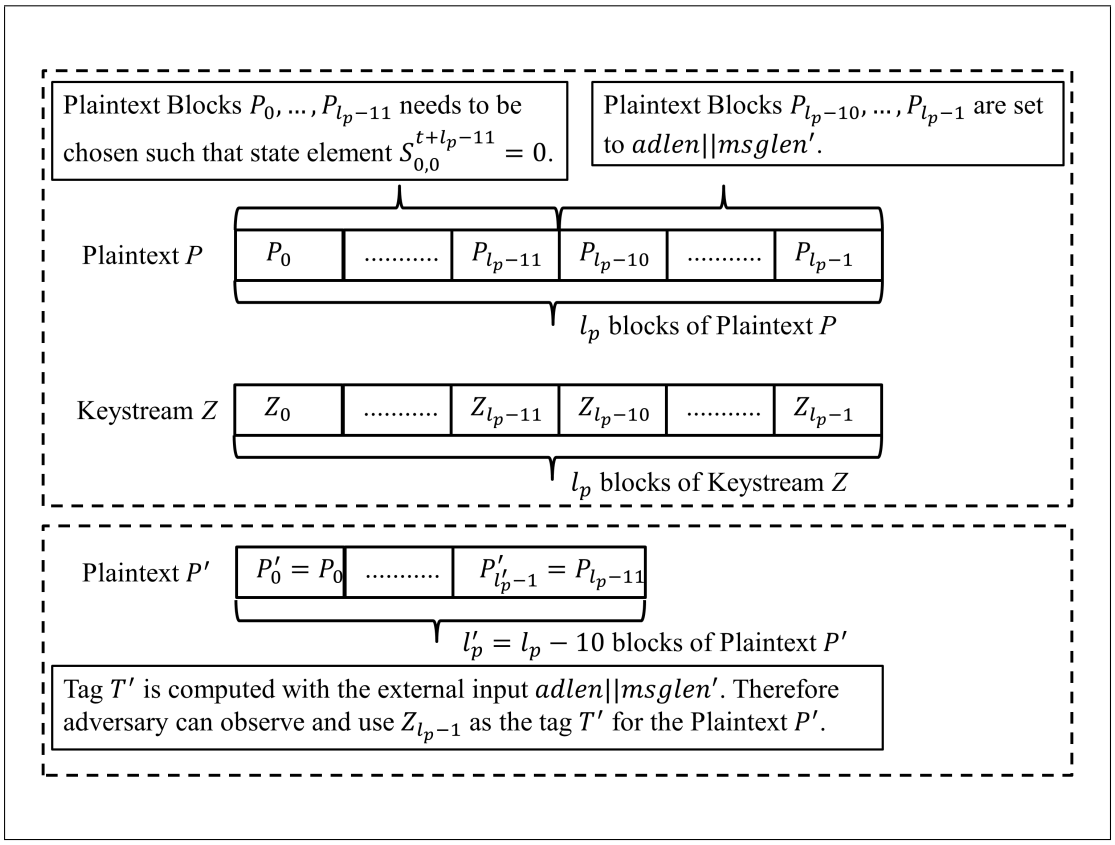

Figure 5.4: Forgery Attack on MORUS using Block Deletion

5.7.2 Fault Based Forgery Attack on MORUS

This section discusses a fault based forgery attack on MORUS. The goal of the attack is to modify the input message by flipping specific message bit(s) and to have this modified message accepted as legitimate at the receiver side.

Recall from Section 3.6 of Chapter 3 that a fault based forgery attack on the authenticated encryption cipher CLOC and SILC was introduced by Roy et al. [146]. Later, in the CAESAR Google discussion forum it was pointed out by Iwata et al. [147] that any authenticated encryption scheme can be forged using faulty encryption queries. This requires an adversary to inject faults in the inputs submitted to the encryption oracle and then use the output of the encryption oracle with the faulty inputs to continue with the forgery. We describe here a

similar fault injected forgery, however; the faults are injected into the internal state instead of the inputs.

The generic fault based forgery attack by Iwata et al. [147] applies specific faults in the message before the message is loaded in to the device. Their attack requires the attacker to enquire about the faulty ciphertext and the faulty tag for a faulty message. However, in our attack the faults are applied in the encryption device after the message is loaded. This attack may be more practical in some applications where the attacker is able to access the encryption device, rather than requiring the attacker to intercept and alter messages being sent to the device. In the following, we discuss the particular details of such fault injection based forgery for MORUS.

In the associated data processing and encryption phases of MORUS, the input associated data and the plaintext are loaded into the internal state of the cipher. Therefore, any changes in the input associated data or the plaintext/ciphertext will affect the internal state. An adversary can apply a forgery attack by injecting faults into the internal state of MORUS to reflect the changes made in the associated data or in the plaintext/ciphertext. In the following, we discuss this idea of the forgery attack using fault injection. For the following we use the term input message M to represent either the associated data or the plaintext.

Suppose the adversary wants to modify the i^{th} bit of the input message block M^t as received by the receiver. Let M'^t denote the modified input message block, where the modification is a bit flip in the i^{th} bit of the original input message block M^t. We observe that the input message block M^t is XOR-ed with the contents of state element $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$. Thus XOR-ing M'^t with the contents of the state element $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$ will flip the i^{th} bit of the respective state element.

To perform a forgery attack on the message M^t, adversary can apply bit flipping faults to the i^{th} bit of the state element $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$. This is equivalent to modifying the input message block from M^t to M'^t at the sender's device, where the modification is a complementation of the i^{th} bit of the original input message block M^t.

Let τ' denote the tag generated for the fault induced version of the state. The adversary can use this faulty tag τ' to perform the forgery attack. Note that the fault was introduced in the internal state bits after the computation of the ciphertext block C^t, thus the fault does not affect the transmitted ciphertext.

Therefore, an adversary needs to modify the i^{th} bit of the corresponding ciphertext block C^t in the case where M^t is a plaintext block. Similarly an adversary needs to modify the i^{th} bit of the corresponding associated data block D^t in the case where M^t is an associated data block.

Adversary can simply intercept and change the transmitted message block M^t to M'^t, and send it with the faulty tag τ' as the output. The adversary does not need to apply any faults at the receiver's side.

In the decryption and tag verification phase, the receiver will XOR the received/recovered message block with the state element $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$ of MORUS. The received/recovered message will be the same as M'^t because of the modified ciphertext block C'^t, or the modified associated data block D'^t. The received/recovered message M'^t is then XOR-ed with the state elements $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$, which complements the i^{th} bit of the respective state elements. The state contents are now the same as the one that resulted after applying the bit-flipping faults at the sender's device.

Once the message is processed, the finalization process at the receiver side generates the tag τ''. Clearly, both of the tags are generated from the same state contents; thus, the tag τ'' generated at the receiver is the same as the tag τ' sent by the sender. Therefore the modified message and faulty tag τ' will be accepted as legitimate at the receiver. Figure 5.5 illustrates the detailed locations of the fault injections to achieve the fault based forgery attack on MORUS.

Let n_f denote the number of bits flipped in an input message block M^t. Then the total number of faults required for the forgery attack is equal to $4 \times n_f$. This is because a single bit flip in the input affects four bits in the state of MORUS.

5.7.2.1 Attack Algorithm for Fault Based Forgery

The steps involved in the fault based forgery attack on MORUS are outlined in Algorithm 5.4.

Algorithm 5.4 Algorithm for Fault Based Forgery Attack on MORUS

1: Insert bit flipping faults at the i^{th} bit of state element $S^t_{1,1}$, $S^t_{2,2}$, $S^t_{3,3}$ and $S^t_{4,4}$ of the sender device.
2: Continue the finalization process at the sender and observe the tag τ'^t.
3: If the goal is associated data modification, then during the transmission complement the i^{th} bit of the associated data from D^t to D'^t. Alternatively, complement the i^{th} bit of the ciphertext from C^t to C'^t.

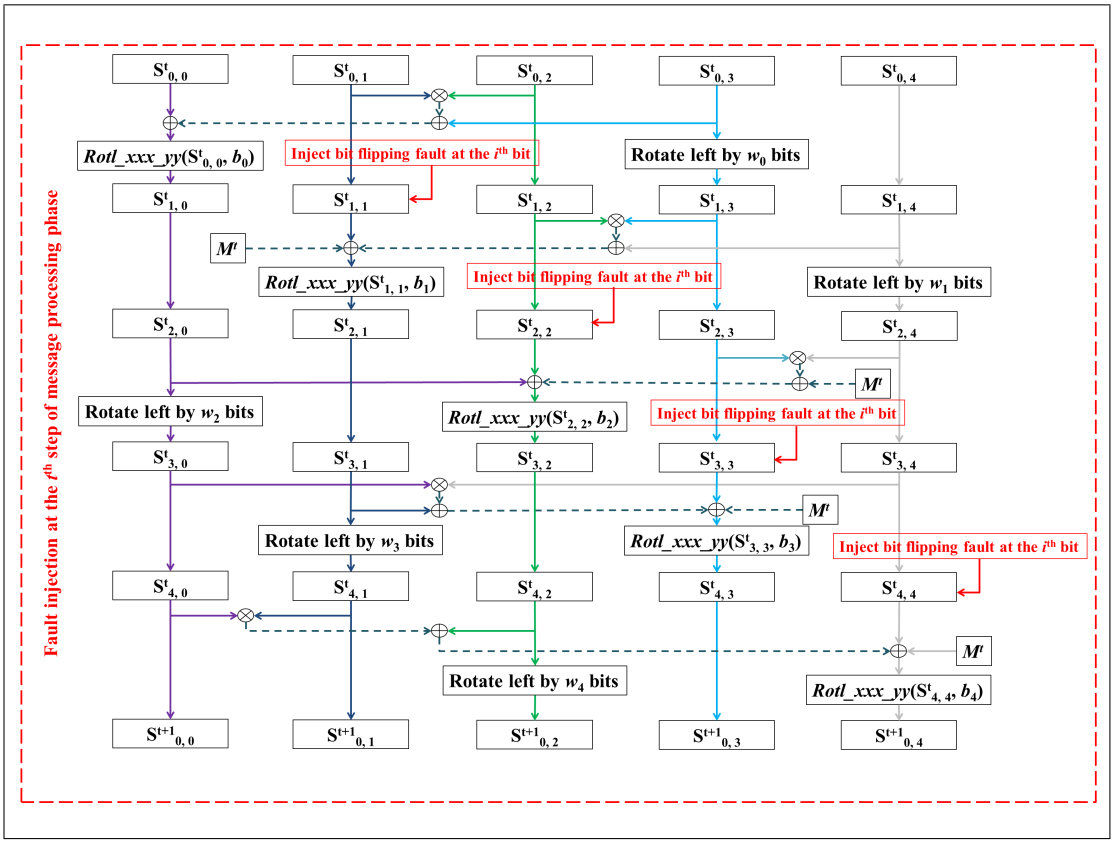

Figure 5.5: Fault based Forgery Attack on MORUS

5.7.3 Summary of Forgery Attack on MORUS

We provided some observations on the forgery attack on MORUS. The first forgery attack described in this chapter is based on the observation that the tag generation function and the output keystream generation function of MORUS is same. Therefore an adversary can use part of the keystream as the authentication tag in a certain scenarios.

The forgery attack using block deletion requires satisfying specific criteria such that state element $S_{0,0}$ consists of all zero bits after processing the first $l_p - 1$ plaintext blocks. The specific conditions required for this forgery attack makes it hard for an adversary to apply the attack without knowing the contents of state elements $S_{0,0}$, $S_{0,1}$, $S_{0,2}$ and $S_{0,3}$. The probability of this forgery attack is 2^{-128} and 2^{-256} for MORUS-640 and MORUS-1280, respectively. This is the same as for brute force search and does not threaten the security claim of MORUS. We note here that MORUS is a candidate in the CAESAR competition aiming to select ciphers for widespread use, and given enough usage, the conditions for this

forgery may occur. So we demonstrate that the attack is possible, not that it is practical. The significance lies in evaluating the resistance of MORUS design to this form of forgery attack. We also note that this forgery attack works because the tag generation function and the keystream generation function of MORUS are the same. This would not be possible if different functions were used for the tag generation and the keystream generation, as is the case for Tiaoxin-346 and AEGIS.

We also introduced a fault based forgery attack on MORUS. This is a simple and trivial attack which works by flipping specific bits in the message and then introducing bit flipping faults in the corresponding bit(s) of state elements where the input message is XOR-ed. The adversary can then observe the faulty ciphertext and faulty tag and use these to get the modified message accepted as legitimate. This attack works because the changes in the input message are reflected in the state elements by introducing bit flipping faults. This is in general true for all the cipher proposals submitted in the CAESAR competition which XOR the input message with the contents of its internal state [147].

The fault based forgery attack is simple and trivial to perform when an adversary has access to the physical implementation of the algorithm. However, there are some argument in the cryptographic community on the goals of an attack based on fault injection. Some researchers argued in the CAESAR discussion forum that the goal of the fault attack is the secret key recovery of the underlying algorithm [147]. Also, it is argued in the same discussion forum that fault based forgery may be relevant [147] because it can compromise one of the security goal of an authenticated encryption algorithm, i.e., compromising the integrity assurance. We think it is important to have necessary countermeasure against fault injection based forgery on MORUS since this can breach the integrity assurance of the cipher.

5.8 Summary on the Security Analysis of MORUS

This chapter examined the feasibility of different attack scenarios on the authenticated encryption cipher MORUS. In particular we analysed the cube attack, fault attack and rotational attack on different variants of MORUS.

Our analysis of cube attack on a reduced version of MORUS-640 with 4-steps of the initialization phase can recover the secret key with a complexity of 2^{10}.

Similarly, the cube attack can recover the secret key for the reduced version of MORUS-1280-128 with a complexity of 2^9. The result shows that key recovery cube attack performs better than the rotational and differential attack applied on reduced version of MORUS. We have also obtained cubes for MORUS-1280 with five initialization steps which can distinguish the cipher output from random with a complexity of 2^8. To date this is the best result obtained on reduced version of MORUS.

We have also introduced a fault based key recovery attack on MORUS-640-128 which can recover the 64 bits of key by introducing 128 bit permanent faults in the 640-bit internal state. Similarly, we introduced a fault attack for MORUS-1280, which can recover the 128 bits of secret key with 256 bits of permanent faults. This is the entire key for MORUS-1280, if the key size is 128-bit. We also analysed the injection of transient faults into the internal state of MORUS. Our analysis shows that the secret keys of MORUS-640-128 and MORUS-1280-256 can be recovered with 384 and 768 bit of transient faults, respectively. Note that all of these fault based key recovery attacks are based on the assumption that there are no associated data. These attacks are not feasible when there is associated data input.

We also investigated the applicability of rotational cryptanalysis on MORUS. We show that all the operations in the state update function of MORUS maintain the rotational pairs when the rotation distance is set to a multiple of the sub-word size. Our investigation also confirms that the rotational pairs can be used as distinguisher for the full version of MORUS if the constants used in MORUS are rotation invariant. However, the actual constants used in MORUS are rotational non-invariant. The introduction of these non-invariant constants in the state update function breaks the symmetry of the rotational pairs. Experimental results show that rotational pairs can be used as distinguishers for only one step of the initialization phase of MORUS. For more than one step, there are not enough known differences in the rotational pairs of MORUS to provide an effective distinguisher. This is due to the XOR-ing of the rotational non-invariant constants at every step. Therefore, it is unlikely for an adversary to construct distinguisher for the full version of MORUS by observing the rotational pairs.

Finally, we provided some observations and analysis on the applicability of forgery attack to MORUS. The forgery attack using block deletion requires to satisfy specific criteria such that state element $S_{0,0}$ consists of all zero bits after

processing the first $l_p - 1$ plaintext blocks. The probability of this forgery attack is 2^{-128} and 2^{-256} for MORUS-640 and MORUS-1280, respectively. Also, note that adversary can manipulate the external inputs to set the contents of state elements $S_{0,0}$ to all zero bits; however, this requires the knowledge of the contents of state element $S_{0,0}$, $S_{0,1}$, $S_{0,2}$ and $S_{0,3}$. Due to the specific conditions required, it is difficult to achieve this forgery attack in practice. We have also illustrated a forgery attack using the bit flipping fault injection. This is a trivial attack when an adversary has access to the implementation of the algorithm in sender's device. This attack requires to introduce four bit faults in MORUS state elements for a single bit flip in the input message. As pointed out by Iwata et al. [147] this type of forgery attack is applicable to other CAESAR candidates.

5.8.1 Security Impact and Possible Recommendations

In this section, we compare the applicability of different attack methods on MORUS. Table 5.14 provides an overall comparison of different attack methods on MORUS.

As illustrated in Table 5.14, the cube attack and rotational attack work only on the reduced version of MORUS. The best of these attacks can work up to 5 steps of the initialization phase, while MORUS has a 16-step initialization phase. This shows that MORUS has a large security margin against cube attacks and rotational attacks.

On the other hand, we see from Table 5.14 that fault based key recovery attacks can be a threat to the full version of MORUS if the associated data processing phase is skipped. This is an attack on the implementation of the algorithm rather than attack on the algorithm itself. Careful consideration should be taken during the physical implementation of MORUS to prevent these type of fault attacks.

We also note that the fault based key recovery attacks described in this chapter works because of the XOR-ing of the secret key bits in to the state element $S_{0,1}$ during the last step of the initialization phase. These attacks would not work if the secret key bits are not XOR-ed in to the state element $S_{0,1}$ after the final step of the initialization phase. We believe the update involving the XOR of secret key with the contents of state element $S_{0,1}$ is performed to make sure that initial state recovery of MORUS does not lead to secret key recovery. That is given the initial state, an adversary can not reverse back to the

Table 5.14: Comparison of Different Attack Methods on MORUS

Algorithm	Attack Method	Attack Type	Initialization Steps	Comments on the Attack
MORUS-640-128	Cube Attack	Key Recovery	4	Complexity: $2^{9.98}$.
MORUS-1280-128	Cube Attack	Key Recovery	4	Complexity: 2^{9}
MORUS-1280-256	Cube Attack	Key Recovery	4	Complexity: $2^{10.14}$
MORUS-640-128	Fault Attack	Key Recovery	16	Recovered Key Bits: 64. Number of Faults Required: 128. Fault Type: Permanent set to zero. Assumption: No associated data.
MORUS-1280-128	Fault Attack	Key Recovery	16	Recovered Key Bits: 128. Number of Faults Required: 256. Fault Type: Permanent set to zero. Assumption: No associated data.
MORUS-1280-256	Fault Attack	Key Recovery	16	Recovered Key Bits: 128. Number of Faults Required: 256. Fault Type: Permanent set to zero. Assumption: No associated data.

MORUS-640-128	Fault Attack	Key Recovery	16	Recovered Key Bits: 128. Number of Faults Required: 384. Fault Type: Transient set to zero. Assumption: No associated data.
MORUS-1280-128	Fault Attack	Key Recovery	16	Recovered Key Bits: 128. Number of Faults Required: 384. Fault Type: Transient set to zero. Assumption: No associated data.
MORUS-1280-256	Fault Attack	Key Recovery	16	Recovered Key Bits: 256. Number of Faults Required: 768. Fault Type: Transient set to zero. Assumption: No associated data.
MORUS-640-128	Cube Testers	Distinguishing	4	Complexity: 2^3.
MORUS-1280-128	Cube Testers	Distinguishing	5	Complexity: 2^8.
MORUS-1280-256	Cube Testers	Distinguishing	5	Complexity: 2^8.
MORUS-640-128	Rotational Attack	Distinguishing	1	Complexity: 2^1.

MORUS-1280-128	Rotational Attack	Distinguishing	1	Complexity: 2^1.
MORUS-1280-256	Rotational Attack	Distinguishing	1	Complexity: 2^1.
MORUS-640-128	Fault Attack	Forgery	16	Forgery Type: Universal forgery. Fault Type: Bit flipping faults. Number of Faults Required: $4 \times n_f$.
MORUS-1280-128	Fault Attack	Forgery	16	Forgery Type: Universal forgery. Fault Type: Bit flipping faults. Number of Faults Required: $4 \times n_f$.
MORUS-1280-256	Fault Attack	Forgery	16	Forgery Type: Universal forgery. Fault Type: Bit flipping faults. Number of Faults Required: $4 \times n_f$.
MORUS-640-128	Block Deletion	Forgery	16	Forgery Type: Existential forgery. Probability of Successful Forgery: 2^{-128}.
MORUS-1280-128	Block Deletion	Forgery	16	Forgery Type: Existential forgery. Probability of Successful Forgery: 2^{-256}.
MORUS-1280-256	Block Deletion	Forgery	16	Forgery Type: Existential forgery. Probability of Successful Forgery: 2^{-256}.

loaded state of MORUS without knowing the secret key. Other than this, the update with the XOR of the secret key does not provide any additional security. Therefore, the designer of MORUS can consider removing this update which can ensure the security against the fault based key recovery attacks presented in this chapter. Alternatively, to provide security against the fault based key recovery of MORUS, an additional state update step can be introduced after this operation and before generating keystream.

From Table 5.14, we can see that the probability of successful forgery using block deletion method is 2^{-128} and 2^{-256} for MORUS-640 and MORUS-1280, respectively. This does not threaten the security claim of MORUS.

Also, Table 5.14 shows that fault based universal forgery attack is trivial with reasonable amount of faults. This type of fault attack is applicable to other CAESAR candidates and it is also arguable that the goal of the fault attack is key recovery rather than to construct a forgery attack [147]. On the other hand, it is also arguable that compromising one of the security goal of any authenticated encryption cipher is sufficient to justify the relevancy of fault based forgery attack [147]. It is important to have physical protections against fault injection based forgery on MORUS since this can compromise the integrity assurance of the cipher.

Based on the overall analysis, no significant threat is detected against the design of the authenticated encryption stream cipher MORUS. Our analysis shows that fault attack may pose some threat on the security of MORUS; however, these type of attacks need to be applied on the physical implementation of MORUS. We conclude that MORUS is a good design and a strong candidate in the final round of the CAESAR competition.

Chapter 6

Conclusions

In recent years, Authenticated Encryption (AE) has received broad interest from the cryptographic community. An AE scheme should be more efficient than applying a two-pass scheme for providing confidentiality and integrity assurance. AE schemes are not yet widely adopted, but the CAESAR competition intends to select secure and efficient AE algorithms that can be widely used.

This thesis reported results from the in depth security analysis of selected AE stream ciphers submitted to the CAESAR competition; namely ACORN, Tiaoxin-346 and MORUS. Also, some results on AEGIS-128L is reported, based on its similarity with Tiaoxin-346. These results were obtained from both theoretical and experimental analysis of the ciphers. The theoretical results were obtained using structural and mathematical analysis of the ciphers. For the experimental analysis, simulations of specific attacks were conducted. All the experiments were conducted using a standard desktop computer with 16GB of memory. The tools used to perform the experiments include: C, Python and Sage software packages.

The rest of the chapter is organised as follows. In Section 6.1 the contributions of this thesis are reviewed. Comparison of the security and efficiency analysis of the selected AE stream ciphers is provided in Section 6.2. Finally, some recommendations for future research are provided in Section 6.3.

6.1 Review of Contributions

The research of this thesis made four contributions. These are the classification of the AE stream ciphers based on several existing approaches, and the security analysis of ACORN, Tiaoxin-346 and MORUS. These contributions are reviewed here.

6.1.1 Classification of AE Stream Ciphers in CAESAR

Chapter 2 provided classifications of 15 AE stream ciphers submitted in the CAESAR competition. These classifications were performed using several existing approaches described by Bellare and Namprempere [13], Katz and Yung [14], and Al-Mashrafi [28]. The classifications are based on three characteristics: the order in which authentication and encryption are performed, the message injection procedure and the number of key-IV pairs used. These classifications were used to determine the similarity among ciphers and potential cryptanalysis techniques for a particular cipher. This is a minor contribution of this thesis.

6.1.2 Analysis of ACORN

Chapter 3 provided an analysis of the authenticated encryption stream cipher ACORN. The results from state collision attacks, cube attacks and forgery attacks on ACORN were reported in this chapter.

The state collision attack was applied to ACORNv1. This attack revealed a weakness in the state update function of ACORNv1, which could be used to obtain state collisions in the internal state of the cipher. Given that a state collision was found, it could be used to apply a forgery attack. The input vectors resulting in collisions were obtained by generating and solving a system of equations. The state collision attack on ACORNv1 described in this thesis is a known key attack. We also investigated the possibility of finding collisions for an unknown key; however, the generated equations get very complex for an unknown key, which makes it infeasible to solve the system of equations. The known key collision attack gives some advantages to the sender or the receiver of a communication channel. Note that this attack is also applicable to the later versions, ACORNv2 and ACORNv3.

The cube attack was applied to ACORNv1 and ACORNv2, with a reduced initialization phase of 477 rounds. The attack recovered the secret key of these

reduced versions with a complexity of 2^{35}. The full version of ACORNv1 and ACORNv2 comprises 2048 rounds, which gives a large security margin against cube attacks. The cube attack on ACORN can possibly be extended to a larger number of rounds, but it will require a significantly large cube size, that will result in a high complexity for the attack. This increase in the complexity was shown in later theoretical results published by Todo et al. [126], which used a variant of the cube attack on ACORNv3 with 704 initialization rounds and required a cube size of 64 with an attack complexity of 2^{122}.

The results reported in Chapter 3 also showed that, in the nonce-reuse scenario, the cube attack can recover the initial state of ACORNv1 with an attack complexity less than the exhaustive search attack. This attack can also be used for the later versions: ACORNv2 and ACORNv3. This attack confirms the designer claim that ACORN should not be used under a nonce-reuse scenario.

Finally, the application of a fault based universal forgery attack on ACORN was demonstrated. This attack used the bit flipping fault model and required a single bit-flipping fault in the internal state for a single bit modification in the input message.

6.1.3 Analysis of Tiaoxin-346

Chapter 4 provided an analysis of the authenticated encryption stream cipher Tiaoxin-346. In this chapter, results on state cycle analysis, cube attacks and fault attacks on Tiaoxin-346 were reported.

In the state cycle analysis, the existence of relatively short cycles in Tiaoxin-346 was investigated. The analysis was conducted using a toy version of Tiaoxin-346, since it is computationally infeasible to explore all the states for the full version. The toy version was created by reducing the internal state word size from 128-bit to 8-bit. For the toy version, relatively short cycles in the individual components were reported. These short cycles in the individual components did not result in relatively short cycles when the components are combined together. The results from the toy version provide a good insight into the full version of Tiaoxin-346.

The construction of 4-round distinguishers on Tiaoxin-346 was reported in Chapter 4, whereas the full version has 15 rounds of initialization. These distinguishers were obtained using cube testers. These can be used to distinguish the output of the 4-round initialization phase of Tiaoxin-346 from random, with a

complexity of 2^3. To date this is the best result reported on a reduced version of Tiaoxin-346.

Two different types of fault attack on Tiaoxin-346 were reported in Chapter 4. The first type of fault injection applied a forgery attack, similar to the one reported for ACORN. This attack used the bit-flipping fault model to obtain a forged message, which would be accepted as legitimate by the receiver.

In the second type of fault attack, the differential fault attack on Tiaoxin-346 described by Dey et. al. [38] was improved. This approach recoverd the secret key of Tiaoxin-346 with 36 random multi-byte faults, with practical complexity. The approach described by Dey et. al. [38] recovers the secret key with 3 multi-byte faults, but requires a bit-flipping fault model. The improvement in the attack is the fault model, which used random faults instead of bit-flipping faults.

At the end of this chapter, we identified structural similarities between Tiaoxin-346 and AEGIS. We demonstrate that, due to these similarities, AEGIS may be vulnerable to similar attacks to those we applied to Tiaoxin-346. In particular, an improved fault based state recovery attack was demonstrated on one of the variants of the AEGIS cipher. The improvement in the attack is the fault model, which used random faults instead of bit-flipping faults used by Dey et. al. [38].

6.1.4 Analysis of MORUS

Chapter 5 provided an analysis of the authenticated encryption stream cipher MORUS. This chapter reported the results on the application of cube attacks, rotational attacks, forgery attacks and fault attacks on MORUS.

The cube attack was applied to a reduced version of MORUS. This application of the cube attack recovered the secret key of a reduced version of MORUS with negligible complexity. The cube attack based key recovery was performed on 4 steps of the initialization phase, whereas the full version has 16 steps. This cube attack on MORUS improves the previous cryptanalytic results based on differential attack [40]. The improvements are both in terms of the number of initialization rounds and the attack complexity.

The application of a cube attack also demonstrated the construction of distinguishers up to 5 steps of the initialization phase of MORUS-1280. To date, this is the best result obtained on a reduced version of MORUS.

The investigation of rotational attack showed the importance of the use of non-invariant constants in the MORUS state update function. This investigation

demonstrated that all the operations used in MORUS are rotationally invariant. This means if the constants used in initializing the MORUS state were rotationally invariant, then a rotational attack could be applied to construct distinguishers. The constants used in MORUS are rotational non-invariant, which provides resistance against the rotational attacks.

Chapter 5 also reported on specific conditions in the MORUS state that can be used to apply a forgery attack. This attack can be performed by deleting some input blocks, when the internal state of MORUS satisfies specific conditions. The probability of this forgery attack does not threaten the MORUS security claims.

Finally, Chapter 5 reported on fault injection based forgery and key recovery attacks on MORUS. The fault based forgery attack is similar to the one reported for ACORN and Tiaoxin-346. This attack uses the bit-flipping fault model and needs access to the physical implementation of Tiaoxin-346 for this attack.

A partial key recovery of MORUS under the permanent set-to-zero fault model was described in Chapter 5. A similar fault attack based on a transient set-to-zero fault model was also described for MORUS. With this model an adversary can recover the entire secret key, but requires more faults. Both of these attacks work under the ciphertext only attack model. An adversary needs to have access to multiple faulty ciphertexts to perform these attacks. These ciphertexts are computed with the same key, but different initialization vectors. Therefore, these attacks work under the nonce-respecting model.

6.2 Comparison of Selected AE Stream Ciphers

A comparison of the AE stream ciphers: ACORN, Tiaoxin-346, AEGIS-128L, and MORUS is provided in this section. The comparison is done both in terms of our security analysis and the performance analysis by Homsirikamol et. al. [153] and Ankele et. al. [154]. Also some recommendations are provided to avoid potential attacks.

6.2.1 Comparison of the Security Analysis

The applicability of different attack methods on the three AE stream ciphers: ACORN, Tiaoxin-346, and MORUS is discussed in this section. A comparison of these ciphers based on the security analysis of this thesis is given in Table 6.1.

As indicated in Table 6.1, a cube attack can work with a reduced initialization phase for all three AE stream ciphers. The results reported in this thesis show that all of these ciphers have a large security margin against the cube attack. The cube attack also validates the designers claim that these ciphers should not be used in the nonce reuse scenario.

For all the three AE stream ciphers of interest, the fault based forgery appears to be trivial. The result from the thesis also shows that a fault attack can be used to recover the secret key of Tiaoxin-346 and MORUS. A fault based key recovery attack on ACORN was not explored in this thesis, since there is already existing contributions on this topic in the literature [127]. The results tabulated in Table 6.1 shows that the fault injected key recovery attack applied to Tiaoxin-346 is a differential fault attack with random fault model, which works under nonce-reuse scenario. Also, Table 6.1 shows that a similar random fault based state recovery attack can be applicable to AEGIS-128L. Table 6.1 illustrates that the fault injected key recovery attack on MORUS can recover partial or full key recovery with different fault models.

6.2.1.1 Recommendations to Avoid Potential Attacks

As illustrated in Table 6.1, the differential fault attack on Tiaoxin-346 provides key recovery, whereas a similar attack on AEGIS-128L only provides state recovery. Key recovery in Tiaoxin-346 is possible because the state updates of the components in Tiaoxin-346 are independent and invertible, and at the beginning of the initialization phase the entire key is loaded in each of the components. A differential fault based key recovery attack on Tiaoxin-346 would be more difficult to perform, if the key was distributed across the entire initial state during the initialization phase.

Additionally, the differential fault based key recovery attack on Tiaoxin-346 will not be possible if the key is used as an external input during the updates of the initialization phase, as is the case for AEGIS-128L. This does not prevent the differential fault based state recovery in Tiaoxin-346, but will prevent the attacker from recovering the secret key.

Also, as illustrated in Table 6.1, unlike Tiaoxin-346, the fault attack on MORUS can work under the nonce-respecting scenario. This is because the secret key of MORUS is XOR-ed with a particular component of the internal state after performing the last round of the initialization phase. This update

Table 6.1: Comparison of the Security Analysis of Selected AE Stream Ciphers from the CAESAR Competition

Algorithm	Attack	Assumption
ACORN	State Collision	Known-key attack
	Cube Attack (Key Recovery)	Reduced initialization phase
	Cube Attack (State Recovery)	Nonce-reuse
	Fault Attack (Forgery)	Bit-flipping fault model
Tiaoxin-346	State Cycle Analysis	Toy version
	Cube Attack (Distinguisher)	Reduced initialization phase
	Fault Attack (Forgery)	Bit-flipping fault model
	Fault Attack (Key Recovery)	Nonce-reuse Random fault model
AEGIS-128L	Fault Attack (State Recovery)	Nonce-reuse Random fault model
MORUS	Cube Attack (Key Recovery)	Reduced initialization phase
	Cube Attack (Distinguisher)	Reduced initialization phase
	Rotational Attack (Distinguisher)	Reduced initialization phase
	Forgery Attack	Low probability condition
	Fault Attack (Forgery)	Bit-flipping fault model
	Fault Attack (Partial Key Recovery)	Permanent set-to-zero fault
	Fault Attack (Full Key Recovery)	Transient set-to-zero fault

is performed to prevent a key recovery attack when the initial internal state is recovered. To provide security against the fault based key recovery of MORUS, an additional state update step can be introduced after this operation and before generating keystream.

Table 6.1 also shows a forgery attack on MORUS, which is applicable based on a very low probability condition. This attack does not threaten the security claim of MORUS. We point out here that this forgery attack works because the tag generation function and the keystream generation function of MORUS are

the same. This attack would not work if different functions were used for the tag generation and the keystream generation, as is the case for Tiaoxin-346 and AEGIS.

Among all the results, the fault attack seems to be the most powerful tool for the AE stream ciphers discussed in this thesis. These are attacks on the implementation rather than the algorithm itself. It is important to have appropriate physical protection to prevent these type of fault attacks. Also several results provided in this thesis validate the designer claims that these ciphers should not be used under the nonce-reuse scenario. Finally, the initialization phase of the cipher should be used as the designer recommended, reducing the initialization phase to increase the speed is not recommended.

6.2.2 Comparison of the Performance Analysis

The performance of the selected AE stream ciphers that are of interest in this research is compared in this section. As indicated in Chapter 1, performance (efficiency) analysis of the AE stream ciphers is out of the research scope of this thesis. For completeness, based on existing literature [153, 154, 155] an efficiency analysis is provided in this section, including the hardware and software efficiency of the AE stream ciphers: ACORN, Tiaoxin-346, AEGIS-128L, and MORUS.

Table 6.2 compares these ciphers based on the hardware efficiency analysis provided by Homsirikamol et. al. [153, 155]. The performance metrics used in Table 6.2 are throughput, area, throughput/area, relative throughput, relative area and relative throughput/area. The throughput determines the speed of the cipher, whereas area determine the hardware space required for the implementation. The relative results, i.e., relative throughput, relative area and relative throughput/area, are the results of the respective ciphers relative to AES-GCM.

As indicated in Table 6.2, AEGIS-128L has the highest throughput, followed by Tiaoxin-346, MORUS-1280-128 and then ACORNv3. In terms of hardware space, ACORN requires the least space. With respect to the performance metric of throughput/area, MORUS provides the best result among these four ciphers. In terms of the performance metric of relative throughput/area, all these four ciphers provide better performance than the AES-GCM. Based on the hardware benchmarking results [153, 155] of the third round candidates, these four ciphers are in the top four of the CAESAR candidates in terms of the performance metric of the throughput/area.

Table 6.3 compares the software efficiency of these AE steam ciphers based on the result provided by Ankele et. al. [154]. The software efficiency is measured in terms of the number of average CPU cycles required to process a single message byte. Table 6.3 tabulates the result for different message sizes. Based on these results, Tiaoxin-346 has the fastest software performance among these four ciphers, when considering a large message size. However, when comparing for a short message size, AEGIS-128L is observed to have the fastest software performance. Overall software performance evaluation of the second round CAESAR candidate [154] claims that Tiaoxin-346 is the fastest among all the second round CAESAR candidates.

6.2.3 Overall Evaluation

An overall evaluation of the three AE stream ciphers ACORN, Tiaoxin-346, and MORUS is provided in this section. Also, we provide some comments on the efficiency analysis of AEGIS-128L, along with these three ciphers. For ACORN, weaknesses in its state update function was demonstrated that can be used to generate collisions. This is a known key attack that is not feasible for an external adversary, but gives some advantages to the sender or receiver of a communication channel.

The results in this thesis confirm that these ciphers should not be used in the nonce-reuse scenario. All these ciphers have shown a strong security margin against the other classical cryptanalytic attacks considered in this thesis.

Unlike classical attacks, the results from the thesis shows that physical attacks such as fault injections can be powerful against all these ciphers. Using a fault attack on MORUS, an adversary can perform a key recovery attack under the nonce-respecting scenario. The fault injection technique described for Tiaoxin-346 is a differential fault attack, which can be performed under the nonce-reuse scenario. The designer of Tiaoxin-346 prohibits the use of Tiaoxin-346 in such a scenario. Additionally, the results demonstrated that a fault based forgery attack can be applied to all these three ciphers. Fault injection is a side channel attack, applied on the implementation rather than the cipher itself; nevertheless, careful measures should be taken to avoid such attacks.

Based on the efficiency analysis provided by the CAESAR committee, Tiaoxin-346 performs the best among these three ciphers, in terms of speed in hardware and software. In fact, Tiaoxin-346 has the best performance in terms of software

Table 6.2: Comparison of the Hardware Efficiency Analysis of Selected AE Stream Ciphers

Algorithm	Throughput (Mbits/s)	Area (No. of LUTs)	Throughput/Area ((Mbits/s)/LUT)	Relative Throughput	Relative Area	Relative Throughput/Area
ACORNv3	3,419	500	6.838	3.49	0.16	8.91
Tiaoxin-346	52,838	7,123	7.418	16.3	2.24	7.27
MORUS-1280-128	49,421	3,406	14.51	15.3	1.07	14.2
AEGIS-128L	70,927	7,592	9.342	21.9	2.39	9.16

Table 6.3: Comparison of the Software Efficiency Analysis of Selected AE Stream Ciphers

Algorithm	Message Length (Byte)	Speed (Cycles per Byte (cpb))
ACORNv2	1	391
	16	116.13
	557	11.65
	1500	9.05
	16000	7.67
	1000000	7.52
MORUS-1280-256v2	1	100.4
	16	28.82
	557	1.73
	1500	1.07
	16000	0.71
	1000000	0.68
Tiaoxin-346	1	50.06
	16	13.93
	557	0.71
	1500	0.38
	16000	0.2
	1000000	0.19
AEGIS-128L	1	45
	16	12.74
	557	0.7
	1500	0.4
	16000	0.22
	1000000	0.21

speed among all the candidates in the third round. On the other hand, ACORN requires the least hardware space among all the third round candidates. Considering the performance metric of throughput/area, MORUS has the best results.

Based on the efficiency analysis, the CAESAR committee has suggested three different use cases. Use case 1 describes the scenario considering lightweight applications, for which the cipher can be implemented using small hardware area and/or small code for 8-bit CPUs. Use case 2 describes the scenario considering high performance applications, for which the cipher can be implemented using 64-bit CPUs and/or dedicated hardware. The CAESAR committee defined Use case 3 as "Defense in depth", which can provide security under the nonce-misuse scenario.

As indicated in this section, all these three ciphers have good performance on specific performance metrics. None of these three ciphers can be used in the nonce-reuse scenario; and therefore are not suitable for the Use case 3. For the Use case 1, ACORN appears to be one of the strongest candidate as it requires the least hardware space among all the candidates. For Use case 2, MORUS, Tiaoxin-346 and AEGIS seem to be strong candidates. Compared to MORUS, Tiaoxin-346 and AEGIS-128L is faster in terms of speed in hardware; however, MORUS requires comparatively less hardware space and thus results in the better performance in terms of throughput/area. On the other hand, Tiaoxin-346 has the fastest performance in software, which is also a critical criterion for Use case 2. For the CAESAR competition, there may be multiple winners announced based on different use cases for the hardware and software profile, as was done for the eStream competition.

6.3 Future Research Directions

An analysis of the state collision attack on ACORN was presented in Chapter 3. Similar analysis of state collision attack can be conducted for MORUS and Tiaoxin-346. Also, the state collisions obtained for ACORN are key dependent, therefore such collision must leak information about the secret key. Future research can be conducted to investigate methods on how this leak can be exploited.

The application of the cube attack on ACORN described in Chapter 3 was improved by Todo et. al. [126], by using the division property. The improved

attack considers the underlying construction as a non-blackbox polynomial. Future research can consider the analysis of MORUS and Tiaoxin-346, using cube attacks based on the division property.

The results reported in this thesis indicate that fault injection attacks can be powerful tool for the AE stream ciphers discussed in this thesis. Future work can focus on investigating the possible remedies against fault based attacks.

The CAESAR third round candidate AEGIS is a stream cipher based construction. Structural similarity between AEGIS and Tiaoxin-346 was discussed in Chapter 4, and it was demonstrated that AEGIS may be vulnerable to similar attacks to those applied to Tiaoxin-346. However, due to the time constraints, an in-depth analysis of AEGIS has not been included in this thesis. Future work can focus on extending the security analysis of AEGIS.

Appendix A

Cube Attack on ACORN

A.1 Cube Attack on ACORN Initialization Phase

Table A.1: Linear Equations for ACORNv1 with 477 Initialization Rounds (Cube Sets are chosen from the Initialization Vector)

Cube Indexes	Output Index	Superpoly
$113, 117, 86, 0, 56$	493	$k_5 \oplus k_7 \oplus k_{10} \oplus k_{11} \oplus k_{44} \oplus 1$
$114, 118, 87, 1, 57$	494	$k_6 \oplus k_8 \oplus k_{11} \oplus k_{12} \oplus k_{45} \oplus 1$
$115, 119, 88, 2, 58$	495	$k_7 \oplus k_9 \oplus k_{12} \oplus k_{13} \oplus k_{46} \oplus 1$
$116, 120, 89, 3, 59$	496	$k_8 \oplus k_{10} \oplus k_{13} \oplus k_{14} \oplus k_{47} \oplus 1$
$117, 121, 90, 4, 60$	497	$k_9 \oplus k_{11} \oplus k_{14} \oplus k_{15} \oplus k_{48} \oplus 1$
$118, 122, 91, 5, 61$	498	$k_{10} \oplus k_{12} \oplus k_{15} \oplus k_{16} \oplus k_{49} \oplus 1$
$119, 123, 92, 6, 62$	499	$k_{11} \oplus k_{13} \oplus k_{16} \oplus k_{17} \oplus k_{50} \oplus 1$
$120, 124, 93, 7, 63$	500	$k_{12} \oplus k_{14} \oplus k_{17} \oplus k_{18} \oplus k_{51} \oplus 1$
$121, 125, 94, 8, 64$	501	$k_{13} \oplus k_{15} \oplus k_{18} \oplus k_{19} \oplus k_{52} \oplus 1$
$122, 126, 95, 9, 65$	502	$k_{14} \oplus k_{16} \oplus k_{19} \oplus k_{20} \oplus k_{53} \oplus 1$
$123, 127, 96, 10, 66$	503	$k_0 \oplus k_{15} \oplus k_{17} \oplus k_{20} \oplus k_{21} \oplus k_{54}$
$13, 106, 97, 0, 104$	482	$k_1 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{14} \oplus k_{18} \oplus k_{21} \oplus k_{25} \oplus k_{29} \oplus k_{30} \oplus k_{31} \oplus k_{32} \oplus k_{35} \oplus k_{38} \oplus k_{40} \oplus k_{43} \oplus k_{44} \oplus k_{68} \oplus k_{77}$
$14, 107, 98, 1, 105$	483	$k_2 \oplus k_3 \oplus k_4 \oplus k_5 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{15} \oplus k_{19} \oplus k_{22} \oplus k_{26} \oplus k_{30} \oplus k_{31} \oplus k_{32} \oplus k_{33} \oplus k_{36} \oplus k_{39} \oplus k_{41} \oplus k_{44} \oplus k_{45} \oplus k_{69} \oplus k_{78}$
$15, 108, 99, 2, 106$	484	$k_3 \oplus k_4 \oplus k_5 \oplus k_6 \oplus k_9 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{20} \oplus k_{23} \oplus k_{27} \oplus k_{31} \oplus k_{32} \oplus k_{33} \oplus k_{34} \oplus k_{37} \oplus k_{40} \oplus k_{42} \oplus k_{45} \oplus k_{46} \oplus k_{70} \oplus k_{79}$
$16, 109, 100, 3, 107$	485	$k_4 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_{10} \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{17} \oplus k_{21} \oplus k_{24} \oplus k_{28} \oplus k_{32} \oplus k_{33} \oplus k_{34} \oplus k_{35} \oplus k_{38} \oplus k_{41} \oplus k_{43} \oplus k_{46} \oplus k_{47} \oplus k_{71} \oplus k_{80}$

$17, 110, 101, 4, 108$	486	$k_0 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_{11} \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{22} \oplus k_{25} \oplus k_{29} \oplus k_{33} \oplus k_{34} \oplus k_{35} \oplus k_{36} \oplus k_{39} \oplus k_{42} \oplus k_{44} \oplus k_{47} \oplus k_{48} \oplus k_{72} \oplus k_{81} \oplus 1$
$18, 111, 102, 5, 109$	487	$k_0 \oplus k_1 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{12} \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{23} \oplus k_{26} \oplus k_{30} \oplus k_{34} \oplus k_{35} \oplus k_{36} \oplus k_{37} \oplus k_{40} \oplus k_{43} \oplus k_{45} \oplus k_{48} \oplus k_{49} \oplus k_{73} \oplus k_{82}$
$19, 112, 103, 6, 110$	488	$k_0 \oplus k_1 \oplus k_2 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{10} \oplus k_{13} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{20} \oplus k_{24} \oplus k_{27} \oplus k_{31} \oplus k_{35} \oplus k_{36} \oplus k_{37} \oplus k_{38} \oplus k_{41} \oplus k_{44} \oplus k_{46} \oplus k_{49} \oplus k_{50} \oplus k_{74} \oplus k_{83} \oplus 1$
$20, 113, 104, 7, 111$	489	$k_1 \oplus k_2 \oplus k_3 \oplus k_8 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{14} \oplus k_{16} \oplus k_{17} \oplus k_{18} \oplus k_{21} \oplus k_{25} \oplus k_{28} \oplus k_{32} \oplus k_{36} \oplus k_{37} \oplus k_{38} \oplus k_{39} \oplus k_{42} \oplus k_{45} \oplus k_{47} \oplus k_{50} \oplus k_{51} \oplus k_{75} \oplus k_{84} \oplus 1$
$21, 114, 105, 8, 112$	490	$k_2 \oplus k_3 \oplus k_4 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{15} \oplus k_{17} \oplus k_{18} \oplus k_{19} \oplus k_{22} \oplus k_{26} \oplus k_{29} \oplus k_{33} \oplus k_{37} \oplus k_{38} \oplus k_{39} \oplus k_{40} \oplus k_{43} \oplus k_{46} \oplus k_{48} \oplus k_{51} \oplus k_{52} \oplus k_{76} \oplus k_{85} \oplus 1$
$22, 115, 106, 9, 113$	491	$k_3 \oplus k_4 \oplus k_5 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{23} \oplus k_{27} \oplus k_{30} \oplus k_{34} \oplus k_{38} \oplus k_{39} \oplus k_{40} \oplus k_{41} \oplus k_{44} \oplus k_{47} \oplus k_{49} \oplus k_{52} \oplus k_{53} \oplus k_{77} \oplus k_{86} \oplus 1$
$23, 116, 107, 10, 114$	492	$k_0 \oplus k_4 \oplus k_5 \oplus k_6 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{17} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{24} \oplus k_{28} \oplus k_{31} \oplus k_{35} \oplus k_{39} \oplus k_{40} \oplus k_{41} \oplus k_{42} \oplus k_{45} \oplus k_{48} \oplus k_{50} \oplus k_{53} \oplus k_{54} \oplus k_{78} \oplus k_{87}$
$24, 117, 108, 11, 115$	493	$k_0 \oplus k_1 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{25} \oplus k_{29} \oplus k_{32} \oplus k_{36} \oplus k_{40} \oplus k_{41} \oplus k_{42} \oplus k_{43} \oplus k_{46} \oplus k_{49} \oplus k_{51} \oplus k_{54} \oplus k_{55} \oplus k_{79} \oplus k_{88} \oplus 1$
$25, 118, 109, 12, 116$	494	$k_1 \oplus k_2 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{26} \oplus k_{30} \oplus k_{33} \oplus k_{37} \oplus k_{41} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{47} \oplus k_{50} \oplus k_{52} \oplus k_{55} \oplus k_{56} \oplus k_{80} \oplus k_{89} \oplus 1$
$26, 119, 110, 13, 117$	495	$k_0 \oplus k_2 \oplus k_3 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{20} \oplus k_{22} \oplus k_{23} \oplus k_{24} \oplus k_{27} \oplus k_{31} \oplus k_{34} \oplus k_{38} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{45} \oplus k_{48} \oplus k_{51} \oplus k_{53} \oplus k_{56} \oplus k_{57} \oplus k_{81} \oplus k_{90}$
$27, 120, 111, 14, 118$	496	$k_1 \oplus k_3 \oplus k_4 \oplus k_8 \oplus k_9 \oplus k_{10} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{18} \oplus k_{21} \oplus k_{23} \oplus k_{24} \oplus k_{25} \oplus k_{28} \oplus k_{32} \oplus k_{35} \oplus k_{39} \oplus k_{43} \oplus k_{44} \oplus k_{45} \oplus k_{46} \oplus k_{49} \oplus k_{52} \oplus k_{54} \oplus k_{57} \oplus k_{58} \oplus k_{82} \oplus k_{91}$
$28, 121, 112, 15, 119$	497	$k_2 \oplus k_4 \oplus k_5 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{16} \oplus k_{17} \oplus k_{18} \oplus k_{19} \oplus k_{22} \oplus k_{24} \oplus k_{25} \oplus k_{26} \oplus k_{29} \oplus k_{33} \oplus k_{36} \oplus k_{40} \oplus k_{44} \oplus k_{45} \oplus k_{46} \oplus k_{47} \oplus k_{50} \oplus k_{53} \oplus k_{55} \oplus k_{58} \oplus k_{59} \oplus k_{83} \oplus k_{92}$
$29, 122, 113, 16, 120$	498	$k_3 \oplus k_5 \oplus k_6 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{17} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{23} \oplus k_{25} \oplus k_{26} \oplus k_{27} \oplus k_{30} \oplus k_{34} \oplus k_{37} \oplus k_{41} \oplus k_{45} \oplus k_{46} \oplus k_{47} \oplus k_{48} \oplus k_{51} \oplus k_{54} \oplus k_{56} \oplus k_{59} \oplus k_{60} \oplus k_{84} \oplus k_{93}$
$30, 123, 114, 17, 121$	499	$k_0 \oplus k_4 \oplus k_6 \oplus k_7 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{24} \oplus k_{26} \oplus k_{27} \oplus k_{28} \oplus k_{31} \oplus k_{35} \oplus k_{38} \oplus k_{42} \oplus k_{46} \oplus k_{47} \oplus k_{48} \oplus k_{49} \oplus k_{52} \oplus k_{55} \oplus k_{57} \oplus k_{60} \oplus k_{61} \oplus k_{85} \oplus k_{94} \oplus 1$

$31, 124, 115, 18, 122$	500	$k_0 \oplus k_1 \oplus k_5 \oplus k_7 \oplus k_8 \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{25} \oplus k_{27} \oplus k_{28} \oplus k_{29} \oplus k_{32} \oplus k_{36} \oplus k_{39} \oplus k_{43} \oplus k_{47} \oplus k_{48} \oplus k_{49} \oplus k_{50} \oplus k_{53} \oplus k_{56} \oplus k_{58} \oplus k_{61} \oplus k_{62} \oplus k_{86} \oplus k_{95}$
$32, 125, 116, 19, 123$	501	$k_1 \oplus k_2 \oplus k_6 \oplus k_8 \oplus k_9 \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{26} \oplus k_{28} \oplus k_{29} \oplus k_{30} \oplus k_{33} \oplus k_{37} \oplus k_{40} \oplus k_{44} \oplus k_{48} \oplus k_{49} \oplus k_{50} \oplus k_{51} \oplus k_{54} \oplus k_{57} \oplus k_{59} \oplus k_{62} \oplus k_{63} \oplus k_{87} \oplus k_{96}$
$33, 126, 117, 20, 124$	502	$k_2 \oplus k_3 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{24} \oplus k_{27} \oplus k_{29} \oplus k_{30} \oplus k_{31} \oplus k_{34} \oplus k_{38} \oplus k_{41} \oplus k_{45} \oplus k_{49} \oplus k_{50} \oplus k_{51} \oplus k_{52} \oplus k_{55} \oplus k_{58} \oplus k_{60} \oplus k_{63} \oplus k_{64} \oplus k_{88} \oplus k_{97}$
$34, 127, 118, 21, 125$	503	$k_0 \oplus k_3 \oplus k_4 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{22} \oplus k_{23} \oplus k_{24} \oplus k_{25} \oplus k_{28} \oplus k_{30} \oplus k_{31} \oplus k_{32} \oplus k_{35} \oplus k_{39} \oplus k_{42} \oplus k_{46} \oplus k_{50} \oplus k_{51} \oplus k_{52} \oplus k_{53} \oplus k_{56} \oplus k_{59} \oplus k_{61} \oplus k_{64} \oplus k_{65} \oplus k_{89} \oplus k_{98} \oplus 1$
$16, 28, 101, 53, 92$	477	$k_3 \oplus k_8 \oplus k_{15} \oplus k_{17} \oplus k_{19} \oplus k_{23} \oplus k_{25} \oplus k_{28} \oplus k_{29} \oplus k_{62}$
$17, 29, 102, 54, 93$	478	$k_4 \oplus k_9 \oplus k_{16} \oplus k_{18} \oplus k_{20} \oplus k_{24} \oplus k_{26} \oplus k_{29} \oplus k_{30} \oplus k_{63}$
$18, 30, 103, 55, 94$	479	$k_5 \oplus k_{10} \oplus k_{17} \oplus k_{19} \oplus k_{21} \oplus k_{25} \oplus k_{27} \oplus k_{30} \oplus k_{31} \oplus k_{64}$
$19, 31, 104, 56, 95$	480	$k_6 \oplus k_{11} \oplus k_{18} \oplus k_{20} \oplus k_{22} \oplus k_{26} \oplus k_{28} \oplus k_{31} \oplus k_{32} \oplus k_{65}$
$20, 32, 105, 57, 96$	481	$k_0 \oplus k_7 \oplus k_{12} \oplus k_{19} \oplus k_{21} \oplus k_{23} \oplus k_{27} \oplus k_{29} \oplus k_{32} \oplus k_{33} \oplus k_{66} \oplus 1$
$21, 33, 106, 58, 97$	482	$k_0 \oplus k_1 \oplus k_8 \oplus k_{13} \oplus k_{20} \oplus k_{22} \oplus k_{24} \oplus k_{28} \oplus k_{30} \oplus k_{33} \oplus k_{34} \oplus k_{67}$
$22, 34, 107, 59, 98$	483	$k_0 \oplus k_1 \oplus k_2 \oplus k_9 \oplus k_{14} \oplus k_{21} \oplus k_{23} \oplus k_{25} \oplus k_{29} \oplus k_{31} \oplus k_{34} \oplus k_{35} \oplus k_{68} \oplus 1$
$23, 35, 108, 60, 99$	484	$k_1 \oplus k_2 \oplus k_3 \oplus k_{10} \oplus k_{15} \oplus k_{22} \oplus k_{24} \oplus k_{26} \oplus k_{30} \oplus k_{32} \oplus k_{35} \oplus k_{36} \oplus k_{69} \oplus 1$
$24, 36, 109, 61, 100$	485	$k_0 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_{11} \oplus k_{16} \oplus k_{23} \oplus k_{25} \oplus k_{27} \oplus k_{31} \oplus k_{33} \oplus k_{36} \oplus k_{37} \oplus k_{70}$
$25, 37, 110, 62, 101$	486	$k_1 \oplus k_3 \oplus k_4 \oplus k_5 \oplus k_{12} \oplus k_{17} \oplus k_{24} \oplus k_{26} \oplus k_{28} \oplus k_{32} \oplus k_{34} \oplus k_{37} \oplus k_{38} \oplus k_{71}$
$26, 38, 111, 63, 102$	487	$k_2 \oplus k_4 \oplus k_5 \oplus k_6 \oplus k_{13} \oplus k_{18} \oplus k_{25} \oplus k_{27} \oplus k_{29} \oplus k_{33} \oplus k_{35} \oplus k_{38} \oplus k_{39} \oplus k_{72}$
$27, 39, 112, 64, 103$	488	$k_0 \oplus k_3 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_{14} \oplus k_{19} \oplus k_{26} \oplus k_{28} \oplus k_{30} \oplus k_{34} \oplus k_{36} \oplus k_{39} \oplus k_{40} \oplus k_{73} \oplus 1$
$28, 40, 113, 65, 104$	489	$k_0 \oplus k_1 \oplus k_4 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_{15} \oplus k_{20} \oplus k_{27} \oplus k_{29} \oplus k_{31} \oplus k_{35} \oplus k_{37} \oplus k_{40} \oplus k_{41} \oplus k_{74}$
$29, 41, 114, 66, 105$	490	$k_1 \oplus k_2 \oplus k_5 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{16} \oplus k_{21} \oplus k_{28} \oplus k_{30} \oplus k_{32} \oplus k_{36} \oplus k_{38} \oplus k_{41} \oplus k_{42} \oplus k_{75}$
$30, 42, 115, 67, 106$	491	$k_2 \oplus k_3 \oplus k_6 \oplus k_8 \oplus k_9 \oplus k_{10} \oplus k_{17} \oplus k_{22} \oplus k_{29} \oplus k_{31} \oplus k_{33} \oplus k_{37} \oplus k_{39} \oplus k_{42} \oplus k_{43} \oplus k_{76}$
$31, 43, 116, 68, 107$	492	$k_0 \oplus k_3 \oplus k_4 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{18} \oplus k_{23} \oplus k_{30} \oplus k_{32} \oplus k_{34} \oplus k_{38} \oplus k_{40} \oplus k_{43} \oplus k_{44} \oplus k_{77} \oplus 1$
$32, 44, 117, 69, 108$	493	$k_1 \oplus k_4 \oplus k_5 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{19} \oplus k_{24} \oplus k_{31} \oplus k_{33} \oplus k_{35} \oplus k_{39} \oplus k_{41} \oplus k_{44} \oplus k_{45} \oplus k_{78} \oplus 1$
$33, 45, 118, 70, 109$	494	$k_0 \oplus k_2 \oplus k_5 \oplus k_6 \oplus k_9 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{20} \oplus k_{25} \oplus k_{32} \oplus k_{34} \oplus k_{36} \oplus k_{40} \oplus k_{42} \oplus k_{45} \oplus k_{46} \oplus k_{79}$

$34, 46, 119, 71, 110$	495	$k_1 \oplus k_3 \oplus k_6 \oplus k_7 \oplus k_{10} \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{21} \oplus k_{26} \oplus k_{33} \oplus k_{35} \oplus k_{37} \oplus k_{41} \oplus k_{43} \oplus k_{46} \oplus k_{47} \oplus k_{80}$
$35, 47, 120, 72, 111$	496	$k_0 \oplus k_2 \oplus k_4 \oplus k_7 \oplus k_8 \oplus k_{11} \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{22} \oplus k_{27} \oplus k_{34} \oplus k_{36} \oplus k_{38} \oplus k_{42} \oplus k_{44} \oplus k_{47} \oplus k_{48} \oplus k_{81} \oplus 1$
$36, 48, 121, 73, 112$	497	$k_1 \oplus k_3 \oplus k_5 \oplus k_8 \oplus k_9 \oplus k_{12} \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{23} \oplus k_{28} \oplus k_{35} \oplus k_{37} \oplus k_{39} \oplus k_{43} \oplus k_{45} \oplus k_{48} \oplus k_{49} \oplus k_{82} \oplus 1$
$37, 49, 122, 74, 113$	498	$k_2 \oplus k_4 \oplus k_6 \oplus k_9 \oplus k_{10} \oplus k_{13} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{24} \oplus k_{29} \oplus k_{36} \oplus k_{38} \oplus k_{40} \oplus k_{44} \oplus k_{46} \oplus k_{49} \oplus k_{50} \oplus k_{83} \oplus 1$
$38, 50, 123, 75, 114$	499	$k_3 \oplus k_5 \oplus k_7 \oplus k_{10} \oplus k_{11} \oplus k_{14} \oplus k_{16} \oplus k_{17} \oplus k_{18} \oplus k_{25} \oplus k_{30} \oplus k_{37} \oplus k_{39} \oplus k_{41} \oplus k_{45} \oplus k_{47} \oplus k_{50} \oplus k_{51} \oplus k_{84} \oplus 1$
$39, 51, 124, 76, 115$	500	$k_4 \oplus k_6 \oplus k_8 \oplus k_{11} \oplus k_{12} \oplus k_{15} \oplus k_{17} \oplus k_{18} \oplus k_{19} \oplus k_{26} \oplus k_{31} \oplus k_{38} \oplus k_{40} \oplus k_{42} \oplus k_{46} \oplus k_{48} \oplus k_{51} \oplus k_{52} \oplus k_{85} \oplus 1$
$40, 52, 125, 77, 116$	501	$k_0 \oplus k_5 \oplus k_7 \oplus k_9 \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{27} \oplus k_{32} \oplus k_{39} \oplus k_{41} \oplus k_{43} \oplus k_{47} \oplus k_{49} \oplus k_{52} \oplus k_{53} \oplus k_{86}$
$41, 53, 126, 78, 117$	502	$k_0 \oplus k_1 \oplus k_6 \oplus k_8 \oplus k_{10} \oplus k_{13} \oplus k_{14} \oplus k_{17} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{28} \oplus k_{33} \oplus k_{40} \oplus k_{42} \oplus k_{44} \oplus k_{48} \oplus k_{50} \oplus k_{53} \oplus k_{54} \oplus k_{87} \oplus 1$
$42, 54, 127, 79, 118$	503	$k_1 \oplus k_2 \oplus k_7 \oplus k_9 \oplus k_{11} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{29} \oplus k_{34} \oplus k_{41} \oplus k_{43} \oplus k_{45} \oplus k_{49} \oplus k_{51} \oplus k_{54} \oplus k_{55} \oplus k_{88} \oplus 1$
$19, 73, 11, 104, 102$	480	$k_1 \oplus k_{16} \oplus k_{18} \oplus k_{21} \oplus k_{22} \oplus k_{55}$
$20, 74, 12, 105, 103$	481	$k_2 \oplus k_{17} \oplus k_{19} \oplus k_{22} \oplus k_{23} \oplus k_{56}$
$21, 75, 13, 106, 104$	482	$k_3 \oplus k_{18} \oplus k_{20} \oplus k_{23} \oplus k_{24} \oplus k_{57}$
$22, 76, 14, 107, 105$	483	$k_4 \oplus k_{19} \oplus k_{21} \oplus k_{24} \oplus k_{25} \oplus k_{58}$
$23, 77, 15, 108, 106$	484	$k_0 \oplus k_5 \oplus k_{20} \oplus k_{22} \oplus k_{25} \oplus k_{26} \oplus k_{59} \oplus 1$
$24, 78, 16, 109, 107$	485	$k_1 \oplus k_6 \oplus k_{21} \oplus k_{23} \oplus k_{26} \oplus k_{27} \oplus k_{60} \oplus 1$
$25, 79, 17, 110, 108$	486	$k_2 \oplus k_7 \oplus k_{22} \oplus k_{24} \oplus k_{27} \oplus k_{28} \oplus k_{61} \oplus 1$
$26, 80, 18, 111, 109$	487	$k_3 \oplus k_8 \oplus k_{23} \oplus k_{25} \oplus k_{28} \oplus k_{29} \oplus k_{62} \oplus 1$
$27, 81, 19, 112, 110$	488	$k_4 \oplus k_9 \oplus k_{24} \oplus k_{26} \oplus k_{29} \oplus k_{30} \oplus k_{63} \oplus 1$
$28, 82, 20, 113, 111$	489	$k_5 \oplus k_{10} \oplus k_{25} \oplus k_{27} \oplus k_{30} \oplus k_{31} \oplus k_{64} \oplus 1$
$29, 83, 21, 114, 112$	490	$k_6 \oplus k_{11} \oplus k_{26} \oplus k_{28} \oplus k_{31} \oplus k_{32} \oplus k_{65} \oplus 1$
$30, 84, 22, 115, 113$	491	$k_0 \oplus k_7 \oplus k_{12} \oplus k_{27} \oplus k_{29} \oplus k_{32} \oplus k_{33} \oplus k_{66}$
$31, 85, 23, 116, 114$	492	$k_0 \oplus k_1 \oplus k_8 \oplus k_{13} \oplus k_{28} \oplus k_{30} \oplus k_{33} \oplus k_{34} \oplus k_{67} \oplus 1$
$32, 86, 24, 117, 115$	493	$k_0 \oplus k_1 \oplus k_2 \oplus k_9 \oplus k_{14} \oplus k_{29} \oplus k_{31} \oplus k_{34} \oplus k_{35} \oplus k_{68}$
$33, 87, 25, 118, 116$	494	$k_1 \oplus k_2 \oplus k_3 \oplus k_{10} \oplus k_{15} \oplus k_{30} \oplus k_{32} \oplus k_{35} \oplus k_{36} \oplus k_{69}$
$34, 88, 26, 119, 117$	495	$k_0 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_{11} \oplus k_{16} \oplus k_{31} \oplus k_{33} \oplus k_{36} \oplus k_{37} \oplus k_{70} \oplus 1$
$35, 89, 27, 120, 118$	496	$k_1 \oplus k_3 \oplus k_4 \oplus k_5 \oplus k_{12} \oplus k_{17} \oplus k_{32} \oplus k_{34} \oplus k_{37} \oplus k_{38} \oplus k_{71} \oplus 1$
$36, 90, 28, 121, 119$	497	$k_2 \oplus k_4 \oplus k_5 \oplus k_6 \oplus k_{13} \oplus k_{18} \oplus k_{33} \oplus k_{35} \oplus k_{38} \oplus k_{39} \oplus k_{72} \oplus 1$
$37, 91, 29, 122, 120$	498	$k_0 \oplus k_3 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_{14} \oplus k_{19} \oplus k_{34} \oplus k_{36} \oplus k_{39} \oplus k_{40} \oplus k_{73}$
$38, 92, 30, 123, 121$	499	$k_0 \oplus k_1 \oplus k_4 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_{15} \oplus k_{20} \oplus k_{35} \oplus k_{37} \oplus k_{40} \oplus k_{41} \oplus k_{74} \oplus 1$
$39, 93, 31, 124, 122$	500	$k_1 \oplus k_2 \oplus k_5 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{16} \oplus k_{21} \oplus k_{36} \oplus k_{38} \oplus k_{41} \oplus k_{42} \oplus k_{75} \oplus 1$
$40, 94, 32, 125, 123$	501	$k_0 \oplus k_2 \oplus k_3 \oplus k_6 \oplus k_8 \oplus k_9 \oplus k_{10} \oplus k_{17} \oplus k_{22} \oplus k_{37} \oplus k_{39} \oplus k_{42} \oplus k_{43} \oplus k_{76}$

$41, 95, 33, 126, 124$	502	$k_1 \oplus k_3 \oplus k_4 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{18} \oplus k_{23} \oplus k_{38} \oplus k_{40} \oplus k_{43} \oplus k_{44} \oplus k_{77}$
$42, 96, 34, 127, 125$	503	$k_0 \oplus k_2 \oplus k_4 \oplus k_5 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{19} \oplus k_{24} \oplus k_{39} \oplus k_{41} \oplus k_{44} \oplus k_{45} \oplus k_{78} \oplus 1$
$113, 44, 117, 0, 56$	493	$k_5 \oplus k_7 \oplus k_8 \oplus k_{11} \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{27} \oplus k_{32} \oplus k_{44} \oplus k_{47} \oplus k_{49} \oplus k_{52} \oplus k_{53} \oplus k_{86}$
$114, 45, 118, 1, 57$	494	$k_0 \oplus k_6 \oplus k_8 \oplus k_9 \oplus k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{17} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{28} \oplus k_{33} \oplus k_{45} \oplus k_{48} \oplus k_{50} \oplus k_{53} \oplus k_{54} \oplus k_{87} \oplus 1$
$115, 46, 119, 2, 58$	495	$k_0 \oplus k_1 \oplus k_7 \oplus k_9 \oplus k_{10} \oplus k_{13} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{29} \oplus k_{34} \oplus k_{46} \oplus k_{49} \oplus k_{51} \oplus k_{54} \oplus k_{55} \oplus k_{88}$
$116, 47, 120, 3, 59$	496	$k_1 \oplus k_2 \oplus k_8 \oplus k_{10} \oplus k_{11} \oplus k_{14} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{30} \oplus k_{35} \oplus k_{47} \oplus k_{50} \oplus k_{52} \oplus k_{55} \oplus k_{56} \oplus k_{89}$
$117, 48, 121, 4, 60$	497	$k_2 \oplus k_3 \oplus k_9 \oplus k_{11} \oplus k_{12} \oplus k_{15} \oplus k_{16} \oplus k_{17} \oplus k_{20} \oplus k_{22} \oplus k_{23} \oplus k_{24} \oplus k_{31} \oplus k_{36} \oplus k_{48} \oplus k_{51} \oplus k_{53} \oplus k_{56} \oplus k_{57} \oplus k_{90}$
$118, 49, 122, 5, 61$	498	$k_0 \oplus k_3 \oplus k_4 \oplus k_{10} \oplus k_{12} \oplus k_{13} \oplus k_{16} \oplus k_{17} \oplus k_{18} \oplus k_{21} \oplus k_{23} \oplus k_{24} \oplus k_{25} \oplus k_{32} \oplus k_{37} \oplus k_{49} \oplus k_{52} \oplus k_{54} \oplus k_{57} \oplus k_{58} \oplus k_{91} \oplus 1$
$119, 50, 123, 6, 62$	499	$k_0 \oplus k_1 \oplus k_4 \oplus k_5 \oplus k_{11} \oplus k_{13} \oplus k_{14} \oplus k_{17} \oplus k_{18} \oplus k_{19} \oplus k_{22} \oplus k_{24} \oplus k_{25} \oplus k_{26} \oplus k_{33} \oplus k_{38} \oplus k_{50} \oplus k_{53} \oplus k_{55} \oplus k_{58} \oplus k_{59} \oplus k_{92}$
$120, 51, 124, 7, 63$	500	$k_1 \oplus k_2 \oplus k_5 \oplus k_6 \oplus k_{12} \oplus k_{14} \oplus k_{15} \oplus k_{18} \oplus k_{19} \oplus k_{20} \oplus k_{23} \oplus k_{25} \oplus k_{26} \oplus k_{27} \oplus k_{34} \oplus k_{39} \oplus k_{51} \oplus k_{54} \oplus k_{56} \oplus k_{59} \oplus k_{60} \oplus k_{93}$
$121, 52, 125, 8, 64$	501	$k_2 \oplus k_3 \oplus k_6 \oplus k_7 \oplus k_{13} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{20} \oplus k_{21} \oplus k_{24} \oplus k_{26} \oplus k_{27} \oplus k_{28} \oplus k_{35} \oplus k_{40} \oplus k_{52} \oplus k_{55} \oplus k_{57} \oplus k_{60} \oplus k_{61} \oplus k_{94}$
$122, 53, 126, 9, 65$	502	$k_3 \oplus k_4 \oplus k_7 \oplus k_8 \oplus k_{14} \oplus k_{16} \oplus k_{17} \oplus k_{20} \oplus k_{21} \oplus k_{22} \oplus k_{25} \oplus k_{27} \oplus k_{28} \oplus k_{29} \oplus k_{36} \oplus k_{41} \oplus k_{53} \oplus k_{56} \oplus k_{58} \oplus k_{61} \oplus k_{62} \oplus k_{95}$
$123, 54, 127, 10, 66$	503	$k_4 \oplus k_5 \oplus k_8 \oplus k_9 \oplus k_{15} \oplus k_{17} \oplus k_{18} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{26} \oplus k_{28} \oplus k_{29} \oplus k_{30} \oplus k_{37} \oplus k_{42} \oplus k_{54} \oplus k_{57} \oplus k_{59} \oplus k_{62} \oplus k_{63} \oplus k_{96}$
$42, 125, 21, 127, 118$	503	$k_2 \oplus k_3 \oplus k_8 \oplus k_{10} \oplus k_{12} \oplus k_{15} \oplus k_{16} \oplus k_{19} \oplus k_{21} \oplus k_{22} \oplus k_{23} \oplus k_{30} \oplus k_{35} \oplus k_{42} \oplus k_{44} \oplus k_{46} \oplus k_{50} \oplus k_{52} \oplus k_{55} \oplus k_{56} \oplus k_{89} \oplus 1$

A.2 Cube Attack on ACORN Encryption Phase

Table A.2: Linear Equations obtained for ACORN by Choosing the Cube Set from the Plaintext

Cube Indices	Output Index	Superpoly
0	58	$s_{73} \oplus s_{78} \oplus s_{119} \oplus s_{214} \oplus s_{217} \oplus s_{251}$
1	59	$s_{74} \oplus s_{79} \oplus s_{120} \oplus s_{215} \oplus s_{218} \oplus s_{252}$
2	60	$s_{75} \oplus s_{80} \oplus s_{121} \oplus s_{216} \oplus s_{219} \oplus s_{253}$
3	61	$s_{76} \oplus s_{81} \oplus s_{122} \oplus s_{217} \oplus s_{220} \oplus s_{254}$
4	62	$s_{77} \oplus s_{82} \oplus s_{123} \oplus s_{218} \oplus s_{221} \oplus s_{255}$

5	63	$s_{78} \oplus s_{83} \oplus s_{124} \oplus s_{219} \oplus s_{222} \oplus s_{256}$
6	64	$s_{79} \oplus s_{84} \oplus s_{125} \oplus s_{220} \oplus s_{223} \oplus s_{257}$
7	65	$s_{80} \oplus s_{85} \oplus s_{126} \oplus s_{221} \oplus s_{224} \oplus s_{258}$
8	66	$s_{81} \oplus s_{86} \oplus s_{127} \oplus s_{222} \oplus s_{225} \oplus s_{259}$
9	67	$s_{82} \oplus s_{87} \oplus s_{128} \oplus s_{223} \oplus s_{226} \oplus s_{260}$
10	68	$s_{83} \oplus s_{88} \oplus s_{129} \oplus s_{224} \oplus s_{227} \oplus s_{261}$
11	69	$s_{84} \oplus s_{89} \oplus s_{130} \oplus s_{225} \oplus s_{228} \oplus s_{262}$
12	70	$s_{85} \oplus s_{90} \oplus s_{131} \oplus s_{226} \oplus s_{229} \oplus s_{263}$
13	71	$s_{86} \oplus s_{91} \oplus s_{132} \oplus s_{193} \oplus s_{196} \oplus s_{227} \oplus s_{230} \oplus s_{264}$
14	72	$s_{87} \oplus s_{92} \oplus s_{133} \oplus s_{194} \oplus s_{197} \oplus s_{228} \oplus s_{231} \oplus s_{265}$
15	73	$s_{88} \oplus s_{93} \oplus s_{134} \oplus s_{195} \oplus s_{198} \oplus s_{229} \oplus s_{232} \oplus s_{266}$
16	74	$s_{89} \oplus s_{94} \oplus s_{135} \oplus s_{193} \oplus s_{199} \oplus s_{230} \oplus s_{233} \oplus s_{267}$
17	75	$s_{90} \oplus s_{95} \oplus s_{136} \oplus s_{194} \oplus s_{200} \oplus s_{231} \oplus s_{234} \oplus s_{268}$
18	76	$s_{91} \oplus s_{96} \oplus s_{137} \oplus s_{195} \oplus s_{201} \oplus s_{232} \oplus s_{235} \oplus s_{269}$
19	77	$s_{92} \oplus s_{97} \oplus s_{138} \oplus s_{196} \oplus s_{202} \oplus s_{233} \oplus s_{236} \oplus s_{270}$
20	78	$s_{93} \oplus s_{98} \oplus s_{139} \oplus s_{197} \oplus s_{203} \oplus s_{234} \oplus s_{237} \oplus s_{271}$
21	79	$s_{94} \oplus s_{99} \oplus s_{140} \oplus s_{198} \oplus s_{204} \oplus s_{235} \oplus s_{238} \oplus s_{272}$
22	80	$s_{95} \oplus s_{100} \oplus s_{141} \oplus s_{199} \oplus s_{205} \oplus s_{236} \oplus s_{239} \oplus s_{273}$
23	81	$s_{96} \oplus s_{101} \oplus s_{142} \oplus s_{200} \oplus s_{206} \oplus s_{237} \oplus s_{240} \oplus s_{274}$
24	82	$s_{97} \oplus s_{102} \oplus s_{143} \oplus s_{201} \oplus s_{207} \oplus s_{238} \oplus s_{241} \oplus s_{275}$
25	83	$s_{98} \oplus s_{103} \oplus s_{144} \oplus s_{202} \oplus s_{208} \oplus s_{239} \oplus s_{242} \oplus s_{276}$
26	84	$s_{99} \oplus s_{104} \oplus s_{145} \oplus s_{203} \oplus s_{209} \oplus s_{240} \oplus s_{243} \oplus s_{277}$
27	85	$s_{100} \oplus s_{105} \oplus s_{146} \oplus s_{204} \oplus s_{210} \oplus s_{241} \oplus s_{244} \oplus s_{278}$
28	86	$s_{101} \oplus s_{106} \oplus s_{147} \oplus s_{205} \oplus s_{211} \oplus s_{242} \oplus s_{245} \oplus s_{279}$
29	87	$s_{61} \oplus s_{66} \oplus s_{102} \oplus s_{107} \oplus s_{148} \oplus s_{206} \oplus s_{212} \oplus s_{243} \oplus s_{246} \oplus s_{280}$
30	88	$s_{62} \oplus s_{67} \oplus s_{103} \oplus s_{108} \oplus s_{149} \oplus s_{207} \oplus s_{213} \oplus s_{244} \oplus s_{247} \oplus s_{281}$
31	89	$s_{63} \oplus s_{68} \oplus s_{104} \oplus s_{109} \oplus s_{150} \oplus s_{208} \oplus s_{214} \oplus s_{245} \oplus s_{248} \oplus s_{282}$
32	90	$s_{64} \oplus s_{69} \oplus s_{105} \oplus s_{110} \oplus s_{151} \oplus s_{209} \oplus s_{215} \oplus s_{246} \oplus s_{249} \oplus s_{283}$
33	91	$s_{65} \oplus s_{70} \oplus s_{106} \oplus s_{111} \oplus s_{152} \oplus s_{210} \oplus s_{216} \oplus s_{247} \oplus s_{250} \oplus s_{284}$
34	92	$s_{61} \oplus s_{71} \oplus s_{107} \oplus s_{112} \oplus s_{153} \oplus s_{211} \oplus s_{217} \oplus s_{248} \oplus s_{251} \oplus s_{285}$
35	93	$s_{62} \oplus s_{72} \oplus s_{107} \oplus s_{108} \oplus s_{111} \oplus s_{113} \oplus s_{154} \oplus s_{212} \oplus s_{218} \oplus s_{249} \oplus s_{252} \oplus s_{286}$
36	94	$s_{63} \oplus s_{73} \oplus s_{108} \oplus s_{109} \oplus s_{112} \oplus s_{114} \oplus s_{155} \oplus s_{213} \oplus s_{219} \oplus s_{250} \oplus s_{253} \oplus s_{287}$
37	95	$s_{64} \oplus s_{74} \oplus s_{109} \oplus s_{110} \oplus s_{113} \oplus s_{115} \oplus s_{156} \oplus s_{214} \oplus s_{220} \oplus s_{251} \oplus s_{254} \oplus s_{288}$
38	96	$s_{65} \oplus s_{75} \oplus s_{110} \oplus s_{111} \oplus s_{114} \oplus s_{116} \oplus s_{157} \oplus s_{215} \oplus s_{221} \oplus s_{230} \oplus s_{235} \oplus s_{252} \oplus s_{255} \oplus s_{289}$
39	97	$s_{66} \oplus s_{76} \oplus s_{111} \oplus s_{112} \oplus s_{115} \oplus s_{117} \oplus s_{158} \oplus s_{216} \oplus s_{222} \oplus s_{231} \oplus s_{236} \oplus s_{253} \oplus s_{256} \oplus s_{290}$
40	98	$s_{67} \oplus s_{77} \oplus s_{112} \oplus s_{113} \oplus s_{116} \oplus s_{118} \oplus s_{159} \oplus s_{217} \oplus s_{223} \oplus s_{232} \oplus s_{237} \oplus s_{254} \oplus s_{257} \oplus s_{291}$
41	99	$s_{68} \oplus s_{78} \oplus s_{113} \oplus s_{114} \oplus s_{117} \oplus s_{119} \oplus s_{160} \oplus s_{218} \oplus s_{224} \oplus s_{233} \oplus s_{238} \oplus s_{255} \oplus s_{258} \oplus s_{292}$

$0, 7$	107	$s_{11} \oplus s_{34} \oplus s_{72} \oplus s_{170} \oplus s_{176} \oplus s_{209}$
$1, 8$	108	$s_{12} \oplus s_{35} \oplus s_{73} \oplus s_{171} \oplus s_{177} \oplus s_{210}$
$2, 9$	109	$s_{13} \oplus s_{36} \oplus s_{74} \oplus s_{172} \oplus s_{178} \oplus s_{211}$
$3, 10$	110	$s_{14} \oplus s_{37} \oplus s_{75} \oplus s_{173} \oplus s_{179} \oplus s_{212}$
$4, 11$	111	$s_{15} \oplus s_{38} \oplus s_{76} \oplus s_{174} \oplus s_{180} \oplus s_{213}$
$5, 12$	112	$s_{16} \oplus s_{39} \oplus s_{77} \oplus s_{175} \oplus s_{181} \oplus s_{214}$
$6, 13$	113	$s_{17} \oplus s_{40} \oplus s_{78} \oplus s_{176} \oplus s_{182} \oplus s_{215}$
$7, 14$	114	$s_{18} \oplus s_{41} \oplus s_{79} \oplus s_{177} \oplus s_{183} \oplus s_{216}$
$8, 15$	115	$s_{19} \oplus s_{42} \oplus s_{80} \oplus s_{178} \oplus s_{184} \oplus s_{217}$
$9, 16$	116	$s_{20} \oplus s_{43} \oplus s_{81} \oplus s_{179} \oplus s_{185} \oplus s_{218}$
$10, 17$	117	$s_{21} \oplus s_{44} \oplus s_{82} \oplus s_{180} \oplus s_{186} \oplus s_{219}$
$11, 18$	118	$s_{22} \oplus s_{45} \oplus s_{83} \oplus s_{181} \oplus s_{187} \oplus s_{220}$
$12, 19$	119	$s_{23} \oplus s_{46} \oplus s_{84} \oplus s_{182} \oplus s_{188} \oplus s_{221}$
$13, 20$	120	$s_{24} \oplus s_{47} \oplus s_{85} \oplus s_{183} \oplus s_{189} \oplus s_{222}$
$14, 21$	121	$s_{25} \oplus s_{48} \oplus s_{86} \oplus s_{184} \oplus s_{190} \oplus s_{223}$
$15, 22$	122	$s_{26} \oplus s_{49} \oplus s_{87} \oplus s_{185} \oplus s_{191} \oplus s_{224}$
$16, 23$	123	$s_{27} \oplus s_{50} \oplus s_{88} \oplus s_{186} \oplus s_{192} \oplus s_{225}$
$17, 24$	124	$s_{28} \oplus s_{51} \oplus s_{89} \oplus s_{154} \oplus s_{160} \oplus s_{187} \oplus s_{193} \oplus s_{226}$
$18, 25$	125	$s_{29} \oplus s_{52} \oplus s_{90} \oplus s_{155} \oplus s_{161} \oplus s_{188} \oplus s_{194} \oplus s_{227}$
$19, 26$	126	$s_{30} \oplus s_{53} \oplus s_{91} \oplus s_{156} \oplus s_{162} \oplus s_{189} \oplus s_{195} \oplus s_{228}$
$20, 27$	127	$s_{31} \oplus s_{54} \oplus s_{92} \oplus s_{157} \oplus s_{163} \oplus s_{190} \oplus s_{196} \oplus s_{229}$
$21, 28$	128	$s_{32} \oplus s_{55} \oplus s_{93} \oplus s_{158} \oplus s_{164} \oplus s_{191} \oplus s_{193} \oplus s_{196} \oplus s_{197} \oplus s_{230}$
$22, 29$	129	$s_{33} \oplus s_{56} \oplus s_{94} \oplus s_{159} \oplus s_{165} \oplus s_{192} \oplus s_{194} \oplus s_{197} \oplus s_{198} \oplus s_{231}$
$23, 30$	130	$s_{34} \oplus s_{57} \oplus s_{95} \oplus s_{154} \oplus s_{166} \oplus s_{193} \oplus s_{195} \oplus s_{198} \oplus s_{199} \oplus s_{232}$
$24, 31$	131	$s_{35} \oplus s_{58} \oplus s_{96} \oplus s_{155} \oplus s_{167} \oplus s_{194} \oplus s_{196} \oplus s_{199} \oplus s_{200} \oplus s_{233}$
$25, 32$	132	$s_{36} \oplus s_{59} \oplus s_{97} \oplus s_{156} \oplus s_{168} \oplus s_{195} \oplus s_{197} \oplus s_{200} \oplus s_{201} \oplus s_{234}$
$26, 33$	133	$s_{37} \oplus s_{60} \oplus s_{98} \oplus s_{157} \oplus s_{169} \oplus s_{196} \oplus s_{198} \oplus s_{201} \oplus s_{202} \oplus s_{235}$
$27, 34$	134	$s_{0} \oplus s_{23} \oplus s_{38} \oplus s_{61} \oplus s_{99} \oplus s_{158} \oplus s_{170} \oplus s_{197} \oplus s_{199} \oplus s_{202} \oplus s_{203} \oplus s_{236}$
$28, 35$	135	$s_{1} \oplus s_{24} \oplus s_{39} \oplus s_{62} \oplus s_{100} \oplus s_{159} \oplus s_{171} \oplus s_{198} \oplus s_{200} \oplus s_{203} \oplus s_{204} \oplus s_{237}$
$29, 36$	136	$s_{2} \oplus s_{25} \oplus s_{40} \oplus s_{63} \oplus s_{101} \oplus s_{160} \oplus s_{172} \oplus s_{199} \oplus s_{201} \oplus s_{204} \oplus s_{205} \oplus s_{238}$
$30, 37$	137	$s_{3} \oplus s_{26} \oplus s_{41} \oplus s_{64} \oplus s_{102} \oplus s_{161} \oplus s_{173} \oplus s_{200} \oplus s_{202} \oplus s_{205} \oplus s_{206} \oplus s_{239}$
$31, 38$	138	$s_{4} \oplus s_{27} \oplus s_{42} \oplus s_{65} \oplus s_{103} \oplus s_{162} \oplus s_{174} \oplus s_{201} \oplus s_{203} \oplus s_{206} \oplus s_{207} \oplus s_{240}$
$32, 39$	139	$s_{5} \oplus s_{28} \oplus s_{43} \oplus s_{66} \oplus s_{104} \oplus s_{163} \oplus s_{175} \oplus s_{202} \oplus s_{204} \oplus s_{207} \oplus s_{208} \oplus s_{241}$
$33, 40$	140	$s_{6} \oplus s_{29} \oplus s_{44} \oplus s_{67} \oplus s_{105} \oplus s_{164} \oplus s_{176} \oplus s_{203} \oplus s_{205} \oplus s_{208} \oplus s_{209} \oplus s_{242}$
$34, 41$	141	$s_{7} \oplus s_{30} \oplus s_{45} \oplus s_{68} \oplus s_{106} \oplus s_{165} \oplus s_{177} \oplus s_{204} \oplus s_{206} \oplus s_{209} \oplus s_{210} \oplus s_{243}$
$35, 42$	142	$s_{8} \oplus s_{31} \oplus s_{46} \oplus s_{61} \oplus s_{66} \oplus s_{69} \oplus s_{107} \oplus s_{166} \oplus s_{178} \oplus s_{205} \oplus s_{207} \oplus s_{210} \oplus s_{211} \oplus s_{244}$
$36, 43$	143	$s_{9} \oplus s_{32} \oplus s_{47} \oplus s_{62} \oplus s_{67} \oplus s_{70} \oplus s_{108} \oplus s_{167} \oplus s_{179} \oplus s_{206} \oplus s_{208} \oplus s_{211} \oplus s_{212} \oplus s_{245}$
$37, 44$	144	$s_{10} \oplus s_{33} \oplus s_{48} \oplus s_{63} \oplus s_{68} \oplus s_{71} \oplus s_{109} \oplus s_{168} \oplus s_{180} \oplus s_{207} \oplus s_{209} \oplus s_{212} \oplus s_{213} \oplus s_{246}$
$38, 45$	145	$s_{11} \oplus s_{34} \oplus s_{49} \oplus s_{64} \oplus s_{69} \oplus s_{72} \oplus s_{110} \oplus s_{169} \oplus s_{181} \oplus s_{208} \oplus s_{210} \oplus s_{213} \oplus s_{214} \oplus s_{247}$

$39, 46$	146	$s_{12} \oplus s_{35} \oplus s_{50} \oplus s_{65} \oplus s_{70} \oplus s_{73} \oplus s_{111} \oplus s_{170} \oplus s_{182} \oplus s_{209} \oplus s_{211} \oplus s_{214} \oplus s_{215} \oplus s_{248}$
$40, 47$	147	$s_{13} \oplus s_{36} \oplus s_{51} \oplus s_{66} \oplus s_{71} \oplus s_{74} \oplus s_{112} \oplus s_{171} \oplus s_{183} \oplus s_{210} \oplus s_{212} \oplus s_{215} \oplus s_{216} \oplus s_{249}$
$41, 48$	148	$s_{14} \oplus s_{37} \oplus s_{52} \oplus s_{67} \oplus s_{72} \oplus s_{75} \oplus s_{113} \oplus s_{172} \oplus s_{184} \oplus s_{211} \oplus s_{213} \oplus s_{216} \oplus s_{217} \oplus s_{250}$
$42, 49$	149	$s_{15} \oplus s_{38} \oplus s_{53} \oplus s_{68} \oplus s_{73} \oplus s_{76} \oplus s_{114} \oplus s_{173} \oplus s_{185} \oplus s_{212} \oplus s_{214} \oplus s_{217} \oplus s_{218} \oplus s_{251}$
$43, 50$	150	$s_{16} \oplus s_{39} \oplus s_{54} \oplus s_{69} \oplus s_{74} \oplus s_{77} \oplus s_{115} \oplus s_{174} \oplus s_{186} \oplus s_{213} \oplus s_{215} \oplus s_{218} \oplus s_{219} \oplus s_{252}$
$44, 51$	151	$s_{17} \oplus s_{40} \oplus s_{55} \oplus s_{70} \oplus s_{75} \oplus s_{78} \oplus s_{116} \oplus s_{175} \oplus s_{187} \oplus s_{214} \oplus s_{216} \oplus s_{219} \oplus s_{220} \oplus s_{253}$
$45, 52$	152	$s_{18} \oplus s_{41} \oplus s_{56} \oplus s_{71} \oplus s_{76} \oplus s_{79} \oplus s_{117} \oplus s_{176} \oplus s_{188} \oplus s_{215} \oplus s_{217} \oplus s_{220} \oplus s_{221} \oplus s_{254}$
$46, 53$	153	$s_{19} \oplus s_{42} \oplus s_{57} \oplus s_{72} \oplus s_{77} \oplus s_{80} \oplus s_{118} \oplus s_{177} \oplus s_{189} \oplus s_{216} \oplus s_{218} \oplus s_{221} \oplus s_{222} \oplus s_{255}$
$47, 54$	154	$s_{20} \oplus s_{43} \oplus s_{58} \oplus s_{73} \oplus s_{78} \oplus s_{81} \oplus s_{119} \oplus s_{178} \oplus s_{190} \oplus s_{217} \oplus s_{219} \oplus s_{222} \oplus s_{223} \oplus s_{256}$
$48, 55$	155	$s_{21} \oplus s_{44} \oplus s_{59} \oplus s_{74} \oplus s_{79} \oplus s_{82} \oplus s_{120} \oplus s_{179} \oplus s_{191} \oplus s_{218} \oplus s_{220} \oplus s_{223} \oplus s_{224} \oplus s_{257}$
$49, 56$	156	$s_{22} \oplus s_{45} \oplus s_{60} \oplus s_{75} \oplus s_{80} \oplus s_{83} \oplus s_{121} \oplus s_{180} \oplus s_{192} \oplus s_{219} \oplus s_{221} \oplus s_{224} \oplus s_{225} \oplus s_{258}$
$50, 57$	157	$s_{0} \oplus s_{46} \oplus s_{61} \oplus s_{76} \oplus s_{81} \oplus s_{84} \oplus s_{122} \oplus s_{154} \oplus s_{160} \oplus s_{181} \oplus s_{193} \oplus s_{220} \oplus s_{222} \oplus s_{225} \oplus s_{226} \oplus s_{259}$
$51, 58$	158	$s_{1} \oplus s_{47} \oplus s_{62} \oplus s_{77} \oplus s_{82} \oplus s_{85} \oplus s_{123} \oplus s_{155} \oplus s_{161} \oplus s_{182} \oplus s_{194} \oplus s_{221} \oplus s_{223} \oplus s_{226} \oplus s_{227} \oplus s_{260}$
$52, 59$	159	$s_{2} \oplus s_{48} \oplus s_{63} \oplus s_{78} \oplus s_{83} \oplus s_{86} \oplus s_{124} \oplus s_{156} \oplus s_{162} \oplus s_{183} \oplus s_{195} \oplus s_{222} \oplus s_{224} \oplus s_{227} \oplus s_{228} \oplus s_{261}$
$53, 60$	160	$s_{3} \oplus s_{49} \oplus s_{64} \oplus s_{79} \oplus s_{84} \oplus s_{87} \oplus s_{125} \oplus s_{157} \oplus s_{163} \oplus s_{184} \oplus s_{196} \oplus s_{223} \oplus s_{225} \oplus s_{228} \oplus s_{229} \oplus s_{262}$
$54, 61$	161	$s_{4} \oplus s_{50} \oplus s_{65} \oplus s_{80} \oplus s_{85} \oplus s_{88} \oplus s_{126} \oplus s_{158} \oplus s_{164} \oplus s_{185} \oplus s_{193} \oplus s_{196} \oplus s_{197} \oplus s_{224} \oplus s_{226} \oplus s_{229} \oplus s_{230} \oplus s_{263}$
$55, 62$	162	$s_{5} \oplus s_{51} \oplus s_{66} \oplus s_{81} \oplus s_{86} \oplus s_{89} \oplus s_{127} \oplus s_{159} \oplus s_{165} \oplus s_{186} \oplus s_{193} \oplus s_{194} \oplus s_{196} \oplus s_{197} \oplus s_{198} \oplus s_{225} \oplus s_{227} \oplus s_{230} \oplus s_{231} \oplus s_{264}$
$56, 63$	163	$s_{6} \oplus s_{52} \oplus s_{67} \oplus s_{82} \oplus s_{87} \oplus s_{90} \oplus s_{128} \oplus s_{160} \oplus s_{166} \oplus s_{187} \oplus s_{194} \oplus s_{195} \oplus s_{197} \oplus s_{198} \oplus s_{199} \oplus s_{226} \oplus s_{228} \oplus s_{231} \oplus s_{232} \oplus s_{265}$
$57, 64$	164	$s_{7} \oplus s_{53} \oplus s_{68} \oplus s_{83} \oplus s_{88} \oplus s_{91} \oplus s_{129} \oplus s_{161} \oplus s_{167} \oplus s_{188} \oplus s_{195} \oplus s_{196} \oplus s_{198} \oplus s_{199} \oplus s_{200} \oplus s_{227} \oplus s_{229} \oplus s_{232} \oplus s_{233} \oplus s_{266}$
$58, 65$	165	$s_{8} \oplus s_{54} \oplus s_{69} \oplus s_{84} \oplus s_{89} \oplus s_{92} \oplus s_{130} \oplus s_{162} \oplus s_{168} \oplus s_{189} \oplus s_{193} \oplus s_{197} \oplus s_{199} \oplus s_{200} \oplus s_{201} \oplus s_{228} \oplus s_{230} \oplus s_{233} \oplus s_{234} \oplus s_{267}$

$59, 66$	166	$s_9 \oplus s_{55} \oplus s_{70} \oplus s_{85} \oplus s_{90} \oplus s_{93} \oplus s_{131} \oplus s_{163} \oplus s_{169} \oplus s_{190} \oplus s_{194} \oplus s_{198} \oplus s_{200} \oplus s_{201} \oplus s_{202} \oplus s_{229} \oplus s_{231} \oplus s_{234} \oplus s_{235} \oplus s_{268}$
$60, 67$	167	$s_{10} \oplus s_{56} \oplus s_{71} \oplus s_{86} \oplus s_{91} \oplus s_{94} \oplus s_{132} \oplus s_{164} \oplus s_{170} \oplus s_{191} \oplus s_{193} \oplus s_{195} \oplus s_{196} \oplus s_{199} \oplus s_{201} \oplus s_{202} \oplus s_{203} \oplus s_{230} \oplus s_{232} \oplus s_{235} \oplus s_{236} \oplus s_{269}$
$61, 68$	168	$s_{11} \oplus s_{57} \oplus s_{72} \oplus s_{87} \oplus s_{92} \oplus s_{95} \oplus s_{133} \oplus s_{165} \oplus s_{171} \oplus s_{192} \oplus s_{194} \oplus s_{196} \oplus s_{197} \oplus s_{200} \oplus s_{202} \oplus s_{203} \oplus s_{204} \oplus s_{231} \oplus s_{233} \oplus s_{236} \oplus s_{237} \oplus s_{270}$
$62, 69$	169	$s_{12} \oplus s_{58} \oplus s_{73} \oplus s_{88} \oplus s_{93} \oplus s_{96} \oplus s_{134} \oplus s_{154} \oplus s_{160} \oplus s_{166} \oplus s_{172} \oplus s_{193} \oplus s_{195} \oplus s_{197} \oplus s_{198} \oplus s_{201} \oplus s_{203} \oplus s_{204} \oplus s_{205} \oplus s_{232} \oplus s_{234} \oplus s_{237} \oplus s_{238} \oplus s_{271}$
$63, 70$	170	$s_{13} \oplus s_{59} \oplus s_{74} \oplus s_{89} \oplus s_{94} \oplus s_{97} \oplus s_{135} \oplus s_{155} \oplus s_{161} \oplus s_{167} \oplus s_{173} \oplus s_{194} \oplus s_{196} \oplus s_{198} \oplus s_{199} \oplus s_{202} \oplus s_{204} \oplus s_{205} \oplus s_{206} \oplus s_{233} \oplus s_{235} \oplus s_{238} \oplus s_{239} \oplus s_{272}$
$64, 71$	171	$s_{14} \oplus s_{60} \oplus s_{75} \oplus s_{90} \oplus s_{95} \oplus s_{98} \oplus s_{136} \oplus s_{156} \oplus s_{162} \oplus s_{168} \oplus s_{174} \oplus s_{195} \oplus s_{197} \oplus s_{199} \oplus s_{200} \oplus s_{203} \oplus s_{205} \oplus s_{206} \oplus s_{207} \oplus s_{234} \oplus s_{236} \oplus s_{239} \oplus s_{240} \oplus s_{273}$
$65, 72$	172	$s_0 \oplus s_{15} \oplus s_{23} \oplus s_{61} \oplus s_{76} \oplus s_{91} \oplus s_{96} \oplus s_{99} \oplus s_{137} \oplus s_{157} \oplus s_{163} \oplus s_{169} \oplus s_{175} \oplus s_{196} \oplus s_{198} \oplus s_{200} \oplus s_{201} \oplus s_{204} \oplus s_{206} \oplus s_{207} \oplus s_{208} \oplus s_{235} \oplus s_{237} \oplus s_{240} \oplus s_{241} \oplus s_{274}$
$66, 73$	173	$s_1 \oplus s_{16} \oplus s_{24} \oplus s_{62} \oplus s_{77} \oplus s_{92} \oplus s_{97} \oplus s_{100} \oplus s_{138} \oplus s_{158} \oplus s_{164} \oplus s_{170} \oplus s_{176} \oplus s_{197} \oplus s_{199} \oplus s_{201} \oplus s_{202} \oplus s_{205} \oplus s_{207} \oplus s_{208} \oplus s_{209} \oplus s_{236} \oplus s_{238} \oplus s_{241} \oplus s_{242} \oplus s_{275}$
$67, 74$	174	$s_2 \oplus s_{17} \oplus s_{25} \oplus s_{63} \oplus s_{78} \oplus s_{93} \oplus s_{98} \oplus s_{101} \oplus s_{139} \oplus s_{159} \oplus s_{165} \oplus s_{171} \oplus s_{177} \oplus s_{198} \oplus s_{200} \oplus s_{202} \oplus s_{203} \oplus s_{206} \oplus s_{208} \oplus s_{209} \oplus s_{210} \oplus s_{237} \oplus s_{239} \oplus s_{242} \oplus s_{243} \oplus s_{276}$
$68, 75$	175	$s_3 \oplus s_{18} \oplus s_{26} \oplus s_{64} \oplus s_{79} \oplus s_{94} \oplus s_{99} \oplus s_{102} \oplus s_{140} \oplus s_{160} \oplus s_{166} \oplus s_{172} \oplus s_{178} \oplus s_{199} \oplus s_{201} \oplus s_{203} \oplus s_{204} \oplus s_{207} \oplus s_{209} \oplus s_{210} \oplus s_{211} \oplus s_{238} \oplus s_{240} \oplus s_{243} \oplus s_{244} \oplus s_{277}$
$69, 76$	176	$s_4 \oplus s_{19} \oplus s_{27} \oplus s_{65} \oplus s_{80} \oplus s_{95} \oplus s_{100} \oplus s_{103} \oplus s_{141} \oplus s_{161} \oplus s_{167} \oplus s_{173} \oplus s_{179} \oplus s_{200} \oplus s_{202} \oplus s_{204} \oplus s_{205} \oplus s_{208} \oplus s_{210} \oplus s_{211} \oplus s_{212} \oplus s_{239} \oplus s_{241} \oplus s_{244} \oplus s_{245} \oplus s_{278}$
$70, 77$	177	$s_5 \oplus s_{20} \oplus s_{28} \oplus s_{66} \oplus s_{81} \oplus s_{96} \oplus s_{101} \oplus s_{104} \oplus s_{142} \oplus s_{162} \oplus s_{168} \oplus s_{174} \oplus s_{180} \oplus s_{201} \oplus s_{203} \oplus s_{205} \oplus s_{206} \oplus s_{209} \oplus s_{211} \oplus s_{212} \oplus s_{213} \oplus s_{240} \oplus s_{242} \oplus s_{245} \oplus s_{246} \oplus s_{279}$
$71, 78$	178	$s_6 \oplus s_{21} \oplus s_{29} \oplus s_{67} \oplus s_{82} \oplus s_{97} \oplus s_{102} \oplus s_{105} \oplus s_{143} \oplus s_{163} \oplus s_{169} \oplus s_{175} \oplus s_{181} \oplus s_{202} \oplus s_{204} \oplus s_{206} \oplus s_{207} \oplus s_{210} \oplus s_{212} \oplus s_{213} \oplus s_{214} \oplus s_{241} \oplus s_{243} \oplus s_{246} \oplus s_{247} \oplus s_{280}$
$72, 79$	179	$s_7 \oplus s_{22} \oplus s_{30} \oplus s_{68} \oplus s_{83} \oplus s_{98} \oplus s_{103} \oplus s_{106} \oplus s_{144} \oplus s_{164} \oplus s_{170} \oplus s_{176} \oplus s_{182} \oplus s_{203} \oplus s_{205} \oplus s_{207} \oplus s_{208} \oplus s_{211} \oplus s_{213} \oplus s_{214} \oplus s_{215} \oplus s_{242} \oplus s_{244} \oplus s_{247} \oplus s_{248} \oplus s_{281}$
$73, 80$	180	$s_8 \oplus s_{23} \oplus s_{31} \oplus s_{61} \oplus s_{66} \oplus s_{69} \oplus s_{84} \oplus s_{99} \oplus s_{104} \oplus s_{107} \oplus s_{145} \oplus s_{165} \oplus s_{171} \oplus s_{177} \oplus s_{183} \oplus s_{204} \oplus s_{206} \oplus s_{208} \oplus s_{209} \oplus s_{212} \oplus s_{214} \oplus s_{215} \oplus s_{216} \oplus s_{243} \oplus s_{245} \oplus s_{248} \oplus s_{249} \oplus s_{282}$
$74, 81$	181	$s_9 \oplus s_{24} \oplus s_{32} \oplus s_{62} \oplus s_{67} \oplus s_{70} \oplus s_{85} \oplus s_{100} \oplus s_{105} \oplus s_{108} \oplus s_{146} \oplus s_{166} \oplus s_{172} \oplus s_{178} \oplus s_{184} \oplus s_{205} \oplus s_{207} \oplus s_{209} \oplus s_{210} \oplus s_{213} \oplus s_{215} \oplus s_{216} \oplus s_{217} \oplus s_{244} \oplus s_{246} \oplus s_{249} \oplus s_{250} \oplus s_{283}$

$75, 82$	182	$s_{10} \oplus s_{25} \oplus s_{33} \oplus s_{63} \oplus s_{68} \oplus s_{71} \oplus s_{86} \oplus s_{101} \oplus s_{106} \oplus s_{109} \oplus s_{147} \oplus s_{167} \oplus s_{173} \oplus s_{179} \oplus s_{185} \oplus s_{206} \oplus s_{208} \oplus s_{210} \oplus s_{211} \oplus s_{214} \oplus s_{216} \oplus s_{217} \oplus s_{218} \oplus s_{245} \oplus s_{247} \oplus s_{250} \oplus s_{251} \oplus s_{284}$
$76, 83$	183	$s_{11} \oplus s_{26} \oplus s_{34} \oplus s_{61} \oplus s_{64} \oplus s_{66} \oplus s_{69} \oplus s_{72} \oplus s_{87} \oplus s_{102} \oplus s_{107} \oplus s_{110} \oplus s_{148} \oplus s_{168} \oplus s_{174} \oplus s_{180} \oplus s_{186} \oplus s_{207} \oplus s_{209} \oplus s_{211} \oplus s_{212} \oplus s_{215} \oplus s_{217} \oplus s_{218} \oplus s_{219} \oplus s_{246} \oplus s_{248} \oplus s_{251} \oplus s_{252} \oplus s_{285}$
$77, 84$	184	$s_{12} \oplus s_{27} \oplus s_{35} \oplus s_{62} \oplus s_{65} \oplus s_{67} \oplus s_{70} \oplus s_{73} \oplus s_{88} \oplus s_{103} \oplus s_{108} \oplus s_{111} \oplus s_{149} \oplus s_{169} \oplus s_{175} \oplus s_{181} \oplus s_{187} \oplus s_{208} \oplus s_{210} \oplus s_{212} \oplus s_{213} \oplus s_{216} \oplus s_{218} \oplus s_{219} \oplus s_{220} \oplus s_{247} \oplus s_{249} \oplus s_{252} \oplus s_{253} \oplus s_{286}$
$78, 85$	185	$s_{13} \oplus s_{28} \oplus s_{36} \oplus s_{63} \oplus s_{66} \oplus s_{68} \oplus s_{71} \oplus s_{74} \oplus s_{89} \oplus s_{104} \oplus s_{109} \oplus s_{112} \oplus s_{150} \oplus s_{170} \oplus s_{176} \oplus s_{182} \oplus s_{188} \oplus s_{209} \oplus s_{211} \oplus s_{213} \oplus s_{214} \oplus s_{217} \oplus s_{219} \oplus s_{220} \oplus s_{221} \oplus s_{248} \oplus s_{250} \oplus s_{253} \oplus s_{254} \oplus s_{287}$
$79, 86$	186	$s_{14} \oplus s_{29} \oplus s_{37} \oplus s_{64} \oplus s_{67} \oplus s_{69} \oplus s_{72} \oplus s_{75} \oplus s_{90} \oplus s_{105} \oplus s_{110} \oplus s_{113} \oplus s_{151} \oplus s_{171} \oplus s_{177} \oplus s_{183} \oplus s_{189} \oplus s_{210} \oplus s_{212} \oplus s_{214} \oplus s_{215} \oplus s_{218} \oplus s_{220} \oplus s_{221} \oplus s_{222} \oplus s_{249} \oplus s_{251} \oplus s_{254} \oplus s_{255} \oplus s_{288}$
$80, 87$	187	$s_{15} \oplus s_{30} \oplus s_{38} \oplus s_{65} \oplus s_{68} \oplus s_{70} \oplus s_{73} \oplus s_{76} \oplus s_{91} \oplus s_{106} \oplus s_{111} \oplus s_{114} \oplus s_{152} \oplus s_{172} \oplus s_{178} \oplus s_{184} \oplus s_{190} \oplus s_{211} \oplus s_{213} \oplus s_{215} \oplus s_{216} \oplus s_{219} \oplus s_{221} \oplus s_{222} \oplus s_{223} \oplus s_{230} \oplus s_{235} \oplus s_{250} \oplus s_{252} \oplus s_{255} \oplus s_{256} \oplus s_{289}$
$81, 88$	188	$s_{16} \oplus s_{31} \oplus s_{39} \oplus s_{61} \oplus s_{69} \oplus s_{71} \oplus s_{74} \oplus s_{77} \oplus s_{92} \oplus s_{107} \oplus s_{112} \oplus s_{115} \oplus s_{153} \oplus s_{173} \oplus s_{179} \oplus s_{185} \oplus s_{191} \oplus s_{212} \oplus s_{214} \oplus s_{216} \oplus s_{217} \oplus s_{220} \oplus s_{222} \oplus s_{223} \oplus s_{224} \oplus s_{231} \oplus s_{236} \oplus s_{251} \oplus s_{253} \oplus s_{256} \oplus s_{257} \oplus s_{290}$
$82, 89$	189	$s_{17} \oplus s_{32} \oplus s_{40} \oplus s_{62} \oplus s_{70} \oplus s_{72} \oplus s_{75} \oplus s_{78} \oplus s_{93} \oplus s_{107} \oplus s_{108} \oplus s_{111} \oplus s_{113} \oplus s_{116} \oplus s_{154} \oplus s_{174} \oplus s_{180} \oplus s_{186} \oplus s_{192} \oplus s_{213} \oplus s_{215} \oplus s_{217} \oplus s_{218} \oplus s_{221} \oplus s_{223} \oplus s_{224} \oplus s_{225} \oplus s_{232} \oplus s_{237} \oplus s_{252} \oplus s_{254} \oplus s_{257} \oplus s_{258} \oplus s_{291}$
$83, 90$	190	$s_{18} \oplus s_{33} \oplus s_{41} \oplus s_{63} \oplus s_{71} \oplus s_{73} \oplus s_{76} \oplus s_{79} \oplus s_{94} \oplus s_{108} \oplus s_{109} \oplus s_{112} \oplus s_{114} \oplus s_{117} \oplus s_{154} \oplus s_{155} \oplus s_{160} \oplus s_{175} \oplus s_{181} \oplus s_{187} \oplus s_{193} \oplus s_{214} \oplus s_{216} \oplus s_{218} \oplus s_{219} \oplus s_{222} \oplus s_{224} \oplus s_{225} \oplus s_{226} \oplus s_{233} \oplus s_{238} \oplus s_{253} \oplus s_{255} \oplus s_{258} \oplus s_{259} \oplus s_{292}$
$0, 21$	121	$s_{83} \oplus s_{88} \oplus s_{127} \oplus s_{129} \oplus s_{131} \oplus s_{174}$
$1, 22$	122	$s_{84} \oplus s_{89} \oplus s_{128} \oplus s_{130} \oplus s_{132} \oplus s_{175}$
$2, 23$	123	$s_{85} \oplus s_{90} \oplus s_{129} \oplus s_{131} \oplus s_{133} \oplus s_{176}$
$3, 24$	124	$s_{86} \oplus s_{91} \oplus s_{130} \oplus s_{132} \oplus s_{134} \oplus s_{177}$
$4, 25$	125	$s_{87} \oplus s_{92} \oplus s_{131} \oplus s_{133} \oplus s_{135} \oplus s_{178}$
$5, 26$	126	$s_{88} \oplus s_{93} \oplus s_{132} \oplus s_{134} \oplus s_{136} \oplus s_{179}$
$6, 27$	127	$s_{89} \oplus s_{94} \oplus s_{133} \oplus s_{135} \oplus s_{137} \oplus s_{180}$
$7, 28$	128	$s_{90} \oplus s_{95} \oplus s_{134} \oplus s_{136} \oplus s_{138} \oplus s_{181}$
$8, 29$	129	$s_{91} \oplus s_{96} \oplus s_{135} \oplus s_{137} \oplus s_{139} \oplus s_{182}$
$9, 30$	130	$s_{92} \oplus s_{97} \oplus s_{136} \oplus s_{138} \oplus s_{140} \oplus s_{183}$
$10, 31$	131	$s_{93} \oplus s_{98} \oplus s_{137} \oplus s_{139} \oplus s_{141} \oplus s_{184}$
$11, 32$	132	$s_{94} \oplus s_{99} \oplus s_{138} \oplus s_{140} \oplus s_{142} \oplus s_{185}$
$12, 33$	133	$s_{95} \oplus s_{100} \oplus s_{139} \oplus s_{141} \oplus s_{143} \oplus s_{186}$
$13, 34$	134	$s_{96} \oplus s_{101} \oplus s_{140} \oplus s_{142} \oplus s_{144} \oplus s_{187}$

$14, 35$	135	$s_{97} \oplus s_{102} \oplus s_{141} \oplus s_{143} \oplus s_{145} \oplus s_{188}$
$15, 36$	136	$s_{98} \oplus s_{103} \oplus s_{142} \oplus s_{144} \oplus s_{146} \oplus s_{189}$
$16, 37$	137	$s_{99} \oplus s_{104} \oplus s_{143} \oplus s_{145} \oplus s_{147} \oplus s_{190}$
$17, 38$	138	$s_{100} \oplus s_{105} \oplus s_{144} \oplus s_{146} \oplus s_{148} \oplus s_{191}$
$18, 39$	139	$s_{101} \oplus s_{106} \oplus s_{145} \oplus s_{147} \oplus s_{149} \oplus s_{192}$
$19, 40$	140	$s_{61} \oplus s_{66} \oplus s_{102} \oplus s_{107} \oplus s_{146} \oplus s_{148} \oplus s_{150} \oplus s_{154} \oplus s_{160} \oplus s_{193}$
$20, 41$	141	$s_{62} \oplus s_{67} \oplus s_{103} \oplus s_{108} \oplus s_{147} \oplus s_{149} \oplus s_{151} \oplus s_{155} \oplus s_{161} \oplus s_{194}$
$21, 42$	142	$s_{63} \oplus s_{68} \oplus s_{104} \oplus s_{109} \oplus s_{148} \oplus s_{150} \oplus s_{152} \oplus s_{156} \oplus s_{162} \oplus s_{195}$
$22, 43$	143	$s_{64} \oplus s_{69} \oplus s_{105} \oplus s_{110} \oplus s_{149} \oplus s_{151} \oplus s_{153} \oplus s_{157} \oplus s_{163} \oplus s_{196}$
$23, 44$	144	$s_{65} \oplus s_{70} \oplus s_{106} \oplus s_{107} \oplus s_{150} \oplus s_{152} \oplus s_{154} \oplus s_{158} \oplus s_{164} \oplus s_{197}$
$24, 45$	145	$s_{61} \oplus s_{71} \oplus s_{107} \oplus s_{108} \oplus s_{151} \oplus s_{153} \oplus s_{155} \oplus s_{159} \oplus s_{165} \oplus s_{198}$
$25, 46$	146	$s_{62} \oplus s_{72} \oplus s_{107} \oplus s_{108} \oplus s_{109} \oplus s_{111} \oplus s_{152} \oplus s_{154} \oplus s_{156} \oplus s_{160} \oplus s_{166} \oplus s_{199}$
$26, 47$	147	$s_{63} \oplus s_{73} \oplus s_{108} \oplus s_{109} \oplus s_{110} \oplus s_{112} \oplus s_{153} \oplus s_{155} \oplus s_{157} \oplus s_{161} \oplus s_{167} \oplus s_{200}$
$27, 48$	148	$s_{64} \oplus s_{74} \oplus s_{107} \oplus s_{109} \oplus s_{110} \oplus s_{113} \oplus s_{154} \oplus s_{156} \oplus s_{158} \oplus s_{162} \oplus s_{168} \oplus s_{201}$
$28, 49$	149	$s_{65} \oplus s_{75} \oplus s_{108} \oplus s_{110} \oplus s_{111} \oplus s_{114} \oplus s_{155} \oplus s_{157} \oplus s_{159} \oplus s_{163} \oplus s_{169} \oplus s_{202}$
$29, 50$	150	$s_{66} \oplus s_{76} \oplus s_{109} \oplus s_{111} \oplus s_{112} \oplus s_{115} \oplus s_{156} \oplus s_{158} \oplus s_{160} \oplus s_{164} \oplus s_{170} \oplus s_{203}$
$30, 51$	151	$s_{67} \oplus s_{77} \oplus s_{110} \oplus s_{112} \oplus s_{113} \oplus s_{116} \oplus s_{157} \oplus s_{159} \oplus s_{161} \oplus s_{165} \oplus s_{171} \oplus s_{204}$
$31, 52$	152	$s_{68} \oplus s_{78} \oplus s_{111} \oplus s_{113} \oplus s_{114} \oplus s_{117} \oplus s_{158} \oplus s_{160} \oplus s_{162} \oplus s_{166} \oplus s_{172} \oplus s_{205}$
$32, 53$	153	$s_{69} \oplus s_{79} \oplus s_{112} \oplus s_{114} \oplus s_{115} \oplus s_{118} \oplus s_{159} \oplus s_{161} \oplus s_{163} \oplus s_{167} \oplus s_{173} \oplus s_{206}$
$33, 54$	154	$s_{70} \oplus s_{80} \oplus s_{113} \oplus s_{115} \oplus s_{116} \oplus s_{119} \oplus s_{160} \oplus s_{162} \oplus s_{164} \oplus s_{168} \oplus s_{174} \oplus s_{207}$
$34, 55$	155	$s_{71} \oplus s_{81} \oplus s_{114} \oplus s_{116} \oplus s_{117} \oplus s_{120} \oplus s_{161} \oplus s_{163} \oplus s_{165} \oplus s_{169} \oplus s_{175} \oplus s_{208}$
$35, 56$	156	$s_{72} \oplus s_{82} \oplus s_{115} \oplus s_{117} \oplus s_{118} \oplus s_{121} \oplus s_{162} \oplus s_{164} \oplus s_{166} \oplus s_{170} \oplus s_{176} \oplus s_{209}$
$36, 57$	157	$s_{73} \oplus s_{83} \oplus s_{116} \oplus s_{118} \oplus s_{119} \oplus s_{122} \oplus s_{163} \oplus s_{165} \oplus s_{167} \oplus s_{171} \oplus s_{177} \oplus s_{210}$
$37, 58$	158	$s_{74} \oplus s_{84} \oplus s_{117} \oplus s_{119} \oplus s_{120} \oplus s_{123} \oplus s_{164} \oplus s_{166} \oplus s_{168} \oplus s_{172} \oplus s_{178} \oplus s_{211}$
$38, 59$	159	$s_{75} \oplus s_{85} \oplus s_{118} \oplus s_{120} \oplus s_{121} \oplus s_{124} \oplus s_{165} \oplus s_{167} \oplus s_{169} \oplus s_{173} \oplus s_{179} \oplus s_{212}$
$39, 60$	160	$s_{76} \oplus s_{86} \oplus s_{119} \oplus s_{121} \oplus s_{122} \oplus s_{125} \oplus s_{166} \oplus s_{168} \oplus s_{170} \oplus s_{174} \oplus s_{180} \oplus s_{213}$
$40, 61$	161	$s_{77} \oplus s_{87} \oplus s_{120} \oplus s_{122} \oplus s_{123} \oplus s_{126} \oplus s_{167} \oplus s_{169} \oplus s_{171} \oplus s_{175} \oplus s_{181} \oplus s_{214}$
$41, 62$	162	$s_{78} \oplus s_{88} \oplus s_{121} \oplus s_{123} \oplus s_{124} \oplus s_{127} \oplus s_{168} \oplus s_{170} \oplus s_{172} \oplus s_{176} \oplus s_{182} \oplus s_{215}$
$42, 63$	163	$s_{79} \oplus s_{89} \oplus s_{122} \oplus s_{124} \oplus s_{125} \oplus s_{128} \oplus s_{169} \oplus s_{171} \oplus s_{173} \oplus s_{177} \oplus s_{183} \oplus s_{216}$
$43, 64$	164	$s_{80} \oplus s_{90} \oplus s_{123} \oplus s_{125} \oplus s_{126} \oplus s_{129} \oplus s_{170} \oplus s_{172} \oplus s_{174} \oplus s_{178} \oplus s_{184} \oplus s_{217}$
$44, 65$	165	$s_{81} \oplus s_{91} \oplus s_{124} \oplus s_{126} \oplus s_{127} \oplus s_{130} \oplus s_{171} \oplus s_{173} \oplus s_{175} \oplus s_{179} \oplus s_{185} \oplus s_{218}$
$45, 66$	166	$s_{82} \oplus s_{92} \oplus s_{125} \oplus s_{127} \oplus s_{128} \oplus s_{131} \oplus s_{172} \oplus s_{174} \oplus s_{176} \oplus s_{180} \oplus s_{186} \oplus s_{219}$
$46, 67$	167	$s_{83} \oplus s_{93} \oplus s_{126} \oplus s_{128} \oplus s_{129} \oplus s_{132} \oplus s_{173} \oplus s_{175} \oplus s_{177} \oplus s_{181} \oplus s_{187} \oplus s_{220}$
$47, 68$	168	$s_{84} \oplus s_{94} \oplus s_{127} \oplus s_{129} \oplus s_{130} \oplus s_{133} \oplus s_{174} \oplus s_{176} \oplus s_{178} \oplus s_{182} \oplus s_{188} \oplus s_{221}$
$48, 69$	169	$s_{85} \oplus s_{95} \oplus s_{128} \oplus s_{130} \oplus s_{131} \oplus s_{134} \oplus s_{175} \oplus s_{177} \oplus s_{179} \oplus s_{183} \oplus s_{189} \oplus s_{222}$
$49, 70$	170	$s_{86} \oplus s_{96} \oplus s_{129} \oplus s_{131} \oplus s_{132} \oplus s_{135} \oplus s_{176} \oplus s_{178} \oplus s_{180} \oplus s_{184} \oplus s_{190} \oplus s_{223}$
$50, 71$	171	$s_{87} \oplus s_{97} \oplus s_{130} \oplus s_{132} \oplus s_{133} \oplus s_{136} \oplus s_{177} \oplus s_{179} \oplus s_{181} \oplus s_{185} \oplus s_{191} \oplus s_{224}$
$51, 72$	172	$s_{88} \oplus s_{98} \oplus s_{131} \oplus s_{133} \oplus s_{134} \oplus s_{137} \oplus s_{178} \oplus s_{180} \oplus s_{182} \oplus s_{186} \oplus s_{192} \oplus s_{225}$
$52, 73$	173	$s_{89} \oplus s_{99} \oplus s_{132} \oplus s_{134} \oplus s_{135} \oplus s_{138} \oplus s_{154} \oplus s_{160} \oplus s_{179} \oplus s_{181} \oplus s_{183} \oplus s_{187} \oplus s_{193} \oplus s_{226}$

$53, 74$	174	$s_{90} \oplus s_{100} \oplus s_{133} \oplus s_{135} \oplus s_{136} \oplus s_{139} \oplus s_{155} \oplus s_{161} \oplus s_{180} \oplus s_{182} \oplus s_{184} \oplus s_{188} \oplus s_{194} \oplus s_{227}$
$54, 75$	175	$s_{91} \oplus s_{101} \oplus s_{134} \oplus s_{136} \oplus s_{137} \oplus s_{140} \oplus s_{156} \oplus s_{162} \oplus s_{181} \oplus s_{183} \oplus s_{185} \oplus s_{189} \oplus s_{195} \oplus s_{228}$
$55, 76$	176	$s_{92} \oplus s_{102} \oplus s_{135} \oplus s_{137} \oplus s_{138} \oplus s_{141} \oplus s_{157} \oplus s_{163} \oplus s_{182} \oplus s_{184} \oplus s_{186} \oplus s_{190} \oplus s_{196} \oplus s_{229}$
$56, 77$	177	$s_{93} \oplus s_{103} \oplus s_{136} \oplus s_{138} \oplus s_{139} \oplus s_{142} \oplus s_{158} \oplus s_{164} \oplus s_{183} \oplus s_{185} \oplus s_{187} \oplus s_{191} \oplus s_{193} \oplus s_{196} \oplus s_{197} \oplus s_{230}$
$57, 78$	178	$s_{94} \oplus s_{104} \oplus s_{137} \oplus s_{139} \oplus s_{140} \oplus s_{143} \oplus s_{159} \oplus s_{165} \oplus s_{184} \oplus s_{186} \oplus s_{188} \oplus s_{192} \oplus s_{194} \oplus s_{197} \oplus s_{198} \oplus s_{231}$
$58, 79$	179	$s_{95} \oplus s_{105} \oplus s_{138} \oplus s_{140} \oplus s_{141} \oplus s_{144} \oplus s_{154} \oplus s_{166} \oplus s_{185} \oplus s_{187} \oplus s_{189} \oplus s_{193} \oplus s_{195} \oplus s_{198} \oplus s_{199} \oplus s_{232}$
$59, 80$	180	$s_{96} \oplus s_{106} \oplus s_{139} \oplus s_{141} \oplus s_{142} \oplus s_{145} \oplus s_{155} \oplus s_{167} \oplus s_{186} \oplus s_{188} \oplus s_{190} \oplus s_{194} \oplus s_{196} \oplus s_{199} \oplus s_{200} \oplus s_{233}$
$60, 81$	181	$s_{61} \oplus s_{66} \oplus s_{97} \oplus s_{107} \oplus s_{140} \oplus s_{142} \oplus s_{143} \oplus s_{146} \oplus s_{156} \oplus s_{168} \oplus s_{187} \oplus s_{189} \oplus s_{191} \oplus s_{195} \oplus s_{197} \oplus s_{200} \oplus s_{201} \oplus s_{234}$
$61, 82$	182	$s_{62} \oplus s_{67} \oplus s_{98} \oplus s_{108} \oplus s_{141} \oplus s_{143} \oplus s_{144} \oplus s_{147} \oplus s_{157} \oplus s_{169} \oplus s_{188} \oplus s_{190} \oplus s_{192} \oplus s_{196} \oplus s_{198} \oplus s_{201} \oplus s_{202} \oplus s_{235}$
$62, 83$	183	$s_{63} \oplus s_{68} \oplus s_{99} \oplus s_{109} \oplus s_{142} \oplus s_{144} \oplus s_{145} \oplus s_{148} \oplus s_{154} \oplus s_{158} \oplus s_{160} \oplus s_{170} \oplus s_{189} \oplus s_{191} \oplus s_{193} \oplus s_{197} \oplus s_{199} \oplus s_{202} \oplus s_{203} \oplus s_{236}$
$63, 84$	184	$s_{64} \oplus s_{69} \oplus s_{100} \oplus s_{110} \oplus s_{143} \oplus s_{145} \oplus s_{146} \oplus s_{149} \oplus s_{155} \oplus s_{159} \oplus s_{161} \oplus s_{171} \oplus s_{190} \oplus s_{192} \oplus s_{194} \oplus s_{198} \oplus s_{200} \oplus s_{203} \oplus s_{204} \oplus s_{237}$
$64, 85$	185	$s_{65} \oplus s_{70} \oplus s_{101} \oplus s_{111} \oplus s_{144} \oplus s_{146} \oplus s_{147} \oplus s_{150} \oplus s_{154} \oplus s_{156} \oplus s_{162} \oplus s_{172} \oplus s_{191} \oplus s_{193} \oplus s_{195} \oplus s_{199} \oplus s_{201} \oplus s_{204} \oplus s_{205} \oplus s_{238}$
$65, 86$	186	$s_{66} \oplus s_{71} \oplus s_{102} \oplus s_{112} \oplus s_{145} \oplus s_{147} \oplus s_{148} \oplus s_{151} \oplus s_{155} \oplus s_{157} \oplus s_{163} \oplus s_{173} \oplus s_{192} \oplus s_{194} \oplus s_{196} \oplus s_{200} \oplus s_{202} \oplus s_{205} \oplus s_{206} \oplus s_{239}$
$66, 87$	187	$s_{67} \oplus s_{72} \oplus s_{103} \oplus s_{113} \oplus s_{146} \oplus s_{148} \oplus s_{149} \oplus s_{152} \oplus s_{154} \oplus s_{156} \oplus s_{158} \oplus s_{160} \oplus s_{164} \oplus s_{174} \oplus s_{193} \oplus s_{195} \oplus s_{197} \oplus s_{201} \oplus s_{203} \oplus s_{206} \oplus s_{207} \oplus s_{240}$
$67, 88$	188	$s_{68} \oplus s_{73} \oplus s_{104} \oplus s_{114} \oplus s_{147} \oplus s_{149} \oplus s_{150} \oplus s_{153} \oplus s_{155} \oplus s_{157} \oplus s_{159} \oplus s_{161} \oplus s_{165} \oplus s_{175} \oplus s_{194} \oplus s_{196} \oplus s_{198} \oplus s_{202} \oplus s_{204} \oplus s_{207} \oplus s_{208} \oplus s_{241}$
$68, 89$	189	$s_{69} \oplus s_{74} \oplus s_{105} \oplus s_{107} \oplus s_{111} \oplus s_{115} \oplus s_{148} \oplus s_{150} \oplus s_{151} \oplus s_{154} \oplus s_{156} \oplus s_{158} \oplus s_{160} \oplus s_{162} \oplus s_{166} \oplus s_{176} \oplus s_{195} \oplus s_{197} \oplus s_{199} \oplus s_{203} \oplus s_{205} \oplus s_{208} \oplus s_{209} \oplus s_{242}$
$69, 90$	190	$s_{70} \oplus s_{75} \oplus s_{106} \oplus s_{108} \oplus s_{112} \oplus s_{116} \oplus s_{149} \oplus s_{151} \oplus s_{152} \oplus s_{155} \oplus s_{157} \oplus s_{159} \oplus s_{161} \oplus s_{163} \oplus s_{167} \oplus s_{177} \oplus s_{196} \oplus s_{198} \oplus s_{200} \oplus s_{204} \oplus s_{206} \oplus s_{209} \oplus s_{210} \oplus s_{243}$
$70, 91$	191	$s_{61} \oplus s_{66} \oplus s_{71} \oplus s_{76} \oplus s_{107} \oplus s_{109} \oplus s_{113} \oplus s_{117} \oplus s_{150} \oplus s_{152} \oplus s_{153} \oplus s_{156} \oplus s_{158} \oplus s_{160} \oplus s_{162} \oplus s_{164} \oplus s_{168} \oplus s_{178} \oplus s_{197} \oplus s_{199} \oplus s_{201} \oplus s_{205} \oplus s_{207} \oplus s_{210} \oplus s_{211} \oplus s_{244}$
$71, 92$	192	$s_{62} \oplus s_{67} \oplus s_{72} \oplus s_{77} \oplus s_{107} \oplus s_{108} \oplus s_{110} \oplus s_{111} \oplus s_{114} \oplus s_{118} \oplus s_{151} \oplus s_{153} \oplus s_{154} \oplus s_{157} \oplus s_{159} \oplus s_{161} \oplus s_{163} \oplus s_{165} \oplus s_{169} \oplus s_{179} \oplus s_{198} \oplus s_{200} \oplus s_{202} \oplus s_{206} \oplus s_{208} \oplus s_{211} \oplus s_{212} \oplus s_{245}$

$72, 93$	193	$s_{63} \oplus s_{68} \oplus s_{73} \oplus s_{78} \oplus s_{107} \oplus s_{108} \oplus s_{109} \oplus s_{112} \oplus s_{115} \oplus s_{119} \oplus s_{152} \oplus s_{154} \oplus s_{155} \oplus s_{158} \oplus s_{160} \oplus s_{162} \oplus s_{164} \oplus s_{166} \oplus s_{170} \oplus s_{180} \oplus s_{199} \oplus s_{201} \oplus s_{203} \oplus s_{207} \oplus s_{209} \oplus s_{212} \oplus s_{213} \oplus s_{246}$
$73, 94$	194	$s_{64} \oplus s_{69} \oplus s_{74} \oplus s_{79} \oplus s_{108} \oplus s_{109} \oplus s_{110} \oplus s_{113} \oplus s_{116} \oplus s_{120} \oplus s_{153} \oplus s_{155} \oplus s_{156} \oplus s_{159} \oplus s_{161} \oplus s_{163} \oplus s_{165} \oplus s_{167} \oplus s_{171} \oplus s_{181} \oplus s_{200} \oplus s_{202} \oplus s_{204} \oplus s_{208} \oplus s_{210} \oplus s_{213} \oplus s_{214} \oplus s_{247}$
$74, 95$	195	$s_{65} \oplus s_{70} \oplus s_{75} \oplus s_{80} \oplus s_{107} \oplus s_{109} \oplus s_{110} \oplus s_{114} \oplus s_{117} \oplus s_{121} \oplus s_{154} \oplus s_{156} \oplus s_{157} \oplus s_{160} \oplus s_{162} \oplus s_{164} \oplus s_{166} \oplus s_{168} \oplus s_{172} \oplus s_{182} \oplus s_{201} \oplus s_{203} \oplus s_{205} \oplus s_{209} \oplus s_{211} \oplus s_{214} \oplus s_{215} \oplus s_{248}$
$75, 96$	196	$s_{66} \oplus s_{71} \oplus s_{76} \oplus s_{81} \oplus s_{108} \oplus s_{110} \oplus s_{111} \oplus s_{115} \oplus s_{118} \oplus s_{122} \oplus s_{155} \oplus s_{157} \oplus s_{158} \oplus s_{161} \oplus s_{163} \oplus s_{165} \oplus s_{167} \oplus s_{169} \oplus s_{173} \oplus s_{183} \oplus s_{202} \oplus s_{204} \oplus s_{206} \oplus s_{210} \oplus s_{212} \oplus s_{215} \oplus s_{216} \oplus s_{249}$
$76, 97$	197	$s_{67} \oplus s_{72} \oplus s_{77} \oplus s_{82} \oplus s_{109} \oplus s_{111} \oplus s_{112} \oplus s_{116} \oplus s_{119} \oplus s_{123} \oplus s_{156} \oplus s_{158} \oplus s_{159} \oplus s_{162} \oplus s_{164} \oplus s_{166} \oplus s_{168} \oplus s_{170} \oplus s_{174} \oplus s_{184} \oplus s_{203} \oplus s_{205} \oplus s_{207} \oplus s_{211} \oplus s_{213} \oplus s_{216} \oplus s_{217} \oplus s_{250}$
$77, 98$	198	$s_{68} \oplus s_{73} \oplus s_{78} \oplus s_{83} \oplus s_{110} \oplus s_{112} \oplus s_{113} \oplus s_{117} \oplus s_{120} \oplus s_{124} \oplus s_{157} \oplus s_{159} \oplus s_{160} \oplus s_{163} \oplus s_{165} \oplus s_{167} \oplus s_{169} \oplus s_{171} \oplus s_{175} \oplus s_{185} \oplus s_{204} \oplus s_{206} \oplus s_{208} \oplus s_{212} \oplus s_{214} \oplus s_{217} \oplus s_{218} \oplus s_{251}$
$78, 99$	199	$s_{69} \oplus s_{74} \oplus s_{79} \oplus s_{84} \oplus s_{111} \oplus s_{113} \oplus s_{114} \oplus s_{118} \oplus s_{121} \oplus s_{125} \oplus s_{158} \oplus s_{160} \oplus s_{161} \oplus s_{164} \oplus s_{166} \oplus s_{168} \oplus s_{170} \oplus s_{172} \oplus s_{176} \oplus s_{186} \oplus s_{205} \oplus s_{207} \oplus s_{209} \oplus s_{213} \oplus s_{215} \oplus s_{218} \oplus s_{219} \oplus s_{252}$
$79, 100$	200	$s_{70} \oplus s_{75} \oplus s_{80} \oplus s_{85} \oplus s_{112} \oplus s_{114} \oplus s_{115} \oplus s_{119} \oplus s_{122} \oplus s_{126} \oplus s_{159} \oplus s_{161} \oplus s_{162} \oplus s_{165} \oplus s_{167} \oplus s_{169} \oplus s_{171} \oplus s_{173} \oplus s_{177} \oplus s_{187} \oplus s_{206} \oplus s_{208} \oplus s_{210} \oplus s_{214} \oplus s_{216} \oplus s_{219} \oplus s_{220} \oplus s_{253}$
$80, 101$	201	$s_{71} \oplus s_{76} \oplus s_{81} \oplus s_{86} \oplus s_{113} \oplus s_{115} \oplus s_{116} \oplus s_{120} \oplus s_{123} \oplus s_{127} \oplus s_{160} \oplus s_{162} \oplus s_{163} \oplus s_{166} \oplus s_{168} \oplus s_{170} \oplus s_{172} \oplus s_{174} \oplus s_{178} \oplus s_{188} \oplus s_{207} \oplus s_{209} \oplus s_{211} \oplus s_{215} \oplus s_{217} \oplus s_{220} \oplus s_{221} \oplus s_{254}$
$81, 102$	202	$s_{72} \oplus s_{77} \oplus s_{82} \oplus s_{87} \oplus s_{114} \oplus s_{116} \oplus s_{117} \oplus s_{121} \oplus s_{124} \oplus s_{128} \oplus s_{161} \oplus s_{163} \oplus s_{164} \oplus s_{167} \oplus s_{169} \oplus s_{171} \oplus s_{173} \oplus s_{175} \oplus s_{179} \oplus s_{189} \oplus s_{208} \oplus s_{210} \oplus s_{212} \oplus s_{216} \oplus s_{218} \oplus s_{221} \oplus s_{222} \oplus s_{255}$
$82, 103$	203	$s_{73} \oplus s_{78} \oplus s_{83} \oplus s_{88} \oplus s_{115} \oplus s_{117} \oplus s_{118} \oplus s_{122} \oplus s_{125} \oplus s_{129} \oplus s_{162} \oplus s_{164} \oplus s_{165} \oplus s_{168} \oplus s_{170} \oplus s_{172} \oplus s_{174} \oplus s_{176} \oplus s_{180} \oplus s_{190} \oplus s_{209} \oplus s_{211} \oplus s_{213} \oplus s_{217} \oplus s_{219} \oplus s_{222} \oplus s_{223} \oplus s_{256}$
$83, 104$	204	$s_{74} \oplus s_{79} \oplus s_{84} \oplus s_{89} \oplus s_{116} \oplus s_{118} \oplus s_{119} \oplus s_{123} \oplus s_{126} \oplus s_{130} \oplus s_{163} \oplus s_{165} \oplus s_{166} \oplus s_{169} \oplus s_{171} \oplus s_{173} \oplus s_{175} \oplus s_{177} \oplus s_{181} \oplus s_{191} \oplus s_{210} \oplus s_{212} \oplus s_{214} \oplus s_{218} \oplus s_{220} \oplus s_{223} \oplus s_{224} \oplus s_{257}$
$84, 105$	205	$s_{75} \oplus s_{80} \oplus s_{85} \oplus s_{90} \oplus s_{117} \oplus s_{119} \oplus s_{120} \oplus s_{124} \oplus s_{127} \oplus s_{131} \oplus s_{164} \oplus s_{166} \oplus s_{167} \oplus s_{170} \oplus s_{172} \oplus s_{174} \oplus s_{176} \oplus s_{178} \oplus s_{182} \oplus s_{192} \oplus s_{211} \oplus s_{213} \oplus s_{215} \oplus s_{219} \oplus s_{221} \oplus s_{224} \oplus s_{225} \oplus s_{258}$

85, 106	206	$s_{76} \oplus s_{81} \oplus s_{86} \oplus s_{91} \oplus s_{118} \oplus s_{120} \oplus s_{121} \oplus s_{125} \oplus s_{128} \oplus s_{132} \oplus s_{154} \oplus s_{160} \oplus s_{165} \oplus s_{167} \oplus s_{168} \oplus s_{171} \oplus s_{173} \oplus s_{175} \oplus s_{177} \oplus s_{179} \oplus s_{183} \oplus s_{193} \oplus s_{212} \oplus s_{214} \oplus s_{216} \oplus s_{220} \oplus s_{222} \oplus s_{225} \oplus s_{226} \oplus s_{259}$
86, 107	207	$s_{77} \oplus s_{82} \oplus s_{87} \oplus s_{92} \oplus s_{119} \oplus s_{121} \oplus s_{122} \oplus s_{126} \oplus s_{129} \oplus s_{133} \oplus s_{155} \oplus s_{161} \oplus s_{166} \oplus s_{168} \oplus s_{169} \oplus s_{172} \oplus s_{174} \oplus s_{176} \oplus s_{178} \oplus s_{180} \oplus s_{184} \oplus s_{194} \oplus s_{213} \oplus s_{215} \oplus s_{217} \oplus s_{221} \oplus s_{223} \oplus s_{226} \oplus s_{227} \oplus s_{260}$
87, 108	208	$s_{78} \oplus s_{83} \oplus s_{88} \oplus s_{93} \oplus s_{120} \oplus s_{122} \oplus s_{123} \oplus s_{127} \oplus s_{130} \oplus s_{134} \oplus s_{156} \oplus s_{162} \oplus s_{167} \oplus s_{169} \oplus s_{170} \oplus s_{173} \oplus s_{175} \oplus s_{177} \oplus s_{179} \oplus s_{181} \oplus s_{185} \oplus s_{195} \oplus s_{214} \oplus s_{216} \oplus s_{218} \oplus s_{222} \oplus s_{224} \oplus s_{227} \oplus s_{228} \oplus s_{261}$
88, 109	209	$s_{79} \oplus s_{84} \oplus s_{89} \oplus s_{94} \oplus s_{121} \oplus s_{123} \oplus s_{124} \oplus s_{128} \oplus s_{131} \oplus s_{135} \oplus s_{157} \oplus s_{163} \oplus s_{168} \oplus s_{170} \oplus s_{171} \oplus s_{174} \oplus s_{176} \oplus s_{178} \oplus s_{180} \oplus s_{182} \oplus s_{186} \oplus s_{196} \oplus s_{215} \oplus s_{217} \oplus s_{219} \oplus s_{223} \oplus s_{225} \oplus s_{228} \oplus s_{229} \oplus s_{262}$
89, 110	210	$s_{80} \oplus s_{85} \oplus s_{90} \oplus s_{95} \oplus s_{122} \oplus s_{124} \oplus s_{125} \oplus s_{129} \oplus s_{132} \oplus s_{136} \oplus s_{158} \oplus s_{164} \oplus s_{169} \oplus s_{171} \oplus s_{172} \oplus s_{175} \oplus s_{177} \oplus s_{179} \oplus s_{181} \oplus s_{183} \oplus s_{187} \oplus s_{193} \oplus s_{196} \oplus s_{197} \oplus s_{216} \oplus s_{218} \oplus s_{220} \oplus s_{224} \oplus s_{226} \oplus s_{229} \oplus s_{230} \oplus s_{263}$
90, 111	211	$s_{81} \oplus s_{86} \oplus s_{91} \oplus s_{96} \oplus s_{123} \oplus s_{125} \oplus s_{126} \oplus s_{130} \oplus s_{133} \oplus s_{137} \oplus s_{159} \oplus s_{165} \oplus s_{170} \oplus s_{172} \oplus s_{173} \oplus s_{176} \oplus s_{178} \oplus s_{180} \oplus s_{182} \oplus s_{184} \oplus s_{188} \oplus s_{193} \oplus s_{194} \oplus s_{196} \oplus s_{197} \oplus s_{198} \oplus s_{217} \oplus s_{219} \oplus s_{221} \oplus s_{225} \oplus s_{227} \oplus s_{230} \oplus s_{231} \oplus s_{264}$
91, 112	212	$s_{82} \oplus s_{87} \oplus s_{92} \oplus s_{97} \oplus s_{124} \oplus s_{126} \oplus s_{127} \oplus s_{131} \oplus s_{134} \oplus s_{138} \oplus s_{160} \oplus s_{166} \oplus s_{171} \oplus s_{173} \oplus s_{174} \oplus s_{177} \oplus s_{179} \oplus s_{181} \oplus s_{183} \oplus s_{185} \oplus s_{189} \oplus s_{194} \oplus s_{195} \oplus s_{197} \oplus s_{198} \oplus s_{199} \oplus s_{218} \oplus s_{220} \oplus s_{222} \oplus s_{226} \oplus s_{228} \oplus s_{231} \oplus s_{232} \oplus s_{265}$
92, 113	213	$s_{83} \oplus s_{88} \oplus s_{93} \oplus s_{98} \oplus s_{125} \oplus s_{127} \oplus s_{128} \oplus s_{132} \oplus s_{135} \oplus s_{139} \oplus s_{161} \oplus s_{167} \oplus s_{172} \oplus s_{174} \oplus s_{175} \oplus s_{178} \oplus s_{180} \oplus s_{182} \oplus s_{184} \oplus s_{186} \oplus s_{190} \oplus s_{195} \oplus s_{196} \oplus s_{198} \oplus s_{199} \oplus s_{200} \oplus s_{219} \oplus s_{221} \oplus s_{223} \oplus s_{227} \oplus s_{229} \oplus s_{232} \oplus s_{233} \oplus s_{266}$
93, 114	214	$s_{84} \oplus s_{89} \oplus s_{94} \oplus s_{99} \oplus s_{126} \oplus s_{128} \oplus s_{129} \oplus s_{133} \oplus s_{136} \oplus s_{140} \oplus s_{162} \oplus s_{168} \oplus s_{173} \oplus s_{175} \oplus s_{176} \oplus s_{179} \oplus s_{181} \oplus s_{183} \oplus s_{185} \oplus s_{187} \oplus s_{191} \oplus s_{193} \oplus s_{197} \oplus s_{199} \oplus s_{200} \oplus s_{201} \oplus s_{220} \oplus s_{222} \oplus s_{224} \oplus s_{228} \oplus s_{230} \oplus s_{233} \oplus s_{234} \oplus s_{267}$
94, 115	215	$s_{85} \oplus s_{90} \oplus s_{95} \oplus s_{100} \oplus s_{127} \oplus s_{129} \oplus s_{130} \oplus s_{134} \oplus s_{137} \oplus s_{141} \oplus s_{163} \oplus s_{169} \oplus s_{174} \oplus s_{176} \oplus s_{177} \oplus s_{180} \oplus s_{182} \oplus s_{184} \oplus s_{186} \oplus s_{188} \oplus s_{192} \oplus s_{194} \oplus s_{198} \oplus s_{200} \oplus s_{201} \oplus s_{202} \oplus s_{221} \oplus s_{223} \oplus s_{225} \oplus s_{229} \oplus s_{231} \oplus s_{234} \oplus s_{235} \oplus s_{268}$
95, 116	216	$s_{86} \oplus s_{91} \oplus s_{96} \oplus s_{101} \oplus s_{128} \oplus s_{130} \oplus s_{131} \oplus s_{135} \oplus s_{138} \oplus s_{142} \oplus s_{154} \oplus s_{160} \oplus s_{164} \oplus s_{170} \oplus s_{175} \oplus s_{177} \oplus s_{178} \oplus s_{181} \oplus s_{183} \oplus s_{185} \oplus s_{187} \oplus s_{189} \oplus s_{195} \oplus s_{196} \oplus s_{199} \oplus s_{201} \oplus s_{202} \oplus s_{203} \oplus s_{222} \oplus s_{224} \oplus s_{226} \oplus s_{230} \oplus s_{232} \oplus s_{235} \oplus s_{236} \oplus s_{269}$
96, 117	217	$s_{87} \oplus s_{92} \oplus s_{97} \oplus s_{102} \oplus s_{129} \oplus s_{131} \oplus s_{132} \oplus s_{136} \oplus s_{139} \oplus s_{143} \oplus s_{155} \oplus s_{161} \oplus s_{165} \oplus s_{171} \oplus s_{176} \oplus s_{178} \oplus s_{179} \oplus s_{182} \oplus s_{184} \oplus s_{186} \oplus s_{188} \oplus s_{190} \oplus s_{196} \oplus s_{197} \oplus s_{200} \oplus s_{202} \oplus s_{203} \oplus s_{204} \oplus s_{223} \oplus s_{225} \oplus s_{227} \oplus s_{231} \oplus s_{233} \oplus s_{236} \oplus s_{237} \oplus s_{270}$
97, 118	218	$s_{88} \oplus s_{93} \oplus s_{98} \oplus s_{103} \oplus s_{130} \oplus s_{132} \oplus s_{133} \oplus s_{137} \oplus s_{140} \oplus s_{144} \oplus s_{156} \oplus s_{162} \oplus s_{166} \oplus s_{172} \oplus s_{177} \oplus s_{179} \oplus s_{180} \oplus s_{183} \oplus s_{185} \oplus s_{187} \oplus s_{189} \oplus s_{191} \oplus s_{197} \oplus s_{198} \oplus s_{201} \oplus s_{203} \oplus s_{204} \oplus s_{205} \oplus s_{224} \oplus s_{226} \oplus s_{228} \oplus s_{232} \oplus s_{234} \oplus s_{237} \oplus s_{238} \oplus s_{271}$

98, 119	219	$s_{89} \oplus s_{94} \oplus s_{99} \oplus s_{104} \oplus s_{131} \oplus s_{133} \oplus s_{134} \oplus s_{138} \oplus s_{141} \oplus s_{145} \oplus s_{157} \oplus s_{163} \oplus s_{167} \oplus s_{173} \oplus s_{178} \oplus s_{180} \oplus s_{181} \oplus s_{184} \oplus s_{186} \oplus s_{188} \oplus s_{190} \oplus s_{192} \oplus s_{198} \oplus s_{199} \oplus s_{202} \oplus s_{204} \oplus s_{205} \oplus s_{206} \oplus s_{225} \oplus s_{227} \oplus s_{229} \oplus s_{233} \oplus s_{235} \oplus s_{238} \oplus s_{239} \oplus s_{272}$
99, 120	220	$s_{90} \oplus s_{95} \oplus s_{100} \oplus s_{105} \oplus s_{132} \oplus s_{134} \oplus s_{135} \oplus s_{139} \oplus s_{142} \oplus s_{146} \oplus s_{154} \oplus s_{158} \oplus s_{160} \oplus s_{164} \oplus s_{168} \oplus s_{174} \oplus s_{179} \oplus s_{181} \oplus s_{182} \oplus s_{185} \oplus s_{187} \oplus s_{189} \oplus s_{191} \oplus s_{196} \oplus s_{199} \oplus s_{200} \oplus s_{203} \oplus s_{205} \oplus s_{206} \oplus s_{207} \oplus s_{226} \oplus s_{228} \oplus s_{230} \oplus s_{234} \oplus s_{236} \oplus s_{239} \oplus s_{240} \oplus s_{273}$
100, 121	221	$s_{91} \oplus s_{96} \oplus s_{101} \oplus s_{106} \oplus s_{133} \oplus s_{135} \oplus s_{136} \oplus s_{140} \oplus s_{143} \oplus s_{147} \oplus s_{155} \oplus s_{159} \oplus s_{161} \oplus s_{165} \oplus s_{169} \oplus s_{175} \oplus s_{180} \oplus s_{182} \oplus s_{183} \oplus s_{186} \oplus s_{188} \oplus s_{190} \oplus s_{192} \oplus s_{197} \oplus s_{200} \oplus s_{201} \oplus s_{204} \oplus s_{206} \oplus s_{207} \oplus s_{208} \oplus s_{227} \oplus s_{229} \oplus s_{231} \oplus s_{235} \oplus s_{237} \oplus s_{240} \oplus s_{241} \oplus s_{274}$
101, 122	222	$s_{61} \oplus s_{66} \oplus s_{92} \oplus s_{97} \oplus s_{102} \oplus s_{107} \oplus s_{134} \oplus s_{136} \oplus s_{137} \oplus s_{141} \oplus s_{144} \oplus s_{148} \oplus s_{154} \oplus s_{156} \oplus s_{162} \oplus s_{166} \oplus s_{170} \oplus s_{176} \oplus s_{181} \oplus s_{183} \oplus s_{184} \oplus s_{187} \oplus s_{189} \oplus s_{191} \oplus s_{196} \oplus s_{198} \oplus s_{201} \oplus s_{202} \oplus s_{205} \oplus s_{207} \oplus s_{208} \oplus s_{209} \oplus s_{228} \oplus s_{230} \oplus s_{232} \oplus s_{236} \oplus s_{238} \oplus s_{241} \oplus s_{242} \oplus s_{275}$
102, 123	223	$s_{62} \oplus s_{67} \oplus s_{93} \oplus s_{98} \oplus s_{103} \oplus s_{108} \oplus s_{135} \oplus s_{137} \oplus s_{138} \oplus s_{142} \oplus s_{145} \oplus s_{149} \oplus s_{155} \oplus s_{157} \oplus s_{163} \oplus s_{167} \oplus s_{171} \oplus s_{177} \oplus s_{182} \oplus s_{184} \oplus s_{185} \oplus s_{188} \oplus s_{190} \oplus s_{192} \oplus s_{197} \oplus s_{199} \oplus s_{202} \oplus s_{203} \oplus s_{206} \oplus s_{208} \oplus s_{209} \oplus s_{210} \oplus s_{229} \oplus s_{231} \oplus s_{233} \oplus s_{237} \oplus s_{239} \oplus s_{242} \oplus s_{243} \oplus s_{276}$
103, 124	224	$s_{63} \oplus s_{68} \oplus s_{94} \oplus s_{99} \oplus s_{104} \oplus s_{109} \oplus s_{136} \oplus s_{138} \oplus s_{139} \oplus s_{143} \oplus s_{146} \oplus s_{150} \oplus s_{154} \oplus s_{156} \oplus s_{158} \oplus s_{160} \oplus s_{164} \oplus s_{168} \oplus s_{172} \oplus s_{178} \oplus s_{183} \oplus s_{185} \oplus s_{186} \oplus s_{189} \oplus s_{191} \oplus s_{196} \oplus s_{198} \oplus s_{200} \oplus s_{203} \oplus s_{204} \oplus s_{207} \oplus s_{209} \oplus s_{210} \oplus s_{211} \oplus s_{230} \oplus s_{232} \oplus s_{234} \oplus s_{238} \oplus s_{240} \oplus s_{243} \oplus s_{244} \oplus s_{277}$
104, 125	225	$s_{64} \oplus s_{69} \oplus s_{95} \oplus s_{100} \oplus s_{105} \oplus s_{110} \oplus s_{137} \oplus s_{139} \oplus s_{140} \oplus s_{144} \oplus s_{147} \oplus s_{151} \oplus s_{155} \oplus s_{157} \oplus s_{159} \oplus s_{161} \oplus s_{165} \oplus s_{169} \oplus s_{173} \oplus s_{179} \oplus s_{184} \oplus s_{186} \oplus s_{187} \oplus s_{190} \oplus s_{192} \oplus s_{197} \oplus s_{199} \oplus s_{201} \oplus s_{204} \oplus s_{205} \oplus s_{208} \oplus s_{210} \oplus s_{211} \oplus s_{212} \oplus s_{231} \oplus s_{233} \oplus s_{235} \oplus s_{239} \oplus s_{241} \oplus s_{244} \oplus s_{245} \oplus s_{278}$
105, 126	226	$s_{65} \oplus s_{70} \oplus s_{96} \oplus s_{101} \oplus s_{106} \oplus s_{111} \oplus s_{138} \oplus s_{140} \oplus s_{141} \oplus s_{145} \oplus s_{148} \oplus s_{152} \oplus s_{154} \oplus s_{156} \oplus s_{158} \oplus s_{162} \oplus s_{166} \oplus s_{170} \oplus s_{174} \oplus s_{180} \oplus s_{185} \oplus s_{187} \oplus s_{188} \oplus s_{191} \oplus s_{193} \oplus s_{198} \oplus s_{200} \oplus s_{202} \oplus s_{205} \oplus s_{206} \oplus s_{209} \oplus s_{211} \oplus s_{212} \oplus s_{213} \oplus s_{232} \oplus s_{234} \oplus s_{236} \oplus s_{240} \oplus s_{242} \oplus s_{245} \oplus s_{246} \oplus s_{279}$
106, 127	227	$s_{61} \oplus s_{71} \oplus s_{97} \oplus s_{102} \oplus s_{107} \oplus s_{112} \oplus s_{139} \oplus s_{141} \oplus s_{142} \oplus s_{146} \oplus s_{149} \oplus s_{153} \oplus s_{155} \oplus s_{157} \oplus s_{159} \oplus s_{163} \oplus s_{167} \oplus s_{171} \oplus s_{175} \oplus s_{181} \oplus s_{186} \oplus s_{188} \oplus s_{189} \oplus s_{192} \oplus s_{194} \oplus s_{199} \oplus s_{201} \oplus s_{203} \oplus s_{206} \oplus s_{207} \oplus s_{210} \oplus s_{212} \oplus s_{213} \oplus s_{214} \oplus s_{233} \oplus s_{235} \oplus s_{237} \oplus s_{241} \oplus s_{243} \oplus s_{246} \oplus s_{247} \oplus s_{280}$
107, 128	228	$s_{62} \oplus s_{72} \oplus s_{98} \oplus s_{103} \oplus s_{107} \oplus s_{108} \oplus s_{111} \oplus s_{113} \oplus s_{140} \oplus s_{142} \oplus s_{143} \oplus s_{147} \oplus s_{150} \oplus s_{156} \oplus s_{158} \oplus s_{164} \oplus s_{168} \oplus s_{172} \oplus s_{176} \oplus s_{182} \oplus s_{187} \oplus s_{189} \oplus s_{190} \oplus s_{193} \oplus s_{195} \oplus s_{200} \oplus s_{202} \oplus s_{204} \oplus s_{207} \oplus s_{208} \oplus s_{211} \oplus s_{213} \oplus s_{214} \oplus s_{215} \oplus s_{234} \oplus s_{236} \oplus s_{238} \oplus s_{242} \oplus s_{244} \oplus s_{247} \oplus s_{248} \oplus s_{281}$

108, 129	229	$s_{63} \oplus s_{73} \oplus s_{99} \oplus s_{104} \oplus s_{108} \oplus s_{109} \oplus s_{112} \oplus s_{114} \oplus s_{141} \oplus s_{143} \oplus s_{144} \oplus s_{148} \oplus s_{151} \oplus s_{157} \oplus s_{159} \oplus s_{165} \oplus s_{169} \oplus s_{173} \oplus s_{177} \oplus s_{183} \oplus s_{188} \oplus s_{190} \oplus s_{191} \oplus s_{194} \oplus s_{196} \oplus s_{201} \oplus s_{203} \oplus s_{205} \oplus s_{208} \oplus s_{209} \oplus s_{212} \oplus s_{214} \oplus s_{215} \oplus s_{216} \oplus s_{235} \oplus s_{237} \oplus s_{239} \oplus s_{243} \oplus s_{245} \oplus s_{248} \oplus s_{249} \oplus s_{282}$
109, 130	230	$s_{64} \oplus s_{74} \oplus s_{100} \oplus s_{105} \oplus s_{109} \oplus s_{110} \oplus s_{113} \oplus s_{115} \oplus s_{142} \oplus s_{144} \oplus s_{145} \oplus s_{149} \oplus s_{152} \oplus s_{158} \oplus s_{160} \oplus s_{166} \oplus s_{170} \oplus s_{174} \oplus s_{178} \oplus s_{184} \oplus s_{189} \oplus s_{191} \oplus s_{192} \oplus s_{195} \oplus s_{197} \oplus s_{202} \oplus s_{204} \oplus s_{206} \oplus s_{209} \oplus s_{210} \oplus s_{213} \oplus s_{215} \oplus s_{216} \oplus s_{217} \oplus s_{236} \oplus s_{238} \oplus s_{240} \oplus s_{244} \oplus s_{246} \oplus s_{249} \oplus s_{250} \oplus s_{283}$
110, 131	231	$s_{65} \oplus s_{75} \oplus s_{101} \oplus s_{106} \oplus s_{110} \oplus s_{111} \oplus s_{114} \oplus s_{116} \oplus s_{143} \oplus s_{145} \oplus s_{146} \oplus s_{150} \oplus s_{153} \oplus s_{154} \oplus s_{159} \oplus s_{160} \oplus s_{161} \oplus s_{167} \oplus s_{171} \oplus s_{175} \oplus s_{179} \oplus s_{185} \oplus s_{190} \oplus s_{192} \oplus s_{193} \oplus s_{196} \oplus s_{198} \oplus s_{203} \oplus s_{205} \oplus s_{207} \oplus s_{210} \oplus s_{211} \oplus s_{214} \oplus s_{216} \oplus s_{217} \oplus s_{218} \oplus s_{237} \oplus s_{239} \oplus s_{241} \oplus s_{245} \oplus s_{247} \oplus s_{250} \oplus s_{251} \oplus s_{284}$
111, 132	232	$s_{61} \oplus s_{76} \oplus s_{102} \oplus s_{112} \oplus s_{115} \oplus s_{117} \oplus s_{144} \oplus s_{146} \oplus s_{147} \oplus s_{151} \oplus s_{155} \oplus s_{161} \oplus s_{162} \oplus s_{168} \oplus s_{172} \oplus s_{176} \oplus s_{180} \oplus s_{186} \oplus s_{191} \oplus s_{193} \oplus s_{194} \oplus s_{197} \oplus s_{199} \oplus s_{204} \oplus s_{206} \oplus s_{208} \oplus s_{211} \oplus s_{212} \oplus s_{215} \oplus s_{217} \oplus s_{218} \oplus s_{219} \oplus s_{238} \oplus s_{240} \oplus s_{242} \oplus s_{246} \oplus s_{248} \oplus s_{251} \oplus s_{252} \oplus s_{285}$
112, 133	233	$s_{62} \oplus s_{77} \oplus s_{103} \oplus s_{113} \oplus s_{116} \oplus s_{118} \oplus s_{145} \oplus s_{147} \oplus s_{148} \oplus s_{152} \oplus s_{156} \oplus s_{162} \oplus s_{163} \oplus s_{169} \oplus s_{173} \oplus s_{177} \oplus s_{181} \oplus s_{187} \oplus s_{192} \oplus s_{194} \oplus s_{195} \oplus s_{198} \oplus s_{200} \oplus s_{205} \oplus s_{207} \oplus s_{209} \oplus s_{212} \oplus s_{213} \oplus s_{216} \oplus s_{218} \oplus s_{219} \oplus s_{220} \oplus s_{239} \oplus s_{241} \oplus s_{243} \oplus s_{247} \oplus s_{249} \oplus s_{252} \oplus s_{253} \oplus s_{286}$
113, 134	234	$s_{63} \oplus s_{78} \oplus s_{104} \oplus s_{114} \oplus s_{117} \oplus s_{119} \oplus s_{146} \oplus s_{148} \oplus s_{149} \oplus s_{153} \oplus s_{154} \oplus s_{157} \oplus s_{160} \oplus s_{163} \oplus s_{164} \oplus s_{170} \oplus s_{174} \oplus s_{178} \oplus s_{182} \oplus s_{188} \oplus s_{193} \oplus s_{195} \oplus s_{196} \oplus s_{199} \oplus s_{201} \oplus s_{206} \oplus s_{208} \oplus s_{210} \oplus s_{213} \oplus s_{214} \oplus s_{217} \oplus s_{219} \oplus s_{220} \oplus s_{221} \oplus s_{240} \oplus s_{242} \oplus s_{244} \oplus s_{248} \oplus s_{250} \oplus s_{253} \oplus s_{254} \oplus s_{287}$
114, 135	235	$s_{64} \oplus s_{79} \oplus s_{105} \oplus s_{107} \oplus s_{111} \oplus s_{115} \oplus s_{118} \oplus s_{120} \oplus s_{147} \oplus s_{149} \oplus s_{150} \oplus s_{154} \oplus s_{155} \oplus s_{158} \oplus s_{161} \oplus s_{164} \oplus s_{165} \oplus s_{171} \oplus s_{175} \oplus s_{179} \oplus s_{183} \oplus s_{189} \oplus s_{194} \oplus s_{196} \oplus s_{197} \oplus s_{200} \oplus s_{202} \oplus s_{207} \oplus s_{209} \oplus s_{211} \oplus s_{214} \oplus s_{215} \oplus s_{218} \oplus s_{220} \oplus s_{221} \oplus s_{222} \oplus s_{241} \oplus s_{243} \oplus s_{245} \oplus s_{249} \oplus s_{251} \oplus s_{254} \oplus s_{255} \oplus s_{288}$
115, 136	236	$s_{65} \oplus s_{80} \oplus s_{106} \oplus s_{108} \oplus s_{112} \oplus s_{116} \oplus s_{119} \oplus s_{121} \oplus s_{148} \oplus s_{150} \oplus s_{151} \oplus s_{155} \oplus s_{156} \oplus s_{159} \oplus s_{162} \oplus s_{165} \oplus s_{166} \oplus s_{172} \oplus s_{176} \oplus s_{180} \oplus s_{184} \oplus s_{190} \oplus s_{195} \oplus s_{197} \oplus s_{198} \oplus s_{201} \oplus s_{203} \oplus s_{208} \oplus s_{210} \oplus s_{212} \oplus s_{215} \oplus s_{216} \oplus s_{219} \oplus s_{221} \oplus s_{222} \oplus s_{223} \oplus s_{230} \oplus s_{235} \oplus s_{242} \oplus s_{244} \oplus s_{246} \oplus s_{250} \oplus s_{252} \oplus s_{255} \oplus s_{256} \oplus s_{289}$
116, 137	237	$s_{61} \oplus s_{81} \oplus s_{107} \oplus s_{109} \oplus s_{113} \oplus s_{117} \oplus s_{120} \oplus s_{122} \oplus s_{149} \oplus s_{151} \oplus s_{152} \oplus s_{156} \oplus s_{157} \oplus s_{160} \oplus s_{163} \oplus s_{166} \oplus s_{167} \oplus s_{173} \oplus s_{177} \oplus s_{181} \oplus s_{185} \oplus s_{191} \oplus s_{196} \oplus s_{198} \oplus s_{199} \oplus s_{202} \oplus s_{204} \oplus s_{209} \oplus s_{211} \oplus s_{213} \oplus s_{216} \oplus s_{217} \oplus s_{220} \oplus s_{222} \oplus s_{223} \oplus s_{224} \oplus s_{231} \oplus s_{236} \oplus s_{243} \oplus s_{245} \oplus s_{247} \oplus s_{251} \oplus s_{253} \oplus s_{256} \oplus s_{257} \oplus s_{290}$
117, 138	238	$s_{62} \oplus s_{82} \oplus s_{108} \oplus s_{110} \oplus s_{114} \oplus s_{118} \oplus s_{121} \oplus s_{123} \oplus s_{150} \oplus s_{152} \oplus s_{153} \oplus s_{157} \oplus s_{158} \oplus s_{161} \oplus s_{164} \oplus s_{167} \oplus s_{168} \oplus s_{174} \oplus s_{178} \oplus s_{182} \oplus s_{186} \oplus s_{192} \oplus s_{197} \oplus s_{199} \oplus s_{200} \oplus s_{203} \oplus s_{205} \oplus s_{210} \oplus s_{212} \oplus s_{214} \oplus s_{217} \oplus s_{218} \oplus s_{221} \oplus s_{223} \oplus s_{224} \oplus s_{225} \oplus s_{232} \oplus s_{237} \oplus s_{244} \oplus s_{246} \oplus s_{248} \oplus s_{252} \oplus s_{254} \oplus s_{257} \oplus s_{258} \oplus s_{291}$

$118, 139$	239	$s_{63} \oplus s_{83} \oplus s_{107} \oplus s_{109} \oplus s_{115} \oplus s_{119} \oplus s_{122} \oplus s_{124} \oplus s_{151} \oplus s_{153} \oplus s_{158} \oplus s_{159} \oplus s_{160} \oplus s_{162} \oplus s_{165} \oplus s_{168} \oplus s_{169} \oplus s_{175} \oplus s_{179} \oplus s_{183} \oplus s_{187} \oplus s_{193} \oplus s_{198} \oplus s_{200} \oplus s_{201} \oplus s_{204} \oplus s_{206} \oplus s_{211} \oplus s_{213} \oplus s_{215} \oplus s_{218} \oplus s_{219} \oplus s_{222} \oplus s_{224} \oplus s_{225} \oplus s_{226} \oplus s_{233} \oplus s_{238} \oplus s_{245} \oplus s_{247} \oplus s_{249} \oplus s_{253} \oplus s_{255} \oplus s_{258} \oplus s_{259} \oplus s_{292}$

Appendix B

Cube Attack on MORUS

B.1 Cube Attack on MORUS-640

Table B.1: Linear Equations for MORUS-640 with 4 Initialization Rounds (Cube Sets are chosen from the Initialization Vector)

Cube Indices	Output Index	Linear Superpoly
45, 106, 9	16	$k_0 \oplus 1$
46, 107, 10	17	$k_1 \oplus 1$
47, 108, 11	18	k_2
48, 109, 12	19	$k_3 \oplus 1$
49, 110, 13	20	k_4
50, 111, 14	21	$k_5 \oplus 1$
51, 112, 15	22	$k_6 \oplus k_{32}$
52, 113, 16	23	k_7
53, 114, 17	24	$k_8 \oplus 1$
54, 115, 18	25	$k_9 \oplus k_{35} \oplus 1$
55, 116, 19	26	$k_{10} \oplus 1$
56, 117, 20	27	k_{11}
57, 118, 21	28	k_{12}
120, 32, 86	37	k_{13}
59, 120, 23	30	k_{14}
60, 121, 24	31	k_{15}
25, 105, 59	15	k_{16}
107, 105, 50	57	k_{17}
108, 106, 51	58	k_{18}
109, 107, 52	59	k_{19}

110, 108, 53	60	k_{20}
111, 109, 54	61	$k_{21} \oplus 1$
36, 97, 0	7	$k_{23} \oplus 1$
37, 98, 1	8	$k_{24} \oplus 1$
38, 99, 2	9	$k_{25} \oplus 1$
39, 100, 3	10	$k_{26} \oplus k_{52}$
40, 101, 4	11	k_{27}
41, 102, 5	12	$k_{28} \oplus 1$
42, 103, 6	13	$k_{29} \oplus 1$
43, 104, 7	14	$k_{30} \oplus 1$
17, 121, 42	65	k_{31}
41, 17, 55	14	$k_{32} \oplus k_{90} \oplus 1$
84, 9, 123	64	k_{33}
85, 10, 124	65	$k_{34} \oplus 1$
44, 58, 82	17	$k_{35} \oplus 1$
87, 12, 126	67	k_{36}
88, 13, 127	68	$k_{37} \oplus 1$
12, 118, 87	11	$k_{38} \oplus 1$
48, 62, 86	21	$k_{39} \oplus k_{65}$
14, 120, 89	13	k_{40}
15, 121, 90	14	$k_{41} \oplus 1$
16, 122, 91	15	k_{42}
17, 123, 92	16	$k_{43} \oplus 1$
53, 35, 29	26	k_{44}
7, 13, 0	108	k_{45}
8, 14, 1	109	$k_{46} \oplus 1$
64, 16, 4	54	$k_{47} \oplus 1$
65, 17, 5	55	$k_{48} \oplus 1$
10, 89, 58	48	k_{49}
67, 19, 7	57	$k_{50} \oplus 1$
12, 91, 60	50	$k_{51} \oplus 1$
69, 21, 9	59	$k_{52} \oplus 1$
71, 23, 11	61	$k_{54} \oplus 1$
97, 78, 17	118	k_{55}
98, 79, 18	119	$k_{56} \oplus 1$
34, 48, 72	7	$k_{57} \oplus k_{83} \oplus 1$
100, 81, 20	121	k_{58}
101, 82, 21	122	k_{59}
102, 83, 22	123	$k_{60} \oplus 1$
3, 109, 78	2	$k_{61} \oplus 1$
39, 15, 53	12	$k_{62} \oplus 1$

5, 111, 80	4	$k_{63} \oplus 1$
82, 58, 107	2	$k_{64} \oplus 1$
83, 59, 108	3	$k_{65} \oplus 1$
84, 60, 109	4	k_{66}
55, 61, 120	2	k_{67}
77, 91, 115	50	$k_{68} \oplus k_{126} \oplus 1$
57, 63, 122	4	$k_{69} \oplus 1$
79, 93, 117	52	k_{70}
80, 94, 118	53	$k_{71} \oplus k_{97}$
81, 95, 119	54	$k_{72} \oplus k_{98}$
82, 64, 58	55	$k_{73} \oplus k_{99} \oplus 1$
83, 65, 59	56	$k_{74} \oplus 1$
38, 37, 64	114	k_{75}
20, 23, 38	98	k_{76}
40, 39, 66	116	k_{77}
41, 40, 67	117	k_{78}
42, 41, 68	118	k_{79}
97, 49, 37	87	k_{80}
44, 43, 70	120	k_{81}
45, 44, 71	121	k_{82}
46, 45, 72	122	$k_{83} \oplus 1$
101, 53, 41	91	k_{84}
102, 54, 42	92	$k_{85} \oplus 1$
103, 55, 43	93	$k_{86} \oplus 1$
104, 56, 44	94	$k_{87} \oplus 1$
51, 50, 77	127	$k_{88} \oplus 1$
76, 36, 51	24	$k_{89} \oplus 1$
67, 81, 105	40	$k_{90} \oplus 1$
78, 38, 53	26	$k_{91} \oplus 1$
79, 39, 54	27	k_{92}
80, 40, 55	28	k_{93}
81, 41, 56	29	$k_{94} \oplus 1$
81, 57, 106	1	k_{95}
19, 70, 24	108	k_{96}
20, 71, 25	109	$k_{97} \oplus 1$
21, 72, 26	110	$k_{98} \oplus 1$
22, 73, 27	111	k_{99}
23, 74, 28	112	k_{100}
24, 75, 29	113	$k_{101} \oplus 1$
25, 76, 30	114	$k_{102} \oplus 1$
26, 77, 31	115	k_{103}

27, 40, 29	116	$k_{104} \oplus 1$
28, 41, 30	117	$k_{105} \oplus 1$
79, 74, 68	0	k_{106}
116, 98, 92	89	k_{107}
117, 99, 93	90	$k_{108} \oplus 1$
118, 100, 94	91	k_{109}
119, 101, 95	92	$k_{110} \oplus 1$
34, 112, 73	22	$k_{111} \oplus 1$
9, 74, 27	111	$k_{112} \oplus 1$
10, 75, 28	112	$k_{113} \oplus 1$
11, 76, 29	113	$k_{114} \oplus 1$
12, 77, 30	114	$k_{115} \oplus 1$
13, 78, 31	115	$k_{116} \oplus 1$
104, 64, 79	52	k_{117}
106, 66, 81	54	k_{119}
107, 67, 82	55	k_{120}
72, 86, 110	45	$k_{95} \oplus k_{121} \oplus 1$
109, 69, 84	57	$k_{122} \oplus 1$
85, 109, 6	61	k_{123}
86, 110, 7	62	k_{124}
87, 111, 8	63	k_{125}
17, 68, 22	106	k_{126}
18, 69, 23	107	$k_{127} \oplus 1$

B.2 Cube Attack on MORUS-1280-128

Table B.2: Linear Equations for MORUS-1280-128 with 4 Initialization Rounds (Cube Sets are chosen from the Initialization Vector)

Cube Indices	Output Index	Superpoly
97, 22	3	$k_0 \oplus 1$
109, 98	128	$k_1 \oplus 1$
110, 99	129	k_2
111, 100	130	$k_3 \oplus 1$
112, 101	131	$k_4 \oplus 1$
113, 102	132	k_5
114, 103	133	$k_6 \oplus 1$
115, 104	134	$k_7 \oplus 1$
116, 105	135	$k_8 \oplus 1$
117, 106	136	k_9

118, 107	137	$k_{10} \oplus 1$
119, 108	138	$k_{11} \oplus 1$
120, 109	139	$k_{12} \oplus 1$
121, 110	140	$k_{13} \oplus 1$
122, 111	141	k_{14}
123, 112	142	k_{15}
124, 113	143	k_{16}
125, 114	144	k_{17}
126, 115	145	k_{18}
127, 116	146	$k_{19} \oplus 1$
79, 80	26	k_{20}
80, 81	27	k_{21}
81, 82	28	k_{22}
82, 83	29	k_{23}
83, 84	30	k_{24}
84, 85	31	$k_{25} \oplus 1$
85, 86	32	k_{26}
86, 87	33	k_{27}
87, 88	34	k_{28}
88, 89	35	k_{29}
89, 90	36	k_{30}
90, 91	37	k_{31}
91, 92	38	$k_{32} \oplus 1$
92, 93	39	$k_{33} \oplus 1$
93, 94	40	k_{34}
94, 95	41	k_{35}
95, 96	42	k_{36}
96, 97	43	k_{37}
97, 98	44	k_{38}
98, 99	45	k_{39}
99, 100	46	$k_{40} \oplus 1$
100, 101	47	k_{41}
101, 102	48	$k_{42} \oplus 1$
102, 103	49	k_{43}
103, 104	50	k_{44}
104, 105	51	k_{45}
105, 106	52	k_{46}
106, 107	53	k_{47}
107, 108	54	k_{48}
108, 109	55	k_{49}
109, 110	56	k_{50}

110, 111	57	$k_{51} \oplus 1$
111, 112	58	k_{52}
112, 113	59	k_{53}
113, 114	60	k_{54}
114, 115	61	k_{55}
115, 116	62	$k_{56} \oplus 1$
116, 117	63	k_{57}
102, 91	185	k_{58}
103, 92	186	k_{59}
104, 93	187	$k_{60} \oplus 1$
105, 94	188	$k_{61} \oplus 1$
106, 95	189	$k_{62} \oplus 1$
107, 96	190	$k_{63} \oplus 1$
78, 62	218	$k_{64} \oplus 1$
45, 34	64	k_{65}
46, 35	65	$k_{66} \oplus 1$
47, 36	66	k_{67}
48, 37	67	$k_{68} \oplus 1$
49, 38	68	k_{69}
50, 39	69	k_{70}
51, 40	70	k_{71}
52, 41	71	k_{72}
53, 42	72	$k_{73} \oplus 1$
6, 5	208	k_{74}
7, 6	209	k_{75}
8, 7	210	$k_{76} \oplus 1$
9, 8	211	k_{77}
10, 9	212	k_{78}
11, 10	213	k_{79}
12, 11	214	$k_{80} \oplus 1$
13, 12	215	k_{81}
14, 13	216	k_{82}
15, 14	217	k_{83}
16, 15	218	$k_{84} \oplus 1$
17, 16	219	$k_{85} \oplus 1$
18, 17	220	k_{86}
19, 18	221	k_{87}
20, 19	222	k_{88}
21, 20	223	$k_{89} \oplus 1$
22, 21	224	k_{90}
23, 22	225	k_{91}

24, 23	226	k_{92}
25, 24	227	k_{93}
26, 25	228	$k_{94} \oplus 1$
27, 26	229	k_{95}
28, 27	230	$k_{96} \oplus 1$
29, 28	231	$k_{97} \oplus 1$
30, 29	232	k_{98}
31, 30	233	k_{99}
32, 31	234	$k_{100} \oplus 1$
33, 32	235	$k_{101} \oplus 1$
34, 33	236	$k_{102} \oplus 1$
35, 34	237	k_{103}
36, 35	238	$k_{104} \oplus 1$
37, 36	239	k_{105}
38, 37	240	$k_{106} \oplus 1$
39, 38	241	k_{107}
40, 39	242	$k_{108} \oplus 1$
41, 40	243	$k_{109} \oplus 1$
42, 41	244	k_{110}
43, 42	245	$k_{111} \oplus 1$
44, 43	246	k_{112}
45, 44	247	k_{113}
46, 45	248	k_{114}
47, 46	249	$k_{115} \oplus 1$
48, 47	250	k_{116}
49, 48	251	$k_{117} \oplus 1$
34, 23	117	$k_{118} \oplus 1$
35, 24	118	k_{119}
36, 25	119	k_{120}
37, 26	120	$k_{121} \oplus 1$
38, 27	121	k_{122}
39, 28	122	k_{123}
40, 29	123	k_{124}
41, 30	124	$k_{125} \oplus 1$
42, 31	125	$k_{126} \oplus 1$
43, 32	126	k_{127}

B.3 Cube Attack on MORUS-1280-256

Table B.3: Linear Equations for MORUS-1280-256 with 4 Initialization Rounds (Cube Sets are chosen from the Initialization Vector)

Cube Indices	Output Index	Superpoly
73, 2	153	$k_0 \oplus k_{97}$
74, 3	154	$k_1 \oplus k_{98}$
102, 97	194	$k_2 \oplus 1$
103, 98	195	$k_3 \oplus 1$
104, 99	196	$k_4 \oplus 1$
105, 100	197	$k_5 \oplus 1$
106, 101	198	$k_6 \oplus 1$
107, 102	199	$k_7 \oplus 1$
68, 67	14	$k_8 \oplus 1$
109, 104	201	$k_9 \oplus 1$
83, 12	163	$k_{10} \oplus k_{107}$
84, 13	164	k_{11}
85, 14	165	k_{12}
86, 15	166	k_{13}
73, 74	20	k_{14}
74, 75	21	k_{15}
75, 76	22	$k_{16} \oplus 1$
76, 77	23	k_{17}
77, 78	24	k_{18}
78, 79	25	k_{19}
79, 80	26	k_{20}
80, 81	27	k_{21}
81, 82	28	k_{22}
82, 83	29	k_{23}
83, 84	30	k_{24}
84, 85	31	$k_{25} \oplus 1$
85, 86	32	k_{26}
86, 87	33	k_{27}
87, 88	34	k_{28}
88, 89	35	k_{29}
89, 90	36	k_{30}
90, 91	37	k_{31}
91, 92	38	$k_{32} \oplus 1$
92, 93	39	$k_{33} \oplus 1$
93, 94	40	k_{34}
94, 95	41	k_{35}

95, 96	42	k_{36}
96, 97	43	k_{37}
97, 98	44	k_{38}
98, 99	45	k_{39}
99, 100	46	$k_{40} \oplus 1$
100, 101	47	k_{41}
101, 102	48	$k_{42} \oplus 1$
102, 103	49	k_{43}
103, 104	50	k_{44}
104, 105	51	k_{45}
105, 106	52	k_{46}
106, 107	53	k_{47}
107, 108	54	k_{48}
108, 109	55	k_{49}
109, 110	56	k_{50}
110, 111	57	$k_{51} \oplus 1$
111, 112	58	k_{52}
112, 113	59	k_{53}
113, 114	60	k_{54}
114, 115	61	k_{55}
115, 116	62	$k_{56} \oplus 1$
116, 117	63	k_{57}
101, 117	127	$k_{58} \oplus 1$
61, 45	26	$k_{59} \oplus k_{92} \oplus 1$
62, 46	27	k_{60}
63, 47	28	k_{61}
71, 0	151	k_{62}
72, 1	152	k_{63}
66, 5	169	$k_{64} \oplus 1$
45, 34	64	k_{65}
68, 7	171	$k_{66} \oplus 1$
69, 8	172	k_{67}
70, 9	173	$k_{68} \oplus 1$
49, 38	68	k_{69}
50, 39	69	k_{70}
51, 40	70	k_{71}
52, 41	71	k_{72}
53, 42	72	$k_{73} \oplus 1$
54, 43	73	k_{74}
55, 44	74	k_{75}
56, 45	75	k_{76}

57, 46	76	$k_{77} \oplus 1$
58, 47	77	k_{78}
59, 48	78	k_{79}
60, 49	79	$k_{80} \oplus 1$
61, 50	80	$k_{81} \oplus 1$
62, 51	81	$k_{82} \oplus 1$
63, 52	82	k_{83}
98, 18	238	$k_{84} \oplus 1$
99, 19	239	$k_{85} \oplus 1$
100, 20	240	k_{86}
101, 21	241	$k_{184} \oplus k_{87}$
102, 22	242	$k_{88} \oplus 1$
103, 23	243	k_{89}
104, 24	244	$k_{187} \oplus k_{90}$
105, 25	245	$k_{91} \oplus 1$
106, 26	246	$k_{92} \oplus 1$
107, 27	247	k_{93}
108, 28	248	$k_{191} \oplus k_{94} \oplus 1$
109, 29	249	k_{95}
110, 30	250	k_{96}
111, 31	251	k_{97}
14, 3	97	k_{98}
15, 4	98	k_{99}
16, 5	99	$k_{100} \oplus 1$
17, 6	100	k_{101}
18, 7	101	k_{102}
19, 8	102	$k_{103} \oplus 1$
20, 9	103	$k_{104} \oplus 1$
21, 10	104	k_{105}
22, 11	105	k_{106}
23, 12	106	$k_{107} \oplus 1$
24, 13	107	k_{108}
25, 14	108	$k_{109} \oplus 1$
26, 15	109	$k_{110} \oplus 1$
27, 16	110	$k_{111} \oplus 1$
28, 17	111	$k_{112} \oplus 1$
29, 18	112	k_{113}
30, 19	113	k_{114}
31, 20	114	$k_{115} \oplus 1$
32, 21	115	$k_{116} \oplus 1$
33, 22	116	$k_{117} \oplus 1$

34, 23	117	$k_{118} \oplus 1$
35, 24	118	k_{119}
36, 25	119	k_{120}
37, 26	120	$k_{121} \oplus 1$
38, 27	121	k_{122}
39, 28	122	k_{123}
40, 29	123	k_{124}
41, 30	124	$k_{125} \oplus 1$
42, 31	125	$k_{126} \oplus 1$
43, 32	126	k_{127}
6, 97	86	$k_{128} \oplus 1$
7, 98	87	$k_{129} \oplus 1$
8, 99	88	k_{130}
9, 100	89	$k_{131} \oplus 1$
10, 101	90	$k_{132} \oplus 1$
11, 102	91	k_{133}
12, 103	92	$k_{134} \oplus 1$
13, 104	93	$k_{135} \oplus 1$
14, 105	94	$k_{136} \oplus 1$
15, 106	95	k_{137}
16, 107	96	$k_{138} \oplus 1$
17, 108	97	$k_{139} \oplus 1$
18, 109	98	$k_{140} \oplus 1$
19, 110	99	$k_{141} \oplus 1$
20, 111	100	k_{142}
21, 112	101	k_{143}
22, 113	102	k_{144}
23, 114	103	k_{145}
24, 115	104	k_{146}
25, 116	105	$k_{147} \oplus 1$
26, 117	106	$k_{148} \oplus 1$
27, 118	107	k_{149}
28, 119	108	k_{150}
29, 120	109	k_{151}
30, 121	110	$k_{152} \oplus 1$
31, 122	111	k_{153}
32, 123	112	$k_{154} \oplus 1$
33, 124	113	k_{155}
34, 125	114	$k_{156} \oplus 1$
35, 126	115	k_{157}
36, 127	116	$k_{158} \oplus 1$

75, 64	158	k_{159}
76, 65	159	$k_{160} \oplus 1$
77, 66	160	k_{161}
78, 67	161	$k_{162} \oplus 1$
79, 68	162	$k_{163} \oplus 1$
80, 69	163	k_{164}
81, 70	164	$k_{165} \oplus 1$
82, 71	165	$k_{166} \oplus 1$
83, 72	166	k_{167}
84, 73	167	k_{168}
85, 74	168	$k_{169} \oplus 1$
86, 75	169	k_{170}
87, 76	170	k_{171}
88, 77	171	k_{172}
89, 78	172	k_{173}
90, 79	173	$k_{174} \oplus 1$
91, 80	174	$k_{175} \oplus 1$
92, 81	175	$k_{176} \oplus 1$
93, 82	176	$k_{177} \oplus 1$
94, 83	177	$k_{178} \oplus 1$
95, 84	178	$k_{179} \oplus 1$
96, 85	179	k_{180}
97, 86	180	$k_{181} \oplus 1$
98, 87	181	k_{182}
99, 88	182	k_{183}
100, 89	183	$k_{184} \oplus 1$
101, 90	184	k_{185}
102, 91	185	k_{186}
103, 92	186	k_{187}
104, 93	187	$k_{188} \oplus 1$
105, 94	188	$k_{189} \oplus 1$
106, 95	189	$k_{190} \oplus 1$
107, 96	190	$k_{191} \oplus 1$
37, 32	129	$k_{193} \oplus 1$
38, 33	130	$k_{194} \oplus 1$
39, 34	131	$k_{195} \oplus 1$
40, 35	132	$k_{196} \oplus 1$
0, 1	203	$k_{197} \oplus 1$
1, 2	204	k_{198}
2, 3	205	k_{199}
3, 4	206	$k_{200} \oplus 1$

4, 5	207	k_{201}
5, 6	208	k_{202}
6, 7	209	k_{203}
7, 8	210	$k_{204} \oplus 1$
8, 9	211	k_{205}
9, 10	212	k_{206}
10, 11	213	k_{207}
11, 12	214	$k_{208} \oplus 1$
12, 13	215	k_{209}
13, 14	216	k_{210}
14, 15	217	k_{211}
15, 16	218	$k_{212} \oplus 1$
16, 17	219	$k_{213} \oplus 1$
17, 18	220	k_{214}
18, 19	221	k_{215}
19, 20	222	k_{216}
20, 21	223	$k_{217} \oplus 1$
21, 22	224	k_{218}
22, 23	225	k_{219}
23, 24	226	k_{220}
24, 25	227	k_{221}
25, 26	228	$k_{222} \oplus 1$
26, 27	229	k_{223}
27, 28	230	$k_{224} \oplus 1$
28, 29	231	$k_{225} \oplus 1$
29, 30	232	k_{226}
30, 31	233	k_{227}
31, 32	234	$k_{228} \oplus 1$
32, 33	235	$k_{229} \oplus 1$
33, 34	236	$k_{230} \oplus 1$
34, 35	237	k_{231}
35, 36	238	$k_{232} \oplus 1$
36, 37	239	k_{233}
37, 38	240	$k_{234} \oplus 1$
38, 39	241	k_{235}
39, 40	242	$k_{236} \oplus 1$
40, 41	243	$k_{237} \oplus 1$
41, 42	244	k_{238}
42, 43	245	$k_{239} \oplus 1$
43, 44	246	k_{240}
44, 45	247	k_{241}

45, 46	248	k_{242}
46, 47	249	$k_{243} \oplus 1$
47, 48	250	k_{244}
48, 49	251	$k_{245} \oplus 1$
49, 50	252	k_{246}
50, 51	253	k_{247}
51, 52	254	$k_{248} \oplus 1$
52, 53	255	k_{249}

Bibliography

[1] Menezes, A., Oorshot, P., and Vanstone, S. *Handbook of Applied Cryptography*, CRC Press LLC, 1997.

[2] Daemen, J., Rijmen, V. The Design of Rijndael: AES –The Advanced Encryption Standard. *Information Security and Cryptography*, Springer, 2002.

[3] Dworkin, M. Recommendation for Block Cipher Modes of Operation. *NIST Special Publication 800-38A, 2001 Edition.* Available from `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf`, Accessed 07 September 2017.

[4] Bernstein, D. J. The Salsa20 Family of Stream Ciphers. In Robshaw, M., Billet, O. (eds.) *New Stream Cipher Designs.* Vol 4986, pp. 84-97. Springer Berlin Heidelberg, 2008.

[5] De Canniére, C. and Preneel, B. Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. In Katsikas, S.K., López, J., Backes, M., Gritzalis. S., Preneel, B. (eds.) *Information Security - ISC 2006.* Vol. 4176, pp. 171 - 186. Springer Berlin Heidelberg, 2006.

[6] Bellare, M., Canetti, R., Krawczyk, H. Message Authentication Using Hash Functions: The HMAC construction. *RSA Laboratories' CryptoBytes*, Vol 2, No. 1, Spring 1996.

[7] Bernstein, D. J. The Poly1305-AES Message-Authentication Code. In Gilbert, H., Handschuh, H. (eds.) *Fast Software Encryption - FSE 2005.* Vol 3557, pp. 32-49. Springer Berlin Heidelberg, 2005.

[8] ISO/IEC 9797, Data Cryptographic Techniques - Data Integrity Mechanism Using A Cryptographic Check Function Employing A Block Cipher Algorithm, 1989.

[9] ETSI/SAGE Specification: Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification. Technical Report, ETSI, Version 1.6, June 2011. Available from `http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/eea3eia3zucv16.pdf`, Accessed 12 September 2014.

[10] Bellovin, S. Problem Areas for the IP Security Protocols. In *Sixth USENIX Security Symposium.* pp. 1-16. 1996.

[11] Vaudenay, S. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS..... In Knudsen, L.R. (ed.) *Advances in Cryptology - EUROCRYPT 2002.* Vol 2332, pp. 534-545. Springer Berlin Heidelberg, 2002.

[12] Black, J. and Urtubia, H. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In Boneh, D. (ed.) *Eleventh USENIX Security Symposium*. pp. 327-338. 2002.

[13] Bellare, M. and Namprempre, C. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. In Okamoto, T. (ed.) *Advances in Cryptology - ASIACRYPT 2000*. Vol. 1976, pp. 531-545. Springer Berlin Heidelberg, 2000.

[14] Katz, J. and Yung, M. Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation. In Schneier, B. (ed.) *Fast Software Encryption - FSE 2000*. Vol. 1978, pp. 284-299. Springer Berlin Heidelberg, 2001.

[15] Bellare, M., Rogaway, P., Wagner, D. The EAX Mode of Operation. In Roy B., Meier W. (eds.) *Fast Software Encryption - FSE 2004*. Vol 3017, pp. 389-407. Springer Berlin Heidelberg, 2004.

[16] Jutla, C. S. Encryption Modes with Almost Free Message Integrity. In Pfitzmann, B. (ed.) *Advances in Cryptology - EUROCRYPT 2001*. Vol. 2045, pp. 529-544. Springer Berlin Heidelberg, 2001.

[17] Rogaway, P., Bellare, M., Black, J., and Krovetz, T. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In Samarati, P. (ed.) *ACM Conference on Computer and Communications Security*. pp. 196-205. ACM Press, 2001.

[18] McGrew, D. and Viega, J., The Galois/Counter Mode of Operation (GCM). Available from `http://csrc.nist.gov/groups/ST/toolkit/BCM/document/pro-posedmodes/gcm/gcm-spec.pdf`, Accessed 10 September 2014.

[19] ETSI/SAGE Specification: Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2, Document 2: SNOW 3G Specification. Technical Report, ETSI, Version 1.1, September 2006. Available from `http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/snow3gspec.pdf`, Accessed 12 September 2014.

[20] Whiting, D., Schneier, B., Lucks, S., and Muller, F. Phelix: Fast Encryption and Authentication in A Single Cryptographic Primitive. eStream Project. Available from `http://www.ecrypt.eu.org/stream/p2ciphers/phelix/phelix_p2.pdf`, Accessed 12 September 2014.

[21] Hawkes, P., McDonald, C., Paddon, M., Rose, G. G., and Miriam, W. V. Specification for NLSv2. In Robshaw, M. and Olivier, B. (eds.) *New Stream Cipher Designs: The eSTREAM Finalists*. Vol. 4986, pp. 57-68. Springer Berlin Heidelberg, 2008.

[22] Hawkes, P. and Rose, G. Primitive Specification for SOBER-128. IACR ePrint Archive. Technical Report 2003/081. Available from `http://eprint.iacr.org/2003/081`, Accessed 12 September 2014.

[23] Braeken, A., Lano, J., Mentens, N., Preneel, B., and Verbauwhede, I. SFINKS: A Synchronous Stream Cipher for Restricted Hardware Environments. *Symmetric Key Encryption Workshop*. Available from `http://cr.yp.to/streamciphers/sfinks/desc.pdf`, Accessed 12 September 2014.

[24] Ågren, M., Hell, M., Johansson, T., and Meier, W. Grain-128a: A New Version of Grain-128 With Optional Authentication. *International Journal of Wireless and Mobile Computing*, 5(1), pp. 48-59. 2011.

[25] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. Available from: `http://competitions.cr.yp.to/index.html`, Accessed 20 September 2014.

[26] Advanced Encryption Standard (AES) Development Effort. Available from: `http://csrc.nist.gov/archive/aes/index2.html#overview`, Accessed 14 September 2017.

[27] SHA-3 Competition. Available from: `http://csrc.nist.gov/groups/ST/hash/sha-3/`, Accessed 14 September 2017.

[28] Al-Mashrafi, M. Analysis of Stream Cipher Based Authenticated Encryption Schemes. PhD Thesis. Queensland University of Technology, 2012.

[29] Wu, H. ACORN: A Lightweight Authenticated Cipher (v1). CAESAR Competition. Available from `http://competitions.cr.yp.to/round1/acornv1.pdf`, Accessed 29 May 2015.

[30] Wu, H. ACORN: A Lightweight Authenticated Cipher (v2). CAESAR Competition. Available from `https://competitions.cr.yp.to/round2/acornv2.pdf`, Accessed 10 September 2015.

[31] Wu, H. ACORN: A Lightweight Authenticated Cipher (v3). CAESAR Competition. Available from `https://competitions.cr.yp.to/round3/acornv3.pdf`, Accessed 10 July 2017.

[32] Nikolić, I. Tiaoxin–346: VERSION 1.0. CAESAR Competition. Available from `http://competitions.cr.yp.to/round1/tiaoxinv1.pdf`, Accessed 29 May 2015.

[33] Nikolić, I. Tiaoxin–346: VERSION 2.0. CAESAR Competition. Available from `https://competitions.cr.yp.to/round2/tiaoxinv2.pdf`, Accessed 26 Aug 2017.

[34] Wu, H. and Huang, T. The Authenticated Cipher MORUS (v1). CAESAR Competition. Available from `https://competitions.cr.yp.to/round1/morusv1.pdf`, Accessed 23 February 2017.

[35] Wu, H. and Huang, T. The Authenticated Cipher MORUS (v2). CAESAR Competition. Available from `https://competitions.cr.yp.to/round3/morusv2.pdf`, Accessed 23 February 2017.

[36] Salam, M. I., Wong, K. K-H., Bartlett, H., Simpson, L., Dawson, E., Pieprzyk, J. Finding State Collisions in the Authenticated Encryption Stream Cipher ACORN. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2016)*. ACM, 2016.

[37] Salam, M. I., Bartlett, H., Dawson, E., Pieprzyk, J., Simpson, L., Wong, K. K. Investigating Cube Attacks on the Authenticated Encryption Stream Cipher ACORN. In Batten L., Li G. (eds.) *Applications and Techniques in Information Security - ATIS 2016.* Vol 651, pp. 15-26. Springer Singapore, 2016.

[38] Dey, P., Rohit, R.S., Sarkar, S., Adhikari, A. Differential Fault Analysis on Tiaoxin and AEGIS Family of Ciphers. In Mueller, P., Thampi, S., Alam, B. M., Ko, R., Doss, R., Alcaraz, C. J. (eds.) *Security in Computing and Communications - SSCC 201.* vol 625., pp. 74-86. Springer Singapore, 2016.

[39] Salam, I., Mahri, H., Simpson, L., Bartlett, H., Dawson, E., Wong, K. K-H. Fault Attacks on Tiaoxin-346. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2018).* ACM, 2018.

[40] Dwivedi, A., Morawiecki, P., Wójtowicz, S. Differential and Rotational Cryptanalysis of Round-reduced MORUS. In *14th International Conference on Security and Cryptography (SECRYPT-2017).* pp. 275-284. 2017.

[41] Salam, I., Simpson, L., Bartlett, H., Dawson, E., Pieprzyk, J., Wong, K. K. Investigating Cube Attacks on the Authenticated Encryption Stream Cipher MORUS. In *Proceedings of the IEEE Trustcom/BigDataSE/ICESS.* pp. 961-966. IEEE Computer Society, 2017.

[42] Salam, I., Simpson, L., Bartlett, H., Dawson, E., Wong, K. K. Fault Attacks on the Authenticated Encryption Stream Cipher MORUS. *Cryptography*, 2(1), 2018.

[43] Bellare, M. and Rogaway, P. Encode-then-Encipher Encryption: How to Exploit Nonces Or Redundancy in Plaintexts for Efficient Cryptography. In Okamoto, T. (ed.) *Advances in Cryptology - ASIACRYPT 2000.* Vol. 1976, pp. 317-330. Springer Berlin Heidelberg, 2000.

[44] Rogaway, P. Authenticated Encryption with Associated Data. In Atluri, V. (ed.) *9th ACM Conference on Computer and Communications Security.* pp. 98-107. ACM, 2002.

[45] Campbell, C. M. Design and Specification of Cryptographic Capabilities. In Brandstad, D.K. (ed.) *Computer Security and the Data Encryption Standard.* pp. 54-66. 1978.

[46] Gligor, V. D. and Lindsay, B. G. Object Migration and Authentication. *IEEE Transactions on Software Engineering.* 5(6), pp. 607-611, 1979.

[47] Jueneman, R., Meyer, C., and Matyas, S. Message Authentication with Manipulation Detection Codes. In *IEEE Symposium on Security and Privacy.* pp. 33-54. IEEE Computer Society Press, 1984.

[48] Preneel, B. Cryptographic Primitives for Information Authentication - State of the Art. In Preneel, B. and Rijmen, V. (eds.) *State of the Art in Applied Cryptography.* Vol. 1528, pp. 49-104. Springer Berlin Heidelberg, 1998.

[49] Rogaway, P. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Lee, P.J. (ed.) *Advances in Cryptology - ASIACRYPT 2004.* Vol. 3329, pp. 16-31. Springer Berlin Heidelberg, 2004.

[50] Krovetz, T. and Rogaway, P. The Software Performance of Authenticated-Encryption Modes. In Joux, A. (ed.) *Fast Software Encryption*. Vol. 6733, pp. 306-327. Springer Berlin Heidelberg, 2011.

[51] Whiting, D., Housley, R., and Ferguson, N. Counter with CBC-MAC (CCM). Available from http://tools.ietf.org/html/rfc3610, Accessed 10 September 2014.

[52] Gligor, V. D. and Donescu, P. Fast Encryption and Authentication: XCBC Encryption and XECB Authentication Modes. In Matsui, M. (ed.) *Fast Software Encryption*. Vol. 2355, pp. 92-108. Springer Berlin Heidelberg, 2002.

[53] Rose, G. Combining Message Authentication and Encryption. In *AUUG 2003 - Open Standards, Open Source, Open Computing*. pp. 77-85. 2003.

[54] Krawczyk, H. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In Kilian, J. (ed.) *Advances in Cryptology - CRYPTO 2001*. Vol. 2139, pp. 310-331. Springer Berlin Heidelberg, 2001.

[55] Black, J. Authenticated Encryption. In Tilborg, H.C., and Jajodia, S. (eds.) *Encyclopedia of Cryptography and Security*. Springer Berlin Heidelberg, 2005.

[56] Moreau, T. The Frogbit Cipher, A Data Integrity Algorithm. eStream Project. Report 2005/001. Available from `http://www.ecrypt.eu.org/stream/frogbit.html`, Accessed 12 September 2014.

[57] O'Neil, S., Gittins, B., and Landman, H. VEST, Hardware-Dedicated Stream Ciphers. IACR ePrint Archive. Report 2005/413. Available from `http://eprint.iacr.org/2005/413`, Accessed 12 September 2014.

[58] Hawkes, P., Paddon, M., Rose, G. G., and Miriam, W. V. Primitive Specification for SSS. eStream Project. Available from `http://www.ecrypt.eu.org/stream/ciphers/sss/sss.pdf`, Accessed 12 September 2014.

[59] Ferguson, N., Whiting, D., Schneier, B., Kelsey, J., Lucks, S., and Kohno, T. Helix: Fast Encryption and Authentication In A Single Cryptographic Primitive. In Johansson, T. (ed.) *Fast Software Encryption - FSE 2003*. Vol. 2887, pp. 330-346. Springer Berlin Heidelberg, 2003.

[60] The eStream Project. Available from: `http://www.ecrypt.eu.org/stream/project.html`, Accessed 20 September 2014.

[61] Bernstein, D. J. Cycle Count for Authenticated Encryption. *Workshop Record of SASC 2007: The state of the art of stream ciphers*. eStream Report 2007/015. Available from `http://cr.yp.to/streamciphers/aecycles-20070118.pdf`, Accessed 12 September 2014.

[62] Sarkar, P. Modes of Operations for Encryption and Authentication Using Stream Ciphers Supporting An Initialisation Vector. *Cryptography and Communications*, 6(3), pp. 189-231, 2014.

[63] Sarkar, P. On Authenticated Encryption Using Stream Ciphers Supporting an Initialisation Vector. IACR Cryptology ePrint Archive. Report 2011/ 299. Available from http://eprint.iacr.org/2011/299.pdf, Accessed 5 September 2014.

[64] Abed, F., Forler, C., and Lucks, S. General Overview of the First-Round CAESAR Candidates for Authenticated Encryption. IACR ePrint Archive. 2014/792. Available from https://eprint.iacr.org/2014/792.pdf, Accessed 25 May 2015.

[65] Wu, H. and Preneel, B. AEGIS: A Fast Authenticated Encryption Algorithm (v1). CAESAR Competition. Available from http://competitions.cr.yp.to/round1/aegisv1.pdf, Accessed 29 May 2015.

[66] Harris, S. The Enchilada authenticated ciphers, v1. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/enchiladav1.pdf, Accessed 29 May 2015.

[67] Taylor, C. The Calico Family of Authenticated Ciphers. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/calicov8.pdf, Accessed 29 May 2015.

[68] Chaza, F., McDonald, C., and Avanzi, R. FASER v1: Authenticated Encryption in A Feedback Shift Register. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/faserv1.pdf, Accessed 29 May 2015.

[69] Henricksen, M., Kiyommoto, S., and Lu, J. The HKC Authenticated Stream Cipher (Ver.1). CAESAR Competition. Available from http://competitions.cr.yp.to/round1/hkcv1.pdf, Accessed 29 May 2014.

[70] Krovetz, T. HS1-SIV (v1). CAESAR Competition. Available from http://competitions.cr.yp.to/round1/hs1sivv1.pdf, Accessed 29 May 2015.

[71] Ye, D., Wang, P., Hu, L., Wang, L., Xie, Y., Sun, S., and Wang, P. PAES v1: Parallelizable Authenticated Encryption Schemes based on AES Round Function. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/paesv1.pdf, Accessed 29 May 2015.

[72] Ye, D., Wang, P., Hu, L., Wang, L., Xie, Y., Sun, S., and Wang, P. PANDA v1. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/pandav1.pdf, Accessed 29 May 2015.

[73] Vuckovac, R. Raviyoyla v1. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/raviyoylav1.pdf, Accessed 29 May 2015.

[74] Zhang, B., Shi, Z., Xu, C., Yao, Y., and Li, Z. Sablier v1. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/sablierv1.pdf, Accessed 29 May 2015.

[75] Chakraborti, A. and Nandi, M. TriviA-ck-v1. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/triviackv1.pdf, Accessed 29 May 2015.

[76] Maxwell, P. Wheesht: an AEAD stream cipher. CAESAR Competition. Available from http://competitions.cr.yp.to/round1/wheeshtv03.pdf, Accessed 29 May 2015.

[77] Schneier, B. *Applied Cryptography*, John Wiley & Sons, 1996.

[78] Mantin, I. and Shamir, A. A Practical Attack on Broadcast RC4. In Matsui, M. (ed.) *Fast Software Encryption - FSE 2001*. Vol. 2355, pp. 152-164. Springer Berlin Heidelberg, 2002.

[79] Shannon, C. E. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4), pp. 656-715, 1949.

[80] Katz, J. and Yung, M. Complete Characterization of Security Notions for Probabilistic Private-Key Encryption. In *32nd Annual ACM Symposium on Theory of Computing*. pp. 245-254. ACM, 2000.

[81] Dolev, D., Dwork, C., and Naor, M. Non-Malleable Cryptography. In *23rd Annual ACM Symposium on Theory of Computing*. pp. 542-552. ACM, 1991.

[82] Goldwasser, S. and Micali, S. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2), pp. 270-299, 1984.

[83] Yuval, G. How to Swindle Rabin. *Cryptologia*, 3(3), pp. 187-191, 1979.

[84] Biham, E. and Shamir, A. Differential Cryptanalysis of DES-like Cryptosystems. In Menezes, A.J. and Vanstone, S.A. (eds.) *Advances in Cryptology - CRYPTO 1990*. Vol. 537, pp. 2-21. Springer Berlin Heidelberg, 1990.

[85] Siegenthaler, T. Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications (Corresp.). *IEEE Transactions on Information Theory*, 30(5), pp. 776-780, 1984.

[86] Siegenthaler, T. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, C-34(1), pp. 81-85, 1985.

[87] Biryukov, A., Wagner, D. Slide attacks, In Knudsen, L. (ed.) *Fast Software Encryption - FSE 1999*. Vol. 1636, pp. 245-259. Springer Berlin Heidelberg, 1999.

[88] Courtois, N. T. and Pieprzyk, J. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Zheng, Y. (ed.) *Advances in Cryptology - ASIACRYPT 2002*. Vol. 2501, pp. 267-287. Springer Berlin Heidelberg, 2002.

[89] Courtois, N. T. and Meier, W. Algebraic Attacks on Stream Ciphers with Linear Feedback. In Biham, E. (ed.) *Advances in Cryptology - EUROCRYPT 2003*. Vol. 2656, pp. 345-359. Springer Berlin Heidelberg, 2003.

[90] Courtois, N.T. Fast Algebraic Attacks on Stream Ciphers with Linear Feedback. In Boneh, D. (ed.) *Advances in Cryptology - CRYPTO 2003*. Vol. 2729, pp. 176-194. Springer Berlin Heidelberg, 2003.

[91] Vielhaber, M. Breaking One.Fivium by AIDA an Algebraic IV Differential Attack. IACR ePrint Archive. 2007/413. Available from `https://eprint.iacr.org/2007/413.pdf`, Accessed 28 May 2016.

[92] Lai, X. Higher Order Derivatives and Differential Cryptanalysis. In Blahut, R.E., Costello, D.J., Maurer, U. and Mittelholzer, T. (eds.) *Communications and Cryptography: Two Sides of One Tapestry*. Vol. 276, pp. 227-233. Springer US, 1994.

[93] Dinur, I. and Shamir, A. Cube Attacks on Tweakable Black Box Polynomials. In Joux, A. (ed.) *Advances in Cryptology - EUROCRYPT 2009*. Vol. 5479, pp. 278-299. Springer Berlin Heidelberg, 2009.

[94] Khovratovich D., Nikolić I., Rotational Cryptanalysis of ARX. In Hong, S., Iwata, T. (eds.) *Fast Software Encryption - FSE 2010*. Vol. 6147, pp. 333-346. Springer Berlin Heidelberg, 2010.

[95] Kocher, P., Jaffe, J., Jun, B., and Rohatgi, P. Introduction to Differential Power Analysis. *Journal of Cryptographic Engineering*, 1(1), pp. 5-27, 2011.

[96] Teo, S. G., Al-Hamdan, A., Bartlett, H., Simpson, L., Wong, K.K-H., Dawson, E. State Convergence in the Initialisation of Stream Ciphers. In Parampalli, U. and Hawkes, P. (eds.) *Australasian Conference on Information Security and Privacy - ACISP 2011*. Vol. 6812. pp. 75-88. Springer Berlin Heidelberg, 2011.

[97] Mroczkowski, P. and Szmidt, J. The Cube Attack on Courtois Toy Cipher. IACR ePrint Archive. 2009/497. Available from `https://eprint.iacr.org/2009/497.pdf`, Accessed 17 June 2016.

[98] Dinur, I. and Shamir, A. Breaking Grain-128 with Dynamic Cube Attacks. In Joux, A. (ed.) *Fast Software Encryption - FSE 2011*. Vol. 6733, pp. 167 - 187. Springer Berlin Heidelberg, 2011.

[99] Sarkar, S., Maitra , S., and Baksi, A. Observing Biases in the State: Case Studies With Trivium and Trivia-SC. *Designs, Codes and Cryptography*, pp. 1-25, 2016.

[100] Blum, M., Luby, M., and Rubinfeld, R. Self-testing/Correcting With Applications to Numerical Problems. *Journal of Computer and System Sciences*, 47, pp. 579-595, 1993.

[101] Aumasson, J. P., Dinur, I., Meier, W., and Shamir, A. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In Dunkelman, O. (ed.) *Fast Software Encryption - FSE 2009*. Vol. 5665, pp. 1-22. Springer Berlin Heidelberg, 2009.

[102] Boneh, D., DeMillo, R.A., Lipton, R.J. On the Importance of Checking Cryptographic Protocols for Faults, In Fumy, W. (ed.) *Advances in Cryptology - EUROCRYPT 1997*. Vol. 1233, pp. 37-51. Springer Berlin Heidelberg, 1997.

[103] Barenghi, A., Breveglieri, L., Koren, I., Naccache, D. Fault injection attacks on cryptographic devices: theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11), pp. 3056-3076, 2012.

[104] Biham, E. New Types of Cryptanalytic Attacks using Related Keys. *J. Cryptology*, 7(4), 229-246, 1994.

[105] Bartlett, H., Al-Mashrafi, M., Simpson, L., Dawson, E., and Wong, K. A General Model for MAC Generation Using Direct Injection. In Kutylowski, M. and Yung, M. (eds.) *Information Security and Cryptology - INSCRYPT 2012*. Vol. 7763, pp. 198-215. Springer Berlin Heidelberg, 2012.

[106] Al-Mashrafi, M., Bartlett, H., Dawson, E., Simpson, L., and Wong, K. Indirect Message Injection for MAC Generation. *Journal of Mathematical Cryptology*, 7(3), pp. 253-277, 2013.

[107] Al-Mashrafi, M., Bartlett, H., Simpson, L., Dawson, E., and Wong, K. Analysis of Indirect Message Injection for MAC Generation Using Stream Ciphers. In Susilo, W., Mu, Y., and Seberry, J. (eds.) *Australasian Conference on Information Security and Privacy - ACISP 2012*. Vol. 7372, pp. 138-151. Springer Berlin Heidelberg, 2012.

[108] Banik, S., Maitra, S., and Sarkar, S. A Differential Fault Attack on Grain-128a Using MACs. In Bogdanov, A. and Sanadhya, S. (eds.) *Security, Privacy, and Applied Cryptography Engineering - SPACE 2012*. Vol 7644, pp. 111-125. Springer Berlin Heidelberg, 2012.

[109] Courtois, N. T. Cryptanalysis of Sfinks. In Won, D.H., and Kim, S. (eds.) *Information Security and Cryptology - ICISC 2005*. Vol. 3935, pp. 261-269. Springer Berlin Heidelberg, 2005.

[110] Watanabe, D. and Furuya, S. A MAC Forgery Attack on SOBER-128. In Roy, B. and Meier, W. (eds.) *Fast Software Encryption - FSE 2004*. Vol. 3017, pp. 472-482. Springer Berlin Heidelberg, 2004.

[111] Cho, J. Y. and Pieprzyk, J. Distinguishing Attack on SOBER-128 With Linear Masking. In Batten, L.M., and Safavi-Naini, R. (eds.) *Australasian Conference on Information Security and Privacy - ACISP 2006*. Vol. 4058, pp. 29-39. Springer Berlin Heidelberg, 2006.

[112] Daemen, J., Lano, J., and Preneel, B. Chosen Ciphertext Attack on SSS. In *SASC Workshop Record - Stream Ciphers Revisited*. pp. 45-51. 2006.

[113] Al-Mashrafi, M., Wong, K., Simpson, L., Bartlett, H., and Dawson, E. Algebraic Analysis of the SSS Stream Cipher. In *4th International Conference on Security of Information and Networks - SIN 2011*. pp. 199-204. ACM Press, 2011.

[114] Cho, J. Y. and Pieprzyk, J. Algebraic Attacks on SOBER-t32 and SOBER-t16 Without Stuttering. In Roy, B. and Meier, W. (eds.) *Fast Software Encryption - FSE 2004*. Vol. 3017, pp. 49-64. Springer Berlin Heidelberg, 2004.

[115] Cho, J. Y. and Pieprzyk, J. Multiple Modular Additions and Modular Crossword Puzzle Attack on NLSv2. In Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) *Information Security - ISC 2007*. Vol. 4779, pp. 230-248. Springer Berlin Heidelberg, 2007.

[116] Wu, H., Huang, T., Nguyen, P. H., Wang, H., and Ling, S. Differential Attacks Against Stream Cipher ZUC. In Wang, X. and Sako, K. (eds.) *Advances in Cryptology - ASIACRYPT 2012*. Vol. 7658, pp. 262-277. Springer Berlin Heidelberg, 2012.

[117] ETSI/SAGE Technical Report: Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report. Technical Report, ETSI, Version 2.0, September 2011. Available from `http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/EEA3_EIA3_Design_Evaluation_v2_0.pdf`, Accessed 12 September 2014.

[118] Fuhr, T., Gilbert, H., Reinhard, J.-R., and Videau, M. Analysis of the Initial and Modified Versions of the Candidate 3GPP Integrity Algorithm 128-EIA3. In Miri, A. and Vaudenay, S. (eds.) *Selected Areas in Cryptography - SAC 2011*. Vol. 7118, pp. 230-242. Springer Berlin Heidelberg, 2012.

[119] Wu, H. and Preneel, B. Differential-Linear Attacks Against the Stream Cipher Phelix. In Biryukov, A. (ed.) *Fast Software Encryption - FSE 2007*. Vol. 4593, pp. 87-100. Springer Berlin Heidelberg, 2007.

[120] Debraize, B. and Corbella, I. M. Fault Analysis of the Stream Cipher SNOW 3G. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*. pp. 103-110. IEEE Computer Society, 2009.

[121] Zhang, B., Shi, Z., Xu, C., Yao, Y., and Li, Z. Sablier v1.1. Cryptographic Competitions Mailiing List. Available from `https://groups.google.com/forum/#!searchin/crypto-competitions/sablier/crypto-competitions/54bwj4NCcaw/1sJOYznnBJAJ`, Accessed 10 June 2015.

[122] Liu, M. and Lin, D. Cryptanalysis of Lightweight Authenticated Cipher Acorn. Cryptographic Competitions Mailing List. Available from `https://groups.google.com/forum/#!topic/crypto-competitions/2mrDnyb9hfM`, Accessed 29 May 2015.

[123] Chaigneau, C., Fuhr, T., Gilbert, H. Full Key-Recovery on ACORN in Nonce-Reuse and Decryption-Misuse Settings. Cryptographic Competitions Mailing List. Available from `https://groups.google.com/forum/#!to-pic/crypto-competitions/RTtZvFZay7k`, Accessed 10 August 2015.

[124] Lafitte, F., Lerman, L., Markowitch, O., Heule, D.V. SAT-based Cryptanalysis of ACORN. IACR ePrint Archive. 2016/521. Available from `https://eprint.iacr.org/2016/521.pdf`, Accessed 10 July 2017.

[125] Dwivedi, A. D., Klouček, M., Morawiecki, P., Nikolić, I., Pieprzyk, J., and Wójtowicz, S. SAT-based Cryptanalysis of Authenticated Ciphers from the CAESAR Competition. IACR ePrint Archive. 2016/1053. Available from `http://eprint.iacr.org/2016/1053.pdf`, Accessed 03 March 2017.

[126] Todo, Y., Isobe, T., Hao, Y., Meier, W. Cube Attacks on Non-Blackbox Polynomials Based on Division Property. IACR Cryptology ePrint Archive. 2017/306. Available from `http://eprint.iacr.org/2017/306.pdf`, Accessed 10 July 2017.

[127] Dey, P., Rohit, R.S., Adhikari, A. Full Key Recovery of ACORN With A Single Fault. *Journal of Information Security and Applications*, 29, 57-64,(2016).

[128] Roy, D. and Mukhopadhyay, S. Some results on ACORN. IACR Cryptology ePrint Archive. 2016/1132. Available from `http://eprint.iacr.org/2016/1132.pdf`, Accessed 10 July 2017.

[129] Jiao, L., Zhang, B., Wang, M. Two Generic Methods of Analyzing Stream Ciphers. In Lopez, J., Mitchell, C. (eds.) *Information Security - ISC 2015*. Vol 9290, pp 379-396. Springer International Publishing, 2015.

[130] Minaud, B. Linear Biases in AEGIS Keystream. In Joux, A. and Youssef, A. (eds.) *Selected Areas in Cryptography - SAC 2014*. Vol 8781, pp. 290-305. Springer International Publishing, 2014.

[131] Dobraunig, C., Eichlseder, M., Mendel, F., and Schlaffer, M. Forgery and Key Recovery Attacks for Calico. Available from `http://ascon.iaik.tugraz.at/files/analysis_calico.pdf`, Accessed 29 May 2015.

[132] Xu, C., Zhang, B., and Feng, D. Linear Cryptanalysis of FASER128/256 and TriviA-ck. In Meier, W. and Mukhopadhyay, D. (eds.) *Progress in Cryptology - INDOCRYPT*. Vol. 8885, pp. 237-254. Springer International Publishing, 2014.

[133] Feng, X. and Zhang, F. A Realtime Key Recovery Attack on the Authenticated Cipher FASER128. IACR ePrint Archive. 2014/258. Available from `https://eprint.iacr.org/2014/258.pdf`, Accessed 29 May 2015.

[134] Saarinen, M.-J. HKC Authentication. Cryptographic Competitions Mailing List. Available from `https://groups.google.com/forum/#!topic/crypto-competitions/wtR0d-M5auw`, Accessed 29 May 2015.

[135] Mileva, A., Dimitrova, V., and Velichkov, V. Analysis of the Authenticated Cipher MORUS (v1). In Pasalic, E. and Knudsen, L.R. (eds.) *Cryptography and Information Security in the Balkans*. Vol. 9540, pp. 45-59. Springer International Publishing, 2015.

[136] Sasaki, Y. and Wang, L. Message Extension Attack Against Authenticated Encryptions: Application to PANDA. In Gritzalis, D., Kiayias, A. and Askoxylakis, I. (eds.) *Cryptology and Network Security*. Vol. 8813, pp. 82-97. Springer International Publishing, 2014.

[137] Feng, X., Zhang, F., and Wang, H. A Practical Forgery and State Recovery Attack on the Authenticated Cipher PANDA-s. IACR ePrint Archive. 2014/325. Available from `https://eprint.iacr.org/2014/325.pdf`, Accessed 29 May 2015.

[138] Jean, J., Nikolić, I., Sasaki, Y., and Wang, L. Practical Cryptanalysis of PAES. In Joux, A. and Youssef, A. (eds.) *Selected Areas in Cryptography - SAC 2014*. Vol. 8781, pp. 228-242. Springer International Publishing, 2014.

[139] Yao, Y., Zhang, B., and Wu, W. A Single Query Forgery Attack on Raviyoyla v1. In *ACM Symposium on Information, Computer and Communications Security*. pp. 671. 2015.

[140] Feng, X. and Zhang, F., Cryptanalysis on the Authenticated Cipher Sablier. In Au, M.H., and Carminati, B. (eds.) *Network and System Security*. Vol. 8792, pp. 198-208. Springer International Publishing, 2014.

[141] Baksi, A., Maitra, S., and Sarkar, S. New Distinguishers for Reduced Round Trivium and Trivia-SC using Cube Testers. IACR ePrint Archive. 2015/223. Available from `https://eprint.iacr.org/2015/223.pdf`, Accessed 29 May 2015.

[142] Canteaut, A. and Leurent, G. Distinguishing and Key-recovery Attacks against Wheesht. Cryptographic Competitions Mailing List. Available from `https://www.rocq.inria.fr/secret/Anne.Canteaut/Publications/wheesht.pdf`, Accessed 29 May 2015.

[143] Courtois, N., Klimov, A., Patarin, J., and Shamir, A. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In Preneel, B. (ed.) *Advances in Cryptology - EUROCRYPT 2000*. Vol. 1807, pp. 392-407. Springer Berlin Heidelberg, 2000.

[144] Sage Mathematics Software (Version 6.4.1), The Sage Developers, 2015, `http://www.sagemath.org`.

[145] Buchberger, B. An Algorithm for Finding the Bases Elements of the Residue Class Ring Modulo A Zero Dimensional Polynomial Ideal (German). PhD Thesis. Univ. of Innsbruck, 1965.

[146] Roy, D., B., Chakraborti, A., Chang, D., Kumar, S., V., D., Mukhopadhyay, D., Nandi, M. Fault Based Almost Universal Forgeries on CLOC and SILC, In Carlet, C., Hasan, M., Saraswat, V. (eds.) *Security, Privacy, and Applied Cryptography Engineering - SPACE 2016*. Vol. 10076, pp. 66-86. Springer International Publishing, 2016.

[147] Iwata, T., Minematsu, K., Guo, J., Morioka, S., Kobayashi, E. Re: Fault Based Forgery on CLOC and SILC. Available from `https://groups.google.com/forum/#!topic/crypto-competitions/_qxORmqcSrY`, Accessed 01 September 2017.

[148] Dawson, E., Nielsen, L. Automated cryptanalysis of XOR plaintext strings. *Cryptologia*, Vol. 20, Number 2, pp. 165-188, 1996.

[149] Dinur, I., Shamir, A. Side Channel Cube Attacks on Block Ciphers. IACR ePrint Archive. Report 2009/127, 2009. Available from `https://eprint.iacr.org/2009/127.pdf,Accessed26Aug2017`.

[150] Biham, E., Shamir, A. Differential Fault Analysis of Secret Key Cryptosystems. In Kaliski, B.S. (ed.) *Advances in cryptology - CRYPTO 1997*. Vol. 1294, pp. 513-525. Springer Berlin Heidelberg, 1997.

[151] Skorobogatov, S.P., Anderson, R.J. Optical Fault Induction Attacks. In Kaliski, B.S., Koç, K., Paar, C. (eds.) *Cryptographic Hardware and Embedded System - CHES 2002*. Vol. 2523, pp. 2-12. Springer Berlin Heidelberg, 2003.

[152] Blömer, J., Seifert, J.P., Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In Wright, R.N. (ed.) Financial *Cryptography - FC 2003*. Vol 2742, pp. 162-181. Springer Berlin Heidelberg, 2003.

[153] Homsirikamol, E., Farahmand, F., Diehl, W., Gaj, K. Benchmarking of Round 3 CAESAR Candidates in Hardware: Methodology, Designs & Results. Available from `https://cryptography.gmu.edu/athena/presentations/CAESAR_R3_HW_Benchmarking.pdf`, Accessed 14 September 2017.

[154] Ankele, R. and Ankele, R. Software Benchmarking of the 2nd round CAESAR Candidates. IACR ePrint Archive. 2016/740. Available from `https://eprint.iacr.org/2016/740.pdf`, Accessed 14 September 2017.

[155] Authenticated Encryption FPGA Ranking. Available from `https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view`, Accessed 14 September 2017.