Analysis and design of authenticated ciphers

Huang, Tao

2016

Huang, T. (2016). Analysis and design of authenticated ciphers. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/65961>

<https://doi.org/10.32657/10356/65961>

NANYANG
TECHNOLOGICAL
UNIVERSITY

ANALYSIS AND DESIGN OF AUTHENTICATED CIPHERS

TAO HUANG

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

2016

ANALYSIS AND DESIGN OF AUTHENTICATED CIPHERS

TAO HUANG

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

A thesis submitted to Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2016

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Prof Wu Hongjun, for introducing me to cryptography, for inspiring and encouraging me to explore various research topics, for providing me with all kinds of opportunities to communicate with researchers in many conferences and workshops. I am also deeply appreciative for his instructive guidance and numerous helpful discussion and suggestions.

I would like to thank Prof Xing Chaoping for supervising my undergraduate research project and encouraging me for my graduate study.

I am very grateful to meet Thomas Peyrin, Axel Poschmann and my seniors Guo Jian and Wei Lei when I began my PhD study in NTU. I learned a lot in the reading group discussions.

I am also pleased to thank Yu hongbo, Yu Sasaki, Wang Meiqin, Wu Wenling, Wu Shengbao, Lu Jiqiang for interesting discussions on various topics in the conferences and workshops.

I am also delighted to have Lim Hoon Wei, Punarbasu Purkayastha, Su Le, Zhang Luchan, Ong Soon Sheng, Deng Yi, Hou Xiaolu, Wang Chenyu, Wang Jing, Tan Ming Ming as my labmates and friends in Singapore. And I am thankful to Pi Shuang for our long lasting friendships from middle school.

Finally, I want to devote my sincere appreciation to my parents for their encouragement and support.

Abstract

An authenticated cipher is a symmetric key cryptographic primitive which protects the confidentiality, integrity and authenticity of the data. It is an integration of the existing symmetric key primitives such as block ciphers, stream ciphers and hash functions, and attracts a lot of research interests in recent years, especially after the announcement of the CAESAR competition.

In this thesis, we study the analysis and designs of the authenticated ciphers. We begin with an introduction to symmetric key cryptography and authenticated ciphers followed by discussing on the typical methods used in the cryptanalysis and design of authenticated ciphers.

Then, several concrete case studies in analyzing the authenticated ciphers are presented. We apply differential-linear cryptanalysis to recover the internal state of ICEPOLE. Differential IV cryptanalysis is used to attack the initialization of the 128-EEA3/128-EIA3 stream cipher ZUC. By exploiting the leaked state from the keystreams, we present a forgery attack on ALE. By exploiting the parameter settings, we present distinguishing and forgery attacks against the authenticated encryption scheme COFFE. We provide a collision attack to break the authentication claim for the authenticated encryption mode IOC.

For the design of authenticated ciphers, we propose two schemes, JAMBU and MORUS fulfilling various features. JAMBU is a lightweight authenticated encryption mode which provides an intermediate level of nonce misuse resistance. MORUS is a nonce-based authenticated cipher which is targeted for high performance in both software and hardware.

Contents

Acknowledgments	iii
Abstract	v
Contents	vii
List of Figures	xv
List of Notations	xvii
List of Abbreviations	xix
1 Introduction	1
1.1 Authenticated Cipher	1
1.1.1 Symmetric Key Cryptography	1
1.1.1.1 Confidentiality	3
1.1.1.2 Integrity and Authenticity	4
1.1.2 Authenticated Encryption	5
1.2 Cryptanalysis of Authenticated Ciphers	8
1.2.1 Exhaustive Key Search Attack	9
1.2.2 The Birthday Attack	10
1.2.3 Differential Cryptanalysis	10
1.2.4 Linear Cryptanalysis	12

1.2.5	Differential-Linear Cryptanalysis	12
1.2.6	Guess and Determine Cryptanalysis	13
1.2.7	Other Cryptanalysis	14
1.3	Design of Authenticated Ciphers	14
1.3.1	Block Cipher Based Authenticated Ciphers	15
1.3.2	Stream Cipher Based Authenticated Ciphers	16
1.3.3	Permutation Based Authenticated Ciphers	17
1.3.4	Compression Function Based Authenticated Ciphers	18
1.4	Outline of this thesis	18
2	Differential-Linear Cryptanalysis of ICEPOLE	21
2.1	Introduction	21
2.2	The ICEPOLE Authenticated Cipher	23
2.2.1	Notations	23
2.2.2	The ICEPOLE permutation P	24
2.2.3	Initialization	25
2.2.4	Processing associated data and plaintext	27
2.2.5	Tag generation	28
2.2.6	Security goals of ICEPOLE	28
2.3	Differential-Linear Distinguishing Attack	29
2.3.1	Constructing the differential characteristics	30
2.3.2	Constructing the linear characteristic	32
2.3.3	Observations to improve the attack	33
2.3.4	Concatenating the differential and linear characteristics	35
2.4	State Recovery Attack on ICEPOLE	37
2.4.1	State recovery attacks on ICEPOLE-128 and ICEPOLE-128a	37
2.4.1.1	Recovering U_0 and U_3	38
2.4.1.2	Recovering U_2	40
2.4.1.3	Recovering U_1	42

2.4.1.4	Correcting the recovered state	44
2.4.1.5	Summary of the attack	44
2.4.2	State recovery attack on ICEPOLE-256a	45
2.4.3	Implications of the state recovery attacks	47
2.5	Experimental Results	48
2.6	Conclusion	49
3	Initialization Differential Attacks on ZUC	51
3.1	Introduction	51
3.2	Preliminaries	52
3.2.1	The Notations	52
3.2.2	The general structure of ZUC 1.4	53
3.2.2.1	Linear feedback shift register(LFSR).	53
3.2.2.2	Bit-reorganization function.	54
3.2.2.3	Nonlinear function F	54
3.2.3	The initialization of ZUC 1.4	55
3.2.3.1	Key and IV loading.	55
3.2.3.2	Running the cipher for 32 steps.	55
3.3	Overview of the Attacks	56
3.4	Attack ZUC 1.4 with Difference at iv_0	59
3.4.1	The weak keys for Δiv_0	60
3.4.2	Detecting weak keys for Δiv_0	63
3.4.3	Recovering weak keys for Δiv_0	64
3.5	Attack ZUC 1.4 with Difference at iv_1	65
3.5.1	Identical keystreams for Δiv_1	65
3.5.2	Key recovery for Δiv_1	66
3.6	Improving ZUC 1.4	68
3.7	Conclusion	68

4	Leaked-State-Forgery Attack on ALE	69
4.1	Introduction	69
4.2	The Specification of ALE	71
4.3	A Basic Leaked-State Forgery Attack on ALE	74
4.3.1	The main idea of the attack	74
4.3.2	Finding a differential characteristic	75
4.3.3	Launching the forgery attack	76
4.4	Optimizing the Leaked-State-Forgery Attack	79
4.4.1	Improving the differential probability	79
4.4.1.1	Summary of the analysis.	84
4.4.2	Reducing the number of known plaintext blocks	86
4.4.2.1	Relaxing conditions on effective active S-boxes.	86
4.4.2.2	Reducing the number of active leaked bytes in the first two rounds.	86
4.4.2.3	Summary of the analysis.	87
4.5	Experiments on a Reduced Version of ALE	87
4.5.1	The “2–8–12–4” differential characteristic	89
4.5.2	The “6–4–6–9” differential characteristic	90
4.6	Conclusion	91
5	Cryptanalysis of COFFE	93
5.1	Introduction	93
5.2	The COFFE Authenticated Cipher	95
5.2.1	Notations	96
5.2.2	Associated Data Processing	98
5.2.3	Initialization	98
5.2.4	Processing plaintext	99
5.2.5	Tag generation	99
5.2.6	Security goals of COFFE	99

5.3	Analysis on the COFFE scheme	101
5.3.1	Modification of β	102
5.3.2	Modification of α	107
5.4	Distinguishing Attack	108
5.5	Ciphertext Forgery Attack	110
5.5.1	Forgery Attack with Constant Message Block Size	110
5.5.2	Forgery Attack with Dynamic Message Block Size	112
5.5.3	Combination of the Existing Attacks	113
5.6	Related Key Recovery Attack	113
5.7	Conclusion	114
5.7.1	Attack Summary	114
5.7.2	Lesson Learned	115
6	Cryptanalysis of the IOC Authenticated Encryption Mode	117
6.1	Introduction	117
6.2	The IOC Authenticated Encryption Mode	118
6.2.1	Notations	118
6.2.2	The IOC specification	119
6.3	The Forgery Attack on IOC mode	122
6.3.1	The basic idea of the forgery attack	122
6.3.2	Constructing suitable pairs of collisions	124
6.3.3	The attack procedure	125
6.4	Conclusion	126
7	The JAMBU Lightweight Authentication Encryption Mode	127
7.1	Introduction	127
7.2	The JAMBU Mode of Operation	127
7.2.1	Preliminary	127
7.2.1.1	Operations	127

7.2.1.2	Notations and Constants	128
7.2.2	Parameters	129
7.2.3	Padding	129
7.2.4	Initialization	129
7.2.5	Processing the associated data	130
7.2.6	Encryption of JAMBU	130
7.2.7	Finalization and tag generation	131
7.2.8	The decryption and verification	133
7.3	The SIMON-JAMBU and AES-JAMBU authenticated ciphers	133
7.4	Security Goals	134
7.5	The Security Analysis of JAMBU: Nonce-Respecting	135
7.5.1	The security of encryption	136
7.5.2	The security of message authentication	136
7.5.2.1	Key/state recovery attack	136
7.5.2.2	The forgery attack	137
7.6	The Security Analysis of JAMBU: Nonce Misuse	139
7.6.1	The security of encryption	139
7.6.2	The security of message authentication	140
7.7	Features	141
7.8	Performance	142
7.8.1	Hardware performance	142
7.8.2	Software performance	142
7.9	Design rationale	144
8	The Authenticated Cipher MORUS	145
8.1	Introduction	145
8.2	Specification of MORUS	146
8.2.1	Notations and Constants	146
8.2.2	Parameters	148

8.2.3	Recommended parameter sets	148
8.2.4	The state update function of MOURS	149
8.2.5	MORUS-640	153
8.2.5.1	The initialization of MORUS-640	153
8.2.5.2	Processing the associated data	153
8.2.5.3	The encryption of MORUS-640	154
8.2.5.4	The finalization of MORUS-640	154
8.2.5.5	The decryption and verification fo MORUS-640 . . .	155
8.2.6	MORUS-1280	156
8.2.6.1	The initialization of MORUS-1280	156
8.2.6.2	Processing the associated data	156
8.2.6.3	The encryption of MORUS-1280	157
8.2.6.4	The finalization of MORUS-1280	157
8.3	Security Goals	158
8.4	Security Analysis	158
8.4.1	The security of the initialization	159
8.4.2	The security of the encryption process	161
8.4.3	The security of message authentication	161
8.4.3.1	Internal state collision	161
8.4.3.2	Attacks on the finalization	165
8.5	Features	165
8.6	Performance	166
8.7	Design rationale	166
8.7.1	State update function	167
8.7.2	Encryption and authentication	168
8.7.3	Selection of rotation constants	168

9 Conclusions

171

Bibliography	173
A Session Key Generation Analysis of COFFE	193
B The List of Possible v and v' for ZUC IV Collision	208
C Procedure for Generating Identical Keystreams for Δiv_1 to attack ZUC	210
D Values of Leaked-Bytes in the ALE attack	212
E Experiments in the ALE Attack	215
E.1 Details of one Forgery in the “2–8–12–4” Experiment	215
E.2 Details of one Forgery in the “6–4–6–9” Experiment	217
F The specification of SIMON family of block ciphers	219
List of Publications	221

List of Figures

2.1	General scheme of ICEPOLE	23
3.1	General structure of ZUC	53
4.1	The positions of the leaked bytes in the even and odd rounds of LEX.	72
4.2	Encryption and authentication of ALE.	72
4.3	Decryption and verification of ALE.	74
4.4	An example of 1-4-16-4 differential characteristic	75
4.5	A differential characteristic of type “1-4-16-4”	77
4.6	An example of differential path of type “4-6-9-6”.	82
4.7	An example of differential path of type “2-8-12-4(3)”	83
4.8	An example of differential path of type “2-3-12-8”.	84
4.9	Differential Path of type “2-8-12-4”	85
4.10	Differential Path of type “6-4-6-9”	88
5.1	General scheme of COFFE	96
6.1	Generation of IOC initializing vectors and integrity check vector [130]121	
6.2	IOC encryption and authentication	121
6.3	An illustration of the forgery attack on the IOC mode	124
7.1	Initialization of JAMBU	130
7.2	Processing associated data.	131
7.3	Processing the plaintext.	132

7.4	Finalization and tag generation.	132
8.1	The state update function of MORUS	152
E.1	Differential Path of type “2–8–12–4”	216
E.2	Differential Path of type “6–4–6–9”	217

List of Notations

$+$	modulo addition of two integers
$\oplus$	bit-wise exclusive-OR operator
$\&$	bit-wise AND operator
$\parallel$	concatenation
$x \gg n$	the shift of x to the right by n bits
$x \ll n$	the shift of x to the left by n bits
$x \ggg n$	the rotation of x to the right by n bits
$x \lll n$	the rotation of x to the left by n bits
$\bar{x}$	the bitwise complement of x
$\lceil x \rceil$	ceiling operation, $\lceil x \rceil$ is the smallest integer not less than x .

List of Abbreviations

3GPP	The 3rd Generation Partnership Project
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
CAESAR	Competition for Authenticated Encryption: Security, Applicability and Robustness
CBC	Cipher-block chaining mode
IOC	Input and Output Chaining
MAC	Message Authenticated Code

Chapter 1

Introduction

1.1 Authenticated Cipher

An *authenticated cipher* (also known as *authenticated encryption scheme*) is a cryptographic primitive that provides encryption and authentication¹ under a shared secret key in the communication between the sender and receiver. In this section, we will review the development of symmetric-key cryptography and introduce the concept of the authenticated ciphers.

1.1.1 Symmetric Key Cryptography

Cryptography has a long history which can be traced back to ancient ages. It was a natural outcome in fulfilling the need for secret communications, especially in the military. Since then, a number of classical ciphers had been developed. They used the same encryption and decryption keys and usually operated on the alphabets or letters. A well-known classical cipher was the Caesar cipher named after Julius Caesar (100 BC – 44 BC), a famous Roman general and Consul. It only shifted the alphabets to hide original message words. The Caesar cipher is an instance of the

¹Here the term “authentication” includes the “integrity” and “authenticity” for the data.

substitution cipher. Other classical ciphers includes the Vigenère cipher and the Playfair cipher. The classical ciphers are generally vulnerable to the exhaustive key search attack or the frequency attack.

During World War I and World War II, cryptography played an important role and received intensive studies. In World War I, the decryption of the Zimmermann Telegram was one of the reasons for the United States' entry into the war [67]. During World War II, the Allies gained a lot of advantages from the decryption of the German rotor machine Enigma [69] and the Japanese cipher machine Purple [68].

It is worth to mention that near the end of World War I, Gilbert Vernam and Joseph Mauborgne invented the one-time pad system (it is considered as a re-invention of the idea by Frank Miller in 1882 [15]). The idea of one-time pad is to generate a random key with the same length as the plaintext and XOR with the plaintext for encryption. Later in 1949, Claude Shannon introduced the "perfect secrecy" and showed that it could only be achieved when length of the secret key is greater than or equal to the plaintext. Hence, the one-time pad is unbreakable. His paper on the mathematical theory of communication [138] was considered to be the starting point for the modern cryptography.

The concept of public-key system was proposed by Whitfield Diffie and Martin Hellman in 1976 [49]. It uses a public key and a private key to construct a trap-door one-way function in the communication. In the rapid spread of the Internet, the public key cryptographic system is the core component for the key exchanges and digital signatures in the digital communications. On the other hand, symmetric key cryptography retains its importance in the modern cryptography. This is primarily due to the efficiency. So far, there is no public key system which is as efficient as the widely used symmetric key ciphers. In this thesis, we will focus on the symmetric key cryptographic primitives.

The main functions of symmetric cryptographic primitives are not only to pro-

1.1. AUTHENTICATED CIPHER

vide the confidentiality through the encryption, but also to protect the data integrity and the authenticity.

1.1.1.1 Confidentiality

The *confidentiality* requires that the protected data should not be disclosed to unauthorized recipients. The encryption schemes are designed to protect the confidentiality. According to the design philosophy, they can be divided into two classes: block ciphers and stream ciphers.

Block cipher. A *block cipher* encrypts the plaintext as blocks of fixed length. To deal with messages with arbitrary length, the *modes of operation*, or simply *modes*, have been introduced to specify how the block ciphers are iteratively used in processing multi-block messages. The encryption of a block cipher can be considered as a pseudo-random permutation of the input message block.

The DES (Data Encryption Standard) cipher is an early block cipher designed by an IBM research group in the 1970s. It was the public standard for encryption [120] until it was replaced by AES (Advanced Encryption Standard) in 2001. In the AES competition, a number of block ciphers were proposed and analyzed, *e.g.*, Rijndael [42], Serpent [3]. Twofish [137], RC6 [131], MARS [39]. Rijndael was the winner of the AES competition and became the most widely used block cipher today.

National Institute of Standard and Technology (NIST) has maintained a website for block cipher modes under its cryptographic toolkit [117] which gives recommendations on the use of block ciphers. Currently, the NIST recommendation [118] defines five confidentiality modes: Electronic Codebook mode (ECB), Cipher Block Chaining mode (CBC), Cipher Feedback mode (CFB), Output Feedback mode (OFB), and Counter mode (CTR).

Stream cipher. A *stream cipher* generates an arbitrary length keystream with a secret key and generates the ciphertext by XORing the plaintext with the keystream.

Compared to block ciphers, stream ciphers with changing state normally require less computation for achieving the same security level. But the initialization vectors (IVs) should be used as a nonce under the same key for a stream cipher.

ECRYPT (2004–2008) has organised the eSTREAM competition [58], which stimulated the study on stream ciphers, and a number of new stream ciphers were proposed, the winners of eSTREAM includes HC-128 [150], Rabbit [36], Salsa20/12 [19], SOSEMANUK [16], Grain v1 [77], MICKEY 2.0 [8] and Trivium [47].

1.1.1.2 Integrity and Authenticity

The *integrity* requires that the protected data should not be modified or destroyed when delivered to the recipients, while the *authenticity* requires that the message should be from the intended sender. The cryptographic hash functions and the message authenticated codes (MACs) are the commonly used cryptographic primitives to protect the integrity and authenticity.

Cryptographic hash function. A *cryptographic hash function* is a deterministic procedure that compresses input strings of arbitrary finite length into output strings of fixed-length (message digest) so that every output of hash function represents a message which is computationally unique. By checking the message digest, the cryptographic hash function can detect whether a message has been modified. Besides, it may be used in the pseudo-random generation, key derivation or password verification *et al.* as well.

MD5 and SHA-1 are two widely used cryptographic hash functions in the last two decades. However, weaknesses of these hash functions were discovered and SHA-2 is currently the recommended cryptographic hash function by NIST as replacement of MD5 and SHA-1. In 2007, the SHA-3 competition [92] was announced to promote the design of new hash standard and Keccak [21] was selected to be the SHA-3 standard in 2012.

Message authenticated codes. A *message authenticated code (MAC)* is a symmetric

1.1. AUTHENTICATED CIPHER

key cryptographic primitive which protects the integrity and authenticity in the communications. Using a MAC, messages are processed under a shared secret key, and a tag is generated for the message to prevent the unauthorized transmission or modification of the messages.

There are two main approaches to design MACs. One approach is based on hash functions, either the universal hash function like UMAC [34] or cryptographic hash functions like MDx-MAC and HMAC [11, 125]. Another approach is to construct MACs based on block ciphers like EMAC [122], OMAC [82] and PMAC [35].

1.1.2 Authenticated Encryption

To protect the confidentiality, integrity and message authentication in the communication, Bellare and Namprempre in 2000 proposed the authenticated encryption (AE) schemes by integrating an encryption scheme with an authentication scheme [13]. In many applications such as the network communication, part of the message (such as the routing information) is public and only needs to be authenticated. The concept of AE was extended to authenticated-encryption with associated-data (AEAD) by Rogaway [132], which was adopted by most of the later designed AE schemes.

An AE scheme takes a secret key, an initialization vector (IV)², associated data (if supported) and a message as input, and produces the ciphertext and authentication tag as output. It protects the confidentiality of the message and the authentications of both the associated data and the message.

Three design approaches of AE were mentioned in [13]:

- Encrypt-then-MAC. The plaintext is encrypted and then the MAC is applied to the ciphertext to generate the MAC tag. It is the standard method in ISO/IEC 19772:2009 [81].

²The term *nonce* is also used to indicate that IV is a number which can be used only once (under the same secret key). In the CAESAR competition, the term *public message number* is used [17].

- Encrypt-and-MAC. The plaintext is encrypted and it is also used to generate a tag for authentication.
- MAC-then-Encrypt. The MAC is applied to the plaintext to generate a tag and then the plaintext and the tag are encrypted. This method was adopted in the SSL/TLS implementation.

It turns out many AE schemes designed by composing an encryption scheme and an authentication scheme are not really satisfactory in applications. The implementations of the composed scheme must be very careful to ensure the security. This turned out to be a difficult task [12, 13, 18, 96]. As mentioned in [95], *“it is very easy to accidentally combine secure encryption schemes with secure MACs and still get insecure authenticated encryption schemes”*.

To improve the general composition schemes, a number of dedicated AE schemes have been proposed. The dedicated AE schemes are designed to provide the encryption and authentication in one specification so as to avoid the misuse in the composition as well as to increase the efficiency.

Many dedicated AE schemes are based on block cipher modes, such as *e.g.*, IAPM [88], OCB [134], OCB 2.0 [133], CCM [146], CWC [95], GCM [107], EAX [14], HBS [84], BTM [83] and McOE [65]. The ISO/IEC 19772:2009 [81] standardized several modes, including EAX, CCM, GCM and OCB 2.0.

AE schemes may be based on stream ciphers as well, *e.g.*, Helix [64], Phelix [147], Hummingbird-2 [60], ASC-1 [85], the 3GPP algorithm 128-EEA3 & 128-EIA3 [1] and Grain-128a [2].

Despite the various existing AE schemes, there are several good reasons to design new AE schemes.

The fast development of the computational devices makes it possible and necessary to design more efficient and more secure AE designs. A decade ago, the ciphers were mostly implemented on 8-bit or 32-bit platforms with single processor. Nowadays, 64-bit platform and multi-core processors are commonly used.

1.1. AUTHENTICATED CIPHER

This allows the designers to introduce new operations and structures which can benefit from the new generation of processors.

The new AE designs can take advantage from the past experiences in the design and cryptanalysis in the symmetric cryptography. In the past 20 years, especially during the focused competitions such as the AES competition [119], the eSTREAM competition [59] and the SHA-3 competition [92], many strategies on the design and analysis of the block ciphers, stream ciphers and hash functions have been developed, which can be applied in the design and analysis of the AE schemes with better efficiency and security. Moreover, those well-designed primitives can be used as building blocks of the AE schemes.

The new AE designs can be targeted in fulfilling various implementation needs. The AE schemes being targeted on the software platform would have different concentration in the design compared with the ones being targeted on the hardware platform. An important trend in the current development of cryptography is to design lightweight cryptographic primitives because of the increasing needs for low-cost embedded systems such as RFID tags, sensor networks and smart cards. Recently, several AE schemes have been proposed for the lightweight usage, such as Hummingbird-2 [60], ALE [37], and FIDES [30].

The new AE designs can have various features. Besides the primary functions of encryption and authentication, there are a lot of desirable features in applications. The nonce misuse resistance indicates that the schemes maintain a certain level of security when the IV is accidentally reused with the same secret key. The parallelism feature processes multiple blocks in parallel, resulting significantly increase of the efficiency. The online feature allows the messages to be processed before being fully loaded. The inverse-free property is to use the same operations of the underlying primitives in both encryption and decryption. The designers may select different sets of preferable features in the AE schemes.

The ongoing CAESAR (Competition for Authenticated Encryption: Security,

Applicability, and Robustness, 2014 – 2017) [40] is held to promote the design of authenticated ciphers with various features better than the existing ones (AES-GCM is considered as a baseline for comparison). This competition attracted 57 first round submissions. The second round candidates were announced in Jul 2015. There are 29 candidates left, including two submissions in this thesis, JAMBU and MORUS. The term *authenticated cipher* is used in the CAESAR website as a replacement of the *authenticated encryption scheme* and we will not distinguish these two terms in this thesis.

1.2 Cryptanalysis of Authenticated Ciphers

Except for the unconditionally secure one-time pad, it is generally difficult to prove the security of a symmetric key cryptographic primitive. In practice, the evaluation of the security of a symmetric key cryptographic primitive is mainly based on applying the known cryptanalysis methods to it.

As both encryption and message authentication are involved, the goals of the cryptanalysis of authenticated ciphers include: recovering the unknown plaintext, recovering the secret key, distinguishing the ciphertext from the random, generating forged ciphertext which can bypass the authentication.

Depending on the building blocks, the authenticated ciphers may be analyzed using the cryptanalysis techniques from the block ciphers, stream ciphers, hash functions and MACs and even some new types of techniques in studying the interaction between the encryption and message authentication. Hence, the cryptanalysis of authenticated ciphers is highly flexible and complicated.

In this section, we will review a number of typical methods in symmetric key cryptanalysis and their applications to authenticated ciphers.

1.2.1 Exhaustive Key Search Attack

The Kerckhoffs' principle is widely adopted as a rule in the design of modern cryptographic primitives, which says the security of the primitive should only be based on the key. Thus, even if all the other information except the key is public, the primitive should be secure.

Under this principle, the exhaustive key search attack, or brute force attack, is a generic attack model. Since the secrecy of an authenticated cipher solely relies on the secret key, it is obvious that after at most 2^k trials of the exhaustively key search, any cipher with k -bit secret key can be broken.

An improvement of the exhaustive key search for the block ciphers is the time-memory trade-off proposed by Hellman [78]. When the key is k -bit, it uses a $2^{2k/3}$ entries precomputed table and $2^{2k/3}$ operations to recover a key. Notice that this attack may have much more cost in real implementation when considering the full cost such as the connecting of processors and large memory, as mentioned by Wiener [148].

In most of the cases nowadays, the exhaustive key search attack is not really harmful for the security, but it is usually considered as a baseline of the attacks. Namely, an attack is valid only if it works better than the exhaustive key search attack.

This attack also affects the selection of the key length. It does not imply that the key length should be very long, as a longer key requires more cost in the key generation, exchange and storage. Generally, when the complexity for the exhaustive key search exceeds the possible computing power available, the primitive can be considered secure under this type of attack. Currently, the most powerful supercomputer Tianhe-2 has a peak speed of 33.86 PFLOPS (or around 2^{80} float point operations per year). Despite the fast development of the computing power, it is still reasonable to consider the ciphers with 128-bit key as secure under the exhaustive key search in the next few decades. Besides, in the lightweight applications

such as RFID tags, the implementation costs are very sensitive to the key length. A shorter key size may still be a good choice for low cost implementation.

1.2.2 The Birthday Attack

The *birthday attack* is based on the well-known birthday paradox [159]. It states that given about $1.2 \cdot 2^{n/2}$ random n -bit values, the probability to find a collision among is larger than 0.5. So usually, $O(2^{n/2})$ is considered as the birthday bound. Using this property, Preneel and van Oorschot proposed a forgery attack which showed that all iterated MACs with n -bit internal state size can be attacked with $O(2^{n/2})$ queries [126].

This generic attack is important for the design and analysis of the authenticated ciphers. It shows that to protect the n -bit security for the authentication, the internal state of the cipher should be at least $2n$ bits. However, this generic attack has the limitation that a large amount of queries are needed in the attack.

The birthday attack is commonly used to produce a collision of certain state in authenticated encryption modes. An example is the collision attack on OCB by Ferguson [63]. In Chapter 6 we will apply it in the analysis of IOC.

1.2.3 Differential Cryptanalysis

Differential cryptanalysis is a typical method in analyzing a symmetric key cryptographic primitive. It was developed by Biham and Shamir in the late 1980s, and was applied to attack the block cipher DES [28, 29]. Differential cryptanalysis introduces a certain difference in the state of a cipher and captures the non-random behavior through difference propagation. Although the method is publicly known for more than 20 years, it is still one of the most influential attacks in today's cryptanalysis. It has been generalized into more advanced forms such as the *impossible differential cryptanalysis* [25], the *truncated differential cryptanalysis* [94] and

1.2. CRYPTANALYSIS OF AUTHENTICATED CIPHERS

higher-order differential cryptanalysis [100]. In addition, differential characteristic is involved in the many more advanced attacks such as the *boomerang attack* [145] and *rebound attack* [108].

Regarding the authenticated cipher, differential cryptanalysis has been included in most of the initial analysis by the designers. Nevertheless, it turns out that differential cryptanalysis is still applicable in breaking a number of schemes. Here we summarize several scenarios in which differential cryptanalysis is applied to authenticated ciphers.

Difference in the Initialization. The initialization is a process that mixes the IV and the secret key and generates a pseudo-random internal state. When an authenticated cipher adopts the stream cipher construction, it is necessary to ensure the randomness of the state after the initialization cannot be exploited. Otherwise, it can lead to distinguishing attack or even the key-recovery attack. In chapter 3, we apply the differential cryptanalysis to the initialization of the ZUC in the 128-EEA3/128-EIA3 to launch a key-recovery attack. And in the MORUS specification in chapter 8, differential cryptanalysis of the initialization is also considered.

Difference in the Message. When the IVs are allowed to be reused in an authenticated cipher, it is possible to study the propagation of differences introduced in the message. In this case, the differential cryptanalysis is similar to the cases in attacking a block cipher or a permutation function. An example is the analysis on McMambo [99]. It was reported to have iterative differential patterns which resulted in a differential of probability 2^{-24} [115]. But this is not a common mistake, we noticed that most of the recent designs have a proper security margin to resist this kind of differential attack.

When a difference is introduced in the ciphertext in an authenticated cipher, it will result in a difference in the plaintext during the decryption. If a differential can

be derived from this initial difference with sufficiently high probability, it may be exploited to attack the cipher. This method has been applied in the forgery attacks of several authenticated ciphers, *e.g.*, ALE [5, 93], LAC [102].

1.2.4 Linear Cryptanalysis

Linear cryptanalysis, introduced by Matsui [106], is another fundamental method used in symmetric key cryptanalysis. The basic idea is to analyze the linear relations of a certain set of bits and to establish linear approximations involving the plaintext bits, key(or subkey) bits and ciphertext bits, which are biased from the random. It was first applied to attack FEAL [106] and DES [105].

Since Matsui's paper, several more advanced linear cryptanalysis techniques have been developed, including *multiple linear cryptanalysis* [32, 89, 116] which considers several linear approximations, the *multidimensional approximation* [79, 80] which relaxes the assumption on the statistical independences of the linear approximations and the *zero-correlation linear cryptanalysis* [38] which is based on the linear approximations with probability exactly $1/2$.

For most of authenticated ciphers, the linear property has been studied by the designers to decide the proper security margin. And it is rare to find the authenticated cipher vulnerable to the classical linear cryptanalysis.

1.2.5 Differential-Linear Cryptanalysis

Differential-Linear Cryptanalysis was introduced by Langford and Hellman [101] in 1994 to attack the block cipher DES. The basic idea is to build a longer differential-linear characteristic by combining a (truncated) differential characteristic and a linear characteristic. Then the differential-linear characteristic can be used to launch the key-recovery attack. This usually results in attacks with more rounds than the pure differential cryptanalysis or linear cryptanalysis. Later Biham, Dunkelman

1.2. CRYPTANALYSIS OF AUTHENTICATED CIPHERS

and Keller extended the original attack [26] which allows the probabilistic differential characteristic. This method has been applied to analyze a number of block ciphers such as Serpent [27,52], CTC2 [54,104] and SHACAL-2 [139].

Regarding the authenticated cipher, the differential-linear cryptanalysis can be used to analyze the property of the randomness of the internal permutations in the cases that IVs are allowed to be reused under a same key. An example of this attack against ICEPOLE [112] will be discussed in Chapter 2.

1.2.6 Guess and Determine Cryptanalysis

Guess and determine cryptanalysis has been a powerful technique used in stream cipher cryptanalysis. The basic idea is to establish relationships between the internal state bits and the keystream bits, then it may be possible to guess some unknown bits in the internal state and determine whether the guessed values are correct by comparing the derived keystream bits from the relations with the actual keystream bits. When the complexity of recovering all the unknown bits is less than the brute force search of the key, the attack is successful. This idea has been applied to analyze the stream cipher SNOW [76], Sober-t32 [7] and SOSEMANUK [61].

When an authenticated cipher adopts the method used in stream ciphers in generating the keystreams, guess and determine attacks might be effectively used for recovering the internal states. Examples can be found in the cryptanalysis of FIDES [50] and Sablier v1 [62].

To resist guess and determine cryptanalysis, it would be helpful to add sufficient diffusion in the updating functions so as to increase the difficulties in establishing the relations.

1.2.7 Other Cryptanalysis

Besides the techniques mentioned above, the padding, domain separation or certain special properties in the authenticated ciphers may also be exploited to attack the authenticated ciphers. Many dedicated attacks are developed in this manner. Several examples arise from the mistakes in the padding schemes, such as the attacks on SCREAM [140], PiCipher [103] and CMCC mode [9]. There are more attacks with very low complexity which are largely due to the mistakes in the specific functions or modes such as the attacks on POET [74] and COBRA mode [114].

1.3 Design of Authenticated Ciphers

Two approaches are commonly used to construct authenticated ciphers. One approach is to perform the encryption and authentication in one-pass, namely, the message is injected into the state in the encryption. Another approach is to separate the encryption and authentication into two phrases: encryption under one primitive such as a block cipher or a stream cipher and authentication under another primitive such as a universal hash functions or a MAC.

Authenticated ciphers may have various interesting features. The performance is always one of the most crucial criteria in evaluating a cipher. To design a hardware-efficient authenticated cipher, simple hardware-efficient operations and low area are preferred. Operations like logical shifts and bit-wise logical operations are of this type, especially when they are used in a short critical path. Using one-pass constructions will be helpful to reduce the internal state size. To design a software-efficient authenticated cipher, minimizing the number of operations and setting an adequate security margin are two important methods. With the SIMD instructions, it is possible to perform the confusion and diffusion efficiently on 128- or 256-bit registers. This is highly useful in the design of fast authenticated ciphers today.

Nonce-reuse security is another important property. Generally, IV should be

1.3. DESIGN OF AUTHENTICATED CIPHERS

used as a nonce which is unique under the same secret key. However, the securities of authenticated ciphers under the nonce-reuse circumstances may vary: completely insecure, partially secure or secure. This influences the design and analysis of the authenticated ciphers. Those ciphers with higher security under IV-reuse circumstance usually have much stronger confusion and diffusion in processing the messages, at the cost of more operations. In practice, the CBC mode with strong IV-reuse security has been a popular choice for many years. But in recent years, we have seen many applications of authenticated ciphers which require the uniqueness of IV. In TLS 1.2 [48], the GCM-Based authenticated ciphers AES-GCM, Camellia-GCM and ARIA-GCM have been included. AES-GCM is also included in other standards like the IEEE 802.1ae [135] and IPsec [144].

Other properties such as being *parallelizable* (the operations being divided and computed in multiple processes), *inverse-free* (no need to inverse operations for decryption) and *online* (messages can be processed without fully loaded) are also desirable in the design of authenticated ciphers.

In this thesis, we classify the authenticated ciphers into four categories according to the underlying primitives: the block cipher based designs, the stream cipher based designs, the permutation based designs and the compression function based designs.

1.3.1 Block Cipher Based Authenticated Ciphers

Block ciphers are the building blocks of many cryptographic primitives. In the construction of authenticated ciphers, block ciphers may be applied in the authenticated encryption modes [107] or in the dedicated designs [86].

There are several advantages of using block ciphers as underlying primitives.

- **Efficient AES-NI.** The widely-used block cipher AES can be implemented very efficiently since the introduction of AES-NI by Intel in 2008. The AES-based designs

are generally more efficient than the non-AES designs in the platforms in which AES-NI is implemented. As a result, a lot of the block cipher based authenticated ciphers use AES or modified AES as the underlying primitives.

- **Easy for analysis.** The security of many block ciphers has been evaluated for a long period. When such block ciphers are used, it is much more easier to analyze than the dedicated new designs.

- **Nonce misuse security.** The fact that a secure block cipher is a strong keyed pseudo-random permutation is important for achieving the nonce misuse resistance property.

To enable the parallelized computation, many designs use the *tweaks*, which are usually derived from the secret key, to allow the different blocks of message being independently processed while keeping the order, such as the COPA [5], KIASU [87] and Joltik [86].

1.3.2 Stream Cipher Based Authenticated Ciphers

Stream ciphers are nonce-based cryptographic primitives which generally require less operations than block ciphers. On the platforms without AES-NI, stream ciphers are very useful building blocks to construct efficient nonce-based authenticated ciphers.

The underlying stream cipher is a crucial component for the authenticated ciphers of this type. During the eSTREAM project, a number of well-designed stream ciphers, either in software or hardware, have been selected as the winners. These stream ciphers provide rich resources for designing two-pass authenticated ciphers. On the other hand, unlike the authenticated encryption modes in block cipher, it is non-trivial to convert a stream cipher to a one-pass authenticated cipher securely. As most of the previous stream ciphers only concentrate on the encryption function, new dedicated designs of stream cipher based authenticated ciphers

1.3. DESIGN OF AUTHENTICATED CIPHERS

such as ACORN [149], MORUS [152] and Sablier [160] are proposed in the CAESAR submissions. To securely integrate the encryption and authentication in one pass, the messages have to be involved in the state updating. Hence, it is necessary to study the differential cancellation in the updating function to prevent the forgery attacks. Moreover, the keystreams should be generated in an secure way so that the leaked information cannot be exploited with the guess and determine attacks.

1.3.3 Permutation Based Authenticated Ciphers

Permutations are popular underlying primitives used in many recent authenticated ciphers. The permutations can be designed in very flexible ways, either keyed or key-less.

When the keyed permutations are employed, a straightforward way is to use the authenticated encryption modes. Prøst-COPA and Prøst-CTR [90,91] are of this type.

A widely-used approach to utilize the key-less permutations for designing authenticated ciphers is the sponge construction [20] or the duplex construction [22] extended from the sponge construction. Quite a few CAESAR submissions are based on this method *e.g.*, Ascon [51], ICEPOLE [112], NORX [6], PRIMATE [4], STRIBOB [136] and π -cipher [70,71].

Then the question is how to construct the specific underlying permutations. The permutations can be from existing block cipher operations like AESQ in PAEQ [33], or hash functions like Keccak- f permutation in Ketje [23]. There are dedicated designed permutations like NORX [6], PRIMATE [4] or ICEPOLE [112].

Depending on the properties of the underlying permutations, the authenticated ciphers have various features. For efficiency, they can be designed with properly chosen state size and optimized operations such as NORX [6]. When robust random permutations are used, it is possible to construct nonce misuse resistant

authenticated schemes such as Keyak [24] and PRIMATES-APE [4]. When small state is used in the permutation, the lightweight authenticated ciphers can be constructed such as Ketje [23] and PRIMATES [4].

1.3.4 Compression Function Based Authenticated Ciphers

Another approach to construct authenticated ciphers is to employ the compression functions, which are widely used in cryptographic hash functions such as MD5, SHA-1 and SHA-2.

An intuition to construct this kind of authenticated ciphers is to utilize the Intel SHA extensions [73] which provide the instructions for the SHA-1 and SHA-256 compression [121]. Like the AES-NI, the instructions are much more efficient than the normal implementations. OMD [41] is a CAESAR submission using this method.

Since the compression functions are the generalization of pseudo-random permutations, it is possible to adopt the ideas in constructing the permutations based authenticated ciphers with slight modifications. For example, OMD uses a modified SHA-2 compression to form a keyed pseudo-random function under a generalized Merkle-Damgård construction. There are not too many authenticated ciphers designed using this method so far, but it may be an interesting task to construct other authenticated ciphers based on the compression functions to achieve other features like nonce-reuse resistance or parallelizable encryption.

1.4 Outline of this thesis

In Chapter 1 (this chapter), we give a general introduction to the authenticated ciphers and review the popular techniques used in the cryptanalysis of authenticated ciphers, followed by the discussion on the design of authenticated ciphers.

1.4. OUTLINE OF THIS THESIS

In Chapter 2, we apply the differential-linear cryptanalysis to study the permutation of the authenticated cipher ICEPOLE under the nonce reuse cases and find a state recovery attack with practical complexity.

In Chapter 3, we present a differential attack on the initialization of the stream cipher ZUC v1.4 in the 128-EEA3/128-EIA3 authenticated encryption standard, which can be exploited to recover the secret key.

In Chapter 4, we propose a leaked-state-forgery-attack on the authenticated encryption mode ALE.

In Chapter 5, we analyze the hash function based authenticated cipher COFFE and show the weakness under certain parameter settings.

In Chapter 6, we describe a forgery attack against the IOC mode through the collision of the internal states.

In Chapter 7, we propose a lightweight authenticated encryption mode JAMBU and the authenticated cipher AES-JAMBU.

In Chapter 8, we propose a very efficient non-AES authenticated cipher MORUS.

In Chapter 9, we conclude this thesis.

Chapter 2

Differential-Linear Cryptanalysis of ICEPOLE

2.1 Introduction

ICEPOLE is a new hardware-oriented single-pass authenticated cipher designed by Morawiecki *et al.* It was submitted to the CAESAR competition [112] and published in CHES 2014 [113]. ICEPOLE is designed to be hardware-efficient. It can achieve 41 Gbits/s on the modern FPGA device Virtex 6 which is over 10 times faster than the equivalent implementation of AES-128-GCM [107]. ICEPOLE adopts the well-known duplex construction by Bertoni *et al.* [22] and uses a Keccak-like permutation as its iterative function.

The ICEPOLE family of authenticated ciphers includes three variants: ICEPOLE-128, ICEPOLE-128a and ICEPOLE-256a. Note that the definition of ICEPOLE-128 has been slightly modified in the CHES 2014 version, which removed the use of secret message number. In this chapter, we will follow the definition in the version submitted to CAESAR. For the security of ICEPOLE, the designers claim that the confidentiality is the same as the key length, which is 128-bit for ICEPOLE-128 and ICEPOLE-128a and 256-bit for ICEPOLE-256a. The authentication security is

128 bits for all the three variants. In particular, the designers claim that the internal permutation is strong enough such that “*in the case of nonce reuse the key-recovery attack is not possible*” and the authenticity “*is not threatened by nonce reuse*”.

In this chapter, we apply the differential-linear cryptanalysis to study the security of the permutation of the ICEPOLE family. Differential-linear cryptanalysis is the combination of differential cryptanalysis [28] and linear cryptanalysis [105]. It was introduced by Langford and Hellman in [101] in 1994 to attack the block cipher DES, and later Biham, Dunkelman and Keller gave an enhanced version of this method [26]. This method has been applied to analyze a number of block ciphers such as Serpent [27, 52], CTC2 [54, 104] and SHACAL-2 [139].

Although the design of authenticated cipher ICEPOLE is different from block ciphers, we manage to exploit the differential-linear property of the permutation when the nonce is reused. We will show that under the nonce-reuse assumption, there exists distinguishing attacks on ICEPOLE with both time and data complexity less than 2^{36} . Furthermore, it is possible to recover the 256 bits unknown state of ICEPOLE-128 and ICEPOLE-128a with practical complexity 2^{46} , and recover the 320 bits unknown state of ICEPOLE-256a with complexity 2^{60} . We experimentally verified our results by recovering the state of ICEPOLE-128 using a 64-core server within 10 days. Thus, the security claims of ICEPOLE do not hold under the nonce-reuse circumstances.

The rest of this chapter is structured as follows: The specification of ICEPOLE is given in Section 2.2. Section 2.3 describes a differential-linear distinguishing attack on ICEPOLE. Section 2.4 introduces the state-recovery attack. Section 2.5 provides our experimental results of the state-recovery attack on ICEPOLE-128. Section 2.6 concludes the chapter.

2.2. THE ICEPOLE AUTHENTICATED CIPHER

Figure 2.1: General scheme of ICEPOLE encryption and authentication (Fig. 1 of [112])

2.2 The ICEPOLE Authenticated Cipher

The ICEPOLE family of authenticated ciphers uses three parameters: key length (128 or 256 bits), secret message number (SMN) length (0 or 128 bits) and nonce length (96 or 128 bits). The one with 128-bit secret message number, 128-bit key and 128-bit nonce is ICEPOLE-128. The other two variants do not have a secret message number and are ICEPOLE-128a and ICEPOLE-256a according to the key length. The nonce length for these two variants is 96-bit. We will briefly describe the specification of the ICEPOLE authenticated cipher. The full specification can be found in [112]. An overview of ICEPOLE-128 is provided in Figure 2.1.

2.2.1 Notations

The ICEPOLE algorithm has a 1280-bit internal state S . It uses the little-endian convention. The organization of internal state is similar to Keccak [21], which uses a 3-dimension structure. Therefore, the 1280-bit state S can be represented as $S[4][5][64]$, or shortly $S[4][5]$, an array of 64-bit words. For $S[x][y][z]$, it is corresponding to the $64(x + 4y) + z$ -th bit of the input. ICEPOLE uses 4×5 slices. Each slice has 4 rows and 5 columns. And $S_{[n]}$ denotes the first n bits of the state.

2.2.2 The ICEPOLE permutation P

The permutation P is applied iteratively on the ICEPOLE state S during the encryption and authentication. Each permutation is called a *round* or R . The 6-round and 12-round of P are represented as P_6 and P_{12} respectively. Each round includes five operations: μ , ρ , π , ψ , and κ .

$$R = \kappa \circ \psi \circ \pi \circ \rho \circ \mu$$

The operations are defined as follows:

μ :

A column vector (Z_0, Z_1, Z_2, Z_3) is multiplied by a constant matrix to produce a vector of four 5-bit words.

$$\begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 1 & 18 & 2 \\ 1 & 2 & 1 & 18 \\ 1 & 18 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{bmatrix} = \begin{bmatrix} 2Z_0 + Z_1 + Z_2 + Z_3 \\ Z_0 + Z_1 + 18Z_2 + 2Z_3 \\ Z_0 + 2Z_1 + Z_2 + 18Z_3 \\ Z_0 + 18Z_1 + 2Z_2 + Z_3 \end{bmatrix}$$

The operations are done in $GF(2^5)$. The irreducible polynomial $x^5 + x^2 + 1$ is used for the field multiplication. μ can be efficiently implemented with simple bitwise equations, see Appendix B in [112].

ρ :

The ρ step is the bitwise rotation on each of the 20 64-bit words. It is defined as:

$$\rho(S[x][y]) = S[x][y] \lll \text{offsets}[x][y] \quad \text{for all } (0 \leq x \leq 3), (0 \leq y \leq 4)$$

The rotation offsets are as follows:

2.2. THE ICEPOLE AUTHENTICATED CIPHER

$\text{offset}[0][0] := 0$	$\text{offset}[0][1] := 36$	$\text{offset}[0][2] := 3$	$\text{offset}[0][3] := 41$
$\text{offset}[0][4] := 18$	$\text{offset}[1][0] := 1$	$\text{offset}[1][1] := 44$	$\text{offset}[1][2] := 10$
$\text{offset}[1][3] := 45$	$\text{offset}[1][4] := 2$	$\text{offset}[2][0] := 62$	$\text{offset}[2][1] := 6$
$\text{offset}[2][2] := 43$	$\text{offset}[2][3] := 15$	$\text{offset}[2][4] := 61$	$\text{offset}[3][0] := 28$
$\text{offset}[3][1] := 55$	$\text{offset}[3][2] := 25$	$\text{offset}[3][3] := 21$	$\text{offset}[3][4] := 56$

π :

π reorders the bits within each slice. It maps $S[x][y]$ to $S[x'][y']$ using following rule:

- $x' := (x + y) \bmod 4$
- $y' := (((x + y) \bmod 4) + y + 1) \bmod 5$

ϕ :

ϕ is the S-box layer. ICEPOLE uses the following 5-bit S-box:

$\{ 31, 9, 18, 11, 5, 12, 22, 15, 10, 3, 24, 1, 13, 4, 30, 7,$
 $20, 21, 6, 23, 17, 16, 2, 19, 26, 27, 8, 25, 29, 28, 14, 0 \}$

The ϕ step applies the 5-bit S-box to all the 256 rows of the state.

κ :

In κ the 64-bit constant is xored with $S[0][0]$. The constants are different for each round. We omit the values here.

2.2.3 Initialization

First, the state S is initialized with a 1280-bit constant:

$S[0][0] := 0\text{xFF97A42D7F8E6FD4}$	$S[0][1] := 0\text{X90FEE5A0A44647C4}$
$S[0][2] := 0\text{X8C5BDA0CD6192E76}$	$S[0][3] := 0\text{XAD30A6F71B19059C}$
$S[0][4] := 0\text{X30935AB7D08FFC64}$	$S[1][0] := 0\text{XEB5AA93F2317D635}$
$S[1][1] := 0\text{XA9A6E6260D712103}$	$S[1][2] := 0\text{X81A57C16DBCF 555F}$
$S[1][3] := 0\text{X43B831CD0347C826}$	$S[1][4] := 0\text{X01F22F1A11A5569F}$
$S[2][0] := 0\text{X05E5635A21D9AE61}$	$S[2][1] := 0\text{X64BEFEF28CC970F2}$
$S[2][2] := 0\text{X613670957BC46611}$	$S[2][3] := 0\text{XB87C5A554FD00ECB}$
$S[2][4] := 0\text{X8C3EE88A1CCF32C8}$	$S[3][0] := 0\text{X940C7922AE3A2614}$
$S[3][1] := 0\text{X1841F924A2C509E4}$	$S[3][2] := 0\text{X16F53526E70465C2}$
$S[3][3] := 0\text{X75F644E97F30A13B}$	$S[3][4] := 0\text{XEAF1FF7B5CECA249}$

Then, the key (K) and the *nonce* are XORed to the state. The *nonce* is 128 bits for ICEPOLE–128. The 96-bit *nonce* for ICEPOLE–128a and ICEPOLE–256a will be padded with 32 zeros to form a 128-bit *nonce*. nonce_0 and nonce_1 denote two 64-bit words of the padded nonce.

For ICEPOLE–128 and ICEPOLE–128a, K_0 and K_1 denote two 64-bit words of the key,

$$\begin{aligned}
 S[0][0] &:= S[0][0] \oplus K_0 \\
 S[1][0] &:= S[1][0] \oplus K_1 \\
 S[2][0] &:= S[2][0] \oplus \text{nonce}_0 \\
 S[3][0] &:= S[3][0] \oplus \text{nonce}_1
 \end{aligned}$$

For ICEPOLE–256a, K_0 , K_1 , K_2 and K_3 denote four 64-bit words of the key,

2.2. THE ICEPOLE AUTHENTICATED CIPHER

$$S[0][0] := S[0][0] \oplus K_0$$

$$S[1][0] := S[1][0] \oplus K_1$$

$$S[2][0] := S[2][0] \oplus K_2$$

$$S[3][0] := S[3][0] \oplus K_3$$

$$S[0][1] := S[0][1] \oplus \text{nonce}_0$$

$$S[1][1] := S[1][1] \oplus \text{nonce}_1$$

After that, the P_{12} permutation is applied to the state S .

2.2.4 Processing associated data and plaintext

ICEPOLE-128 uses a 128-bit secret message number (SMN) σ^{SMN} . It will be processed before associated data and the plaintext. Since ICEPOLE-128a and ICEPOLE-256a do not have an SMN, only the associated data σ_i^{AD} and the plaintext σ_i^P will be processed.

For ICEPOLE-128 and ICEPOLE-128a, the length of blocks σ_i^{AD} and σ_i^P is in the range $[0, 1024]$ bits. The blocks will be padded to 1026 bits according to following rule. First, a *frame bit* is appended. It is set to '1' for the last σ^{AD} block and all σ_i^P except the last one. For all other blocks, it is set to '0'. Then, a bit '1' is appended, following by '0's to make the length of padded block be 1026-bit. The number of blocks under a single key is less than 2^{126} .

For ICEPOLE-256a, the associated data and plaintext blocks have length in the range $[0, 960]$. The same padding rule is applied and the padded blocks have length 962-bit. The number of blocks under a single key is less than 2^{62} .

The process of secret message number is as below for ICEPOLE-128:

$$c_{SMN} = S_{[128]} \oplus \sigma^{SMN}$$

$$\sigma^{SMN} := pad(\sigma^{SMN})$$

$$S_{[1026]} := S_{[1026]} \oplus \sigma^{SMN}$$

$$S := P_6(S)$$

The process of associated data and plaintext blocks is as below:

for all blocks σ_i^{AD} {

$$\sigma_i^{AD} := pad(\sigma_i^{AD})$$

$$S_{[1026]} := S_{[1026]} \oplus \sigma_i^{AD}$$

$$S := P_6(S)$$

}

for all blocks σ_i^P {

$$c_i := S_{[l]} \oplus \sigma_i^P \text{ (} l \text{ is the length of } \sigma_i^P \text{)}$$

$$\sigma_i^P := pad(\sigma_i^P)$$

$$S_{[1026]} := S_{[1026]} \oplus \sigma_i^P$$

$$S := P_6(S)$$

}

2.2.5 Tag generation

After the AD and P are processed, the 128-bit tag T is derived: (T_0 and T_1 are two 64-bit words of T).

$$T_0 := S[0][0]$$

$$T_1 := S[0][1]$$

The decryption and verification is trivial and we omit it here.

2.2.6 Security goals of ICEPOLE

The main security goals of ICEPOLE are: 128-bit encryption security for ICEPOLE-128 and ICEPOLE-128a; 256-bit encryption security for ICEPOLE-256a; and 128-bit

2.3. DIFFERENTIAL-LINEAR DISTINGUISHING ATTACK

authentication security for all variants.

An important property of ICEPOLE is the intermediate level of robustness under nonce-misuse circumstance. It is claimed that

1. *"...in the case of nonce reuse the key-recovery attack is not possible".*
2. *"Authenticity (integrity) in the duplex construction does not need a nonce requirement, thus is not threatened by nonce reuse."*

2.3 Differential-Linear Distinguishing Attack on ICEPOLE

In [112], the designers have performed initial cryptanalysis on ICEPOLE, including differential cryptanalysis, linear cryptanalysis and rotational cryptanalysis. In this section, we will revisit the cryptanalysis in [112] and introduce our analysis on ICEPOLE based on differential-linear cryptanalysis. We emphasize that our attacks only work under the assumption that the *nonce* and *secret message number* can be reused.

The main idea is to query messages with certain input difference and analyze the statistics of the differences of chosen bits (according to the linear mask) in the output. When the XORed differences of the chosen bits have a significant bias from 0.5, the adversary can distinguish the cipher from a random permutation. In [104], Lu studied the implicit assumptions made in [101] and [26] and gave a theorem to compute the probability for the differential-linear distinguisher under the original two assumptions:

1. The involved round functions behave independently.
2. The two inputs $E_0(P)$ and $E_0(P \oplus \alpha)$ of the linear characteristic for E_1 behave as independent inputs with respect to the linear characteristic, where E_0 is

the encryption for the differential rounds and E_1 is the encryption for the linear rounds, P is the plaintext and α is the input difference.

When the analysis of encryption is divided as differential step followed by linear step. Let $\hat{p}$ be probability that the input mask of the linear step has no XORed difference after the differential step. Let ϵ be the linear characteristic bias for the linear step. The theorem says that the probability that the XORed difference of the output linear mask is 0 is $1/2 + 2(2\hat{p} - 1)\epsilon^2$.

Hence, when the bias is large enough, it is possible to distinguish it from a random permutation. In the analysis on ICEPOLE, we first divide the encryption P_6 into two steps with equal number of rounds. So the first 3 rounds will be the differential step and the last 3 rounds will be the linear step.

Our task is to find good 3-round differential characteristics and 3-round linear characteristics. Unless otherwise specified, we are discussing the ICEPOLE-128 and ICEPOLE-128a in this section under the nonce-misuse assumption. The ICEPOLE-256a is similar and we will discuss it later.

2.3.1 Constructing the differential characteristics

In ICEPOLE, the S-box is the only non-linear operation, and the maximum differential probability of the S-box is 2^{-2} . The differential probability is largely determined by the number of active S-boxes. Although the designers expect that only 3% of the difference transitions has the maximum probability 2^{-2} , this probability is not rare in the early rounds, because most of the active S-boxes have 1 bit input difference after the diffusion. The differential probability of a single ICEPOLE S-box is 2^{-2} when the input and output differences are identical with weight 1. Those 1-to-1 identical difference transitions are preferred to the 1-to- n ($n \geq 2$) cases even with the same probability as they will propagate to a lower number of active S-boxes in the next round.

2.3. DIFFERENTIAL-LINEAR DISTINGUISHING ATTACK

In [112], the designers analyzed the minimum number of active S-boxes using a SAT solver and found that the minimum number is 9 for 3 rounds ICEPOLE. There is an intuitive way to construct the differential characteristics that reach this lower bound. We can place 1 active S-box in the middle round and let it propagate backward and forward for one round. Then the number of active S-boxes will be in the form of 4–1–4, which reaches the minimum number 9.

However, the above 3 round differential path is not feasible. In fact, only at most 1024 bits (for ICEPOLE–256a, 960 bits) out of the 1280 bits in a state can be affected by the plaintext and are feasible to introduce the difference. Thus, we have the following observation on the operation μ .

Observation 2.3.1. *For any 20-bit slice, when the output difference is one bit after μ , there is at least one bit input difference at the last column ($S[\cdot][4]$).*

This observation can be easily verified by analyzing the inverse operation of μ .

Therefore, when an active S-box is propagated backward, it is likely that a number of active bits will be propagated to the last column of the input state, which is infeasible to introduce. And we programmed to verify that it is impossible to find such feasible 4–1–4 differential path.

It implies that for any feasible differential characteristics of ICEPOLE, the minimum number of active S-boxes is 2 in the first round. As a result, we will consider differential characteristics with two active S-boxes in the first round. The ideal case is that the output difference of each active S-box is only 1 bit. Then after μ , this 1-bit difference in a slice will propagate to 4 bits. So we expect the good differential characteristics will have 8 active S-boxes in round 2 and no more than 32 active S-boxes in round 3.

We searched for good 3-round differential characteristics, and found 5 possible initial differences in Table 2.1 (the rotated differences on the 64-bit word are not considered here), which can lead to feasible 3-round differential characteristics. We will name them as D_1, D_2, D_3, D_4 and D_5 .

(a) Initial difference 1

0x0	0x0	0x1	0x0	0x0
0x1	0x0	0x1	0x0	0x0
0x1	0x1	0x1	0x1	0x0
0x0	0x1	0x0	0x0	0x0

(b) Initial difference 2

0x0	0x0	0x1	0x0	0x0
0x1	0x1	0x1	0x1	0x0
0x0	0x1	0x0	0x1	0x0
0x1	0x0	0x1	0x0	0x0

(c) Initial difference 3

0x1	0x1	0x0	0x0	0x0
0x1	0x0	0x0	0x1	0x0
0x0	0x0	0x0	0x1	0x0
0x1	0x1	0x1	0x0	0x0

(d) Initial difference 4

0x0	0x0	0x0	0x0	0x0
0x1	0x0	0x1	0x0	0x0
0x1	0x1	0x1	0x1	0x0
0x0	0x1	0x0	0x1	0x0

(e) Initial difference 5

0x0	0x0	0x0	0x0	0x0
0x0	0x1	0x0	0x1	0x0
0x1	0x0	0x1	0x0	0x0
0x1	0x1	0x1	0x1	0x0

Table 2.1: The initial differences. Each entry represents a 64-bit word.

The total number of 3-round active S-boxes is 42, which implies that the differential probability is at most 2^{-84} . But in fact, the only requirement is that the input linear mask does not have XORed difference at the beginning of round 4. Hence, a large number of differential characteristics are satisfied. As long as the number of active S-boxes is relatively low (note that 32 is only 1/8 of the total 256 S-boxes), there is a high probability for the differential step.

2.3.2 Constructing the linear characteristic

The bias of an ICEPOLE linear characteristic is determined by the S-boxes involved in the linear characteristic. We use the following method to construct the 3-round linear characteristics.

Take a single bit in both the input and output masks of an S-box in the middle round. Then find related bits in the input of the first round (*input linear mask*) and the output of the third round before the S-box (*output linear mask*), assuming that the S-boxes have input masks equal to the output masks. The ICEPOLE S-box has maximum bias 2^{-2} for a linear characteristic. All the 1-bit linear approximations where the input mask equals the output mask have bias 3/16 which is nearly optimal.

The most essential criterion for the input and output masks is low weight. When the input linear mask has lower weight, the probability that the input linear

2.3. DIFFERENTIAL-LINEAR DISTINGUISHING ATTACK

mask does not have XORed difference after the 3-round differential step will be higher. When the output linear mask has lower weight, a lower number of round 6 S-boxes will be involved in the linear relation.

Two good linear characteristics we found are given in Table 2.2. They are denoted as L_1 and L_2 .

(a) Linear characteristic 1, input linear mask

0x0	0x8000000000000000	0x2000000000	0x20000	0x8000000000000040
0x40	0x0	0x8000000000000000	0x2000020000	0x0
0x0	0x2000000000	0x8000000000000000	0x40	0x2000020000
0x0	0x0	0x800002000020000	0x0	0x40

(b) Linear characteristic 1, output linear mask

0x40000	0x1	0x0	0x800000000000	0x0
0x0	0x4	0x0	0x0	0x0
0x0	0x200000	0x2000000000000000	0x0	0x0
0x0	0x0	0x20000000000	0x1000000000000000	0x0

(c) Linear characteristic 2, input linear mask

0x120000004000000	0x8000000000000000	0x100000000000	0x0	0x0
0x8020000000000000	0x4000000	0x8000100000000000	0x0	0x1000000000000000
0x8100000000000000	0x2000000000000000	0x0	0x100000000000	0x4000000
0x4000000	0x8100100000000000	0x0	0x0	0x2010000000000000

(d) Linear characteristic 2, output linear mask

0x40000	0x1	0x0	0x0	0x0
0x8000	0x4	0x0	0x0	0x0
0x0	0x0	0x2000000000000000	0x0	0x0
0x0	0x400	0x20000000000	0x1000000000000000	0x0

Table 2.2: The linear characteristics, with input mask = output mask on the S-boxes

2.3.3 Observations to improve the attack

The following observation will be helpful in the selection of the differential characteristics.

Observation 2.3.2. *When the 1024 bits $S[0..3][0..3]$ of an ICEPOLE state S are known, we are able to determine $S[0][2]$, $S[1][3]$, $S[2][4]$, $S[3][0]$, $S[3][2]$ after μ , ρ and π operations are performed on S .*

According to the above observation, we are able to determine the values of certain input bits to the S-boxes in the first round. Thus, we can update the differential tables for the S-boxes with fixed input values.

The following example shows how this will help to improve the differential probability. Suppose the input of an S-box is in the form “1 * 0 * *”, where the ‘*’s are the unknown bits and the ‘1’ and ‘0’ are the bits with fixed values, and the input difference is 2, then the output difference is 2 with probability 1, which is larger than the 0.25 in the general case. To get the fixed values, we can manipulate the input bits according to the mask in Table 2.3.

0x0	0x800000010	0x200000001	0x0	0x0
0x800000010	0x0	0x0	0x200000001	0x0
0x0	0x800000010	0x0	0x200000001	0x0
0x800000010	0x0	0x800000010	0x200000001	0x0

Table 2.3: Input mask for the fixed bits in round 1.

By setting the bits in $S[0][1]$ selected by the mask 0x800000010 as ‘1’ and all the other bits selected by the other masks as ‘0’, the two active S-boxes in round 1 will satisfy the above condition on the input values.

Our next observation is about the S-box in the round 6 of P_6 .

Observation 2.3.3. *When 4 of the 5 bits in the output of an S-box are known, it is possible to recover some of the input bits from the output bits of the S-boxes. Table 2.4 provides the probability that we can recover a bit at each position of the S-box input.*

Since the ciphertext is generated by XORing the keystream and the plaintext, we can obtain the values of at most 1024 bits in the state after 6 rounds. For

Position	0	1	2	3	4
Probability	$\frac{6}{8}$	$\frac{5}{8}$	$\frac{4}{8}$	$\frac{4}{8}$	$\frac{1}{8}$

Table 2.4: Probability that the input bits can be recovered for an S-box at round 6.

2.3. DIFFERENTIAL-LINEAR DISTINGUISHING ATTACK

0x0	0x0	0x0	0x0	0x0
0x0	0x0	0x0	0x0	0x0
0x0	0x0	0x0	0x0	0x0
0x0	0x400	0x20000000000	0x0	0x0

Table 2.5: The difference of state before the first S-box in D_2 .

ICEPOLE-128 and ICEPOLE-128a, 4 of the 5 bits in the output of all the S-boxes are known. And for ICEPOLE-256a, 4 of the 5 bits for 192 S-boxes and 3 of the 5 bits for 64 S-boxes in the output are known.

Therefore, instead of using the bias of the linear relation in the round 6, we can recover the chosen input bits in round 6 before S-box, by enumerating the values of output bits. And thus compute the value of the bit in the linear relation in the output of round 5.

For example, if we consider the output linear mask of L_1 , the probability that we can recover the 8 bits can be computed as

$$Pr_{L_1} = (3/4) \times (5/8)^3 \times (1/2)^4 = 2^{-6.45}.$$

And using these 8 bits, we can compute the value of bit $S[1][1][0]$ at the output of round 5.

Similarly, the probability for L_2 is

$$Pr_{L_2} = (3/4)^2 \times (5/8)^3 \times (1/2)^3 = 2^{-5.86}.$$

2.3.4 Concatenating the differential and linear characteristics

After constructing the 3-round differential characteristics and 3-round linear characteristics, we can concatenate them to form 6-round differential-linear characteristics.

First, we choose the initial difference D_2 because the two active S-boxes in round 1 are in the last row which has two known bits. The difference of state before the S-box for D_2 is given in Table 2.5.

To choose the 3-round linear characteristic, we consider all the possible rotations of the two linear characteristics L_1 and L_2 given in Section 2.3.2. There are 128 possible rotated linear characteristics. The selection of linear characteristic is done experimentally:

1. Randomly pick 1024-bit plaintext blocks pairs with the chosen initial difference D_2 and the fix values (1, 0) for the two known bits at positions 0 and 2 in the round 1 active S-boxes.
2. After 5 rounds, verify whether the XORed difference of the states under the linear characteristic is zero or not. If it is zero, add it to a counter *cntSame*.
3. Repeat the above process, and choose the one with highest bias at *cntSame* from 0.5.

For D_2 , the highest bias is $2^{-9.2}$ and the linear characteristic is L_1 left-rotated by 33 bits.

We remark that the experimentally determined bias is significantly higher than the one computed using the Theorem 1 in [104]. The reason may be that Assumption 2 does not hold in this case. Namely, the correlation between two linear steps cannot be ignored. Therefore, we will use the experimental results for the biases throughout this chapter.

From the bias above, we immediately get a distinguishing attack on ICEPOLE.

1. Generate $2^{33.9}$ pairs of two-block plaintext such that the first 1024-bit plaintext block has initial difference D_2 and the fixed values as specified above. All the other bits are random.
2. Use ICEPOLE to encrypt the plaintext blocks and then decide the 1024 bits input and output state of the first P_6 . Discard those pairs if the two bits in round 5 output in the linear characteristic L_1 left-rotated by 33 bits cannot be recovered. Note that for each bit, the probability is $2^{-6.45}$ to recover. So there

2.4. STATE RECOVERY ATTACK ON ICEPOLE

are 2^{21} pairs left. Then compute the bit at position $S[1][1][33]$ of the output of round 5 using the recovered bits from round 6.

3. Analyze the bias of the XORed difference of those two bits ($S[1][1][33]$). If the bias is larger than $2^{-10.2}$ we conclude that it is the ICEPOLE encryption.

The success probability is computed by using the normal distribution to approximate the binomial distribution of the bias. For ICEPOLE, the bias is a random variable $X \sim N(n(1/2 + 2^{-9.2}), n(1/2 + 2^{-9.2})(1/2 - 2^{-9.2}))$, where n is the number of pairs of the recovered 5 rounds output. When $n = 2^{21}$, the probability that $X \geq 2^{-10.2}$ is 99.3%. Assuming for a random permutation, the bias is a random variable $Y \sim N(n/2, n/4)$. The value is larger than $2^{-10.5}$ with probability 0.7%. Hence, we have a very good chance to distinguish the ICEPOLE encryption from a random permutation by using $2^{35.9}$ plaintext blocks (a pair of two-block messages are counted as 4 plaintext blocks), assuming the nonce can be reused.

2.4 State Recovery Attack on ICEPOLE

In this section, we will use the differential-linear characteristics to launch state recovery attacks on ICEPOLE.

2.4.1 State recovery attacks on ICEPOLE-128 and ICEPOLE-128a

For ICEPOLE-128 and ICEPOLE-128a, there are 256 unknown bits in the state before P_6 . They are in the last column of each slice. For convenience, we denote those four 64-bit unknown words in the last column as $\{U_0, U_1, U_2, U_3\}$ according to the row index. We will recover them step by step.

values	bias (log based 2)
$b_1 = 0, b_3 = 0$	-13.0
$b_1 = 1, b_3 = 0$	-7.3
$b_1 = 0, b_3 = 1$	-13.9
$b_1 = 1, b_3 = 1$	-11.9

2.4.1.1 Recovering U_0 and U_3

To recover the unknown bits in the state, we will analyze the input values at the active S-boxes. For D_2 , there are two active S-boxes in round 1. One has input difference 2 and the other has input difference 4. We will focus on the one with input difference 4 and denote the five input bits as $b_0, b_1, \dots, b_4$ according to their positions (b_0 is the least significant bit).

When the two bits of an ICEPOLE S-box are fixed with values $b_0 = 1$ and $b_2 = 0$, there are 8 possible values for the remaining three bits. When the input difference is 4 (at b_2), the output difference have weight 1 only when $b_1 = 1$ and $b_3 = 0$. Intuitively, the lower the weight of the output difference after the first round, the higher the probability that there is no XORed difference at the output linear mask after 5 rounds. Hence, it is possible to relate the input value of the active S-box in round 1 to the bias of XORed output difference in round 5.

We experimentally find the following biases for different values of the input bit b_1 and b_3 . Note that we collected 2^{30} data in the experiments to compute the bias and repeated for several times. When the bias is less than 2^{-14} , the experimental results were not very stable, and the average number is listed in the table. In fact, it is not necessary to consider those low biases as only the highest ones could be useful for our analysis.

Note that b_1 is related to an unknown bit in U_3 , and b_3 is related to two unknown

2.4. STATE RECOVERY ATTACK ON ICEPOLE

bits, in U_0 and U_3 . So we have following relations:

$$b_1 = U_3^{31} \oplus a_0$$

and

$$b_3 = U_0^{49} \oplus U_3^{49} \oplus a_1,$$

where U^x is the x -th bit in the 64-bit word U ; a_0 and a_1 are constants can be computed from the 1024-known bits.

We describe the state recovery process given as below:

1. Generate $2^{33.9}$ pairs of two-block plaintext satisfied following requirements.
The first block of the plaintext has difference D_2 and each active S-box has fixed values '1' and '0' in bit 0 and 2 respectively. All the other bits are random.
2. Use ICEPOLE to encrypt the plaintext blocks and then decide the 1024 bits input and output state of the first P_6 . Discard the pairs if the two bits in round 5 output in the linear characteristic L_1 left-rotated by 33 bits cannot be recovered. There are 2^{21} pairs left. Then compute the bit at position $S[1][1][33]$ of the output of round 5 using the recovered bits from round 6.
3. If the two bits at the position $S[1][1][33]$ are the same, we compute the values of a_0 and a_1 according to the input and increase the counter for the value of (a_0, a_1) by 1.
4. Suppose the largest counter of (a_0, a_1) takes value $a_0 = v_0$ and $a_1 = v_1$, we guess that $U_3^{31} = v_0 \oplus 1$ and $U_0^{49} \oplus U_3^{49} = v_1$.
5. By rotating the differential-linear characteristic for the other 63 bits, we can recover the two 64-bit unknown words U_0 and U_3 .

To assess the success probability, it is equivalent to compute the probability that a random variable $X \sim N(n(1/2 + 2^{-7.3}), n(1/2 + 2^{-7.3})(1/2 - 2^{-7.3}))$ has value

0x0	0x0	0x0	0x0	0x0
0x0	0x4	0x0	0x0	0x0
0x0	0x200000	0x0	0x0	0x0
0x0	0x0	0x0	0x0	0x0

Table 2.6: The difference of state before the first S-box in D_2 .

greater than the random variable $Y \sim N(n(1/2 + 2^{-11.9}), n(1/2 + 2^{-11.9})(1/2 - 2^{-11.9}))$. When $n = 2^{19}$, the probability is almost 1. Since we have 2^{21} pairs of input, each of the four choices of b_0 and b_1 will be around 2^{19} pairs.

We remark that here the probability is high enough such that even if the experiment is repeated for 64 times, the success probability is still close to 1.

2.4.1.2 Recovering U_2

Assuming that U_0 and U_3 have been recovered correctly, we can use similar method to recover U_2 . In this case, we use D_1 and L_2 left-rotated by 58 bits as the differential-linear characteristic.

The difference of state before the first round S-box for D_1 is given in the Table 2.6.

Fixed bits: With the knowledge of U_0 and U_3 , we fix bit 0, 2, 3 with the values (1, 0, 0) respectively for the active S-box at the second row. This is to ensure the output difference of this active S-box has weight exactly 1. And we fix bit 0, 4 with the values (1, 1) for the active S-box at the third row. Then, the weight of output difference can be distinguished from the input value of bit 2, which is denoted as b_2 .

We experimentally find the following biases for b_2 after 5 rounds. The biases are based on the difference of the output bit at position $S[3][1][58]$.

2.4. STATE RECOVERY ATTACK ON ICEPOLE

values	bias (log base 2)
$b_2 = 0$	-11.0
$b_2 = 1$	-15.4

From the active S-box at the third round, it is possible to find following relation:

$$b_2 = U_2^{24} \oplus a,$$

where a is a constant can be computed from the known input bits.

The state recovery process process:

1. Generate $2^{36.7}$ pairs of two-block plaintext satisfying following requirements.
The first block of the plaintext has difference D_1 and each active S-box has fixed values according to the above paragraph. All the other bits are random.
2. Use ICEPOLE to encrypt the plaintext blocks and then decide the 1024 bits input and output state of the first P_6 . Discard the pairs if the two bits in the linear relation (according to L_2 rotated by 58 bits) in the output of round 5 cannot be recovered. There are 2^{25} pairs left. Then compute the bit at position $S[3][1][58]$ of the output of round 5 using the recovered bits from round 6.
3. If the two bits at the position $S[3][1][58]$ are the same, we compute the value of a according to the input and increase the counter for that value by 1.
4. Suppose the largest counter of a take value $a = v$, we guess that $U_1^0 = v$.
5. By rotating the differential-linear characteristic for the other 63 bits, we can recover the 64-bit unknown word U_1 .

The estimated success probability is 99.6% for each bit.

0x0	0x0	0x8000000000000000	0x0	0x0
0x8000	0x0	0x0	0x0	0x0
0x0	0x0	0x0	0x0	0x0
0x0	0x0	0x0	0x0	0x0

Table 2.7: The differential of state before the first S-box in D_3 .

2.4.1.3 Recovering U_1

At this stage, we assume that U_0 , U_2 and U_3 have been recovered correctly. In this case, we select D_3 and L_2 left-rotated by 35 bits as the differential-linear characteristic.

The differential of state before the first round S-box for D_3 is given in the Table 2.7.

Fixed bits: We fix bit 1, 2, 4 with the values (1, 0, 1) for the active S-box at the first row. It is to ensure that the weight of output difference of this active S-box is determined by the value of input bit 3 which is denoted as $b_{0,3}$. And we fix bit 0, 2, 3, 4 with the values (0, 1, 1, 1) for the active S-box at the second row. It is to ensure that the weight of output difference is determined by the value of input bit 1 which is denoted as $b_{1,1}$.

The $b_{0,3}$ and $b_{1,1}$ are related to the unknown bits:

$$b_{0,3} = a_0 \oplus U_1^{12}$$

$$b_{1,1} = a_1 \oplus U_1^{13}$$

where a_0 and a_1 are constants can be computed from the known input.

We experimentally find the biases for different values of the input bit $b_{0,3}$ and $b_{1,1}$ in Table 2.8. The biases are based on the difference of the output bit at position $S[3][1][35]$.

We remark that the biases other than the first row in the table may not be very

2.4. STATE RECOVERY ATTACK ON ICEPOLE

values	bias (log base 2)
$b_{0,3} = 0, b_{1,1} = 0$	-11.2
$b_{0,3} = 1, b_{1,1} = 0$	-15.2
$b_{0,3} = 0, b_{1,1} = 1$	-16.4
$b_{0,3} = 1, b_{1,1} = 1$	-14.8

Table 2.8: Biases for different values of the input bits $b_{0,3}$ and $b_{1,1}$.

accurate because they are quite small numbers.

The state recovery process process:

1. Generate $2^{37.7}$ pairs of two-block plaintext satisfied following requirements.
The first block of the plaintext has difference D_3 and each active S-box has fixed values according to the above paragraph. All the other bits are random.
2. Use ICEPOLE to encrypt the plaintext blocks and then decide the 1024 bits input and output state of the first P_6 . Discard the pairs if the two bits in the linear relation (according to L_2 rotated by 35 bits) in the output of round 5 cannot be recovered. There are 2^{26} pairs left. Then compute the bit at position $S[3][1][35]$ of the output of round 5 using the recovered bits from round 6.
3. If the two bits at the position $S[3][1][35]$ are the same, we compute the value of a_0 and a_1 according to the input and increase the counter for (a_0, a_1) by 1.
4. Suppose the largest counter of (a_0, a_1) take value $a_0 = v_0$ and $a_1 = v_1$, we guess that $U_1^{12} = v_0$ and $U_1^{13} = v_1$.
5. By rotating the differential-linear characteristic for the other 31 even bits less than 64, we can recover the 64-bit unknown word U_1 .

Note that in each rotation of the differential-linear characteristic, we are able to recover two consecutive bits, so we only need to test 32 rotations to recover the 64-bit U_1 . The estimated success probability is 98.7% for each bit.

2.4.1.4 Correcting the recovered state

In the previous state recovery attack, only the first 128 bits can be correctly recovered with probability almost 1. For the other 128 bits, although the success probability is around 99%, it is still possible that the state we recovered has some error bits. Whether the state is correct can be easily verified by encrypting some new messages and compare the ciphertext.

We can correct up to 7 error bits with relatively low complexity. To correct i error bits is to choose any i bits from the 128 bits and flip the values. Then test whether the modified unknown state is correct.

Suppose that for the 128 bits U_1 and U_2 , the probability that each bit is correct is 0.99, we can compute the probability that the number of error bits is less than 8 as

$$\sum_{i=0}^7 \binom{128}{i} \times .99^{128-i} \times .01^i = 0.99995.$$

The total number of encryptions to correct up to 7 error bits is $2^{37.5}$, which is negligible to the whole attack.

2.4.1.5 Summary of the attack

The data complexity is:

- U_0 and U_3 : $2 \times 2 \times 2^{33.9} \times 2^6$. The first 2 is from the two blocks and the second 2 is from the pair of messages.
- U_2 : $2 \times 2 \times 2^{36.7} \times 2^6$
- U_1 : $2 \times 2 \times 2^{37.7} \times 2^5$

2.4. STATE RECOVERY ATTACK ON ICEPOLE

- Total: $2^{45.8}$

The time complexity is the $2^{45.8}$ encryptions of the one block plaintext and the possible $2^{37.5}$ encryptions for correction.

The memory cost is mainly on the storage of some counters, which is negligible.

The success rate of this attack is close to 1, and can be adjusted through the number of input messages.

2.4.2 State recovery attack on ICEPOLE-256a

In the case of ICEPOLE-256a, there are 320 unknown bits in the input and output states in the encryption of a block. In addition to the U_0 to U_3 in the previous subsection, we use U_4 to denote the unknown 64-bit word $S[3][3]$.

Different from the ICEPOLE-128 and ICEPOLE-128a, in the last row of the output in ICEPOLE-256a, there are only 3 known bits instead of 4 known bits. Consequently, it is impossible to recover the input bits given the output bits for that row. To deal with this issue, we have to consider the linear relation of the input mask of the S-box. When the value of the input mask is less than 8, the largest bias is $3/16$, but if the value of the input mask is 8, the largest bias is $1/16$.

For L_1 , there are two mask bits placed in the last row, one is 4 and the other is 8. From the Piling-Up Lemma [105], the bias is $2^{-5.4}$.

For L_2 , there are three mask bits placed in the last row, 2, 4, and 8. By the Piling up Lemma, the bias is $2^{-6.8}$.

To increase the bias of the last row, we introduce another linear characteristic L_3 (Table 2.4.2), which has only 1 bit in the last row of the input mask of round 6 S-box.

For L_3 , the bias is $3/16$ and the probability to recover other bits is $2^{-9.6}$.

To recover U_4 , we use the differential characteristic D_2 with linear characteristic L_1 left-rotated by 33 bits. Here we don't know the value of $S[3][2]$, so we will only fix the active bits in $S[3][0]$ as 1.

(a) input linear mask

0x0	0x1000000000000000	0x20000	0x820000000000	0x4000000000000000
0x4100000000000000	0x0	0x800000000000	0x20000	0x2000000000
0x4000000000000000	0x1000000000000000	0x200000000000	0x800000020000	0x0
0x4100000000000000	0x0	0x1000000000000000	0x2000020000	0x800000000000

(b) output linear mask

0x40000	0x0	0x0	0x800000000000	0x200000000000
0x8000	0x4	0x0	0x0	0x0
0x8	0x200000	0x0	0x0	0x1000000000000
0x0	0x0	0x200000000000	0x0	0x0

Table 2.9: The linear characteristic 3, assuming S-boxes are identical mappings

values	bias (log based 2)
$b_{3,2} = 0$	-9.2
$b_{3,2} = 1$	-15.3 (negative)

Table 2.10: Biases for different values of the input bit $b_{3,2}$

We will find the value of bit 2 in the input of active S-box related to $S[3][1]$ with difference $0x400$. We denote this bit $b_{3,2}$, and we have $b_{3,2} = a \oplus U_4^{33}$, where a is a constant from the known input.

We experimentally find the biases for the input bit values $b_{3,2}$ after 5 rounds (Table 2.10). The biases are based on the difference of the output bit at position $S[1][1][33]$.

Considering the linear relation of this XORed value to the 4 bits in the two output states, the bias of the XORed difference of the 4 bits becomes 2^{-18} by the Piling up lemma. To make the guess have probability of success nearly 1, we need around 2^{40} pairs of two-block plaintext. The total data needed is around $2^{58.7}$.

Next, the complexity of recovering the other unknown words need to be amended. We omit the similar attack process and discuss the estimated complexities here.

For U_0 and U_3 , the increased bias is $2^{-8.8}$ which needs to be compensated by around $2^{17.6}$ addition messages. The probability that recovers the bits from the round 6 S-box layer will be increased by 2^4 times. The total effect is the data com-

2.4. STATE RECOVERY ATTACK ON ICEPOLE

plexity becomes $2^{55.5}$.

For U_2, D_1 with L_3 left-rotated by 13 bits will be used and for U_1, D_3 with L_1 left-rotated by 2 bits will be used. The increased bias is roughly $2^{-2.8}$ for both cases (including the increased 5-round bias). And the decreased probability for recovering the values before round 6 S-box is $2^{-7.5}$ times. Therefore, the total data complexity for recovering each bit is increased by $2^{13.1}$, which is $2^{57.8}$.

Therefore, the total data complexity of the state recovery attack for ICEPOLE-256a is estimated to be $2^{59.8}$, which is less than the constraint 2^{62} .

2.4.3 Implications of the state recovery attacks

For ICEPOLE-128, the state recovery attack implies the failure of encryption security if both the nonce and the secret message number are reused. When an adversary has the full knowledge of a state, he can invert the cipher until the secret message number is injected. Thus, the adversary can decrypt arbitrary plaintext blocks. It also implies a forgery attack on the authentication of plaintext and associated data since the valid tag for any modified associated data or plaintext block can be computed. Since both the key and secret message number are unknown, the adversary cannot recover the key.

For ICEPOLE-128a and ICEPOLE-256a, the state recovery attack implies the whole security is broken if the nonce is reused. The initialization of ICEPOLE-128a is invertible, so the adversary can directly recover the secret key from the known state. Then both the encryption and authentication are insecure.

Summary of the security of ICEPOLE under our analysis when nonce is reused is given in Table 2.11.

	ICEPOLE-128	ICEPOLE-128a	ICEPOLE-256a
confidentiality for the plaintext	46	46	60
confidentiality for the secret message number	128	-	-
integrity for the plaintext	46	46	60
integrity for the associated data	46	46	60
integrity for the secret message number	46	-	-
integrity for the public message number	46	46	60

Table 2.11: Number of bits of security when nonce and secret message number can be reused.

2.5 Experimental Results

We experimentally verified the state recovery attack on ICEPOLE-128a. And we managed to recover the 256 unknown bits practically. The experiments used the state after the initialization with all zeros IV and key. The unknowns states are:

$$U_0 = 0x1e7aed5bfaeb535f$$

$$U_1 = 0xe0dcc6422595e5ba$$

$$U_2 = 0x892bf76586876c23$$

$$U_3 = 0x8b2ef3bf50e902f6$$

To recover U_0 and U_3 , we run the attack in Sect. 2.4.1.1 on a server with 64 cores (AMD Opteron(tm) Processor 6276). Instead of checking the constants from input, we used an equivalent method: directly extract the input bits of the active S-box in the first round, and decide whether they are the estimated ones. The number of plaintext pairs we used is 2^{34} for each bit. The attacks takes 15.3 hours and all the 128 bits recovered are correct.

To recover U_2 , we run the attack in Sect. 2.4.1.2 on the server. The number of plaintext pairs we used is 2^{37} for each bit. The attacks takes 3.5 days and all the bits are correct.

To recover U_1 , we run the attack in Sect. 2.4.1.3 on the server. The number of plaintext pairs we used is 2^{38} for each bit. The attacks takes 3.5 days and there is

2.6. CONCLUSION

one error bit.

Since the number of error bits is very small, the experiments show that our state recovery attack indeed works for ICEPOLE-128a.

2.6 Conclusion

In this chapter, we analyzed the security of the ICEPOLE family of authenticated ciphers using differential-linear cryptanalysis when the nonce is misused. ICEPOLE is strong against differential cryptanalysis since only part of the input difference is affected by message in the attack (so the best differential attack against the permutation cannot be applied to break ICEPOLE); and ICEPOLE is strong against linear cryptanalysis since only part of the input and output of the permutation are known in the attack (so the best linear attack against the permutation cannot be applied to break ICEPOLE). We successfully developed the differential-linear cryptanalysis against ICEPOLE by bypassing the input/output constraints of ICEPOLE. Our attacks show that the states of all the ICEPOLE variants can be recovered, and the secret key of ICEPOLE-128a and ICEPOLE-256a can also be recovered. The security claim of ICEPOLE is incorrect.

From the attacks against ICEPOLE, the lesson we learned is that when we are designing a strong cipher based on a permutation, it is better to consider the best attacks against the permutation without the input/output constraints. If the input/output constraints are exploited to improve the efficiency of a cipher, the designers should analyze whether the input/output constraints could be bypassed in the attacks.

Chapter 3

Initialization Differential Attacks on ZUC

3.1 Introduction

The 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3 were proposed for inclusion in the “4G” mobile standard LTE (Long Term Evolution). Stream cipher ZUC was designed by the Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences. In July 2010, the ZUC 1.4 [141] was made public for evaluation.

We developed two key recovery attacks against ZUC 1.4 [153], and our attacks directly led to the tweak of ZUC 1.4 into ZUC 1.5 [142] in Jan 2011. The latest version, ZUC 1.6 [143], was released in June 2011 (ZUC 1.6 and ZUC 1.5 have almost the same specifications).

In this chapter, we present the details of our differential attacks against ZUC 1.4. Our attacks against ZUC are similar to the differential attacks against Py, Py6 and Pypy [155], in which different IVs result in identical keystreams. In the first attack against ZUC 1.4, the difference is at the first byte of the IV, and one in $2^{15.4}$ keys results in identical keystreams after testing $2^{13.3}$ IV pairs for each key. Once

identical keystreams are detected, the key can be recovered with complexity $2^{99.4}$. In the second attack against ZUC 1.4, the difference is at the second byte of the IV, and identical keystreams can be obtained after testing 2^{54} IVs. The key can be recovered with complexity 2^{67} .

This chapter is organized as follows. The notations and the description of ZUC 1.4 are give in Section 3.2. The overview of the attack is is given in Section 3.3. In Section 3.4 and 3.5, we present the key recovery attack with difference at the first byte and the second byte of IV, respectively. We suggest the tweak to fix the flaw in Section 3.6. Section 3.7 concludes the chapter.

3.2 Preliminaries

3.2.1 The Notations

In this chapter, we follow the notations used in the ZUC specifications [141].

a_H	the most significant 16 bits of integer a
a_L	the least significant 16 bits of integer a
0_n	the sequence of n bits 0
1_n	the sequence of n bits 1
$\bar{y}$	The bitwise complement of y
$(a_1, a_2, \dots, a_n) \rightarrow (b_1, b_2, \dots, b_n)$	It assigns the values of a_i to b_i in parallel.

An integer a can be written in different formats. For example,

a	$=$	25	decimal representation
	$=$	0x19	hexadecimal representation
	$=$	00011001 ₂	binary representation

3.2. PRELIMINARIES

We number the least significant bit with 1 and use $A[i]$ to denote the i th bit of a A . And use $B[i..j]$ to denote the bit i to bit j of B .

3.2.2 The general structure of ZUC 1.4

ZUC is a word-oriented stream cipher with 128-bit secret key and a 128-bit initial vector. It consists of three main components: the linear feedback shift register (LFSR), the bit-reorganization (BR) and a nonlinear function F . The general structure of the algorithm is illustrated in Fig. 3.1.

Figure 3.1: General structure of ZUC

3.2.2.1 Linear feedback shift register(LFSR).

It consists of sixteen 31-bit registers $s_0, s_1, \dots, s_{15}$, and each register is an integer in the range $\{1, 2, \dots, 2^{31} - 1\}$. During the keystream generate stage, the LFSR is updated as follows:

LFSRUpdate():

1. $s_{16} = (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0) \bmod (2^{31} - 1);$
2. If $s_{16} = 0$ then set $s_{16} = 2^{31} - 1;$
3. $(s_1, s_2, \dots, s_{15}, s_{16}) \rightarrow (s_0, s_1, \dots, s_{14}, s_{15}).$

3.2.2.2 Bit-reorganization function.

It extracts 128 bits from the state of the LFSR and forms four 32-bit words X_0 , X_1 , X_2 and X_3 as follows:

Bitreorganization():

1. $X_0 = s_{15H} || s_{14L};$
2. $X_1 = s_{11L} || s_{9H};$
3. $X_2 = s_{7L} || s_{5H};$
4. $X_3 = s_{2L} || s_{0H};$

3.2.2.3 Nonlinear function F .

It contains two 32-bit memory words R_1 and R_2 . The description of F is given below. In function F , S is the Sbox layer and L_1 and L_2 are linear transformations as defined in [141]. The output of function F is a 32-bit word W . The keystream word Z is given as $Z = W \oplus X_3$.

$F(X_0, X_1, X_2)$:

1. $W = (X_0 \oplus R_1) \boxplus R_2;$

3.2. PRELIMINARIES

2. $W_1 = R_1 \boxplus X_1;$
3. $W_2 = R_2 \oplus X_2;$
4. $R_1 = S(L_1(W_{1L}||W_{2H}));$
5. $R_2 = S(L_2(W_{2L}||W_{1H}));$

3.2.3 The initialization of ZUC 1.4

The initialization of ZUC 1.4 consists of two steps: loading the key and IV into the register, and running the cipher for 32 steps with the keystream word being used to update the state.

3.2.3.1 Key and IV loading.

Denote the 16 key bytes as k_i ($0 \leq i \leq 15$), the 16 IV bytes as iv_i ($0 \leq i \leq 15$). We load the key and IV into the register as: $s_i = (k_i||d_i||iv_i)$. The values of the constants d_i are given in [141]. The two memory words R_1 and R_2 in function F are set as 0.

3.2.3.2 Running the cipher for 32 steps.

At the initialization stage, the keystream word Z is used to update the LFSR as follows:

LFSRWithInitialisationMode(u):

1. $v = (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0) \bmod (2^{31} - 1);$
2. If $v = 0$ then set $v = 2^{31} - 1;$
3. $s_{16} = v \oplus u;$

4. If $s_{16} = 0$ then set $s_{16} = 2^{31} - 1$;
5. $(s_1, s_2, \dots, s_{15}, s_{16}) \rightarrow (s_0, s_1, \dots, s_{14}, s_{15})$.

The cipher runs for 32 steps at the initialization stage as follows:

```

InitializationStage():
  for  $i = 0$  to 31 {
    1. Bitreorganization();
    2.  $Z = F(X_0, X_1, X_2) \oplus X_3$ ;
    3. LFSRWithInitialisationMode( $Z \ll 1$ ).
  }
```

3.3 Overview of the Attacks

We notice that the LFSR in ZUC is defined over $GF(2^{31} - 1)$, with the element 0 being replaced with $2^{31} - 1$. To the best of our knowledge, it is the first time that $GF(2^{31} - 1)$ is used in the design of stream cipher. In the initialization of ZUC 1.4, we notice that XOR is involved in the update of LFSR ($s_{16} = v \oplus u$). When XOR is applied to the elements in $GF(2^{31} - 1)$, we obtain the following undesirable property:

Property 3.3.1. *Suppose that a and a' are two elements in $GF(2^{31}-1)$, $a \neq a'$, and $\bar{a} = a'$. If $b = a$ or $b = \bar{a}$, then $a \oplus b \bmod (2^{31}-1) = a' \oplus b \bmod (2^{31}-1) = 0$.*

The Property 3.3.1 shows that the difference between a and a' can get eliminated with an XOR operation! In the rest of this chapter, we exploit this property to attack ZUC 1.4 by eliminating the difference in the state.

3.3. OVERVIEW OF THE ATTACKS

In our attacks, we try to eliminate the difference in the state without the difference in the state being injected into the nonlinear function F . The reason is that if a difference is injected into F , then Sboxes would be involved, and the difference would remain in F until additional difference being injected into F , thus the probability that the difference in the state being eliminated would get significantly reduced.

We now investigate what are the IV differences that would result in the difference in the state being eliminated with high probability. The IV differences are classified into the following three types:

Type 1. $\Delta iv_i \neq 0$ for at least one value of i ($7 \leq i \leq 15$).

After loading this type of IVs into LFSR, the difference would appear at the least significant byte of at least one of the LFSR elements $s_7, s_8, \dots, s_{15}$. Note that the least significant byte of s_7 is part of X_2 in the Bit-reorganization function since $X_2 = s_{7L} || s_{5H}$, and X_2 is an input to function F . Due to the shift of LFSR, the difference at the least significant byte of $s_7, s_8, \dots, s_{15}$ would be injected into F . Thus we would not use this type of IV difference in our attacks.

Type 2. $\Delta iv_i = 0$ for $7 \leq i \leq 15$, $\Delta iv_i \neq 0$ for at least one value of i ($2 \leq i \leq 6$).

After loading this type of IVs into LFSR, the difference would appear at the least significant byte of at least one of the LFSR elements $s_2, s_3, \dots, s_6$. Note that the least significant byte of s_2 is part of X_3 in the Bit-reorganization function since $X_3 = s_{2L} || s_{0H}$, X_3 is XORed with the output of F to generate keystream word Z , and Z is used to update the LFSR. Two steps later, the difference in iv_2 would appear in the feedback function to update LFSR. It means that if there is difference in iv_2 , the difference in s_2 would be used to update the LFSR twice, and the probability that the difference would be eliminated is very small. Due to the shift of LFSR, the difference at $s_2, s_3, \dots, s_7$ would be eliminated with very small probability.

Thus we did not use this type of IV difference in our attacks.

Type 3. $\Delta iv_i = 0$ for $2 \leq i \leq 15$, $\Delta iv_0 \neq 0$ or $\Delta iv_1 \neq 0$.

The focus of our attacks is on this type of IV differences. In order to increase the chance of success, we consider the difference at only one byte of the IV. We discuss below how the difference in the state can be eliminated when there is difference in s_0 (the analysis for the difference in s_1 is similar). At the first step in the initialization,

$$s_0 = (k_0 || d_0 || iv_0), \quad (3.3.1)$$

$$v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0 \mod (2^{31} - 1), \quad (3.3.2)$$

$$s_{16} = v \oplus u. \quad (3.3.3)$$

Suppose that the difference is only at iv_0 , and $iv_0 - iv'_0 = \Delta iv_0 > 0$. From (3.3.1) and (3.3.2) we know that

$$\begin{aligned} v - v' &= (1 + 2^8)(iv_0 - iv'_0) \mod (2^{31} - 1) \\ &= \Delta iv_0 \parallel \Delta iv_0. \end{aligned} \quad (3.3.4)$$

If we need to eliminate the difference in s_{16} , from Property 1 and (3.3.3), the following condition should be satisfied:

$$v \oplus v' = 1_{31} \quad (3.3.5)$$

$$u = v \quad \text{or} \quad u = v' \quad (3.3.6)$$

According to (3.3.5), v and v' have XOR difference in the left-most 15 bits (i.e. $v[17..31]$ and $v'[17..31]$), while according to (3.3.4), the subtraction difference of those bits are 0. The only possible reason is that the 15 bits, $v[17..31]$, are all affected by the carries from the addition of Δiv_0 to v' . After testing all the one-byte differences, we found that v must be in one of the following four forms (the values of v and v' can

3.4. ATTACK ZUC 1.4 WITH DIFFERENCE AT IV_0

be swapped):

$$\begin{aligned}
 v &= 1111111111111111_2 \parallel y \parallel 1_2 \parallel y \\
 \text{or } v &= 0111111111111111_2 \parallel y \parallel 0_2 \parallel y \\
 \text{or } v &= 0000000000000000_2 \parallel \bar{y} \parallel 0_2 \parallel \bar{y} \\
 \text{or } v &= 1000000000000000_2 \parallel \bar{y} \parallel 1_2 \parallel \bar{y}
 \end{aligned} \tag{3.3.7}$$

(y is a 7-bit integer.)

There are 510 possible values of v ($v = 1_{31}$ and $v = 0_{31}$ are excluded since one of v and $\bar{v}$ cannot be 0). All the (v, v') pairs and their differences are given in Appendix B. Notice that we ignored the order of v and v' as they are exchangeable. We have obtained all the possible values of v and u for generating identical keystreams.

We highlight the following property in the table: the difference between v and v' uniquely determines the value of pair (v, v') in the table. As a result, if we know the difference of IVs that results in the collision of the state, we can determine the value of (v, v') immediately.

By eliminating the difference in the state as illustrated above, we developed two attacks against ZUC 1.4. The first attack is to exploit the difference at iv_0 , and the second attack is to exploit the difference at iv_1 . The details are given in the following two sections.

3.4 Attack ZUC 1.4 with Difference at iv_0

In this section, we present our first differential attack on the initialization by using IV difference at iv_0 and generating identical keystream. The keys that generate the same keystream are called weak keys in this attack. We will show that a weak key exists with probability $2^{-15.4}$, and a weak key can be detected with about $2^{13.3}$ chosen IVs. Once a weak key is detected, its effective key size is reduced from 128 bits to around 100 bits.

3.4.1 The weak keys for Δiv_0

We will show that when there is difference at iv_0 , about one in $2^{15.4}$ keys would result in identical keystream. For a random key, we will check whether there exists a pair of IVs such that (3.3.5), (3.3.6) and (3.3.7) can be satisfied.

We start with analyzing how keys and IVs are involved in the expression of u and v in the first step of initialization. From the specifications of the initialization, we have

$$\begin{aligned} u = Z \ll 1 &= (X_0 \oplus X_3) \ll 1 = ((s_{15H} \parallel s_{14L}) \oplus (s_{2L} \parallel s_{0H})) \ll 1 \\ &= ((k_{15} \parallel iv_2 \parallel k_0 \parallel iv_{14}) \oplus 0x6b8f9a89) \ll 1 \end{aligned} \quad (3.4.1)$$

In (3.3.2) and (3.4.1), there are 5 bytes of key, $\{k_0, k_4, k_{10}, k_{13}, k_{15}\}$, and 7 bytes of IV, $\{iv_0, iv_2, iv_4, iv_{10}, iv_{13}, iv_{14}, iv_{15}\}$ being involved in the computation of u and v . The complexity would be very high if we directly try all possible combinations of the keys and IVs. However, with analysis on the expressions of u and v , we can reduce the search space from 2^{96} to around $2^{26.3}$.

Solving (3.3.5), (3.3.6), (3.3.7) and (3.4.1), we obtain the following four groups of solutions:

Group 1.

$$\begin{aligned} u = v &= 1111111111111111_2 \parallel y \parallel 1_2 \parallel y \\ k_{15} &= 0x94 \\ iv_2 &= 0x70 \\ k_0 &= 0x9a \oplus (y \parallel 1_2) \\ iv_{14} \ll 1 &= 0x44 \oplus y \end{aligned} \quad (3.4.2)$$

3.4. ATTACK ZUC 1.4 WITH DIFFERENCE AT IV_0

Group 2.

$$\begin{aligned}
u &= v = 0111111111111111_2 \parallel y \parallel 0_2 \parallel y \\
k_{15} &= 0x14 \\
iv_2 &= 0x70 \\
k_0 &= 0x9a \oplus (y \parallel 0_2) \\
iv_{14} \ll 1 &= 0x44 \oplus y
\end{aligned} \tag{3.4.3}$$

Group 3.

$$\begin{aligned}
u &= v = 0000000000000000_2 \parallel \bar{y} \parallel 0_2 \parallel \bar{y} \\
k_{15} &= 0x6b \\
iv_2 &= 0x8f \\
k_0 &= 0x9a \oplus (\bar{y} \parallel 0_2) \\
iv_{14} \ll 1 &= 0xbb \oplus \bar{y}
\end{aligned} \tag{3.4.4}$$

Group 4.

$$\begin{aligned}
u &= v = 1000000000000000_2 \parallel \bar{y} \parallel 1_2 \parallel \bar{y} \\
k_{15} &= 0xeb \\
iv_2 &= 0x8f \\
k_0 &= 0x9a \oplus (\bar{y} \parallel 1_2) \\
iv_{14} \ll 1 &= 0xbb \oplus \bar{y}
\end{aligned} \tag{3.4.5}$$

Furthermore, from (3.3.2) we compute v as follows (note that the property $2^k s_i \bmod (2^{31} - 1) = s_i \ggg k$):

$$\begin{aligned}
v &= (1 + 2^{23})k_0 + 2^7k_{15} + 2^9(k_{13} + 2^3k_4 + 2^4k_{10}) + (1 + 2^8)iv_0 \\
&\quad + 2^{15}(iv_{15} + 2^2iv_{13} + 2^5iv_4 + 2^6iv_{10}) + 0x451bfe1b \bmod (2^{31} - 1)
\end{aligned} \tag{3.4.6}$$

Let $sum_1 = k_{13} + 2^3k_4 + 2^4k_{10}$, $sum_2 = iv_{15} + 2^2iv_{13} + 2^5iv_4 + 2^6iv_{10}$. The value of sum_1 ranges from 0 to 6375, and the value of sum_2 ranges from 0 to 25755. We developed Algorithm 1 to search for weak keys.

Algorithm 1 Find weak keys for Δiv_0

```

for  $(k_{15}, iv_2)$  in each of the 4 groups of solutions (3.4.2), (3.4.3), (3.4.4), (3.4.5) do
  for  $y = 0$  to 127 do
    determine  $iv_{14} \ll 1$  and  $k_0$ 
    for  $sum_1 = 0$  to 6375 do
      for  $iv_0 = 0$  to 255 do
         $keySum \leftarrow 2^7 k_{15} + (2^{23} + 1)k_0 + 2^9 sum_1 \pmod{2^{31} - 1}$ 
         $sum_2 \leftarrow (u - keySum - (1 + 2^8)iv_0 - 0x451bfe1b)/2^{15} \pmod{2^{31} - 1}$ 
        if  $sum_2$  is less than 25756 then
           $v = u; v' = u \oplus 1_{32};$ 
          if  $(v - v') \pmod{2^{31} - 1}$  is a multiple of  $1 + 2^8$  then
             $\Delta iv_0 = (v - v') \pmod{(2^{31} - 1)/(1 + 2^8)};$ 
             $iv'_0 = iv_0 - \Delta iv_0;$ 
          else
             $\Delta iv_0 = (v' - v) \pmod{(2^{31} - 1)/(1 + 2^8)};$ 
             $iv'_0 = iv_0 + \Delta iv_0;$ 
          end if
          output  $u, k_0, k_{15}, sum_1, iv_0, iv'_0, iv_2, iv_{14} \ll 1, sum_2$ 
        end if
      end for
    end for
  end for
end for

```

3.4. ATTACK ZUC 1.4 WITH DIFFERENCE AT IV_0

Each output from Algorithm 1 gives the value of $(k_{15}, k_0, sum_1, iv_0, iv'_0, iv_2, iv_{14}, sum_2)$ that results in identical keystreams. Running Algorithm 1, we found $9934 = 2^{13.28}$ different outputs. We note that on average, each sum_1 from the output of the algorithm represents $2^{24}/6376 = 2^{11.36}$ possible choices of (k_4, k_{10}, k_{13}) . Thus there are $2^{13.3} \times 2^{11.4} = 2^{24.7}$ weak values of $(k_0, k_4, k_{10}, k_{13}, k_{15})$. Hence, there are $2^{24.7}$ weak keys out of 2^{40} possible values of the 5 key bytes. The probability that a random key is weak for IV difference at iv_0 is $2^{-15.4}$. The complexity of Algorithm 1 is $4 \times 128 \times 6376 \times 256 = 2^{26.3}$.

Identical key streams. We give below a weak key and an IV pair with difference at iv_0 that result in identical keystreams.

$$key = 87, 4, 95, 13, 161, 32, 199, 61, 20, 147, 56, 84, 126, 205, 165, 148$$

$$IV = 166, 166, 112, 38, 192, 214, 34, 211, 170, 25, 18, 71, 4, 135, 68, 5$$

$$IV' = 116, 166, 112, 38, 192, 214, 34, 211, 170, 25, 18, 71, 4, 135, 68, 5$$

The identical keystreams are: 0xbfe800d5 0360a22b 6c4554c8 67f00672 2ce94f3f f94d12ba 11c382b3 cbaf4b31....

3.4.2 Detecting weak keys for Δiv_0

We have shown above that a random key is weak with probability $2^{-15.4}$. In the attack against ZUC, we will first detect a weak key, then recover it. To detect a weak key, our approach is to use the IV pairs generated from Algorithm 1 to test whether identical keystreams are generated. Note that for a particular value of sum_2 , we can always find a combination of $(iv_4, iv_{10}, iv_{13}, iv_{15})$ that satisfies $sum_2 = iv_{15} + 2^2 iv_{13} + 2^5 iv_4 + 2^6 iv_{10}$. Thus a pair of IVs $(iv_0, iv_2, iv_4, iv_{10}, iv_{13}, iv_{14}, iv_{15})$ and $(iv'_0, iv_2, iv_4, iv_{10}, iv_{13}, iv_{14}, iv_{15})$ can be determined by each output of Algorithm 1. Using this result, we developed Algorithm 2 to detect weak keys for Δiv_0 .

Algorithm 2 Detecting weak keys for Δiv_0

1. Choose one of the $2^{13.28}$ outputs of Algorithm 1.
 2. Find the pair of IVs determined by this output (if iv_j does not appear in the first initialization step, set it as some fixed constant).
 3. Use the IV pair to generate two keystreams.
 4. If the keystreams are identical, output the IVs and conclude the key is weak.
 5. If all outputs of Algorithm 1 have been checked, and there are no identical keystreams, we conclude that the key is not weak.
-

In Algorithm 2, we need to test at most $2^{13.3}$ pairs of IVs to determine if a key is weak for difference at iv_0 .

3.4.3 Recovering weak keys for Δiv_0

After detecting a weak key, we proceed to recover the weak key. Once a key is detected as weak (as given from Algorithm 2), from the IV pair being used to generate identical keystreams, we immediately know the value of k_0 , k_{15} and sum_1 . Note that $sum_1 = (k_{13} + 2^3 k_4 + 2^4 k_{10})$. In the best situations, the sum is 0 or 25755, then we can uniquely determine k_4 , k_{10} and k_{13} . In the worst situation, there are 2^{12} possible choices for k_4 , k_{10} and k_{13} , and therefore, we need 2^{12} tests to determine the correct values for k_4 , k_{10} and k_{13} . On average, for each value of sum_1 , we need to test $2^{11.4}$ combinations of (k_4, k_{10}, k_{13}) .

Since there are only five key bytes being recovered in our attack, the remaining 11 key bytes should be recovered with exhaustive search. Hence, the complexity to recover all key bits is $2^{88} \times 2^{11.4} = 2^{99.4}$. From the analysis above, we also know that the best complexity is 2^{88} and the worst complexity is 2^{100} .

3.5 Attack ZUC 1.4 with Difference at iv_1

In this section, we present the differential attack on ZUC 1.4 for IV difference at iv_1 . Different from the attack in Section 4, we need to consider the computation of u and v in the second step of the initialization. For this type of IV difference, for every key, there are some IV pairs that result in identical keystreams since more IV bytes are involved. Once we found such an IV pair, we can recover the key with complexity around $2^{68.3}$.

3.5.1 Identical keystreams for Δiv_1

The computation of u and v in the second initialization step involves more key and IV bytes. The v in the second initialization step is computed as:

$$\begin{aligned} v &= (2^{15}s_{16} + 2^{17}s_{14} + 2^{21}s_{11} + 2^{20}s_5 + (1 + 2^8)s_1) \bmod (2^{31} - 1), \\ s_{16} &= ((2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0) \bmod (2^{31} - 1)) \\ &\quad \oplus (((k_{15} \parallel iv_2 \parallel k_0 \parallel iv_{14}) \oplus 0x6b8f9a89) \ll 1) \end{aligned} \quad (3.5.1)$$

And u is given as:

$$\begin{aligned} u &= (((X_0 \oplus R_1) + R_2) \oplus X_3) \ll 1 \\ X_0 &= (s_{16H} \parallel 10101100_2 \parallel iv_{15}) \\ X_3 &= (01011110_2 \parallel iv_3 \parallel k_1 \parallel 01001101_2) \\ R_1 &= S(L_1(s_{9H} \parallel s_{7L})) = f_1(iv_7, k_9) \\ R_2 &= S(L_2(s_{5H} \parallel s_{11L})) = f_2(iv_{11}, k_5) \end{aligned} \quad (3.5.2)$$

where f_1 and f_2 are some deterministic non-linear functions.

There are 10 IV bytes involved in the expression of v , i.e. $(iv_0, iv_1, iv_2, iv_4, iv_5, iv_{10}, iv_{11}, iv_{13}, iv_{14}, iv_{15})$ and 8 IV bytes involved in the expression of u , i.e. $(iv_0, iv_3, iv_4, iv_7, iv_{10}, iv_{11}, iv_{13}, iv_{15})$. In total, there are 12 IV bytes being involved in the computation of u and v , and every bit of u and v can be affected by IV. Thus

for every key, the conditions (3.3.5) and (3.3.6) can be satisfied, and identical key streams can be generated. To confirm that identical keystreams can be generated from every key, we tested 1000 random keys. Our experimental result shows that we can always find two IVs that result in identical keystreams.

In the attack, a random key and a random iv pair with difference at iv_1 , the probability that v and u satisfy the conditions (3.3.5) and (3.3.6) is $2^{-31} \times 2^{-31} \times 2 = 2^{-61}$. Choosing 2^8 ivs with difference at iv_1 , we have around 2^{15} pairs. The identical keystream pair appears with probability $2^{-61+15} = 2^{-46}$ with 2^8 IVs. We thus need about $2^{46} \times 2^8 = 2^{54}$ IVs to obtain identical keystreams.

Identical key streams. We give below a key and an IV pair with difference at iv_1 that result in identical keystreams. The algorithm being used to find the IV pair is given in Appendix C. The algorithm is a bit complicated since a number of optimization tricks are involved. The explanation of the optimization details is omitted here since our focus is to develop a key recovery attack.

$key = 123, 149, 193, 87, 42, 150, 117, 4, 209, 101, 85, 57, 46, 117, 49, 243$

$IV = 92, 80, 241, 10, 0, 217, 47, 224, 48, 203, 0, 45, 204, 0, 0, 17$

$IV' = 92, 182, 241, 10, 0, 217, 47, 224, 48, 203, 0, 45, 204, 0, 0, 17$

The identical keystreams are: 0xf09cc17d 41f12d3f 453ac0c3 cadcef9f f98fb964 ca6e576e b48b813 6c43da22 . . .

3.5.2 Key recovery for Δiv_1

After identical keystreams are generated from an IV pair with difference at iv_1 , we proceed to recover the secret key. From the table in Appendix B, we know the value of (v, v') since we know the difference at iv_1 of the chosen IV pair, and we also know the value of u since $u = v$ or $u = v'$. In the following, we illustrate a key recovery attack after identical keystreams have been detected.

3.5. ATTACK ZUC 1.4 WITH DIFFERENCE AT IV_1

1. In the expression of u in (3.5.2), (k_1, k_5, k_9, s_{16H}) is involved. Note that there are only two possible values of the 31-bit u . We try all the possible values of (k_1, k_5, k_9, s_{16H}) , then there would be $2^{8 \times 3 + 16} \times 2^{-31} \times 2 = 2^{10}$ possible values of (k_1, k_5, k_9, s_{16H}) that generate the two possible values of u . The complexity of this step is 2^{40} .
2. Next we use the expression of s_{16} in (3.5.1). For each of the 2^{10} possible values of (k_1, k_5, k_9, s_{16H}) , we try all the possible values of $(k_0, k_4, k_{10}, k_{13}, k_{15})$ and check whether the values of s_{16H} is computed correctly or not. There would be $2^{8 \times 5} \times 2^{-16} = 2^{24}$ possible values of $(k_0, k_4, k_{10}, k_{13}, k_{15})$ left. Considering that there are 2^{10} possible values of (k_1, k_5, k_9, s_{16H}) , about $2^{10} \times 2^{24} = 2^{34}$ possible values of $(k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H})$ remain. The complexity of this step is $2^{8 \times 5} \times 2^{10} = 2^{50}$.
3. Then we use the expression of v in (3.5.1). For each of the 2^{34} possible values of $(k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H})$, we try all the possible values of (k_{11}, k_{14}) and check whether the value of v is correct or not. A random value of (k_{11}, k_{14}) would pass the test with probability $2^{8 \times 2} \times 2^{-31} = 2^{-15}$. Considering that there are 2^{34} possible values of $(k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H})$, about $2^{34} \times 2^{-15} = 2^{19}$ possible values of $(k_0, k_1, k_4, k_5, k_9, k_{10}, k_{11}, k_{13}, k_{14}, k_{15})$ remain. The complexity of this step is $2^{8 \times 2} \times 2^{34} = 2^{50}$.
4. For each of the 2^{19} possible values of $(k_0, k_1, k_4, k_5, k_9, k_{10}, k_{11}, k_{13}, k_{14}, k_{15})$, we recover the remaining 6 key bytes $(k_2, k_3, k_6, k_7, k_8, k_{12})$ by exhaustive search. The complexity of this step is $2^{19} \times 2^{8 \times 6} = 2^{67}$.

The overall computational complexity to recover a key is $2^{40} + 2^{50} + 2^{50} + 2^{67} \approx 2^{67}$. We need about 2^{54} IVs in the attack. Note that the complexity in the first, second and third steps can be significantly reduced with optimization since we are dealing with simple functions. For example, meet-int-the-middle attack can be used in the first step, and the sum of a few key bytes can be considered in the

second and third steps. However, the complexity of those three steps has little effect on the overall complexity of the attack, so we do not present the details of the optimization here.

3.6 Improving ZUC 1.4

From the analysis in Sect. 3, the weakness of the initialization comes from the non-injective update of the LFSR. To fix the flaw, we proposed the tweak in the rump session of Asiacrypt 2010. Instead of using the XOR operation, it is better to use addition modulo operation over $GF(2^{31} - 1)$. More specifically, the operation $s_{16} = v \oplus u$ is changed to $s_{16} = v + u \mod (2^{31} - 1)$. With this tweak, the difference in v would always result in the difference in s_{16} if there is no difference in u , and the attack against ZUC 1.4 can no longer be applied. In the later versions ZUC 1.5 and 1.6 (ZUC 1.5 and 1.6 have almost the same specifications), the computation of s_{16} is modified using our suggested method.

3.7 Conclusion

In this chapter, we developed two chosen IV attacks against the initialization of ZUC 1.4. In our attacks, identical keystreams are generated from different IVs, then key recovery attacks are applied. Our attacks are independent of the number of steps in initialization. The lesson from this chapter is that when non-injective functions are used in cipher design, we should pay special attention to ensure that the difference can not be eliminated with high probability.

Chapter 4

Leaked-State-Forgery Attack on ALE

4.1 Introduction

ALE. ALE (Authenticated *Lightweight Encryption*) is an AES-based authenticated encryption algorithm proposed by Bogdanov *et al.* at FSE 2013 [37]. It is designed for the low-cost embedded systems (such as RFID tags and smart cards) and provides single-pass authenticated encryption with associated data. The keystream generation of ALE uses the idea of the LEX stream cipher [31], and the tag generation uses the idea of Pelican MAC [45]. It has 256-bit internal state and aims to have a probability of success at most 2^{-128} for a forgery attack.

Pelican MAC is an extremely simple MAC based on AES. In Pelican MAC, any difference being introduced in the forgery attack passes through at least four AES rounds. It ensures that the success rate of a forgery attack is at most 2^{-128} . The state size of Pelican MAC is only 128 bits. The small state size means that the number of messages being authenticated under the same key should be less than 2^{64} . Yuan *et al.* delivered a state recovery attack against the Pelican MAC by exploiting the state collision when more than 2^{64} authentication tags are generated from the same key [158]. The attack given in [158] cannot be applied to ALE. In ALE, the state size is increased to 256 bits, and a new nonce is needed for generating each authentication

tag when the same key is used.

The stream cipher LEX is based on AES, and four keystream bytes are extracted from the AES state after each round. LEX suffers from two attacks. The slide attack against LEX recovers the key with negligible complexity when around 2^{60} nonces are used with the same secret key [154]. Another attack recovers the key with around 2^{100} simple operations and 2^{40} keystream bytes [53, 55]. ALE is not vulnerable to these two attacks due to its large state and the changing AES round keys (the round keys in LEX are fixed for the same key).

The design of ALE is similar to the authenticated encryption algorithm ASC-1 [85].

In ASC-1, a leaked byte is protected by an additional key byte before it is extracted as keystream byte. However, the additional key byte is not used in ALE for better hardware efficiency. Unfortunately, the lacking of additional key bytes in ALE allows part of the AES state being leaked as keystream, and such leaked state information can be exploited to improve the forgery attack, as demonstrated in this chapter.

In this chapter, we propose a new attack – leaked-state-forgery attack (LSFA) against ALE. The general idea of this attack is to exploit the leaked state information so as to increase the differential probability. For ALE, there exists four-round AES differential characteristics with probability much larger than 2^{-128} after taking into account the leaked state information. The forgery attack against ALE can reach the success rate of 2^{-97} , which is 2^{31} higher than the claimed probability. We implemented our attack on a small version of ALE, in which 64-bit block and 4-bit-to-4-bit S-box are used. The experimental results match well with the theoretical results.

Very recently, Khovratovich and Rechberger independently proposed an attack against ALE in SAC 2013 [93] which also exploits the weakness of the ALE scheme. However, we notice that their attack is applied to a variant of ALE when the four

4.2. THE SPECIFICATION OF ALE

bytes are leaked after SubByte. In this work, we optimized the differential characteristics used in our attacks so that lower complexities can be obtained in this chapter.

This chapter is organized as follows. The specification of ALE is given in Section 4.2. Section 4.3 describes a basic forgery attack against ALE. Section 4.4 optimizes the forgery attack. Section 4.5 gives the experimental results on ALE with reduced block size. Section 4.6 concludes this chapter.

4.2 The Specification of ALE

In this section, we give a brief description of the ALE. The full specifications of ALE can be found in the original paper [37].

AES round function. AES-128 is used as an underlying primitive of ALE. A full specification of AES can be found in [44]. There are four operations in an AES round: SubBytes(SB), ShiftRows(SR), MixColumns(MC) and AddRoundKey(ARK).

```
AESRound(State, ExpandedKey[i])
{
    SubBytes(State);
    ShiftRows(State);
    MixColumns(State);
    AddRoundKey(State, ExpandedKey[i]);
}
```

LEX keystream extraction. In the stream cipher LEX, AES round functions are repeatedly applied to a state (the subkeys are fixed). At the end of each AES round, 4 bytes from the state are extracted as the keystream [31]. The positions of leaked bytes are shown in Fig. 4.1.

4.2. THE SPECIFICATION OF ALE

nonce ν as inputs. It outputs ciphertext γ of the same length as message and a 128-bit tag τ . The initialization of ALE is given as follows: the nonce ν is encrypted using AES-128 under the master key κ . The 128-bit output is used as the initial key state. A message with value 0 is encrypted using AES-128 under the master key κ to give the data state. The 128-bit output $\text{AES}_{\kappa}(0)$ is encrypted again using the initial key state as the key. The key state is updated by applying round key schedule of AES-128 to the final round key of last AES encryption with round constant x^{10} in $\mathbb{F}_{2^8}$.

To process a 16-byte message block, the data state is encrypted with 4 rounds of AES using the key state as key. 16 bytes are leaked from the data state in the 4 AES rounds in accordance with the LEX keystream extraction. According to the code provided by the authors of ALE, five round keys are used during the 4 AES rounds, namely, an initial whitening key is used. And at each AES round, four bytes are leaked after the `AddRoundKey()` function. The leak is xored to the current 16-byte block M for encryption. The final round subkey is updated one more time using the AES round key schedule with byte round constant x^4 in $\mathbb{F}_{2^8}$ to get the key state. The current message block M is xored to the data state so that it would pass through the next 4 AES rounds for authentication purpose (similar to that in Pelican MAC).

The decryption/verification is similar to the encryption/authentication, except that the ciphertext block is xored to the keystream to get the message, as shown in Fig. 4.3. We provide this figure here since the decryption/verification is important in our attack.

The designers of ALE claim that any forgery attack not involving key recovery/internal state recovery has a success probability at most 2^{-128} . It is stated that each secret key is used to protect at most 2^{48} message bits. Such restriction on message bits does not affect the success rate of our forgery attack.

Figure 4.3: Decryption and verification of ALE.

4.3 A Basic Leaked-State Forgery Attack on ALE

In this section, we present a basic forgery attack against ALE. The chance of successful forgery attack is 2^{-106} , which is 2^{22} larger than the claimed success rate 2^{-128} . This attack requires 2^{41} known plaintext blocks.

4.3.1 The main idea of the attack

The following property of active S-box will be used in our attack:

Property 4.3.1. *For an active S-box, if the values of an input and the input/output difference are known, the output/input difference is known with probability 1.*

Here the active S-box is the S-box with non-zero input difference. In the rest of the chapter, we will use a new term *active leaked byte* to denote a leaked byte with difference on it.

In the security analysis of Pelican MAC [45] and ALE [37], the probability of four-round differential characteristic of ALE follows the analysis of AES. It has been shown that for any four-round AES differential characteristic, the number of active S-boxes is at least 25 [43]. For each S-box, the differential probability is at most 2^{-6} . Hence, there is a trivial upper bound for the four-round AES differential

4.3. A BASIC LEAKED-STATE FORGERY ATTACK ON ALE

Figure 4.4: An example of 1-4-16-4 differential characteristic. Gray squares denote leaked bytes. Squares marked with broken line denote active bytes.

probability which is 2^{-150} . However, different from the Pelican MAC, 4 state bytes are leaked at the end of every round in ALE. Using Property 4.3.1, it is possible to bypass some active S-boxes with probability 1 when the input bytes to those active S-boxes are leaked. It means that the overall differential probability could be significantly increased.

4.3.2 Finding a differential characteristic

The first step of the attack is to find a valid four-round AES differential characteristic which passes through 25 (or close to 25) active S-boxes and the differences pass through several leaked bytes in the first three rounds.

There are many differential characteristics for four AES rounds. To categorize those differential characteristics, we use the number of active bytes before the S-box layer in each round to represent a certain type of differential characteristics. For example, the differential characteristic shown in Fig. 4.4 falls in the type “1-4-16-4”. Note that the positions of active bytes are not unique for each type.

In our basic attack, we use the type of differential characteristic shown in Fig. 4.4. There are 25 active S-boxes in the differential characteristic, and 8 active leaked bytes are located in the first three rounds.

Next we need to find a differential characteristic with high probability. Note that it is not always guaranteed that the differential probability of each active S-box can reach the maximum value 2^{-6} . The AES S-box has a property that for any input difference δ_1 and output difference δ_2 , the probability that equation $S(x) \oplus S(x \oplus$

$\delta_1) = \delta_2$ has a solution is 127/256. Among the 127 solutions, there are 126 solutions that have probability 2^{-7} and only one solution has probability 2^{-6} . Hence, for an active S-box, there is a unique output difference that reaches the probability 2^{-6} for difference propagation. It shows that the conditions to set active S-boxes with difference propagation probability 2^{-6} will limit the number of choices for the possible differential characteristics.

It is thus not surprising that we found no differential characteristic such that every active S-box (except those involving the leaked ones) has the maximum differential probability 2^{-6} after testing all the possible positions of the type “1–4–16–4”. In order to find a differential characteristic, we need to allow some active S-box with differential probability 2^{-7} . We managed to find a number of differential characteristics. One of them is given in Fig. 4.5, and we will use this differential characteristic to demonstrate our basic attack. The differential probability of this differential characteristic is given as $2^{-6 \times 16 + (-7) \times 9} = 2^{-159}$ (differential probability 2^{-6} for 16 active S-boxes, 2^{-7} for 9 active S-boxes).

Three differences in Fig. 4.5 will be used in our attack: the input difference Δ_{in} , the output difference Δ_{out} and the keystream difference Δ_s :

$$\Delta_{in} = (0,0,0,0; 0,0,0,0; 0,0,0,0; 0,96,0,0);$$

$$\Delta_{out} = (B1,DE,6F,6F; 0,0,0,0; B8,5C,82,55; 0,0,0,0);$$

$$\Delta_s = (0,0,E,F3; 59,37,6E,F2; 0,81,6C,0; 0,0,0,0);$$

Note that the values in Δ_s are obtained by simply concatenating the bytes extracted from the states. The order of those bytes has no effect on the attack, as long as this order is fixed.

4.3.3 Launching the forgery attack

After finding a four-round AES differential characteristic, we need to determine the values of the leaked bytes on the differential characteristic so as to improve the

4.3. A BASIC LEAKED-STATE FORGERY ATTACK ON ALE

Figure 4.5: A differential characteristic of type "1-4-16-4". The hexadecimal numbers indicate the difference values. The empty squares indicate no difference. The squares of leaked bytes are marked with gray color.

differential probability. The values of the leaked bytes are important for locating the ciphertext bytes that will be modified in the forgery attack.

In the differential characteristic shown in Fig. 4.5, the differences at the positions of leaked bytes are known before and after the S-box. Hence, we solve for the values of the active leaked bytes. There are either two or four possible solutions depending on the output difference. We store the possible values of leaked bytes in a table T (Table D.1 in Appendix D). Notice that we ignore the conditions on the leaked bytes in the fourth round because for any leaked values at the end of Round 3, we can always derive the corresponding difference in Round 4.

If the value of a keystream block s_i falls into one of the possible values of table T , we modify the previous ciphertext block c_{i-1} and the current ciphertext block c_i using the differences given in Fig. 4.5. More specifically, $c'_{i-1} = c_{i-1} \oplus \Delta_{\text{in}}$; $c'_i = c_i \oplus \Delta_{\text{out}} \oplus \Delta_s$. The modified ciphertext is sent for decryption/verification.

We illustrate here how the above attack works. From the decryption, the difference $\Delta m_{i-1} = (c_{i-1} \oplus s_{i-1}) \oplus (c'_{i-1} \oplus s'_{i-1}) = \Delta_{\text{in}}$ because $\Delta s_{i-1} = 0$; the difference $\Delta m_i = (c_i \oplus s_i) \oplus (c'_i \oplus s'_i) = \Delta_{\text{out}}$ because $c'_i \oplus c_i = \Delta_{\text{out}} \oplus \Delta_s$. Then Δm_{i-1} is introduced to the data state, and after four rounds, Δm_i is introduced to cancel the difference in the state. The difference propagation follows that in Fig. 4.5.

Complexity of the attack. In the attack above, the differential probability of the differential characteristic is 2^{-159} before considering the leaked bytes. There are eight leaked bytes being involved in the differential characteristic, with 5 of them being introduced to the active S-boxes with probability 2^{-7} , and another 3 of them being introduced to the active S-boxes with probability 2^{-6} . According to Property 4.3.1, the differential probabilities of those eight active boxes involving the leaked bytes become 1. The overall differential probability becomes $2^{-159} \times 2^{7 \times 5} \times 2^{6 \times 3} = 2^{-106}$. The success rate of the above attack is thus 2^{-106} .

In this attack, eight leaked keystream bytes are considered, and the values of 6

4.4. OPTIMIZING THE LEAKED-STATE-FORGERY ATTACK

leaked bytes (from the first two rounds) should be one of the 128 entries in Table T (as explained above). A random keystream block satisfies the requirement with probability $128/2^{6 \times 8} = 2^{-41}$. We thus need 2^{41} known plaintext blocks in this attack.

4.4 Optimizing the Leaked-State-Forgery Attack against ALE

In this section, we optimize the LSFA against ALE. In Sect. 4.4.1, we improve the success rate of the forgery attack. The optimal success rate of a forgery attack can reach 2^{-97} , while 2^{56} known plaintext blocks are needed. In Sect. 4.4.2, the number of known plaintext blocks can be reduced to $2^{8.4}$ for achieving a success rate 2^{-102} . Note that the known plaintext blocks can be related to different keys or different nonces.

4.4.1 Improving the differential probability

From the attack presented in Sect. 4.3, we notice that the success rate of the forgery attack is determined by the probability of the differential path after taking into account of the leaked bytes. To evaluate the success rate of the forgery attack against ALE, we use the term *effective active S-boxes* to represent the active S-boxes which cannot be bypassed by exploiting the leaked bytes. In the following, we will analyze different cases to find the smallest number of effective active S-boxes.

We start with recalling some properties of the AES round function. The function *MixColumns* has a property that if it is active, the total number of active bytes in the input and output will be at least five (the property of the maximum distance separable code). By referring to Lemma 9.4.1 from [44], we have the following lemma.

Lemma 4.4.1. *The number of active S-boxes of any two-round AES differential trail is lower bounded by $5N$, where N is the number of active columns in the first round.*

In the four AES rounds in ALE, there are 16 leaked bytes. But the leaked bytes from the fourth round cannot be exploited in the attack as they do not pass through S-boxes directly. Therefore only the leaked bytes in the first three rounds can be exploited, and there are at most 12 active leaked bytes. We use $[l_1, l_2, l_3]$ to indicate the number of active leaked bytes in the first three rounds respectively. For instance, the number of active leaked bytes in the differential path in Fig. 4.4 is $[2, 4, 2]$. And we use n_i^A ($i = 1, 2, 3, 4$) to denote the number of active S-boxes at each S-box layer, which will be used in later analysis.

In the following, we will analyze five cases which give a large number of active leaked bytes. We expect that the differential path with the smallest number of effective active S-boxes can be found from these five cases. The first case is $[l_1, l_2, l_3]$ with $l_1, l_2, l_3 \geq 3$; the next three cases are $[x, 4, 4]$, $[4, 4, x]$ and $[4, x, 4]$, where the value of x is at most 2; the last case is $[l_1, l_2, l_3]$ with l_1, l_2, l_3 being different from each other, and $l_1, l_2, l_3 \in \{2, 3, 4\}$.

Case 0. $[l_1, l_2, l_3]$ with $l_1, l_2, l_3 \geq 3$. For this case, the number of effective active S-boxes is at least 18, as given in the Theorem 4.4.1 below.

Theorem 4.4.1. *If a four-round AES differential path in ALE processing message step has active leaked bytes $[l_1, l_2, l_3]$ where $l_1, l_2, l_3 \geq 3$, then the number of effective active S-boxes of this differential path is at least 18.*

Proof. We denote the column with leaked bytes as *leaking column* so the column which does not have leaked bytes is *non-leaking column*. And the column with active bytes is called *active column*.

Given the special positions of the leaked bytes, it is easy to verify that any difference in a leaking column of one round will not propagate to a leaking column in the next round. And if there are at least three out of four active leaked bytes in

4.4. OPTIMIZING THE LEAKED-STATE-FORGERY ATTACK

one round, both of the two non-leaking columns in next round will be active.

We will first analyze the number of active S-boxes in Round 1 and Round 2. Consider the number of active columns at the input of MixColumns. It must be no less than two, since we have two active columns in the output ($l_1 \geq 3$). Applying Lemma 4.4.1, we have $n_1^A + n_2^A \geq 5 \times 2 = 10$.

Then, we will analyze the number of active S-boxes in Round 3 and Round 4. Consider the number of active columns at the input of MixColumns in Round 3. We claim this number must be 4, *i.e.*, all the input columns are active. It is obvious that two leaking columns in Round 3 must be active as $l_3 \geq 3$. Now we look at the non-leaking column in Round 3. When we trace back to Round 2, there are two leaking columns that have at least 3 active leaked bytes inside as $l_2 \geq 3$. As we mentioned before, those three active leaked bytes will not propagate to the leaking columns in Round 3, after SubBytes and ShiftRows. If two of them are in the first row, they will stay at the original positions. If two of them are in the third row, they will interchange their positions. In both cases, the two non-leaking columns at the input of MixColumns in Round 3 are active. Hence, $n_3^A + n_4^A \geq 5 \times 4 = 20$ from Lemma 4.4.1.

From the above analysis, we can conclude that when $l_1, l_2, l_3 \geq 3$, the minimum number of total active S-boxes is 30. At most 12 S-boxes can be bypassed through the leaked bytes. Hence, the number of effective active S-boxes is at least $30 - 12 = 18$. $\square$

Case 1. $[x, 4, 4]$. For this case, the number of effective active S-boxes is at least 16, as illustrated below. The last two numbers of active leaked bytes are greater than three, we have $n_3^A + n_4^A \geq 20$ following the proof of Theorem 4.4.1. Then we discuss the value of x .

$x = 2$. In Round 1, the number of active S-boxes is at least 1. If the two active leaked bytes are in the same column, they will propagate to two non-leaking

Figure 4.6: An example of differential path of type “4–6–9–6”.

columns after ShiftRows in Round 2. Then, there are four active columns at the input of MixColumns in Round 3. Thus, $n_2^A + n_3^A \geq 20$. At the input of Round 4, there are at least 6 active S-boxes (4 leaked bytes plus the minimum 2 active bytes in non-leaking columns). The minimum number of effective active S-boxes is $1 + 20 + 6 - (2 + 4 + 4) = 17$. If the two active leaked bytes in Round 1 are not in the same column, the number of active columns at the input of MixColumns in Round 1 would be at least 2, which implies $n_1^A + n_2^A \geq 10$. Then the total number of active S-boxes would be at least 30 and the minimum number of effective active S-boxes is 20.

$x = 1$. With similar reasoning as the case $x = 2$, we have $n_1^A + n_2^A \geq 5$ and $n_2^A + n_3^A \geq 15$. Only one type of differential path, “2–3–12–8”, reaches both lower bounds. It has 16 effective active S-boxes and Fig. 4.6 describes an example of this differential path.

$x = 0$. The maximum possible number of effect active S-boxes is $25 - 8 = 17$, which is larger than the case when $x = 1$, so we may ignore this case.

Case 2. $[4, 4, x]$. For this case, the number of effective active S-boxes is at least 16, as illustrated below. For the first two numbers of active leaked bytes, the situation is similar to the previous discussion. Hence, we get $n_2^A + n_3^A \geq 20$. Then we discuss the value of x .

$x = 2$. In Round 1, it is obvious that the number of active columns at the input of MixColumns is at least 2. Thus, $n_1^A \geq 2$. And in Round 4, we have $n_4^A \geq 4$ since there are at least two actives bytes will appear in two non-leaking columns

4.4. OPTIMIZING THE LEAKED-STATE-FORGERY ATTACK

Figure 4.7: An example of differential path of type “2–8–12–4(3)”. The byte with triangle inside can be either active or non-active.

and at least two active leaked bytes by the assumption. There is only one possible type of differential path, “2–8–12–4”, which attains both of the lower bounds in the inequality (Fig. 4.7). The number of effective active S-boxes is $26 - 10 = 16$.

$x = 1$. This case is quite similar to $x = 2$, except for the number of active S-boxes in Round 4 is 3 instead of 4 (Fig. 4.7). The number of effective active S-boxes is $25 - 9 = 16$.

$x = 0$. We ignore this case as it is obviously worse than the cases $x = 2$ and $x = 1$.

Case 3. $[4, x, 4]$. For this case, the number of effective active S-boxes is at least 16, as illustrated below. Like previous cases, we discuss the value of x .

$x = 2$. If the two active leaked bytes in Round 2 are in the same column, they will propagate to two non-leaking columns in Round 3. Then, the number of active columns at the input of MixColumns in Round 3 would be 4, which gives $n_3^A + n_4^A \geq 20$. In Round 1, the number of active columns at the input of the MixColumns is at least 2. Therefore, the total number of active S-boxes is at least $20 + 10 = 30$, which is too large to have a good differential probability. If the two active leaked bytes in Round 2 are in different columns, the number of active columns at the input of MixColumns in Round 2 is 4. Then, $n_2^A + n_3^A \geq 20$. It is not difficult to see $n_1^A \geq 4$ and $n_4^A \geq 6$. Again, the total number of

Figure 4.8: An example of differential path of type “2–3–12–8”.

active S-boxes is at least $4 + 20 + 6 = 30$. So we conclude that no good differential path exists for this case.

$x = 1$. In this case, we have $n_1^A + n_2^A \geq 10$ and $n_3^A + n_4^A \geq 15$ using similar analysis as the previous discussions. By checking the possible differential paths, the only type of differential path attaining both of the lower bounds is “4–6–9–6” (Fig. 4.8). The number of effective active S-boxes is 16 in this case.

$x = 0$. We ignore the discussion as it is always worse than the case when $x = 1$.

Case 4. For the remaining undiscussed differential paths, only the ones with number of active leaked bytes $[l_1, l_2, l_3]$ such that (l_1, l_2, l_3) is one of the permutations of $(2, 3, 4)$ can possibly have less than 17 effective active S-boxes. However, we can always increase the number “3” to “4” to make it fall in one of the discussed cases without violating any of the previous analysis. Since there is one more active leaked byte, the cases previously discussed are always better.

4.4.1.1 Summary of the analysis.

From the above discussion, we conclude that the number of effective active S-boxes is at least 16 in a differential characteristic. There are four types of differential characteristics, “2–3–12–8”, “2–8–12–4”, “2–8–12–3” and “4–6–9–6”, which can reach this lower bound.

After testing these four types of differential characteristics, we conclude that there is no differential characteristic in which each of the effective active S-box reaches the maximum differential probability 2^{-6} . The differential characteristic

4.4. OPTIMIZING THE LEAKED-STATE-FORGERY ATTACK

Figure 4.9: Differential Path of type "2-8-12-4". The hexadecimal numbers indicate the difference values. The empty squares indicate no difference. The squares of leaked bytes are marked with gray color.

with the best probability is of the type "2-8-12-4", the details are given in Fig. 4.9. In this differential characteristic, the probability of one effective active S-box is 2^{-7} . So the overall probability of the differential characteristic is $2^{-6 \times 15 + (-7)} = 2^{-97}$. This is the best success rate of the forgery attack against ALE. For this differential characteristic, the values of 8 leaked bytes (from the first two rounds) should be one of the 2^8 values given in Table D.2 in Appendix D. And the probability of random keystream block satisfying the requirement is $2^8 / 2^{8 \times 8} = 2^{-56}$. If each key is restricted to protect 2^{48} message bits (2^{41} message blocks), we need to observe 2^{15} keys to find a weak keystream block to launch the attack. The experimental results of this attack on a small version of ALE are given in Sect. 4.5.1.

4.4.2 Reducing the number of known plaintext blocks

There are two approaches to reduce the number of known plaintext blocks required in the attack. One approach is to allow differential probability of 2^{-7} for some effective active S-boxes; another approach is to reduce the number of active leaked bytes in a differential characteristic. In these two approaches, with the reduced success rate, we are able to reduce the number of known plaintext blocks drastically.

4.4.2.1 Relaxing conditions on effective active S-boxes.

When we try to find the best probability for the differential characteristics, it is important to restrict as many as possible effective active S-boxes to have probability 2^{-6} for the input and output differences. However, if we are not satisfied with the large number of plaintext blocks required to launch the attack, we can relax the condition on some active S-boxes to have probability 2^{-7} . For instance, the probability of random keystream satisfying the requirements for leaked bytes in the differential characteristic presented in Sect. 4.4.1 is 2^{-56} . However, if we relax the probabilities on two effective active S-boxes to 2^{-7} , this probability increases to at least 2^{-50} because the increased number of differential characteristics is at least 2^6 by our test. It can be increased further if more conditions on effective active S-boxes are relaxed.

4.4.2.2 Reducing the number of active leaked bytes in the first two rounds.

Another way to reduce the number of known plaintext blocks is to reduce the active leaked bytes in the first two rounds. The reason is that only the active leaked bytes in first two rounds are related to the conditions on leaked bytes. No matter what values the active leaked bytes take in Round 3, we can determine the corresponding differences after the S-box layer according to the leaked values. The

4.5. EXPERIMENTS ON A REDUCED VERSION OF ALE

only cost is an additional pre-computed look-up table. One good choice is let the number of active leaked bytes to be $[4, 0, 4]$, and the type of differential characteristic is “6-4-6-9”. In this case, we only need to check conditions on the four active leaked bytes in the first round, yet we can still have a relatively good differential probability. There are 762408 possible differential characteristics in the first two rounds when all the 17 effective active S-boxes are with probability 2^{-6} , resulting in a success rate 2^{-102} for the forgery attack. The average number of solutions for an active S-box is estimated as $2 \times 126/127 + 4 \times 1/127 = 2^{1.01}$. Therefore, the probability for a random keystream satisfying the conditions on leaked bytes is $2^{1.01 \times 4} \times 762408/2^{32} = 2^{-8.4}$. The details of one of the 762408 differential characteristics are provided in Fig. 4.10.

4.4.2.3 Summary of the analysis.

From the above discussion, the whitening key layer is important for ALE. Once it is removed, more internal information will be leaked to an attacker, resulting in forgery attacks with higher success rates and less required plaintext blocks. The success rate of a forgery attack now is about $2^{-93.1}$ to $2^{-94.1}$, and at most 2 plaintext blocks are needed.

4.5 Experiments on a Reduced Version of ALE

As a proof of concept, we would apply our attacks to ALE (with the whitening key). However, it is impossible to directly attack the original ALE as the complexity is too high. Instead, we choose to attack a reduced ALE construction based on an AES-like light-weight block cipher, LED [75].

The LED block cipher has similar round function as AES except that the operation `AddConstants` is used before the S-box layer in each round, and the round keys are added every four rounds. The S-box in LED has difference propagation

Figure 4.10: Differential Path of type "6-4-6-9". The hexadecimal numbers indicate the difference values. The empty squares indicate no difference. The squares of leaked bytes are marked with gray color.

4.5. EXPERIMENTS ON A REDUCED VERSION OF ALE

probability at most 2^{-2} . Unlike the AES S-box, the output difference may not be unique to attain the best difference propagation probability. And for input difference 14, the probability 2^{-2} can never be obtained. So we need to take care of these differences in the attack.

In our experiments, we modified the LED round function so that it has the same ordered operations: `SubCells`, `ShiftRows`, `MixColumns`, `AddRoundKeys` as AES. Since the differential characteristic is not related to the key schedule, we use random round keys rather than deriving them from the key schedule. In addition, we simplified the input message to the two-block case without considering the initialization, padding and the associated data. The initial state is randomly generated.

4.5.1 The “2–8–12–4” differential characteristic

In the optimized forgery attacks presented in Sect. 4.4.1, the differential characteristic of type “2–8–12–4” is one of those have the highest success rate. We will experimentally verify the results on this type of differential characteristics.

Estimations. Using the above reduced ALE, we searched the differential characteristics of type “2–8–12–4”. Like the case discussed in original ALE, we need to relax the difference propagation probability of one effective active S-box to find a valid differential characteristic. Fig. E.1 in Appendix E.1 illustrates one of the differential characteristics we found.

To estimate the probability that a random keystream block is vulnerable to the attack, we analyze the number of solutions for the values of active leaked bytes in first two rounds. In each of the first two rounds, there are 2^6 possible solutions for the values of the four active leaked bytes. Therefore, the probability that a random keystream block satisfies the conditions on leaked bytes is estimated as $2^6 \times 2^6 \times 2^{(-4) \times 8} = 2^{-20}$. The average number of plaintext blocks needed to get a vulnerable keystream block is thus $1 + 1/2^{-20} = 1 + 2^{20}$. Notice that we need an

extra plaintext block to introduce the initial differences.

There are 16 effective active S-boxes in the chosen differential characteristic: 15 of the active effective S-boxes with differential probability 2^{-2} , and one with probability 2^{-3} . So the estimated probability of the differential characteristic is $2^{(-2) \times 15 + (-3) \times 1} = 2^{-33}$ which is also the success rate of the forgery attack.

Experimental results. First, we check the probability of the vulnerable keystream blocks. After encrypting $2^{27.1}$ random plaintext blocks, we found 2^7 vulnerable keystream blocks. Hence, the average plaintext blocks needed to find a vulnerable keystream block is $2^{27.1-7} = 2^{20.1}$ which matches the estimated value.

Then, we verify the success rate of the forgery attack. For a vulnerable keystream block, the value of final state is xored with the second message block and stored as t_1 . The differences in the final state (thus the leaked bytes) in Round 4 are determined by the values of leaked bytes in Round 3. Then we compute two forged ciphertext blocks similar to the attack procedure in Sect. 4.3 (but using the difference shown in Fig. E.1 in Appendix E.1). We decrypt the forged ciphertext blocks and xor the second plaintext block from decryption with the final state to get t_2 . If the two internal states t_1 and t_2 collide, we get a successful forgery. After examining $2^{36.36}$ vulnerable keystream blocks, we managed to get 10 collisions at the internal states after two blocks. So the average probability for one successful forgery is $2^{-33.04}$. One of the successful forgeries is given in Appendix E.1.

4.5.2 The “6–4–6–9” differential characteristic

In Sect. 4.4.2, the differential characteristics of type “6–4–6–9” (Fig. 4.10) are observed to require a small number of known plaintext blocks yet have good success rate. We experimentally tested this case on the reduced version of ALE.

Estimations. For this type, we found 1400 differential characteristics for the first two rounds, resulting in 21311 different values for the leaked bytes in the first

4.6. CONCLUSION

round. Details of one of the differential characteristics are given in Fig. E.2 in Appendix E.2. It is interesting to notice that certain leaked values may be used in more than one differential characteristic. If we take this into consideration, there are 28657 different leaked values related to the 1400 differential characteristics. Since there are only four active leaked bytes in the first two rounds, the probability that a random keystream is vulnerable is $28657/2^{4 \times 4} = 2^{-1.12}$. Thus, the estimated number of plaintext blocks needed to find a vulnerable keystream block is $1 + 1/2^{-1.12} = 2^{1.7}$.

There are 17 effective active S-boxes in the differential characteristic. All of them attain the maximum differential probability 2^{-2} . So the estimated probability of the differential characteristic is $2^{(-2) \times 17} = 2^{-34}$, which is also the success rate of the forgery attack.

Experimental results. In our experiments, $2^{20.7}$ vulnerable keystream blocks are generated from the encryption of $2^{21.6}$ random 2-block plaintexts. So the average number of blocks needed to find one vulnerable keystream block is $2 \times 2^{21.6}/2^{20.7} = 2^{1.9}$, which is close to the estimated value.

After querying $2^{37.7}$ forged ciphertexts, we found 10 collisions in the internal states. So the average probability of successful forgery is around $2^{-34.4}$ which is close to the estimated 2^{-34} . One of the successful forgeries is given in Appendix E.2.

4.6 Conclusion

The ALE authenticated encryption algorithm is claimed with a forgery success rate of 2^{-128} . In this chapter, we show that the success rate is significantly higher than the claimed rate. By applying the proposed leaked-state-forgery attack, the success rate can reach 2^{-97} . For a success rate 2^{-102} , every one out of $2^{8.4}$ plaintext blocks is vulnerable to the forgery attack. Our attacks are well-supported by the experi-

mental results on a reduced version of ALE. Our attack confirms again that “it is very easy to accidentally combine secure encryption schemes with secure MACs and still get insecure authenticated encryption schemes” [95]. Hence, in the design of authenticated encryption algorithms, we should be very cautious in analyzing the interaction between encryption and authentication.

Chapter 5

Cryptanalysis of the Authenticated Encryption Algorithm COFFE

5.1 Introduction

Authenticated encryption is a symmetric encryption scheme aiming to provide authenticity at the same time as confidentiality to the message. Initially, Bellare and Namprempre proposed the authenticated encryption(AE) schemes by integrating an encryption scheme with an authentication scheme in 2000, [13]. In 2001, Krawczyk published a paper [96] that studies the possibility to solve this problem by applying the existing symmetric key cryptosystem and hash function one after another.

The difficulty of the general composition approach is that although the security of the parts individually is well-studied, the application of one function may affect the security of the other. Furthermore, from implementation point of view, it is not very efficient and error-prone considering it is required to have two different primitives, one for encryption, one for plaintext integrity.

To tackle the first difficulty, a lot of dedicated designs to simultaneously encrypt and authenticate the message have been proposed, among which the authenticated

encryption mode is a commonly used design approach. Some examples of these mode of operations are IAPM [88], OCB [134], Jambu [3], GCM [57], CCM [56] and ELmD [46].

The consideration for the efficiency comes from the fact that encryption and authentication is done independently with each of their own primitive. So one way to solve this is to consider using the same primitive for both purposes. The initial direction that research goes was to construct a block-cipher based hash function for the authentication purpose such as the ones found in [109] and [124].

Another way to solve this problem is to purely use a hash function for both encryption and authentication purposes. Some of AE modes that are based on hash functions are OMD [41] and COFFE [66].

COFFE is a hash-function-based authenticated encryption scheme designed by Forler *et al.* It was published in ESC 2013 [66]. COFFE is designed to be secure in a computationally constrained environment. As mentioned above, COFFE utilises a hash function for both encryption and authentication without introducing any block cipher primitive. According to [66], COFFE is one of the first authenticated encryption scheme that is based purely on a hash function. This alternative direction of constructing an authenticated encryption system is interesting.

The designers claim that COFFE is secure against a chosen-plaintext attack in the nonce-respecting scenario. It is also claimed to have ciphertext-integrity even in the nonce-misuse scenario. In particular, it is claimed that the ciphertext integrity of COFFE is at least strong as the indistinguishability of the hash function used. That is, forging a ciphertext with a valid tag should be as hard as finding collision in the underlying hash function. Furthermore, it also provides additional features. Firstly, it provides failure-friendly authenticity, that is, COFFE provides reasonable authenticity in the case of weaker underlying hash function. Secondly, it also provides side channel resistance in the nonce-respecting scenario.

In this chapter, we first analyse the design of COFFE. During the analysis we

5.2. THE COFFE AUTHENTICATED CIPHER

consider the scheme firstly in the nonce-repeating scenario. Instead of using any specific hash function for the underlying primitive, we analyse it on the generic construction case with an ideal underlying hash function. We show that under these settings, some instances of COFFE with particular parameters are vulnerable to a distinguishing attack, a ciphertext forgery attack, or a related key recovery attack. Thus, the security claim of COFFE does not hold for these parameters.

The attacks come from the consideration that intermediate state values of two different COFFE instantiations can be made the same while having different inputs. The vulnerability comes from the fact that having most of the parameters to be variables, different set of parameters can be chosen and combined to create the collision. The attack starts by first trying to find a specific value for the parameters where this can happen. Having found these parameters, different approaches are made to exploit this discovery to launch either distinguishing attack, forgery attack, or key recovery attack. In this chapter, we found that for the distinguishing and forgery attack, if we use the same secret key for all the instantiations, the success probability is close to 1.

The rest of this chapter is structured as follows: The generic specification of COFFE is given in Section 5.2. Section 5.3 provides some analysis and observation of COFFE. Section 5.4 introduces the distinguishing attack. Section 5.5 provides two variants of ciphertext forgery attack. We propose a related key recovery attack in Section 5.6. Lastly, Section 5.7 concludes the chapter.

5.2 The COFFE Authenticated Cipher

The COFFE family of authenticated ciphers uses six parameters: key length, nonce length, block size, hash function input and output size, and tag length. We will briefly describe the specification of the COFFE authenticated cipher. The full specification can be found in [66]. An overview of COFFE is provided in Figure 5.1.

Figure 5.1: General scheme of COFFE encryption and authentication (Fig. 2 of [66])

5.2.1 Notations

Throughout this chapter, we will be using the following notations:

- $\mathcal{F}$: Underlying Hash function
 - γ : Input size for $\mathcal{F}$ assuming “one compression function invocation per hash function call”
 - δ : Output size for $\mathcal{F}$
- $\mathcal{L}_K$: Secret key length expressed in bits. The length of this string should be a multiple of a byte
- $\mathcal{L}_V$: Nonce length expressed in bits. The length of this string should be a multiple of a byte
- $\mathcal{L}_T$: Tag length expressed in bits. The length of this string should be a multiple of a byte, $\mathcal{L}_T \leq \delta$
- α : Message block size
- β : Last message block size, $\beta \leq \alpha \leq \delta$

5.2. THE COFFE AUTHENTICATED CIPHER

- x : Bits for domain value. The number of bytes used for x equals the number of bytes needed to express $(\beta + 5)$ in bits.
- Let v be a binary string and b be a positive integer.
 - $|v|$: The length of v in bits.
 - $|v|_b$: A b -bit binary representation of v .
 - $\lceil v \rceil$: The length of v in byte.
 - $\lceil v \rceil_b$: A b -byte binary representation of v .
 - $\mathfrak{b}$: $\lceil \beta + 5 \rceil$.
- $\mathcal{S}_1 || \mathcal{S}_2$: Concatenation of string $\mathcal{S}_1$ followed by $\mathcal{S}_2$.
- $\mathcal{S}_1 \oplus_{\ell} \mathcal{S}_2$: The ℓ -bit string obtained by XOR-ing the ℓ least significant bits of $\mathcal{S}_1$ and $\mathcal{S}_2$.
 - $\mathcal{S}_1 =_b \mathcal{S}_2$: The last b bits of both $\mathcal{S}_1$ and $\mathcal{S}_2$ are the same.
- $\mathcal{S}_1 || 0^* || \mathcal{S}_2$: When it is known that the total length should be, say a bits, concatenate $\mathcal{S}_1$ with 0-bits then with $\mathcal{S}_2$ with the number of 0-bits being the difference between a and the total length of $\mathcal{S}_1$ and $\mathcal{S}_2$.
- $\mathcal{K}$: Secret key string
- $\mathcal{V}$: Nonce
- $\mathcal{L}$: The number of message blocks for the encryption
- $\mathcal{S}$: Session Key with length δ bits
- $\mathcal{H}$: Associated Data
- $\mathcal{M}[i], 1 \leq i \leq \mathcal{L}$: The i -th message block, an α bit string except for $\mathcal{M}[\mathcal{L}]$ having length β bits.

- $\mathcal{C}[0]$: The initial vector
- $\mathcal{C}[i], 1 \leq i \leq \mathcal{L}$: The i -th ciphertext block with the same length as $\mathcal{M}[i]$
- $\mathcal{T}[i], 0 \leq i \leq L$: Chaining values for the scheme each of which having length δ bits
- $\mathcal{T}$: Message Tag.

5.2.2 Associated Data Processing

The method of processing the associated data, $\mathcal{H}$, can be divided into three cases based on the length of the associated data.

- If the length of $\mathcal{H}$ is less than δ bits, it is appended by 1 followed by appropriate number of zeros to reach δ bits. This is defined as $\mathcal{T}[0]$ and a domain value x is defined to be 1.
- If the length of $\mathcal{H}$ is exactly δ bits, this is directly defined as $\mathcal{T}[0]$ while the domain value x is set to be 2.
- If the length of $\mathcal{H}$ is more than δ bits, feed $\mathcal{H}$ to $\mathcal{F}$ and the resulting hash output is used as the value of $\mathcal{T}[0]$ and x is defined as 3.

5.2.3 Initialization

There are two values that need to be computed in the initialization phase, $\mathcal{S}$ and $\mathcal{C}[0]$. Firstly, the session key, $\mathcal{S}$ which is defined based on $\mathcal{K}, \mathcal{V}, \mathcal{L}_K, \mathcal{L}_V$, and $\mathfrak{b}$. The value of $\mathcal{S}$ is defined to be $\mathcal{F}(\mathcal{K}||\mathcal{V}||0^*||\mathcal{L}_K||\mathcal{L}_V||[0]_{\mathfrak{b}})$. Note that here 0^* is used to pad the string to make the length equal to γ .

Next, the constant $\mathcal{C}[0]$ which depends only on the message block size α . $\mathcal{C}[0]$ is defined to be the first $\frac{\alpha}{4}$ post-decimal values of π interpreted as a hexadecimal

5.2. THE COFFE AUTHENTICATED CIPHER

string. So for example, since the decimal values of π is $.14159\dots$, if $\alpha = 16$, Then $\mathcal{C}[0] = 0x1415 = 00010100000010101$.

5.2.4 Processing plaintext

Plaintext is encrypted to obtain the ciphertext after the generation of session key $\mathcal{S}$, the initialization vector $\mathcal{C}[0]$, initial chain value $\mathcal{T}[0]$ and the domain value, x . The plaintext blocks are processed as follow:

$$\mathcal{T}[1] = \mathcal{F}((\mathcal{S} \oplus \mathcal{T}[0]) \parallel \mathcal{C}[0] \parallel 0^* \parallel [x]_b)$$

$$\mathcal{C}[1] = \mathcal{M}[1] \oplus_{\alpha} \mathcal{T}[1]$$

for all blocks $\mathcal{M}[i], 2 \leq i \leq \mathcal{L} - 1\{$

$$\mathcal{T}[i] = \mathcal{F}((\mathcal{S} \oplus \mathcal{T}[i-1]) \parallel \mathcal{C}[i-1] \parallel 0^* \parallel [4]_b)$$

$$\mathcal{C}[i] = \mathcal{M}[i] \oplus_{\alpha} \mathcal{T}[i]$$

$\}$

$$\mathcal{T}[\mathcal{L}] = \mathcal{F}((\mathcal{S} \oplus \mathcal{T}[\mathcal{L}-1]) \parallel \mathcal{C}[\mathcal{L}-1] \parallel 0^* \parallel [4]_b)$$

$$\mathcal{C}[\mathcal{L}] = \mathcal{M}[\mathcal{L}] \oplus_{\beta} \mathcal{T}[\mathcal{L}].$$

5.2.5 Tag generation

After the associated data and plaintext are processed, the $\mathcal{L}_T$ -bit tag $\mathcal{T}$ is derived:

$$\mathcal{T} = \mathcal{F}((\mathcal{S} \oplus \mathcal{T}[\mathcal{L}]) \parallel \mathcal{C}[\mathcal{L}] \parallel 0^* \parallel \mathcal{L}_T \parallel \beta + 5)$$

The decryption is trivial and we omit it here. For the verification, only the $\mathcal{L}_T$ least significant bits of the tags are checked.

5.2.6 Security goals of COFFE

COFFE is claimed to have the INT-CTXT (ciphertext integrity) and IND-CPA(indistinguishabl under chosen plaintext attack) property in any nonce-respecting scenario.

In particular, in lemma 1 of [66], we have:

Lemma 5.2.1. *Let Π be a COFFE scheme as defined above with $\mathcal{F}$ as its underlying hash function. Then the advantage of adversary $\mathcal{A}$ in any nonce-respecting scenario with q queries and ℓ message blocks to the encryption oracle with time bounded by t can be upper bounded by:*

$$\mathbf{Adv}_{\Pi}^{\text{CPA}}(q, \ell, t) \leq \frac{8\ell^2 + 3q^2}{2^n} + 2 \cdot \mathbf{Adv}_{\mathcal{F}}^{\text{PRF-XRK}}(q, \ell, t).$$

In other words, distinguishing COFFE from a random function with chosen input under the bound of (q, ℓ, t) should be at least as hard as distinguishing $\mathcal{F}$ from a random function $\$: \{0, 1\}^\gamma \Rightarrow \{0, 1\}^\delta$.

Additionally, COFFE has some other security claim under different circumstances. Firstly, under the nonce-misuse scenario, it claimed that

- “..., the integrity of the ciphertext does not depend on a nonce, but only on the security of $\mathcal{F}$ ”.

In particular, in lemma 2 of [66], we have:

Lemma 5.2.2. *Let Π be a COFFE scheme as defined above with $\mathcal{F}$ as its underlying hash function. Then in the nonce-ignoring adversary scenario with q queries for ℓ message blocks and t times, we have*

$$\mathbf{Adv}_{\Pi}^{\text{INT-CTXT}}(q, \ell, t) \leq \frac{3\ell^2 + 2q^2}{2^\delta} + \frac{q}{2^{\mathcal{L}_T}} + \mathbf{Adv}_{\mathcal{F}}^{\text{PRF}}(q + \ell, O(t)).$$

This implies that the hardness of forging a ciphertext with a valid tag should be at least as hard as distinguishing $\mathcal{F}$ from a random function from $\{0, 1\}^\gamma$ to $\{0, 1\}^\delta$.

Secondly, COFFE also provides a failure-friendly authenticity. That is, under a weaker assumption on the security of the underlying hash function $\mathcal{F}$, the authenticity of the message is still kept.

Lastly, COFFE also provides a reasonable resistance against side channel attack. This is so because “for each encryption process, a new short term key is derived from a nonce and the long term key” [66].

5.3 Analysis on the COFFE scheme

In our analysis, we will assume $\mathcal{F} : \{0, 1\}^* \Rightarrow \{0, 1\}^\delta$ to be an arbitrary ideal hash function with γ being the largest possible length of the input to ensure exactly one compression function invocation per hash function call. Here we are assuming the possibility of the parameters to have length more than 255 bits. In other words, it is possible that it requires more than 1 byte to represent $\mathcal{L}_K, \mathcal{L}_V, \mathcal{L}_T$ in their binary format.

The first observation is about the input for the hash function call. Note that since we are only considering concatenation, there is not always a way, given the concatenated string, to uniquely determine the value for each string before the concatenation. For example, if $a||b = 11011$, it is possible that $a = 110, b = 11$ or $a = 1, b = 1011$. This leads to the possibility that two different sets of strings are concatenated to the same string.

Observation 5.3.1. *For the input of any hash function call, due to the absence of separator between substrings and changeable elements lengths, it is possible to have two different sets of strings that are concatenated to the same string.*

On the following subsections, we analyse this observation further to find whether it is possible to utilise this to cause a collision in the intermediate state value of the COFFE. We first consider the case when we fix the message block size while allowing two different last message block sizes, β_1 and β_2 , to be used. The analysis is focused on the case when $|\beta_1 - \beta_2|$ is a multiple of 256. The analysis on this can be found in Section 5.3.1. Next we also consider the possibility of having identical intermediate state values when we change the message block size, α , while keeping β fixed. The analysis is focused on how α should be chosen in such a way for the first message block encryption of both instantiations to have identical hash value output. This is discussed in Section 5.3.2.

5.3.1 Modification of β

We fix α and consider different values of β . In our next observation, with large enough α , it is possible to have $\beta_1 < \beta_2 \leq \alpha$ such that $\beta_2 - \beta_1$ is a multiple of 256. This implies that the last byte of the input of $\mathcal{F}$ in the tag generation for the two different plaintexts can be the same. As discussed above, however, we want the collision to happen in the whole input string for any $\mathcal{F}$ input. If both $\beta_1 + 5$ and $\beta_2 + 5$ require 2 bytes to be represented in binary format, the second to last byte will never agree. So for collision to happen, we need $\beta_1 + 5 < 256, 256 \leq \beta_2 + 5 < 65536$ and $\beta_2 = \beta_1 + 256\rho$ for some integer $1 \leq \rho \leq 255$.

To further analyse this observation, we consider the note by the designers regarding the increase in the number of bytes required for the binary representation of the domain. In [66], it is stated that if $\beta + 5$ exceeds one byte, all domain representations in the current COFFE will be encoded as two-byte values instead of one. So this is important in our analysis on the possibility of exploiting this observation to introduce a successful attack.

Note that in the message processing, assuming that γ is big enough, there are enough bits of the zero padding between $\mathcal{C}[i]$ and the domain values for the encryption to absorb the additional byte for the domain values in case $\beta + 5$ is increased from one byte to two bytes value. So the parts that need to be taken care of for this to happen are the session key generation and tag generation.

In the session key generation, we consider the last several bytes of the input of $\mathcal{F}$. Here we have the input to be $\dots \parallel a \parallel \mathcal{L}_K \parallel \mathcal{L}_V \parallel 0$. Note that when we expand the domain value from 1- to 2-byte value, the domain value should still have the same value. So the second to last byte for the input must be 0. This gives us our next observation.

Observation 5.3.2. *To ensure that a collision can occur when extending the domain from 1- to 2-byte value, the initial value of $\mathcal{L}_V$ must be a multiple of 256. This means that if the initial $\mathcal{L}_V$ is a 1-byte value, it must be 0, that is, no nonce in the first instance.*

5.3. ANALYSIS ON THE COFFE SCHEME

Our primary goal in this section is to investigate the possibilities to have two different (key, nonce) pairs, $(\mathcal{K}_1, \mathcal{V}_1)$ and $(\mathcal{K}_2, \mathcal{V}_2)$ with lengths $\mathcal{L}_{K_1}, \mathcal{L}_{V_1}, \mathcal{L}_{K_2}, \mathcal{L}_{V_2}$ respectively such that

$$(\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^* \parallel \mathcal{L}_{K_1} \parallel \mathcal{L}_{V_1} \parallel [0]_1) = (\mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^* \parallel \mathcal{L}_{K_2} \parallel \mathcal{L}_{V_2} \parallel [0]_2).$$

For simplicity, let

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^* \parallel \mathcal{L}_{K_1} \parallel \mathcal{L}_{V_1} \parallel [0]_1),$$

$$\mathcal{S}_2 = (\mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^* \parallel \mathcal{L}_{K_2} \parallel \mathcal{L}_{V_2} \parallel [0]_2).$$

Here, we note that a collision is indeed possible as illustrated by the following example: Let SHA-512 be our hash function. We can choose any 256-bit string, say $\mathcal{K}$, and set $\mathcal{K}_1 = \mathcal{K}_2 = \mathcal{K}$. Now we use any one bit value (0 or 1) as our $\mathcal{V}_2$ while $\mathcal{V}_1$ is set to be $\mathcal{V}_2$ appended by 255 zeros. Now if we use 472 zero paddings on the first string while using 727 bits for the second string we will have

$$\mathcal{S}_1 = \mathcal{S}_2 = (\mathcal{K} \parallel 1 \parallel 0^{255} \parallel 0^{472} \parallel [1]_1 \parallel [0]_1 \parallel [1]_1 \parallel [0]_1 \parallel [0]_1).$$

In the remainder of this section, we will try to analyse whether such collision is possible for other instances of COFFE. Here we analyse different cases of $\mathcal{S}_1$ on the possibility of having $\mathcal{S}_1 = \mathcal{S}_2$. The factors that we need to consider are the number of bytes required for $\mathcal{L}_{K_i}, \mathcal{L}_{V_i}$ and whether there is any zero padding required. Note that if $\mathcal{L}_{K_i}$ or $\mathcal{L}_{V_i}$ is a 3-byte value, the value will be at least 65536 which is too big. To simplify our discussion, for this chapter, we will only consider the key and nonce to have length whose binary format can be represented as at most a 2-byte value. Due to the large number of cases we need to consider and the similarity of the cases, we will just discuss one case as example and a full analysis of the other cases can be found in the Appendix A while the Table 5.1 containing the conclusion is provided for reference.

I.3.c Case I.3.c.: $\mathcal{S}_1$ has no zero paddings, $[\mathcal{L}_{K_1}] = 1, [\mathcal{L}_{V_1}] = 2, [\mathcal{L}_{K_2}] = 1, [\mathcal{L}_{V_2}] = 2$.

By observation 5.3.2, $\mathcal{L}_{V_1} = 256\mathbf{b}$ and $\mathcal{L}_{K_1} = \mathbf{a}$ where $1 \leq \mathbf{a}, \mathbf{b} \leq 255$. Both $\mathbf{a}$ and $\mathbf{b}$ are nonzero because of the following reasons. First of all, since $\mathcal{L}_{K_1} = \mathbf{a}$, if $\mathbf{a} = 0$, then there is no secret key, in which case, no confidentiality for the message. So we can disregard the case when $\mathcal{L}_K = 0$. Next, since $[\mathcal{L}_{V_1}] = 2$, this should mean that $\mathcal{L}_{V_1} \geq 256$ since otherwise, $[\mathcal{L}_{V_1}] = 1$. So if $\mathbf{b} = 0$, this implies $\mathcal{L}_{V_1} = 0$ which violates the requirement $\mathcal{L}_{V_1} \geq 256$. Hence

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{d}$ where $\mathbf{d}$ is the last byte of $\mathcal{V}_1$. So

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}'_1 \parallel \mathbf{d} \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Consider the alternative string $\mathcal{S}_2$. Recall that here we want $\mathcal{S}_1 = \mathcal{S}_2$ where $\mathcal{S}_2$ has its domain value represented as a 2-bytes value. This implies that the $[0]_2$ in the last 2 bytes of $\mathcal{S}_1$ must appear as the domain for $\mathcal{S}_2$. So this implies that $\mathcal{L}_{K_2} = \mathbf{d}$ and $\mathcal{L}_{V_2} = 256\mathbf{a} + \mathbf{b}$. Let t' be the number of zero padding in $\mathcal{S}_2$ where $t' \geq 0$. Equating $\mathcal{S}_1$ with $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. Now comparing the length of these substrings, we have $\mathbf{a} + 256\mathbf{b} - 8 = \mathbf{d} + 256\mathbf{a} + \mathbf{b} + t'$ or equivalently, $255(\mathbf{b} - \mathbf{a}) = \mathbf{d} + 8 + t'$. Consider the family:

$$\mathcal{F}_{(1,2),(1,2)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{d}, t') : 1 \leq \mathbf{a}, \mathbf{b}, \mathbf{d} \leq 255, t' \geq 0, 255(\mathbf{b} - \mathbf{a}) = \mathbf{d} + 8 + t'\}.$$

Now we consider the feasibility of each element of $\mathcal{F}_{(1,2),(1,2)}$. Feasibility here means the possibilities of using these values as the parameters to have the collision. Let $(\mathbf{a}, \mathbf{b}, \mathbf{d}, t') \in \mathcal{F}_{(1,2),(1,2)}$. Note that the collision may not happen with probability 1 due to the case when $\mathcal{K}_1 \neq \mathcal{K}_2$. Note that since the key is the first part of the collided string, this can only happen when $\mathcal{L}_{K_1} \neq \mathcal{L}_{K_2}$.

Before going on to the analysis, we have an assumption first. Suppose that $\mathcal{L}_{K_1} > \mathcal{L}_{K_2}$. Since $\mathcal{K}_1$ is the first $\mathcal{L}_{K_1}$ bits of $\mathcal{S}_1$, $\mathcal{K}_2$ is the first $\mathcal{L}_{K_2}$ bits of $\mathcal{S}_2$ and

5.3. ANALYSIS ON THE COFFE SCHEME

we need $\mathcal{S}_1 = \mathcal{S}_2$, the first $\mathcal{L}_{K_2}$ bits of $\mathcal{K}_1$ must be $\mathcal{K}_2$. Instead of assuming that this happens by chance, we will assume the following: The user has 2 different instantiations of COFFE scheme with different parameters and different key lengths. However, the keys chosen by the user are not independent. The longer key is an extension of the shorter key by a random secret string. We note that this assumption is only made for the related key setting attack and not for the general attack.

Based on this assumption, we then have the probability of $\mathcal{K}_1$ to have its first $\mathcal{L}_{K_2}$ bits to be the same as $\mathcal{K}_2$ is exactly 1.

Now back to our case, we have that $\mathcal{L}_{K_1} = a$ and $\mathcal{L}_{K_2} = d$. So the length difference of the two keys is $|a - d|$ bits. Now if $a = d$, then we have $\mathcal{K}_1 = \mathcal{K}_2$. Now the rest of the two strings are $\mathcal{V}'_1 \parallel d \parallel a \parallel b \parallel [0]_2$ and $\mathcal{V}_2 \parallel 0^{t'} \parallel d \parallel a \parallel b \parallel [0]_2$. So we have $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^{t'} \parallel d$. Now since $\mathcal{V}_1$ can be controlled by the attacker, we can easily set this to be true. So the probability of the two strings to collide is 1 if $a = d$.

Now consider when $a \neq d$, specifically, $a > d$. The other case can be analysed using exactly the same way. Now let $\mathcal{K}_1 = \mathcal{K}_2 \parallel \mathcal{K}'_1$ where $\mathcal{K}'_1$ is the last $a - d$ bits of $\mathcal{K}_1$. We have $\mathcal{K}'_1 \parallel \mathcal{V}'_1 \parallel d \parallel a \parallel b \parallel [0]_2$ and $\mathcal{V}_2 \parallel 0^{t'} \parallel d \parallel a \parallel b \parallel [0]_2$ as the remaining part of the two strings truncating the first d bits. Thus, $\mathcal{K}'_1 \parallel \mathcal{V}'_1 = \mathcal{V}_2 \parallel 0^{t'}$. Note that since $1 \leq a, d \leq 255$, $a - d \leq 255$, $\mathcal{V}_2$ has length $256a + b \geq 256$. So the entire $\mathcal{K}'_1$ is in $\mathcal{V}_2$. In other words, for the two strings to collide, the last $a - d$ bits of $\mathcal{V}_2$ must be equal to $\mathcal{K}'_1$. Since $\mathcal{K}'_1$ is supposed to be unknown, the probability of this collision is 2^{a-d} . It is easy to see that the remaining substring can be set to collide with probability 1. So the probability of collision to happen is $2^{-(a-d)}$. Using exactly the same analysis, we will see that when $d > a$, the probability of collision to happen is $2^{-(d-a)}$. Hence, for any non-negative integer k , we can define a subfamily

of $\mathcal{F}_{(1,2),(1,2)}$,

$$\mathcal{F}_{(1,2),(1,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{d}, t') \in \mathcal{F}_{(1,2),(1,2)} : |\mathbf{a} - \mathbf{d}| \leq k\}.$$

Then for any quadruplet $(\mathbf{a}, \mathbf{b}, \mathbf{d}, t') \in \mathcal{F}_{(1,2),(1,2),k}$ we take as parameter, the collision probability is at least 2^{-k} .

We remark that this probability is applicable for any choices of $\mathcal{K}_1$ and $\mathcal{K}_2$. This observation is essential in our attacks later to decide whether the attacks are only applicable to a family of key, or to any value of key with the given length.

We include in Table 5.1 the full list of conclusions of the session key generation analysis. Here we use t to represent the zero padding for $\mathcal{S}_1$ and t' for $\mathcal{S}_2$. In the Collision column, No means a collision in this case is impossible, yes means a collision will always happen for any choice of the parameter values (with appropriate choice of key and nonce). Lastly, $\mathcal{F}_{(a,b),(c,d)}$ or $\mathcal{F}_{p,(a,b),(c,d)}$ is the family of parameter values that belongs to the respected case that collision is possible. The definition of each of the family can be found in the complete analysis of each case that is either can be found above or in the Appendix A.

Recall that in the tag generation, given $\mathcal{T}[\mathcal{L}]$, $\mathcal{S}$, and $\mathcal{C}[\mathcal{L}]$, the input of $\mathcal{F}$ is

$$\left((\mathcal{S} \oplus \mathcal{T}[\mathcal{L}]) \parallel \mathcal{C}[\mathcal{L}] \parallel 0^* \parallel \mathcal{L}_T \parallel \beta + 5 \right).$$

We note here that here we do not use the byte-aligned assumption in our analysis. The analysis can be restricted to a byte-aligned one by adding a restriction on the families to have some of the values to be divisible by 8. Here the values that are related to the remainder of any value divided by 256 must be divisible by 8. So for example, in the case when $[\mathcal{L}_K] = 2$, if $\mathcal{L}_K = (\mathbf{a} \parallel \mathbf{b})$, then we do not need $\mathbf{a}$ to be divisible by 8. We just need $\mathbf{b}$ to be divisible by 8. This will not change the existence of any of the families. However, it will certainly requires a bigger parameter value. For example, for the case when $[\mathcal{L}_{K_1}] = [\mathcal{L}_{V_1}] = [\mathcal{L}_{K_2}] = 2$ and $[\mathcal{L}_{V_2}] = 1$, if

5.3. ANALYSIS ON THE COFFE SCHEME

$[\mathcal{L}_{K_1}]$	$[\mathcal{L}_{V_1}]$	t	$[\mathcal{L}_{K_2}]$	$[\mathcal{L}_{V_2}]$	t'	Collision?	Probability	Key restriction
1	1	Any	Any	Any	Any	No	0	Not Applicable
2	1	Any	1	1	Any	No	≈ 0	Not Applicable
2	1	0	2	1	Any	$\mathcal{F}_{(2,1),(2,1)}$	$2^{-(\mathcal{L}_{K_1} - \mathcal{L}_{K_2})}$	$\mathcal{K}_1 = (t'+8) \ 0^{t'} \parallel \lfloor \frac{\mathcal{L}_{K_2}}{256} \rfloor$
2	1	$0 < t < 8$	2	1	Any	$\mathcal{F}_{p,(2,1),(2,1)}$	$2^{-(\mathcal{L}_{K_1} - \mathcal{L}_{K_2})}$	$\mathcal{K}_1 = (t'+8-t) \ 0^{t'} \parallel \lfloor \frac{\mathcal{L}_{K_2}}{28+t} \rfloor$
2	1	Any	Any	2	Any	No	0	Not Applicable
1	2	Any	1	1	$255\mathcal{L}_{V_2} + t$	Yes	1	No
1	2	Any	2	1	Any	No	≈ 0	Not Applicable
1	2	0	1	2	Any	$\mathcal{F}_{(1,2),(1,2)}$	$2^{- \mathcal{L}_{K_1} - \mathcal{L}_{K_2} }$	No
1	2	$0 < t < 8$	1	2	Any	$\mathcal{F}_{p,(1,2),(1,2)}$	$2^{- \mathcal{L}_{K_1} - \mathcal{L}_{K_2} }$	No
1	2	0	2	2	Any	$\mathcal{F}_{(1,2),(2,2)}$	$2^{-(\mathcal{L}_{K_2} - \mathcal{L}_{K_1})}$	No
1	2	$0 < t < 16$	2	2	Any	$\mathcal{F}_{p,(1,2),(2,2)}$	$2^{-(\mathcal{L}_{K_2} - \mathcal{L}_{K_1})}$	No
2	2	Any	1	1	Any	No	0	Not Applicable
2	2	Any	2	1	$255\mathcal{L}_{V_2} + t$	Yes	1	No
2	2	Any	1	2	Any	No	≈ 0	Not Applicable
2	2	0	2	2	Any	$\mathcal{F}_{(2,2),(2,2)}$	$2^{- \mathcal{L}_{K_1} - \mathcal{L}_{K_2} }$	$\mathcal{K}_1 = \max(0, t' - (\mathcal{L}_{V_1} - 8)) \ 0$
2	2	$0 < t < 8$	2	2	Any	$\mathcal{F}_{p,(2,2),(2,2)}$	$2^{- \mathcal{L}_{K_1} - \mathcal{L}_{K_2} }$	$\mathcal{K}_1 = \max(0, t' - (\mathcal{L}_{V_1} - (8-t))) \ 0$

Table 5.1: Session Key Generation Input Collision

we want all the values to be byte aligned, the smallest parameters we can use is when $\mathcal{L}_{K_1} = \mathcal{L}_{K_2} = 256$, $\mathcal{L}_{V_2} = 8$, and $\mathcal{L}_{V_1} = 2048$. This leads to the input size for the hash function to be at least $2048 + 256 + 5 \times 8 = 2394$ bits. Here we set $\mathcal{V}_2 = 128$ and $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^{2040}$ and the other settings to be the same as the previous example.

We move on to the tag generation when $\beta + 5$ changes from a 1-byte value to a 2-byte value. Note that the only possible source of this 1-byte value is from $\mathcal{L}_T$. So, in the second instantiation where $\beta + 5$ is changed to a 2-byte value, the tag length will be different from the initial one. In fact, the first tag length needs to be a 2-bytes value, say $a \parallel b$ and the second tag length needs to be a while the difference between the two β s needs to be $256 \times b$.

5.3.2 Modification of α

This section discusses a special case of the analysis in which the user has at least two instantiations of COFFE where they have different message block sizes but the same (or related) key. Since we are considering changing α , the one we really need to take care of is just the generation of $\mathcal{C}[0]$. This is because for any other place

where α affects the system, it is generated by the previous chain in which we can truncate easily.

Recall that $\mathcal{C}[0]$ is the first $\frac{\alpha}{4}$ post decimal values of π interpreted as hexadecimal values. Suppose that we want the difference of the two block sizes, α_1 and α_2 , to be k with α_2 being the larger value. Since we are assuming the ideality of $\mathcal{F}$, we want the input of $\mathcal{F}$ in this point for both instantiation to coincide. So in other words, if the initial vector of the first instantiation is the α_1 bit $\mathcal{C}_1$ and the second one to be the α_2 bit $\mathcal{C}_2$, the additional k bits of α_2 should be absorbed by the next substring of the input, which is the zero padding. Hence, the last k bits of $\mathcal{C}_2$ should all be zeros. In other words, the value of α_1 so that it can coincide with the positions in the post decimal values of π to have a consecutive 0^k bits. So for example, if we want $\alpha_2 = \alpha_1 + 8$, and α_1 and α_2 to be a multiple of 8, then we will need to wait until the 306-th decimal place to get the 8 bits of consecutive zeros. In this case, $\alpha_1 = 1224$ and $\alpha_2 = 1232$. The requirement that α_1 and α_2 are divisible by 8 comes if we are assuming that the design is byte-aligned. Note that different α s can be found along the places where we can find k consecutive zeros in the binary representation of π .

5.4 Distinguishing Attack

In this section, to form a distinguishing attack, we use the session key collision discussed in the previous section and the Appendix A. Assuming that we have the same secret key, a different nonce, and a different number of bytes of domain value but the same session key, as before, we assume that now the session key for each instantiation is the same, with the proper number of bytes of domain value used.

We set the parameters $\alpha, \beta_1, \beta_2, \mathcal{L}_{T_1}$ and $\mathcal{L}_{T_2}$ as follows:

1. $\beta_1 + 5 < 256 < \beta_2 + 5 \leq \alpha + 5 \leq \delta + 5$
2. $\beta_2 - \beta_1 = 256\rho$ for some positive integer ρ

5.4. DISTINGUISHING ATTACK

3. $\mathcal{L}_{T_1} < 256 \leq \mathcal{L}_{T_2}$ where $\mathcal{L}_{T_2} = 256\mathcal{L}_{T_1} + \rho$.

Set the first plaintext to be a two-blocks message, $\mathcal{M}_1 = (\mathcal{MB}_1 \parallel \mathcal{MB}_2)$ such that $\mathcal{MB}_1$ has α bits and $\mathcal{MB}_2$ has β_1 bits with tag length set to be $\mathcal{L}_{T_2}$. Assume the ciphertext is $\mathcal{C}_1 = (\mathcal{CB}_1 \parallel \mathcal{CB}_2)$ with $\mathcal{T}_1$ as the tag.

The second message block, is then chosen to be $\mathcal{M}_2 = (\mathcal{MB}_1 \parallel \mathcal{MB}'_2)$ such that $\mathcal{MB}'_2$ has β_2 bits. Here we will use the tag length to be $\mathcal{L}_{T_1}$. We also assume the ciphertext is $\mathcal{C}_2 = (\mathcal{CB}'_1 \parallel \mathcal{CB}'_2)$ with $\mathcal{T}_2$ as the tag.

As discussed above, since the first blocks of both message are the same, $\mathcal{MB}_1$, we should have $\mathcal{CB}_1 = \mathcal{CB}'_1$ and $\mathcal{T}[1]$ and $\mathcal{T}[2]$ should also be the same. Now remember that $\mathcal{MB}_2 \oplus \mathcal{CB}_2$ and $\mathcal{MB}'_2 \oplus \mathcal{CB}'_2$ tells us the last β_1 and β_2 bits of $\mathcal{T}[2]$ respectively. So if $\mathcal{C}_1$ and $\mathcal{C}_2$ are both from COFFE instantiation, we must have $\mathcal{CB}_1 = \mathcal{CB}'_1$ and the last β_1 bits of $\mathcal{CB}_2$ and the last β_1 bits of $\mathcal{CB}'_2$ should agree. So we will guess that it is a COFFE instantiation instead of a random function if these requirements are met. Note that this can happen if it is a random function with probability $2^{-(\alpha+\beta_1)}$.

Recall that a distinguishing attack works as follows. An oracle randomly chooses whether it uses a random function or a COFFE instantiation with the given parameter. Then as an attacker, we can request for encryption for some plaintext. Then an adversary tries to decide whether the oracle uses a random function or a COFFE instantiation. The distinguishing attack described above has error probability 0 if we conclude that the oracle uses a random function. However, if we guess that the oracle uses a COFFE instantiation, there is a probability of $2^{-(\alpha+\beta_1)}$ that the function is actually a random function instead of COFFE. Note that since $\alpha \geq 256$ in our attack, the failure probability is at most 2^{-256} which is very small. Therefore, with 2 enquiries to the oracle with 4 message blocks, COFFE with ideal underlying hash function in nonce-respecting scenario can be distinguished with probability close to 1. So in these instantiations of COFFE, the security claim given in Lemma 5.2.1 is not satisfied.

5.5 Ciphertext Forgery Attack

In this section, we will propose two different ciphertext forgery attacks. The first attack is based on the observation on subsection 3.1. It exploits the possibility of having an identical intermediate state value for two different instantiations when we fix α while using different values of β . The detail of the attack can be found in section 5.1. Similarly, Section 5.2. discusses the forgery attack based on the discussion on subsection 3.2. Here we try to forge a valid ciphertext in the case when there exists two different COFFE instantiations with the same key for different message block size. Here both attacks require 2 enquiries and can forge a valid ciphertext with probability one. The success probability 1 is applicable whenever we assume for both instantiations, the secret key used is the same instead of one key being an extension of the other. Lastly, we will also discuss the possibility of combining the two forgery attacks. This can be found in subsection 5.3

5.5.1 Forgery Attack with Constant Message Block Size

Take any $(\mathcal{K}_1, \mathcal{V}_1), (\mathcal{K}_2, \mathcal{V}_2)$ (key, nonce) pairs from the discussion session such that they generate the same session key, one with a 1-byte domain value, the other with two. Let $\mathcal{S}_1$ be the input for session key generation with the 1-byte domain value and $\mathcal{S}_2$ be the input for the session key generation with the 2-bytes domain value. Here we assume that the input for session key generation is chosen accordingly based on the number of bytes of domain value. Hence, after this point, we can ignore the secret key and nonce and we can just assume that for each instantiation, we are using the same session key and associated data.

Note that any full block plaintext-ciphertext pair leaks the least significant α bits of the output of the hash function for a fixed input, while any β -bit block plaintext-ciphertext pair leaks only the least significant β bits of it. So since $\beta \leq \alpha$, it is always more desirable to get a full-block plaintext-ciphertext pairs since they

5.5. CIPHERTEXT FORGERY ATTACK

leak more.

Here we set the parameters $\alpha, \beta_1, \beta_2, \mathcal{L}_{T_1}$ and $\mathcal{L}_{T_2}$ as described before in Section 5.4.

Next we define the first message $\mathcal{M}_1$, a 3-block message $(\mathcal{MB}_1 \parallel \mathcal{MB}_2 \parallel \mathcal{MB}_3)$ such that $|\mathcal{MB}_1| = |\mathcal{MB}_2| = \alpha, |\mathcal{MB}_3| = \beta_2$. Let the ciphertext of this message be $\mathcal{C}_1 = (\mathcal{CB}_1 \parallel \mathcal{CB}_2 \parallel \mathcal{CB}_3)$ with tag $\mathcal{T}_1$ with $\mathcal{L}_T$ set to any value. Here we can compute the values of $\mathcal{CB}_1$ and $\mathcal{CB}_2$ since $\mathcal{MB}_1 \oplus \mathcal{CB}_1$ gives us the last α bits of $\mathcal{T}[1]$ and $\mathcal{M}_2 \oplus \mathcal{C}_2$ gives us the last α bits of $\mathcal{T}[2]$ which are essential in the attack.

We define our second message $\mathcal{M}_2$, a 2-block message $(\mathcal{MB}_1 \parallel \mathcal{MB}'_2)$ with the length of $\mathcal{MB}'_2$ to be β_1 bits and tag length to be $\mathcal{L}_{T_2}$. The first block is chosen to be exactly the same as before to ensure the value of $\mathcal{T}[1]$ and $\mathcal{T}[2]$ can be kept constant. Based on the previous message, the least significant α bits of both values are known. Suppose that the ciphertext of this plaintext is $\mathcal{C}_2 = (\mathcal{CB}_1 \parallel \mathcal{CB}'_2)$ with tag $\mathcal{T}_2$.

Using the information we obtain, we generate the following valid ciphertext. Define another 2-block message $\mathcal{M}_3 = (\mathcal{MB}_1 \parallel \mathcal{MB}''_2)$. Here we set $\mathcal{MB}_1$ to be the same as the first block from the previous message blocks. This is again to ensure that the value of $\mathcal{T}[1]$ and $\mathcal{T}[2]$ can be kept constant. We let the length of $\mathcal{MB}''_2$ be β_2 and choose $\mathcal{MB}''_2$ such that $\mathcal{MB}''_2 \oplus_{\beta_2} \mathcal{T}[2] = \mathcal{C}_2 \parallel 0^{\beta_2 - \beta_1}$. Here, $\mathcal{MB}''_2$ can be calculated since we know the last α bits of $\mathcal{T}[2]$ and $\alpha > \beta_2$. Using this message, it is easy to see that the tag generation will have the same input as before, although $\mathcal{L}_T$ is now $\mathcal{L}_{T_1}$. So the tag for this ciphertext will be the last β_1 bits of $\mathcal{T}_2$.

This attack has success probability equal to the probability that the two strings used as the input session key generation are the same. As we have discussed before, for some parameters such as the ones in case I.3 and case II.3, this can even be 1. In other words, in the case when the success probability is one, the attack above proves that the ciphertext integrity of this cipher does not satisfy the bound given in lemma 5.2.2 even in an ideal hash function situation.

Note that here we use three COFFE instantiations for each attack (2 for enquiry and 1 for the guess), while in our discussion on session key generation collision, we only consider the collision for two (key, nonce) pairs. So the same attack cannot directly work for a nonce-respecting scenario unless we can find three (key, nonce) pairs that collide to the same session key.

5.5.2 Forgery Attack with Dynamic Message Block Size

In this section, we are assuming the existence of two different instantiations of COFFE with different message block size but the same secret key and constant last message block size β . Now we pick $\alpha_1 < \alpha_2$ such that $\alpha_2 - \alpha_1 = k$. Next we find the valid size of α_1 and α_2 based on our discussion in the discussion section. Here since we assume constant last message block size, β , we assume $\beta \leq \alpha_1$. Since we are using constant last message block size, to get the same session key, we can consider the nonce-misuse scenario where we use the same key and nonce for both instantiations. Note that this means the tag length should still be kept the same.

First, we generate message, $\mathcal{M}_1 = (\mathcal{MB}_1 \parallel \mathcal{MB}_2)$ with $|\mathcal{MB}_1| = \alpha_2$ and $|\mathcal{MB}_2| = \beta$. Now assume that we get the ciphertext $\mathcal{C}_1 = (\mathcal{CB}_1 \parallel \mathcal{CB}_2)$ with tag $\mathcal{T}_1$. In this pair, our objective is to find the last α_2 bits of $\mathcal{T}[1]$ which can be obtained by calculating $\mathcal{MB}_1 \oplus \mathcal{CB}_1$.

We then consider the following message: $\mathcal{M}_2 = (\mathcal{MB}'_1 \parallel \mathcal{MB}'_2)$ with $|\mathcal{MB}'_1| = \alpha_2$ and $|\mathcal{MB}'_2| = \beta$. We further require the last k bits of $\mathcal{MB}'_2 \oplus_{\alpha_1} \mathcal{T}[1]$ are all zeros. Note that $\mathcal{MB}'_2$ can be generated easily with the knowledge of the last α_1 bits of $\mathcal{T}[1]$. Assume that the ciphertext is $\mathcal{C}_2 = (\mathcal{CB}'_1 \parallel \mathcal{CB}'_2)$ with tag $\mathcal{T}_2$. Here the last k bits of $\mathcal{CB}'_1$ are all zero and $\mathcal{CB}'_2 \oplus \mathcal{MB}'_2$ tells us the last β bits of $\mathcal{T}[2]$ when the first block of the message is $\mathcal{MB}'_1$.

Now having this information, we will proceed to our forgery attack. The message block that we will use is $\mathcal{M}_3 = (\mathcal{MB}''_1 \parallel \mathcal{MB}'_2)$. Here we have $|\mathcal{MB}''_1| = \alpha_1$ and

5.6. RELATED KEY RECOVERY ATTACK

$\mathcal{MB}_1''$ is chosen such that

$$\left(\mathcal{MB}_1'' \bigoplus_{\alpha_1} \mathcal{T}[1] \right) \parallel 0^k = \mathcal{CB}_1'.$$

We also note that the second block is exactly the same used in $\mathcal{M}_2$. Then it is easy to see that the ciphertext is $\mathcal{C}_3 = (\mathcal{CB}_1'' \parallel \mathcal{CB}_2')$ where $\mathcal{CB}_1'' = \mathcal{MB}_1'' \bigoplus_{\alpha_1} \mathcal{T}[1]$. Furthermore, the tag is exactly $\mathcal{T}_2$.

This forgery attack requires 2 enquiries to the oracle with 4 message blocks. So this attack provides a family of instances of COFFE that cannot provide ciphertext integrity as claimed in Lemma 5.2.2 under nonce-misuse scenario.

5.5.3 Combination of the Existing Attacks

In our previous two subsections, we change one of the parameters (α, β) while letting the other remain constant. This is done to simplify the analysis. However, it is possible for us to combine both attacks to generate a new attack, that is, we change α and β at the same time. Notice that by combining the two attacks, the “nonce-misuse” requirement is not a must anymore. As discussed in Section 5.5.1, as long as we can find a triple of (key, nonce) pairs that generate the same session key, we can launch the attack in the nonce-respecting scenario.

5.6 Related Key Recovery Attack

Note that, in most of the attacks we have mentioned, we are assuming the same secret key. In this section, we will discuss the case with two different instances of COFFE with different key lengths. As discussed in our observations, in this case we assume that the longer key is obtained by extending the shorter key with a secret string. As we discuss in the Appendix A, there are $(\mathcal{K}_1, \mathcal{V}_1), (\mathcal{K}_2, \mathcal{V}_2)$ pairs that lead to the same session key (with one of them using a one-byte domain value while the other using a two-byte value) with different key length. Here we will use

the pairs with k -bits key length difference and the difference are also in the nonce of the corresponding shorter key. Now assume that $|\mathcal{L}_{K_1}| > |\mathcal{L}_{K_2}|$.

We again choose the parameters $\alpha, \beta_1, \beta_2, \mathcal{L}_{T_1}$, and $\mathcal{L}_{T_2}$ as in Section 5.4. The attack here is an adaptation of the distinguishing attack we proposed earlier. We use the two messages $\mathcal{M}_1$ and $\mathcal{M}_2$ as described in section 5.4. The difference here is that for $\mathcal{M}_2$ with two bytes domain value and shorter key length, we will enquire 2^k different blocks of it with different k most significant bits of $\mathcal{V}_2$. Note that if the k most significant bits of $\mathcal{V}_2$ coincide with the k -bit extension of the secret key, then $\mathcal{CB}_1 = \mathcal{CB}'_1$ and the last β_1 bits of $\mathcal{CB}_2$ and the last β_1 bits of $\mathcal{CB}'_2$ should agree. So by using this approach, we can guess the k -bits extension of the secret key with the same complexity as exhaustive search for a k -bits secret key.

As discussed in the distinguishing attack section, when we decide that the guessed k bits are wrong, the probability that the k bits are actually the correct extension key is 0. So there will not be a false negative. However, when the k bits we guess are wrong, the probability of false positive is, as discussed in the distinguishing attack, $2^{\alpha+\beta_1}$ which is at least 2^{-255} which is negligible.

So for the related key recovery attack, to recover the k -bit extension of the secret key in nonce-respecting scenario, we will need $2^k + 1$ plaintext-ciphertext pairs with success probability approximately 1. Note that the exact same attack can be adapted to the case when $|\mathcal{K}_1| < |\mathcal{K}_2|$.

5.7 Conclusion

5.7.1 Attack Summary

From the discussion above, we see that the security claim for the nonce-misusing scenario is not met for many different parameters. The same attack can be adapted to give a distinguishing attack for the nonce-respecting scenario for some subfamilies of the parameters mentioned above.

5.7. CONCLUSION

Lastly, having two different instances of COFFE with different key lengths with the longer key being the extension of the shorter key may not be a good idea. This is because if the parameter used belongs to the family we have found earlier, the extension of the key can be recovered with exhaustive search in the same way as if the secret key is just k bits.

In conclusion, COFFE does not satisfy any of the two security claims for some of the parameters that we have discussed before. The problem arises from the fact that concatenation of strings cannot be inverted uniquely and hence giving the opportunity of having two different set of strings concatenated to the same resulting strings.

5.7.2 Lesson Learned

Here we see that the forgery and distinguishing attacks are feasible due to the possibility to have different (key,nonce) pairs to generate the same session key. This can be fixed by fixing the space for every given parameters. If some parameters are variables (such as the message block size in COFFE), we should ensure that the values of the variables get authenticated so as to prevent the forgery attack.

Chapter 6

Cryptanalysis of the IOC Authenticated Encryption Mode

6.1 Introduction

The IOC (Input and Output Chaining) mode is a lightweight authenticated encryption mode proposed on the NIST web site by Recacha. The initial proposal was posted in March 2013 [129]. After 9 months of public evaluation, the designer revised it recently with modifications to the initialization and the finalization [130]. The IOC mode is a simplified and strengthened version of the IOBC (Input and Output Block Chaining) mode designed by the same author in 1996 [127]. In the IOC specification, the following features are mentioned: low implementation cost; only one key is used; low success rate of forgery attack ($2^{-(n-5/4)}$).

In 2007, the modified version of IOBC, EPBC, was broken by Mitchell [110]. In 2013, the first cryptanalysis result on the original IOBC, after it was translated to English [128], was published by Mitchell [111], which reduced the complexity of known-plaintext forgery attack to around $2^{n/3}$. The IOC mode can resist all the attacks against EPBC and IOBC.

We notice that Rogaway and Murray have developed an attack against the

initialization of the initial IOC mode, and the attack was posted on the IOBC blog [128]. This attack exploits a flaw in the recommended method in generating the initialization vectors IV_a and IV_b which may lead to a forgery attack with success probability $1/2$. This attack can be resisted without any modifications to the iteration function if these initialization vectors IV_a and IV_b are carefully generated. And the flaw is fixed in the revised version of IOC.

Our contributions. In the revised IOC mode, it mentioned that “the core integrity mechanism proposed by IOC proposal has resisted so far all the critical reviews known by the author”. In this chapter, we point out that the security analysis is flawed, and the success rate of forgery attack is much higher than the claimed $2^{-(n-4/5)}$.

As a proof of concept, we present a forgery attacks against the IOC mode. Our forgery attack requires $O(2^{n/2})$ blocks of messages, and the success rate is close to one.

The rest of this chapter is organized as follows. Section 6.2 describes the IOC authenticated encryption mode. In Section 6.3, we develop a forgery attack against IOC mode. Section 6.4 concludes this chapter.

6.2 The IOC Authenticated Encryption Mode

In this section, we briefly describe the recent revised IOC mode using the notations from [130].

6.2.1 Notations

The following notations are used in the IOC mode and later part of this chapter.

6.2. THE IOC AUTHENTICATED ENCRYPTION MODE

n	: the block size of the block cipher used in IOC mode
N	: the length of the plaintext, in n -bit blocks
S	: a unique counter assigned to each message
P	: plaintext
P_i	: an n -bit plaintext block
C	: ciphertext
C_i	: an n -bit ciphertext block
$E_K(X)$	: the result of the block encryption of X , using the key K
MDC	: Modification Detection Code
IV_x	: an n -bit initialization vector
ICV	: Integrity Check Vector, a secret random n -bit value
$+ \text{ or } \boxplus$	: addition modulo 2^n
$\oplus$	: bit-wise exclusive OR
$\parallel$	: concatenation

6.2.2 The IOC specification

The IOC mode iteratively uses an underlying block cipher to encrypt and authenticate a message. When a plaintext P is submitted for IOC authenticated encryption, a ciphertext C and an authentication tag, MDC, will be generated. In contrast to the IOBC mode, the IOC mode itself does not impose any restriction on the length of input message. Hence, the maximum number of blocks that can be processed is solely determined by the requirements of the underlying block cipher.

When the length of the message is not a multiple of n , padding is needed. The padding scheme of the IOC mode is that “a single ‘1’ bit shall be appended to the last plaintext bit and as many as required ‘0’ bits will be appended, if necessary, to complete the last block P_N ”.

In the initialization of IOC, two initialization vectors IV_a and IV_b are generated. According to [130], the following four requirements are needed for IV_a , IV_b and

ICV:

1. IV_a , IV_b , and ICV shall be shared between the sender and the receiver and shall be secret;
2. IV_a , IV_b , and ICV shall be random, or pseudo-random and probabilistically different among them;
3. IV_a , IV_b , and ICV shall be renewed for each one of the messages;
4. IV_a , IV_b , and ICV generation shall avoid any weak composition with IOC chaining logics.

The specifications for generating the initialization vectors and integrity check vector are provided by the designer (Fig. 6.1):

- a. Set $IV_a = E_{K+S}(0)$, and $IV_b = E_{K+S}(IV_a)$.
- b. When dealing with subsequent messages, IV_a can be set as internal state O_{N+1} , and IV_b can be set as internal state I_{N+1} from the previous message. Alternatively, the IV may be reset to fresh values according to part a.
- c. For each message, ICV is computed as $ICV = (S \oplus IV_a) + (N \oplus IV_b)$.

After the IV_a and IV_b are chosen, two internal state I and O are initialized as $I_0 = IV_b$ and $O_0 = IV_a$. Then, the encryption of IOC can be described as Alg. 3.

Algorithm 3 IOC Encryption

- 1: **for** $i = 1$ to N **do**
 - 2: $I_i \leftarrow P_i \oplus O_{i-1}$
 - 3: $O_i \leftarrow E_K(I_i)$
 - 4: $C_i \leftarrow O_i + I_{i-1}$
 - 5: **end for**
-

6.2. THE IOC AUTHENTICATED ENCRYPTION MODE

Figure 6.1: Generation of IOC initializing vectors and integrity check vector [130]

When all the plaintext blocks are processed, an authentication tag, or Modification Detection Code (MDC) in the specification, is generated using following operation:

$$MDC = E_K(O_N \oplus ICV) + I_N.$$

The whole encryption and authentication are shown in Fig. 6.2.

Figure 6.2: IOC encryption and authentication

Since the decryption/verification is straightforward, we omit the details here.

They can be found in the full IOC specification [130].

6.3 The Forgery Attack on IOC mode

In this section, we will describe a chosen-plaintext forgery attack on the IOC mode. This attack requires $O(2^{n/2})$ block cipher invocations and the success rate is almost 1.

6.3.1 The basic idea of the forgery attack

Internal-State Collision. Our attack is based on internal-state collisions of IOC mode. A lemma on the IOC internal-state collision is given below.

Lemma 6.3.1. *In a chosen-plaintext scenario, when $O(2^{n/2})$ message blocks are processed using the IOC mode, an internal-state collision can be detected with probability almost 1.*

Proof. First, we notice that the IOC mode has two n -bit internal-state variables I_i and O_i , and they have the relation $O_i = E_K(I_i)$. So a collision in I_i will immediately imply a collision in O_i . Then, from the birthday paradox, a collision (I_i, I_j) is expected to occur after processing $O(2^{n/2})$ blocks of message.

To detect the collisions, we choose $O(2^{n/2})$ identical plaintext blocks in one message (there may be other plaintext blocks with different values), and look at the corresponding ciphertext blocks. Suppose we have $P_{i+1} = P_{j+1}$ and $C_{i+1} = C_{j+1}$, there is a significant probability that $I_i = I_j$ and $O_i = O_j$. The other possibility is $E_K(I_i \oplus I_j) = O_i \oplus O_j$ which has a probability 2^{-n} . To increase the probability of correct detection of the internal collisions, we can use a pair of consecutive identical plaintext blocks. If the second ciphertext blocks of two pairs (P_i, P_{i+1}) and (P_j, P_{j+1}) are the same, the probability is only $1/2^n$ for a false detection. Hence, for sufficient large 2^n , the probability that an internal collision can be correctly detected is almost 1. $\square$

6.3. THE FORGERY ATTACK ON IOC MODE

Forgery Based on Internal State Collisions. We notice that the IV and message length N are used in the finalization to generate the authentication tag. Because of the use of IV in the finalization, it is very difficult to exploit the internal-state collision between two messages in the forgery attack. In our attack, we will exploit the internal-state collisions in one message.

Because of the use of the message length in the finalization, we will exploit two internal-state collisions in the forgery attack so that we can swap two segments of the message without affecting the message length. The IOC designer has considered this type of attack with two state collisions, but wrongly considered that the two distances (the distance here is the one between two identical states) should be the same. The following observation shows that the analysis of the IOC designer is wrong.

Observation 6.3.1. *(as illustrated in Fig. 6.3) Suppose that plaintext blocks, $(P_1, P_2, \dots, P_N)$, are processed using the IOC authenticated encryption mode. $(C_1, C_2, \dots, C_N)$ are the corresponding ciphertext blocks, and authentication tag is MDC. If there are two collisions in the internal states such that $I_i = I_k$, and $I_j = I_l$ for $1 \leq i < j < k < l \leq N$ (I indicates the inputs to block cipher), then the following modified ciphertext sequence gives the same MDC through the decryption and verification:*

$$(C_1, \dots, C_i, C_{k+1}, \dots, C_l, C_{j+1}, \dots, C_k, C_{i+1}, \dots, C_j, C_{l+1}, \dots, C_N)$$

In the above observation, we swap two message segments $(C_{i+1}, \dots, C_j$ and $C_{k+1}, \dots, C_l)$ without modifying the authentication tag. In the following, We explain this observations in details. First, until C_i the ciphertext blocks are the same as original, so P_1 to P_i will be decrypted and the internal states are (O_i, I_i) . According to the collision, they are the same as (O_k, I_k) . Hence, $(C_{k+1}, \dots, C_l)$ will be decrypted to $(P_{k+1}, \dots, P_l)$ with internal states updated to (O_l, I_l) . Since (O_l, I_l) can be written as (O_j, I_j) , the blocks $(C_{j+1}, \dots, C_k)$ will be decrypted to the corresponding plaintext blocks, resulting in internal states (O_k, I_k) . Similar analysis

Figure 6.3: An illustration of the forgery attack on the IOC mode. Two colliding pairs of internal states are (I_i, I_k) and (I_j, I_l) .

can be applied to later blocks $(C_{i+1}, \dots, C_j)$ so that the internal states at the input of C_{l+1} are $O_l = O_j$ and $I_l = I_j$. Hence, the blocks $(C_{l+1}, \dots, C_N)$ will be decrypted exactly the same as the original ciphertext. Then all the states O_N, I_N, S and N are the same as original ciphertext, the MDC remains unchanged.

Our forgery attack is based on the above observation. Once we can find two such pairs of collisions, we can develop a successful forgery attack.

6.3.2 Constructing suitable pairs of collisions

Based on the above idea, we illustrate an example below on how to construct two collisions which are suitable for the forgery attack. (In practice, a slightly more efficient method can be used, but the following example is easy to illustrate.)

Let $\{P_i\}$ be a sequence of $4 \cdot 2^{n/2}$ random n -bit numbers. For P_i and P_k with $1 \leq i \leq 2^{n/2}$ and $2 \cdot 2^{n/2} < k \leq 3 \cdot 2^{n/2}$, there are 2^n pairs of (P_i, P_k) , each with

6.3. THE FORGERY ATTACK ON IOC MODE

collision probability 2^{-n} . Therefore, the probability to find at least one collision pair (P_{i0}, P_{k0}) equals approximately $1 - (1/e) \approx 0.63$. Similarly, for $2^{n/2} \leq j \leq 2 \cdot 2^{n/2}$ and $3 \cdot 2^{n/2} \leq l \leq 4 \cdot 2^{n/2}$, the probability to find least one colliding pair (P_{j0}, P_{l0}) is approximately 0.63. Since $1 \leq P_{i0} < P_{j0} < P_{k0} < P_{l0}$, these two pairs satisfy the condition of the Observation 6.3.1. And the probability is $0.63^2 = 0.4$.

6.3.3 The attack procedure

Now we give the details of the chosen-plaintext forgery attack on IOC mode by applying Observation 6.3.1.

1. Construct a plaintext P in the following form:

$$P = Q_1 || Q_2 || \cdots || Q_q,$$

where Q_i is the concatenation of three plaintext blocks $P_{3i-2} || P_{3i-1} || P_{3i}$, $q = 4 \cdot 2^{n/2}$, and all the P_{3i-2} are different, all the (P_{3i-1}, P_{3i}) are the same for $i = 1, \dots, q$.

2. Use the IOC mode to process the chosen plaintext P . We get $3q$ blocks of ciphertext, $(C_1, C_2, \dots, C_{3q})$, and a MDC. If there is a collision of the internal state I_{3i-2} , then the collision can be detected with probability almost 1 by comparing (C_{3i-1}, C_{3i}) for $i = 1, 2, \dots, q$. If there are two pairs of collisions which satisfy the condition in the Observation 6.3.1, we proceed to next step with the colliding pairs (I_{3a-2}, I_{3c-2}) and (I_{3b-2}, I_{3d-2}) . Otherwise, we go back to step 1 to choose another plaintext.

3. Ask for the decryption/verification of following forged ciphertext:

$$(C_1, \dots, C_{3a-2}, C_{3c-1}, \dots, C_{3d-2}, C_{3b-1}, \dots, C_{3c-2}, C_{3a-1}, \dots, C_{3b-2}, C_{3d-1}, \dots, C_{3q}).$$

The authentication tag remains unchanged.

The probability that two desirable pairs of collisions can be found in the plaintext generated in step 1 is 0.4 according to Sect. 6.3.2. When there are more messages available, the success rate of the attack would increase.

6.4 Conclusion

In this chapter, we developed a forgery attack against the IOC mode. The forgery attack requires about $O(2^{n/2})$ message blocks, and the success rate is almost 1. The security of IOC authentication is much weaker than claimed.

Our attack demonstrates that in the design of block cipher authenticated encryption modes, using IV (message counter) and message length in the finalization stage to generate the authentication tag is useless for achieving authentication security beyond the birthday bound. Instead, we should increase the effective state size of the authenticated encryption algorithm. We expect that this conclusion is useful for the design of secure authenticated encryption algorithms.

Chapter 7

The JAMBU Lightweight Authentication Encryption Mode

7.1 Introduction

In this chapter, we propose a lightweight authenticated encryption mode JAMBU and use AES-128 as the underlying block cipher to construct an authenticated cipher – AES-JAMBU.

7.2 The JAMBU Mode of Operation

7.2.1 Preliminary

7.2.1.1 Operations

The following operations are used in JAMBU:

- $\oplus$: bit-wise exclusive OR.
- $\parallel$: concatenation.

7.2.1.2 Notations and Constants

The following notations are used in JAMBU specifications.

- 0^a : a bit of '0's.
- AD : associated data (this data will not be encrypted or decrypted).
- $adlen$: bit length of the associated data with $0 \leq adlen < 2^{64}$.
- C : ciphertext.
- C_i : a ciphertext block (the last block may be a partial block).
- E_K : encryption of one block using the secret key K .
- IV : initialization vector used in JAMBU.
- K : secret key used in JAMBU.
- $msglen$: bit length of the plaintext/ciphertext with $0 \leq msglen < 2^{64}$.
- m_i : a data block.
- n : half of the block size used in JAMBU.
- N : number of the associated data blocks and plaintext blocks after padding. $N = N_A + N_P$
- N_A : number of the associated data blocks after padding.
- N_P : number of the plaintext blocks after padding.
- P : plaintext.
- P_i : a plaintext block (the last block may be a partial block).
- R : an additional state used for encryption. The size is half of the block size.
- S : an internal state which will be used for encryption.
- T : authentication tag.
- t : bit length of the authentication tag

7.2. THE JAMBU MODE OF OPERATION

7.2.2 Parameters

As an authenticated encryption mode, JAMBU accepts the underlying block ciphers with even bits block size which is denoted by $2n$. The key size is the same as the one used in the block cipher. The tag length is n bits. We limit the maximum length of messages to be 2^n bits under a single key.

7.2.3 Padding

The following padding scheme is used in JAMBU. For associated data, a '1' bit is padded followed by the least number of '0' bits to make the length of padded associated data a multiple of n -bit. Then the same padding method is applied to the plaintext.

7.2.4 Initialization

JAMBU uses an n -bit initialization vector(IV). The initialization vector (public message number) is public. Each key/IV pair should be used only once to achieve the maximum security of the scheme.

Let (X, Y) represent the composition of n -bit states X and Y which results in a state of $2n$ -bit. The initial state is set as $S_{-1} = (0^n, IV)$. The following operations are used for initialization.

1. $(X_{-1}, Y_{-1}) = E_K(S_{-1});$
2. $R_0 = X_{-1};$
3. $S_0 = (X_{-1}, Y_{-1} \oplus 5).$

The initialization of JAMBU is shown in Fig. 7.1.

Figure 7.1: Initialization of JAMBU .

7.2.5 Processing the associated data

The associated data is divided into n -bit blocks and processed sequentially. For the last block, the padding scheme is applied to make it a full block. Note that at least one block is processed in the processing of AD. Namely, if the length of AD, $adlen$, is 0, a padded block $1 \parallel 0^{n-1}$ will be processed. Let N_A be the number of AD blocks after padding, the AD is processed as follows.

- For $i = 0$ to $N_A - 1$, we update the states:

$$\begin{aligned} (X_i, Y_i) &= E_K(S_i); \\ U_{i+1} &= X_i \oplus A_i; \\ V_{i+1} &= Y_i \oplus R_i \oplus 1; \\ S_{i+1} &= (U_{i+1}, V_{i+1}); \\ R_{i+1} &= R_i \oplus U_{i+1}. \end{aligned}$$

Fig. 7.2 shows the processing of two blocks of associated data.

7.2.6 Encryption of JAMBU

In the encryption of JAMBU, the plaintext is divided into blocks of n -bit. The last block is padded using the padding scheme specified previously. In each step of the encryption, a plaintext block P_i is encrypted to a ciphertext block C_i .

If the last plaintext block is a full block, a block of $"1 \parallel 0^{n-1}"$ is processed without any output. Fig. 7.3 shows the encryption of two plaintext blocks.

7.2. THE JAMBU MODE OF OPERATION

Figure 7.2: Processing associated data.

Let N_P be the number of plaint blocks after padding, the encryption is described as follows:

- For $i = N_A$ to $N_A + N_P - 1$, we perform encryption and update the state:

$$(X_i, Y_i) = E_K(S_i);$$

$$U_{i+1} = X_i \oplus P_{i-N_A};$$

$$V_{i+1} = Y_i \oplus R_i;$$

$$S_{i+1} = (U_{i+1}, V_{i+1});$$

$$R_{i+1} = R_i \oplus U_{i+1}.$$

$$C_{i-N_A} = P_{i-N_A} \oplus V_{i+1} \text{ if } i < N_A + N_P - 1 \text{ or the last plaintext block is a partial block; otherwise, } C_{N_P-1} \text{ will not be computed.}$$

- The final ciphertext block is truncated to the actual length of last plaintext block from the most significant bit side.

7.2.7 Finalization and tag generation

After all the padded plaintext blocks are processed, the state is S_{N+1} and R_{N+1} ($N = N_A + N_P - 1$), we use following steps to generate the authentication tag, see Fig. 7.4.

Figure 7.3: Processing the plaintext.

Figure 7.4: Finalization and tag generation.

1. $(X_{N+1}, Y_{N+1}) = E_K(S_{N+1});$
2. $U_{N+2} = X_{N+1};$
3. $V_{N+2} = Y_{N+1} \oplus R_{N+1} \oplus 3;$
4. $R_{N+2} = R_{N+1} \oplus X_{N+1};$
5. $S_{N+2} = (X_{N+2}, Y_{N+2});$
6. The authentication tag is generated as $T = R_{N+2} \oplus X_{N+2} \oplus Y_{N+2}.$

7.2.8 The decryption and verification

The decryption and verification are similar to the encryption and authentication, except that the ciphertext block is XORed with the sub-state V to compute the plaintext block. For the final block, the ciphertext is padded using the same scheme as the plaintext before the XOR operation. If the length of ciphertext block is a multiple of n , another block of “1||0 ^{$n-1$} ” is processed similar as in the encryption.

A tag T' is generated after the decryption and is compared to the tag T . If the two tags match, the plaintext is outputted.

7.3 The SIMON-JAMBU and AES-JAMBU authenticated ciphers

To construct a lightweight authenticated cipher using JAMBU, there are many choices of underlying block ciphers. In this specification, we use SIMON [10] as our primary choice of underlying block ciphers. We remark that JAMBU is capable to be used with any other block cipher, especially those which are designed for lightweight applications.

The SIMON-JAMBU authenticated ciphers

In this specification, we take SIMON as our primary choice of the application of JAMBU. SIMON is a family of lightweight blocks cipher published by NSA in 2013. It specifies 10 block ciphers with block sizes 32, 48, 64, 96, 128 bits and key sizes 64, 72, 96, 128, 144, 192, 256 bits in the SIMON family. SIMON uses the Feistel network with simple round operations which are efficient in both hardware and software. The Feistel network can be easily adopted in JAMBU as the internal state of the block cipher is naturally divided into two sub-states. In this document, we will use SIMON64/96, SIMON96/96, SIMON128/128 in our designs of SIMON-JAMBU authenticated ciphers.

We include a description of SIMON block cipher in the Appendix F from the original paper [10] so as to make the SIMON-JAMBU self-contained.

The AES-JAMBU authenticated cipher

We also apply the JAMBU mode to the most widely used block cipher AES-128 and construct an authenticated cipher AES-JAMBU. Although AES itself is not designed for lightweight usage, it is often employed in constrained devices. AES-JAMBU will be a good choice when authenticated encryption is needed based on AES in such cases for the very small additional area.

Recommended parameter sets

- **Primary recommendation:** SIMON-JAMBU_{96/96}
96-bit key, 48-bit nonce, 144-bit state, 48-bit tag
Reason: lightweight state size with reasonable security.
- **Secondary recommendation:** SIMON-JAMBU_{64/96}
96-bit key, 32-bit nonce, 96-bit state, 32-bit tag
Reason: very small state size to fit extremely constrained condition.
- **Tertiary recommendation:** SIMON-JAMBU_{128/128}
128-bit key, 64-bit nonce, 192-bit state, 64-bit tag
Reason: provide high security level with SIMON_{128/128}.
- **Quaternary recommendation:** AES-JAMBU
128-bit key, 64-bit nonce, 192-bit state, 64-bit tag
Reason: provide high security level with widely used AES-128.

7.4 Security Goals

The security goals of JAMBU are given in Table 7.1. There is no secret message number. The public message number is a nonce. To achieve the maximum security,

7.5. THE SECURITY ANALYSIS OF JAMBU: NONCE-RESPECTING

Table 7.1: Security Goals of JAMBU. (κ -bit key, $2n$ -bit block size.)

	Confidentiality (bits)	Integrity (bits)
JAMBU	κ	n

each key and IV pair should be used to protect only one message. If verification fails, the new tag and the decrypted ciphertext should not be given as output.

However, in case that nonce is reused under the same key, the confidentiality of JAMBU is partially compromised. If the first i plaintext blocks are the same, then the $(i + 1)$ -th and $(i + 2)$ -th plaintext blocks are insecure. (It is obvious that the security of the $(i + 1)$ -th plaintext block is insecure when nonce is reused. It was shown in [123] that if the first i plaintext blocks are the same, then the $(i + 2)$ -th plaintext block is also insecure if the attacker can repeat the attack using the same nonce and chosen plaintexts for $2^{n/2}$ times.) The integrity security of JAMBU remains as n -bit.

If the ciphertext is released when verification fails, the security of JAMBU is similar to that of nonce reuse.

Notice that the integrity security in Table 7.1 includes the integrity security of plaintext, associated data and nonce and under the assumption that a block cipher with $2n$ -bit block size and κ -bit encryption security is employed and n -bit tag is generated. The table also assumes that the total length of message (plaintext and associated data) protected by a single key is limited to 2^n bits.

7.5 The Security Analysis of JAMBU in a Nonce-Respecting Scenario

In this section, we analyze the security of JAMBU which includes the security of encryption and the security of authentication when the nonce is unique under the

same key.

7.5.1 The security of encryption

The encryption of JAMBU can be seen as a variant of the Cipher Feedback (CFB) mode from the NIST recommendation [118]. The main difference is that in JAMBU the message block and an additional state block R are XORed with the internal state. The encryption of JAMBU is expected to be as strong as the underlying block cipher as long as each key/IV is used to protect only one message. Thus, the plaintext block and the additional state will not affect the randomness of the output of the CFB mode, and the confidentiality of JABMU can be implied from CFB.

7.5.2 The security of message authentication

In this section, we will give our security analysis on message authentication of JAMBU from the key/state recovery attack and the probability for a successful forgery in the nonce-respecting scenario.

7.5.2.1 Key/state recovery attack

Since the secret key of is protected by the underlying block cipher which is considered ideal, JAMBU is not vulnerable to the key recovery attack. JAMBU also has strong resistance against state recovery attacks as there are $2n$ unknown state bits at each step and the secret key is used in the encryption of each step. With the leaked n bits keystream, it is very difficult to recover the whole state provided the underlying block cipher is secure.

7.5.2.2 The forgery attack

The tag of JAMBU is generated by XORing three n -bit words in the internal state: R_{N+2} , X_{N+2} and Y_{N+2} . Suppose an adversary makes q queries with at most m blocks message. In a forgery attack, the goal of the adversary is to produce a valid tag for a (AD, C) which has never been queried. We consider two cases:

Case 1: the internal state (R, U, V) does not collide with any previous queries before the finalization. Then at least one of the R_{N+1} , U_{N+1} and V_{N+1} is new. Then we discuss the input of the final encryption (U_{N+2}, V_{N+2}) :

- (U_{N+2}, V_{N+2}) is new. Since the final encryption will have a new input, the tag is valid with probability $1/2^n$.
- (U_{N+2}, V_{N+2}) has been queried before. In this case, the corresponding R_{N+1} must be different from any other queries, otherwise, the input would be a collision which violates our assumption. The difference in R_{N+1} will propagate to the input difference at (U_{N+1}, V_{N+1}) and the output difference must be a fixed value to eliminate the difference in R_{N+1} . Then the probability is bounded by $(q - 1)/2^{2n} \leq 1/2^n$ when $q \leq 2^n$.

Case 2: the internal state (R, U, V) before the finalization collides with the internal state of some previous queries. A general approach to construct an internal collision is from the birthday attack. This was discussed by Preneel and van Oorschot [125] which showed that all iterated MACs with n -bit internal state can be attacked with $O(2^{n/2})$ queries. For JAMBU, the internal state size is $3n$ bits. Thus, around $2^{3n/2}$ messages are needed for an internal collision using the birthday attack. When the number of blocks encrypted is $2^{n-\log(n)}$ (the maximum value defined in the specification), there are around $2^{2n-2\log(n)}$ pairs of internal states. So

the collision probability is around $2^{-3n+2n-2\log(n)} = 2^{-n-2\log(n)}$ which is less than $1/2^n$.

We can derive the necessary conditions for an internal collision. Suppose we have the first internal collision such that $(R_{i+1}, U_{i+1}, V_{i+1}) = (R_{j+1}, U_{j+1}, V_{j+1})$. We have $R_{i+1} = R_i \oplus U_{i+1}$, $U_{i+1} = X_i \oplus P_i$, and $V_{i+1} = Y_i \oplus R_i$, and similar expressions hold for the states at step $j + 1$. Hence, we can derive the following necessary conditions for the internal collision:

$$Y_i = Y_j \tag{7.5.1}$$

$$R_i = R_j \tag{7.5.2}$$

$$X_i \oplus X_j = P_i \oplus P_j \tag{7.5.3}$$

Note that the condition (7.5.1) and (7.5.2) imply $V_{i+1} = V_{j+1}$ which can be observed from the keystream block. By generating around $2^{n/2}$ blocks of keystream, the probability that we observe a collision is approximately 0.5. But this collision is only a necessary condition, and we still need either (7.5.1) or (7.5.2) holds.

For condition (7.5.1), Y_i and Y_j are the output of AES encryption. Without the collision on the input (U_i, V_i) and (U_j, V_j) , $Y_i = Y_j$ has probability 2^{-n} . On the other hand, if there is collision on (U_i, V_i) and (U_j, V_j) , we get an internal collision on previous state (R_i, U_i, V_i) and (R_j, U_j, V_j) which violates our assumption.

For condition (7.5.2), both R_i and R_j are unknown and during the encryption. So the condition (7.5.2) can only be satisfied probabilistically, which is $1/2^n$.

For condition (7.5.3), when there is no difference on X_i and X_j , it will lead to a collision in previous internal state, which is assumed impossible. And if there is difference, the difference is uncontrollable by an attacker. Hence, the probability for condition (7.5.3) is $1/2^n$.

Given the above analysis, the probability for internal collision given the collision on keystream is $1/2^{2n}$. Therefore, even if we can detect the maximum $2^{(n-\log(n)) \times 2} / (2^n) = 2^{n-2\log(n)}$ collisions on keystream blocks, the probability that an internal collision

occurs is less than $1/2^n$.

Hence, the probability of a successful forgery is upper bounded by the the sum of probability in the above two cases, which is $2/2^n$. Hence, JAMBU is strong against the attacks on message authentication.

7.6 The Security Analysis of JAMBU under Nonce Misuse Scenario

In this section, we analyze the security of AES-JAMBU which includes the security of encryption and the security of authentication when the nonce is unique under the same key.

7.6.1 The security of encryption

For JAMBU, the encryption has intermediate level of robustness in the nonce reuse circumstances. More specifically, after the identical blocks in the prefix, the first and second message blocks are insecure.

Regarding to the encryption security under IV misuse cases, it is not that meaningful to consider the distinguishing attack, as it can be trivially done. For a key recovery attack, it is as difficult as breaking the underlying block cipher. Here we will discuss the plaintext recovery attack without the knowledge of the key.

Suppose that a nonce-misuse chosen plaintext adversary wants to decrypt a secure ciphertext block, say C_{i+2} . If he can find the correct plaintext with probability greater than $1/2^n$, he has a better chance than the random guess. Otherwise the ciphertext is secure. In our setting, the adversary may query messages with common blocks up to P_{i-1} so that the C_{i+2} is secure. To decrypt C_{i+2} , $Y_{i+2} \oplus R_{i+2}$ must be known. Since, $X_{i+2}||Y_{i+2} = E_K(U_{i+2}||V_{i+2})$, if $U_{i+2}||V_{i+2}$ has never been queried before, Y_{i+2} will be random and the adversary can not win the game. Thus, the ad-

versary must be able to obtain a collision of $U_{i+2}||V_{i+2}$. Note that $V_{i+2} = Y_{i+1} \oplus R_{i+1}$ does not have common prefix the any other queries and the value of R_{i+1} is secret, this condition can only be satisfied with probability $1/2^n$. But since C_{i+1} is known and the plaintext can be chosen, it is possible to obtain a collision on V_{i+2} . Suppose that there is some V_j satisfies that $V_{i+2} = V_j$, the probability that $U_{i+2} = U_j$ is $1/2^n$. To see this, we write the condition as $X_{i+1} \oplus P_{i+1} = X_j \oplus P_j$. Since P_{i+1} has unique prefix by our assumption, the value is fixed, and X_{i+1}, X_j are the output of encryption which can not be controlled, we have $P_j = P_{i+1} \oplus X_{i+1} \oplus X_j$ has probability $1/2^n$. Therefore, the probability to obtain a collision of $U_{i+2}||V_{i+2}$ is at most $1/2^n$. Hence, except the first two blocks after the common prefix, the blocks are secure.

7.6.2 The security of message authentication

Like the nonce misuse scenario, we consider two cases: *Case 1:* the internal state (R, U, V) does not collide with any previous queries before the finalization. Same analysis as the Section 7.5.2.2 applies here and the probability to obtain a valid tag is $1/2^n$.

Case 2: the internal state (R, U, V) has collision with some previous queries before the finalization. The three necessary conditions for internal collision still hold, but the analysis would have some difference in the nonce misuse setting.

First, it is obvious that the internal collisions can happen with identical prefix and a same input block, but this trivial internal collision will not lead to a valid forgery. So we are only interested in the non-trivial internal collisions which do not have common prefix.

Next, if the first difference appears in the i -th block, then it is impossible to eliminate it in the $(i + 1)$ -th block. This is easy to see as either R_{i+1} or U_{i+1} will have a difference.

For condition (1), the probability for a single collision is the same as the unique nonce case. But the difference is that when a collision of Y is found, it can be fixed

7.7. FEATURES

by using the same prefix when the other conditions are considered. Thus, if an adversary queried at most m blocks of messages, the probability to find $m(m-1)/2 \times 2^{-n}$ collisions on Y is approximately 0.5.

For condition (2), since the collision blocks will not have the identical prefix, this conditions has probability 2^{-n} as we discussed in previous analysis.

For condition (3), when the nonce is reused, it is possible to choose the difference of the message block. Then if the difference on X_i and X_j is known, condition (3) can be satisfied with probability 1. According to the method used in [123], the difference in X and R can be derived by $6 \cdot 2^{n/2}$ queries under nonce reused assumption. Here we use $2^{n/2}$ as the lower bound of the number of queries need to find the difference in X and R .

Now if we find N_c collisions of Y which satisfies condition (1), the probability of an internal collision can be computed as $N_c/2^n$ with $N_c \times 2^{n/2}$ queries. On the other hand when N_c random queries are made, the probability of a successful forgery has probability $N_c/2^n$. Therefore, when $N_c \times 2^{n/2}$ queries are made, the probability of internal collision is less than the probability of the trivial attack.

Hence, the probability for a successful forgery is bounded by $2/2^n$.

7.7 Features

- Lightweight. In addition to the registers used in the underlying block cipher, the JAMBU authenticated encryption mode only requires one additional register with half of the block size. For AES-GCM, two additional registers¹ are needed and each has equal length as the block size. For fast implementation of GCM operations, a look up table is very helpful. However, when the table is used, a much larger amount of memory will be needed. It makes

¹The two registers are used to: store the length of P and AD; store the chaining value for authentication.

AES-GCM not suitable for lightweight implementations.

- Partial resistance against IV reuse. When the IV is accidentally reused under the same key, the security of encryption and authentication is not completely compromised. Notice that in AES-GCM, the nonce reuse will lead to the loss of all confidentiality and integrity.

7.8 Performance

7.8.1 Hardware performance

The design of JAMBU is hardware-oriented. In the hardware implementation of authenticated ciphers, the state size is an important factor, especially for low-cost embedded systems. To compare the hardware efficiency of the authenticated encryption modes in term of area, We look at the state size when an authenticated encryption mode is applied to a $2n$ -bit block cipher. We compare the state size in JAMBU with the existing authentication modes. The results are given in Table 7.2. As a lightweight authenticated encryption mode, JAMBU provides the minimum state size for the hardware implementation.

7.8.2 Software performance

We implemented SIMON-JAMBU AES-JAMBU in C code. The AES-NI instruction is used in AES-JAMBU. We tested the speed on the Intel Core i7-4770 3.4GHz processor (Haswell) running 64-bit Linux 14.04. The turbo boost is turned off, so the CPU runs at 3.4GHz in the experiment. The compiler being used is gcc 4.8.2, and the options “-O3 -msse2 -maes” are used. The test is performed by encrypting/decrypting a message repeatedly, and printing out the final message. To ensure that the tag generation is not removed during the compiler optimization process, we use the tag as the IV for processing the next message. To ensure that

7.8. PERFORMANCE

Table 7.2: The comparison (in state size) for authenticated encryption modes, assuming the underlying block cipher has block size $2n$ bits

Modes	State size	Increments
CCM	$4n$	$2n$
GCM	$6n$	$4n$
OCB3	$6n$	$4n$
EAX	$8n$	$6n$
COPA	$6n$	$4n$
CPFB	$6n$	$4n$
ELmD	$8n$	$6n$
SILC	$4n$	$2n$
CLOC	$4n$	$2n$
JAMBU	$3n$	n

the tag verification is not removed during the compiler optimization process, we sum up the number of failed verifications and print out the final result.

We tested the speed of CTR, OCB3, GCM, CCM (AES-128 is used in these modes) on the same machine for comparison. The testing programs of CTR, OCB3, GCM and CCM are downloaded following the description given Krovetz and Rogaway in the OCB3 paper [98] and their website [97]. The performance comparison is given in Table 7.3.

For 4096-byte messages, the speed of SIMON-JAMBU64/96 is 51.94 cpb which is about two times of the SIMON64/96 speed, 27.3 cpb, mentioned in [10]. The speed of AES-JAMBU is about 9.98 cpb. Since in AES-JAMBU, each AES encryption is used to process only 8 byte of message, we expect that the operations of AES-JAMBU will be about two times more than AES-GCM.

Table 7.3: The software speed comparison (in cycles per byte) for different message length on Intel Haswell.

	64B	128B	256B	512B	1024B	4096B
AES-128-CTR	1.71	1.52	1.13	1.09	1.00	0.97
AES-128-CCM	6.62	5.56	5.03	4.76	4.63	4.53
AES-128-GCM	5.93	3.84	2.91	2.46	2.24	2.07
AES-128-OCB3	3.46	2.15	1.43	1.09	0.93	0.78
SIMON-JAMBU64/96	83.24	62.78	57.21	54.79	53.21	51.94
SIMON-JAMBU96/96	124.72	95.67	84.93	79.67	76.93	75.08
SIMON-JAMBU128/128	76.11	58.26	49.55	45.61	43.06	41.45
AES-JAMBU	24.41	17.08	13.41	11.57	10.65	9.98

7.9 Design rationale

JAMBU is designed to be a lightweight authenticated encryption mode which can offer partial resistance against IV reuse.

To make this mode lightweight, we introduce only an n -bit extra register for a $2n$ -bit block size. We only use the bit-wise XOR operations in the JAMBU mode.

The padding scheme used in JAMBU does not require the length information to be stored in a register. This reduces the memory requirements.

To offer a certain level of security against IV reuse, we use a block cipher encryption in the state update and only half of the state is leaked after encryption. The plaintext is injected into the other half of the state which is unknown for the attacker.

Several constants are XORed with the state in JAMBU. They are used to separate the initialization, associate data processing, plaintext processing, and finalization.

SIMON-JAMBU takes advantage of the lightweight block cipher SIMON. It can achieve a very lightweight hardware implementation.

AES-JAMBU uses AES as the underlying block cipher. It can take advantages from the security analysis AES as well as the fast implementation of AES using AES-NI.

Chapter 8

The Authenticated Cipher MORUS

8.1 Introduction

In this chapter, we specify the MORUS family of authenticated ciphers with two different internal state sizes: 640 bits and 1280 bits, and two different key sizes: 128 bits and 256 bits. Three MORUS algorithms – MORUS-640-128, MORUS-1280-128, and MORUS-1280-256 are recommended in this specification.

MORUS is a dedicated authenticated cipher which achieves encryption and authentication simultaneously. The encryption/decryption is done by XORing the plaintext with the keystream. The authentication/verification is done by injecting the message into state and generating/verifying an authentication tag.

The MORUS algorithms are designed to be very fast in hardware, since only shifts, AND, and XOR operations are used in the cipher, and the critical path is very short. MORUS is very efficient in software. The speed of MORUS-1280 (on Intel i7 4770 Haswell, 64-bit Ubuntu 13.10 and GCC 4.8.1) is 0.69 cpb for 16384 blocks of message.

This chapter is organized as follows. The MORUS specification is introduced in Section 8.2. The security of MORUS is discussed in Section 8.3 and Section 8.4. The features of MORUS are discussed Section 8.5. The performance of MORUS is

given in Section 8.6. The design rationale is given in Section 8.7.

8.2 Specification of MORUS

8.2.1 Notations and Constants

The following notations and constants are used in MORUS:

0^n	: n bits of '0's.
1^n	: n bits of '1's.
AD	: associated data (this data will not be encrypted or decrypted).
AD_i^{128}	: a 16-byte associated data block (the last block may be a partial block).
AD_i^{256}	: a 32-byte associated data block (the last block may be a partial block).
$adlen$	: bit length of the associated data with $0 \leq adlen < 2^{64}$.
$Rotl_128_32(x, n)$	: Divide a 128-bit block x into 4 32-bit words, rotate each word left by n bits.
$Rotl_256_64(x, n)$	: Divide a 256-bit block x into 4 64-bit words, rotate each word left by n bits.
b_i	: rotation constants used in $Rotl_xxx_yy$, $0 \leq i \leq 4$, which are given in Table 2.
C	: ciphertext.
C_i	: a ciphertext block (the last block may be a partial block).

8.2. SPECIFICATION OF MORUS

$const$	: a 32-byte constant in the hexadecimal format; $const = 00 \parallel 01 \parallel 01 \parallel 02 \parallel 03 \parallel 05 \parallel 08 \parallel 0d \parallel 15 \parallel 22 \parallel 37 \parallel 59 \parallel 90 \parallel e9 \parallel 79 \parallel 62 \parallel db \parallel 3d \parallel 18 \parallel 55 \parallel 6d \parallel c2 \parallel 2f \parallel f1 \parallel 20 \parallel 11 \parallel 31 \parallel 42 \parallel 73 \parallel b5 \parallel 28 \parallel dd$. This is the Fibonacci sequence modulo 256.
$const_0$	: the first 16 bytes of $const$.
$const_1$	: the second 16 bytes of $const$.
IV_{128}	: 128-bit initialization vector used in MORUS-640.
K_{128}	: 128-bit key used in MORUS.
K_{256}	: 256-bit key used in MORUS.
$msglen$	: bit length of the plaintext/ciphertext with $0 \leq msglen < 2^{64}$.
P	: plaintext.
P_i	: a 16-byte plaintext block (the last block may be a partial block).
S^i	: state at the beginning of i th step.
S_j^i	: state at the beginning of j th round at i th step.
$S_{j,k}^i$	: k th element of the state S_j^i . In MORUS-640, each element is 128-bit; in MORUS-1280, each element is 256-bit.
T	: authentication tag.
t	: bit length of the authentication tag.
u	: number of AD blocks after padding, $u = \lceil \frac{adlen}{128} \rceil$ for MORUS-640; $u = \lceil \frac{adlen}{256} \rceil$ for MORUS-128.
v	: number of plaintext blocks after padding, $v = \lceil \frac{msglen}{128} \rceil$ for MORUS-640; $v = \lceil \frac{msglen}{256} \rceil$ for MORUS-1280.
w_i	: constants used in left rotations, which are given in Table 3.

Table 8.2: The MORUS parameters.

Parameters	Length in bits		
	MORUS-640-128	MORUS-1280-128	MORUS-1280-256
Plaintext (P)	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$
Associated data (AD)	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$
Key (K)	128	128	256
Tag (T)	128	128	128
Initialization vector (IV)	128	128	128
State (S)	640	1280	1280

8.2.2 Parameters

MORUS is a family of authenticated ciphers with two internal state sizes: 640 and 1280 bits. 128-bit and 256-bit key sizes are supported in MORUS. The associated data length and the plaintext length are less than 2^{64} bits. The authentication tag is less than or equal to 128 bits. We strongly recommend the use of a 128-bit tag. We do not require any secret message number in our design, and the public message number (IV) is 128-bit in MORUS. Table 8.2 summarizes the parameters.

8.2.3 Recommended parameter sets

- Primary recommendation: MORUS-1280-128
128-bit key, 128-bit nonce, 1280-bit state, 128-bit tag
Reason: high speed for both software and hardware applications.
- Secondary recommendation: MORUS-640-128
128-bit key, 128-bit nonce, 640-bit state, 128-bit tag
Reason: smaller state.
- Tertiary recommendation: MORUS-1280-256

8.2. SPECIFICATION OF MORUS

Table 8.3: Rotation constants used in Rot1_xxx_yy 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

256-bit key, 128-bit nonce, 1280-bit state, 128-bit tag

Reason: MORUS-1280-256 uses 256-bit secret key.

8.2.4 The state update function of MOURS

In each step of MORUS, there are 5 rounds with similar operations to update the state S . Notice that the message block is used in the updates of *Round 2* to *Round 5* but not in *Round 1*. The operation Rot1_xxx_yy is to divide an xxx-bit state element into 4 words of yy-bit, and perform left rotation operation for every yy-bit word. Rot1_128_32 is used in MORUS-640 and Rot1_256_64 is used in MORUS-1280. The rotation constants for each round are defined in Table 8.3.

Besides the the Rot1_xxx_yy operation, the left rotation of a whole state element is used for diffusion. The rotation constants for the left rotation are listed in Table 8.4.

Table 8.4: Rotation constants used in left rotation in MORUS

	MORUS-640	MORUS-1280
w_0	32	64
w_1	64	128
w_2	96	192
w_3	64	128
w_4	32	64

$S^{i+1} = \text{StateUpdate}(S^i, p_i)$ is given as follows:

$$\text{Round 1 : } S_{1,0}^i = \text{Rotl_xxx_yy}(S_{0,0}^i \oplus (S_{0,1}^i \& S_{0,2}^i) \oplus S_{0,3}^i, b_0);$$

$$S_{1,3}^i = S_{0,3}^i \ggg w_0;$$

$$S_{1,1}^i = S_{0,1}^i;$$

$$S_{1,2}^i = S_{0,2}^i;$$

$$S_{1,4}^i = S_{0,4}^i;$$

$$\text{Round 2 : } S_{2,1}^i = \text{Rotl_xxx_yy}(S_{1,1}^i \oplus (S_{1,2}^i \& S_{1,3}^i) \oplus S_{1,4}^i \oplus m_i, b_1);$$

$$S_{2,4}^i = S_{1,4}^i \ggg w_1;$$

$$S_{2,0}^i = S_{1,0}^i;$$

$$S_{2,2}^i = S_{1,2}^i;$$

$$S_{2,3}^i = S_{1,3}^i;$$

8.2. SPECIFICATION OF MORUS

$$\text{Round 3 : } S_{3,2}^i = \text{Rotl_xxx_yy}(S_{2,2}^i \oplus (S_{2,3}^i \& S_{2,4}^i) \oplus S_{2,0}^i \oplus m_i, b_2);$$

$$S_{3,0}^i = S_{2,0}^i \ggg w_2;$$

$$S_{3,1}^i = S_{2,1}^i;$$

$$S_{3,3}^i = S_{2,3}^i;$$

$$S_{3,4}^i = S_{2,4}^i;$$

$$\text{Round 4 : } S_{4,3}^i = \text{Rotl_xxx_yy}(S_{3,3}^i \oplus (S_{3,4}^i \& S_{3,0}^i) \oplus S_{3,1}^i \oplus m_i, b_3);$$

$$S_{4,1}^i = S_{3,1}^i \ggg w_3;$$

$$S_{4,0}^i = S_{3,0}^i;$$

$$S_{4,2}^i = S_{3,2}^i;$$

$$S_{4,4}^i = S_{3,4}^i;$$

$$\text{Round 5 : } S_{0,4}^{i+1} = \text{Rotl_xxx_yy}(S_{4,4}^i \oplus (S_{4,0}^i \& S_{4,1}^i) \oplus S_{4,2}^i \oplus m_i, b_4);$$

$$S_{0,2}^{i+1} = S_{4,2}^i \ggg w_4;$$

$$S_{0,0}^{i+1} = S_{4,0}^i;$$

$$S_{0,1}^{i+1} = S_{4,1}^i;$$

$$S_{0,3}^{i+1} = S_{4,3}^i;$$

The state update function is shown in Fig.8.1

Figure 8.1: The state update function of MORUS. In $Rotl_xxx_yy$, xxx_yy is 128_32 for MORUS-640 and 256_64 for MORUS-1280.

8.2. SPECIFICATION OF MORUS

8.2.5 MORUS-640

MORUS-640 uses 5 128-bit registers in its internal state. In each step, it processes one block of 128-bit associated data or plaintext.

8.2.5.1 The initialization of MORUS-640

The initialization of MORUS-640 consists of loading the key and IV into the state, and running the cipher for 16 steps. The key and IV are loaded into the state as follows:

$$S_{0,0}^{-16} = IV_{128};$$

$$S_{0,1}^{-16} = K_{128};$$

$$S_{0,2}^{-16} = 1^{128};$$

$$S_{0,3}^{-16} = const_0;$$

$$S_{0,4}^{-16} = const_1;$$

After loading the key and IV, the internal state is updated 16 times using the state update function:

$$\text{For } i = -16 \text{ to } -1, S^{i+1} = \text{StateUpdate}(S^i, 0);$$

Then the key is xored to the state again:

$$S_{0,1}^0 = S_{0,1}^0 \oplus K_{128}.$$

8.2.5.2 Processing the associated data

After the initialization, the associated data AD is processed using the state update function.

1. If the last associated data block is not a full block, use '0' bits to pad it to 128 bits for MORUS-640, and use the padded full block to update the state. Note that if $adlen = 0$, the state will not be updated.
2. For $i = 0$ to l , we update the state:

$$S^{i+1} = \text{StateUpdate}(S^i, AD_i^{128});$$

where $l = \lceil \frac{adlen}{128} \rceil - 1$.

8.2.5.3 The encryption of MORUS-640

After processing the associated data, at each step of the encryption, a 16-byte plaintext block P_i is used to update the state, and P_i is encrypted to C_i .

1. If the last plaintext block is not a full block, use '0' bits to pad it to 128 bits, and use the padded full block to update the state. But only the partial block is encrypted. Note that if $msglen = 0$, the state will not get updated, and there is no encryption.
2. Let $u = \lceil \frac{adlen}{128} \rceil$ and $v = \lceil \frac{msglen}{128} \rceil$. For $i = 0$ to $v - 1$, we perform encryption and update state:

$$C_i = P_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \ggg 96) \oplus (S_2^{u+i} \& S_3^{u+i});$$

$$S^{u+i+1} = \text{StateUpdate}(S^{u+i}, P_i);$$

8.2.5.4 The finalization of MORUS-640

After encrypting all the plaintext blocks, we generate the authentication tag using eight more steps. The length of the associated data and the length of the message are used to update the state.

8.2. SPECIFICATION OF MORUS

1. $tmp = S_3^{u+v} \oplus (adlen \parallel msglen)$, where $adlen$ and $msglen$ are represented as 64-bit integers.
2. $S_4^{u+v} = S_4^{u+v} \oplus S_0^{u+v}$.
3. For $i = u + v$ to $u + v + 7$, we update the state:

$$S^{i+1} = \text{StateUpdate}(S^i, tmp);$$

4. We generate the authentication tag from the state S^{u+v+8} as follows:

$$T' = \oplus_{i=1}^4 S_i^{u+v+8};$$

8.2.5.5 The decryption and verification fo MORUS-640

The exact values of key size, IV size, and tag size should be known to the decryption and verification processes. The decryption starts with the initialization and the processing of authenticated data. Then the ciphertext is decrypted as follows:

1. If the last ciphertext block is not a full block, decrypt only the partial ciphertext block. The partial plaintext block is padded with 0 bits, and the padded full plaintext block is used to update the state.
2. For $i = 0$ to $v - 1$, we perform decryption and update the state.

$$P_i = C_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \ggg 96) \oplus (S_2^{u+i} \& S_3^{u+i});$$

$$S^{u+i+1} = \text{StateUpdate}(S^{u+i}, P_i);$$

The finalization in the decryption process is the same as that in the encryption process. We emphasize that if the verification fails, the ciphertext and the newly generated authentication tag should not be given as output; otherwise, the state of MORUS-640 is vulnerable to known-plaintext or chosen-ciphertext attacks (using a fixed IV). This requirement also applies to MORUS-1280.

8.2.6 MORUS-1280

MORUS-1280 uses 5 256-bit registers in its internal state. In each step, it processes one block of 256-bit associated data or plaintext.

8.2.6.1 The initialization of MORUS-1280

The initialization of MORUS-1280 consists of loading the key and IV into the state, and running the cipher for 16 steps. Let k_0 set as follows:

- When the key size is 128-bit, $k_0 = K_{128} \parallel K_{128}$.
- When the key size is 256-bit, $k_0 = K_{256}$.

the key and IV are loaded into the state as follows:

$$S_{0,0}^{-16} = IV_{128} \parallel 0^{128};$$

$$S_{0,1}^{-16} = k_0;$$

$$S_{0,2}^{-16} = 1^{256};$$

$$S_{0,3}^{-16} = 0^{256};$$

$$S_{0,4}^{-16} = const_0 \parallel const_1.$$

After loading the key and IV, the internal state is updated 16 steps:

$$\text{For } i = -16 \text{ to } -1, S^{i+1} = \text{StateUpdate}(S^i, 0);$$

Then the key is XORed with the state again:

$$S_{0,1}^0 = S_{0,1}^0 \oplus k_0;$$

8.2.6.2 Processing the associated data

After the initialization, the associated data AD is used to update the state.

8.2. SPECIFICATION OF MORUS

1. If the last associated data block is not a full block, use '0' bits to pad it to 256 bits for MORUS-1280, and use the padded full block to update the state. Note that if $adlen = 0$, the state will not be updated.
2. For $i = 0$ to l , we update the state:

$$S^{i+1} = \text{StateUpdate}(S^i, AD_i^{256});$$

where $l = \lceil \frac{adlen}{256} \rceil - 1$.

8.2.6.3 The encryption of MORUS-1280

After processing the associated data, at each step of the encryption, a 32-byte plaintext block P_i is used to update the state, and P_i is encrypted to C_i .

1. If the last plaintext block is not a full block, use '0' bits to pad it to 256 bits, and use the padded full block to update the state. But only the partial block is encrypted. Note that if $msglen = 0$, the state will not get updated, and there is no encryption.
2. Let $u = \lceil \frac{adlen}{256} \rceil$ and $v = \lceil \frac{msglen}{256} \rceil$. For $i = 0$ to $v - 1$, we perform encryption and update state:

$$C_i = P_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \ggg 192) \oplus (S_2^{u+i} \& S_3^{u+i});$$
$$S^{u+i+1} = \text{StateUpdate}(S^{u+i}, P_i);$$

8.2.6.4 The finalization of MORUS-1280

After encrypting all the plaintext blocks, we generate the authentication tag using eight more steps. The length of the associated data and the length of the message are used to update the state.

1. $tmp = S_3^{u+v} \oplus (adlen \parallel msglen \parallel 0^{128})$, where $adlen$ and $msglen$ are represented as 64-bit integers.
2. $S_4^{u+v} = S_4^{u+v} \oplus S_0^{u+v}$.
3. For $i = u + v$ to $u + v + 7$, we update the state:

$$S^{i+1} = \text{StateUpdate}(S^i, tmp);$$

4. We generate the authentication tag from the state S^{u+v+8} as follows:

$$T' = \oplus_{i=1}^4 S_i^{u+v+8};$$

The authentication tag T consists of the first t bits of T' .

8.3 Security Goals

The security goals of MORUS are given in Table 8.5. In MORUS, each key and IV pair should be used to protect only one message. If verification fails, the new tag and the decrypted ciphertext should not be given as output.

Note that the encryption security is under the assumption that the attacker could not forge a message through repeated trials. The integrity security in Table 8.5 includes the integrity security of plaintext, associated data and nonce and under the assumption that the secret key is unknown to the attacker, and a 128-bit tag is used.

8.4 Security Analysis

In this section, we will give our initial analysis on the security of MORUS. Most of the analysis is on MORUS-640, but can be extended to MORUS-1280 trivially.

8.4. SECURITY ANALYSIS

Table 8.5: Security Goals of MORUS.

	Confidentiality (bits)	Integrity (bits)
MORUS-640-128	128	128
MORUS-1280-128	128	128
MORUS-1280-256	256	128

8.4.1 The security of the initialization

The initialization of MORUS is designed to ensure the IV and key are properly mixed so that the internal state after initialization is secret. We consider the differential attack using IV differences as the main threat to the initialization of MORUS. The reason is that the IV is the only input in the initial state controlled by an attacker under our security assumptions (differences in key are considered uncontrollable by an attacker).

In order to maximize the differential probability, we will model the AND operation in following way:

- Any difference that passes through the AND operation will be eliminated unless it is able to cancel a difference in XOR operations to reduce the weight of the updated state element.

For example, suppose that S_A, S_B, S_C , and S_D are four state elements, and $S_A = S_B \oplus (S_C \& S_D)$ is computed. If there is a difference at bit i of either S_C or S_D , the difference in the output of AND operation for bit i will be the same as the i th bit of S_B to ensure the i th bit of S_A has no difference.

This approximation does not always lead to the minimal number of active bits. But in most of the circumstances we analyzed, especially when the number of active bits is small, it does give a good approximation to the minimal number of active bits.

We discuss the differential probability of the initialization of MORUS-640 with input differences on IV through following two cases.

- Case 1. There is only 1 bit difference in IV . Notice that MORUS uses bit operations in the state update function. As a result, the position of the one bit difference will not affect the differential probability. We assume the difference is at bit 0 of the IV in our analysis. Using the above mentioned approximation, we find that after 6 steps, the differential probability is 2^{-201} when the value of initial state is assumed unknown. After 7 steps, the differential probability is 2^{-357} . Since the initial state is given except the key, the differential probability can be 1 for some bits in the first two steps. However, after two steps, the values of state elements are mixed with the key and thus can be considered unknown. We may exclude the differential probability in first three steps, which is 2^{-10} for a more accurate bound.
- Case 2. There are 2- or 3-bit difference in IV . We fix a bit difference at bit 0 of IV , and enumerated all the cases that there are 2 and 3 active bits in the IV . For 2 bits cases, there are 127 possible ways to choose the position of the other bit. And for 3-bit case, there are $\frac{127 \times 126}{2} = 8001$ possible ways to choose the positions of the other two bits. After 5 steps, the differential probability is 2^{-401} for the 2-bit differences on IV and 2^{-432} for 3-bit difference on IV .
- Case 3. When the difference on IV is more than 3 bits, we expect that the differential probability will decrease until the weight in the internal state elements is high enough so that a large number of active bits get canceled. But in that case, the weight of active bits will increase fast in the early steps and differential probability is likely to fall below 2^{-256} more quickly than the 1 bit difference case.

Hence, it is unlikely that there is any differential characteristic with probability hight than 2^{-128} after 16 steps of MORUS initialization.

8.4.2 The security of the encryption process

The MORUS encryption is a stream cipher with a large state which is updated continuously. The attacks against a block cipher cannot be applied directly to MORUS. We emphasize that the security of encryption is under the assumption that the *IV* is not reused for the same key. Once the *IV* is reused, it will lead to serious attacks on recovering the state.

Statistical Attacks. If the *IV* is used only once for each key, it is impossible to apply a differential attack to the encryption process. In general, statistical attacks are difficult to launch against MORUS. Non-linear operations are used in the state update and keystream generation, which makes it difficult to recover the state by applying algebraic attacks.

8.4.3 The security of message authentication

The security of message authentication of MORUS is related to the length of the tag generated. In our analysis, we will only consider the case that the tag length is 128-bit since it implies the security of the cases with shorter tag length. To analyze the authentication of the MORUS, we will compute the probability that a forged message would bypass the verification.

8.4.3.1 Internal state collision

Construction of internal state collision is a typical method used in attacking the message authentication. For MORUS, since the internal state size is at least 640-bit, any internal collision through birthday attack requires about 2^{-320} blocks to be encrypted, which is far too expensive for an attacker. Another method to construct an internal state collision is to inject a message difference at a certain step and cancel it at a later step. Our analysis will show that this attack will lead to an internal-state collision with probability below 2^{-128} .

First, we consider the case that the difference gets eliminated in two steps, *i.e.*, the difference is injected to the internal state at one step and gets eliminated immediately at the next step. There are 10 rounds (2 steps) in this case, which we number as Round 1,..., Round 10. Recall that two state elements get updated in each round: one is to compute a state element using 4 of the previous state elements and the other is to left rotate a state element computed two rounds ago. For simplicity, we will slightly change the order of these operations so that only one state element is updated in each round. We call this state element as CV_i when it is updated at Round i . What we do is to perform the left rotation immediate after one state element get updated and rotate back when it is used after two rounds. Hence the state update in Round i becomes:

$$CV_i = \text{Rotl_xxx_yy}(CV_{i-5} \oplus (CV_{i-4} \& CV_{i-3}) \oplus (CV_{i-2} \lll w_{(i-1) \bmod 5}) \oplus m_i)$$

Notice that m_i is the plaintext block used in each step and $m_i = 0$ if $i = 0 \bmod 5$. And the difference in plaintext will inject to Round 2 and be the same in Round 3-5.

To eliminate the difference after two steps, we need that $CV_6, \dots, CV_{10}$ have no difference. In our study, we will focus on following two conditions:

- 1: No difference at CV_6 . This is because CV_6 is completely determined by the previous state elements and has nothing to do with the plaintext block in the second step.
- 2: For each difference at bit i in CV_3 or CV_4 there must be a difference at bit i in CV_5 . Otherwise, is impossible to eliminate the difference using the difference in the second plaintext block.

Then we searched the input difference bits to find a lower bound for the number of bits with difference (active bits) in the input. We found that for the input difference with weight less than or equal to 25, there is no valid 10-round differential

8.4. SECURITY ANALYSIS

characteristics for MORUS. Now we may evaluate the bound for the differential probabilities. When input difference is n bits, there are n bits differences at CV_2 , CV_3 and CV_5 . Since each bit difference will be involved in two AND operations, and each AND operation on one bit has differential probability 2^{-1} , the differential probability is at most 2^{-5n} (5 AND operations for CV_i and CV_{i+1} , $i = 1, 2, 3, 4, 5$). The differential probability is less than $2^{-26} \times 5 = 2^{-130}$.

Next, we consider the case that the input difference get eliminated in 3 steps. If there are 3 active bits in the input, the differential probability after 3 steps is 2^{-132} by our approximation. Note that the difference is not eliminated through the approximation. Much stronger conditions are needed to eliminate the differences. Hence the probability that the input difference get eliminated after 3 steps will be much lower than 2^{-132} when the number of active bits is 3. When we increase the number of active bits in the input, the trend is to increase the weight of active bits in the states, which we can observe in the previous cases. Intuitively, this can be explained as when the weight of active bits is low, the number of new active bits exceeds the number of active bits get eliminated. And when the weight is high enough such that the number of eliminated active bits exceeds the new active bits, we can expect the overall weight will be much higher than the single difference case in the first 3 steps. Hence, although it is impossible to enumerate all the input differences, we believe that there is no differential characteristic with probability higher than 2^{-128} which can eliminate the input difference in 3 steps.

Now we deal with the cases that the number of active bits in the input is less than three.

- Only one active bit in the input. Since the position of active has no impact on the differentials, we assume the active bit is at bit 0. Then, we propagate the difference up to 3 steps (15 rounds), assuming no input difference at next two steps. Now, we enumerate the input difference at step 2 such that following two conditions are satisfied:

1. There is no difference at Round 11. Again, it is because the difference cannot be eliminated through the message in step 3.
2. The active bits at CV_{10} cover the active bits at CV_8 and CV_9 .

Our search shows that even if we increase the number of active bits to 20 in the input of the second step, it is impossible to find a differential characteristic satisfying the above conditions. With a similar method, we also evaluated differential probability introduced by the initial difference. We conclude that the probability that the internal state collision is less than 2^{-128} in this case.

- Two active bits in the input. By our approximation, the differential probability is at most 2^{-101} for any two active bits propagate to 3 steps. We think it is safe to consider the probability for internal state collision to be less than 2^{-128} if the number of active bits in the second step is larger than 20, in spite that some difference in the internal state may be canceled. In our search, we fix one bit difference at bit 0 and try to impose a difference at the other 127 possible positions. The search result confirms that no valid differential characteristic is found when the number of active bits is less than 21.

Now, consider the rest cases: the difference get eliminated after at least 4 steps. If there is one bit difference at the input, the differential probability is at least 2^{-196} using our approximation, which is much lower than 2^{-128} . If we want to eliminate the differences, more conditions are required. Hence, it is reasonable to consider the probability to eliminate the internal difference in these cases to be less than 2^{-128} . This conclude our analysis when the internal state collision is constructed through injection of plaintext differences.

8.5. FEATURES

8.4.3.2 Attacks on the finalization

In addition to the internal state collision, we analyze the security of the finalization. When there is a difference in the internal state before the finalization, we expect that the differential probability is less than 2^{-128} after 8 steps of the MORUS state update function. Hence, the difference at the tag is not predictable in this case. Another situation is that there is no difference at the internal state but there is some difference at the *adlen* or *msglen*. Then the difference will be used as message difference in the state update function for 8 steps. So the differential probability is expected to be less than 2^{-128} in this case as well.

8.5 Features

MORUS has the following advantages over previous ciphers:

1. MORUS is efficient in software. According to the previous section, the speed of MORUS-1280 is 0.69 cpb on Intel Haswell processors for long messages, which is around 30% faster than AES-GCM [107].
2. MORUS is fast in hardware performance. In MORUS, the critical path to generate a keystream block is 3 AND gates and 8 XOR gates.
3. MORUS is efficient across platforms. In constructing authenticated encryption schemes, AES is frequently used as a building block. There are authenticated encryption modes so that the AES can be used as underlying block cipher, *e.g.*, EAX [14], CCM [146], GCM [107] and OCB 2.0 [133]. A number of dedicated AE schemes use AES round function, *e.g.*, AEGIS [156] and ALE [37]. These schemes can benefit from the AES-NI which performs one round AES encryption/decryption in a single instruction. On the other hand, although the widely use of AES, there are platforms which do not support

the AES-NI instruction set. The performance of AES based authenticated encryption schemes will be notably slower on these platforms. In contrast, the MORUS family offer a more steady performance across platforms since its performance does not rely on the use of AES-NI instruction set.

4. Secure. MORUS provides 128-bit authentication security, stronger than AES-GCM.

8.6 Performance

We implemented MORUS in C code. We tested the speed on the Intel Core i7-4770 processor (Haswell) running 64-bit Ubuntu 13.01. Turbo boost is turned off in the experiment. The compiler being used is gcc 4.8.1, and the options “-O3 -mavx2” are used. The test is performed by encrypting/decrypting a message repeatedly, and printing out the final message. To ensure that the tag generation is not removed during the compiler optimization process, we use the tag as the IV for processing the next message. To ensure that the tag verification is not removed during the compiler optimization process, we sum up the number of failed verifications and print out the final result.

Table 8.6 shows the speed comparison of the MORUS. For long message, the speed of MORUS-640 and MOURS-1280 is about 1.11 cpb and 0.69 cpb, respectively. The speed of MOURS-1280 is faster than that of AES-128-GCM on the Haswell, which is 1.03 cpb [72].

8.7 Design rationale

In our design of MORUS, we are trying to design a fast authenticated cipher which is not based on AES so that this cipher can run fast in platforms with no AES-NI. Our design is aimed at achieving the following goals:

8.7. DESIGN RATIONALE

Table 8.6: The speed comparison (in cycles per byte) for different message length on Intel Haswell. EA means encryption-authentication; DV means decryption-verification.

	16B	64B	512B	1024B	4096B	16384B
MORUS-640(EA)	28	7.72	1.95	1.58	1.18	1.11
MORUS-640(DV)	28	7.99	1.97	1.56	1.23	1.16
MORUS-1280(EA)	33.9	8.28	1.59	1.12	0.78	0.69
MORUS-1280(DV)	35.8	8.46	1.63	1.13	0.80	0.69

- Simple
- Secure
- Fast in hardware
- Efficient in software
- Avoid using AES round function

8.7.1 State update function

The construction of state update function of MORUS is based on 5 small round functions with similar operations. In each round function, only XOR, AND and rotations are used. The diffusion of MORUS is from two types of rotations: the rotations on the whole registers ($\ggg$) and the rotations on four partial words inside a register (Rotl_xxx_yy). The later operation takes advantage of the SSE2 and AVX instructions in which the shifts on four word can be done in one instruction. We choose the AND non-linear function since it can be easily and efficiently implemented in both software and hardware. Two internal state elements get updated in a round function. Hence, every internal state element will get updated twice in a

step. It is remarkable that MORUS is constructed using simple bit-wise operations, which makes it fast in hardware implementations.

8.7.2 Encryption and authentication

The encryption of MORUS adopts the method used in stream ciphers. The key and nonce are mixed into the state during initialization and after that, the cipher generates keystreams and XORs the keystreams with the plaintext to produce ciphertext. In MORUS, message blocks are injected into its state update function so as to authenticate the message simultaneously with the encryption.

In the initialization of MORUS, we use 16 steps of state update function (80 rounds). This is to ensure the state cannot be recovered and the differential probability is small after the initialization.

In the finalization, we introduce an extra XOR operation to distinguish the finalization from the encryption and we use a similar method as used in AEGIS: mixing the length of associated data and plaintext is XORed to one of the internal state elements and used as a message block to update the states for 8 steps. In this way, any change in the internal state or the length of message will be involved in computing the tag.

8.7.3 Selection of rotation constants

The diffusion in MORUS relies on the 10 rotations. Therefore, the rotation constants need to be carefully chosen. We use following rules in the selection of rotations constants:

1. The rotation constants should exclude the multiples of 8.
2. No rotation constant should be a multiple of another rotation constant.

8.7. DESIGN RATIONALE

3. The sum of any two constants modular 32 (or 64 for MORUS-1280) is not equal to 0 or another constant.

We enumerate the possible choices of rotation constants satisfying the above requirements and propagate a 1-bit difference on message to count the weight after four steps for MORUS-640 and five steps for MORUS-1280. Then we select a set of the rotation constants which results in high weight.

Chapter 9

Conclusions

In this thesis, we have investigated various techniques and principles used in the cryptanalysis and design of authenticated ciphers. After the overview of the authenticated cipher concept, we discuss the typical methods in the authenticated cipher cryptanalysis. Then, we present five attacks on authenticated ciphers and proposed the authenticated encryption mode JAMBU and authenticated cipher MORUS.

Designing a secure and efficient authenticated cipher which integrates the two functionalities encryption and message authentication is a challenging work. We learn several lessons from this thesis.

- Classical differential cryptanalysis remains important in studying the security of authenticated ciphers. The differential properties on the initialization, message encryption and authentication tag generation may all be exploited to attack authenticated ciphers or the underlying primitives which were demonstrated in our analysis of ALE and ZUC.
- The interaction between encryption and authentication should not be overlooked. Especially when the internal state is used in the keystream generation, information from the ciphertext may be used to attack the authentica-

tion or to recover the state. On the other hand, when weak authentication is used, the encryption is under the risk of chosen ciphertext attacks which may be effective in certain designs.

- For the permutation based authenticated ciphers, the property of nonce reuse resistance requires strong confusion and diffusion of the underlying permutation function. Even if the input/output strings are only partially leaked, it is better to consider the best attacks against the fully known permutation to ensure the security.
- When parameters are used as input of an authenticated cipher, it is important to ensure the variant values of parameters will not affect the security.
- It is not easy to achieve beyond-birthday-bound security for the authentication in an authenticated-encryption mode. Designers must be very caution in considering all the possible attack scenarios which can be exploited to produce internal collisions.
- In the design of authenticated cipher, it is very useful to follow the advances in the processors. The latest instructions such as AES-NI, the SIMD instructions and the Intel SHA extensions can be adopted in the designs to speed up the operations.
- When the encryption and authentication share the same internal state in an authenticated cipher, it requires less internal state and is usually more efficient than performing the encryption and authenticated in two phrases.

Analysis and design are two highly related and mutually reinforcing aspects in cryptography. Together with the advances of the computational techniques and application needs, they have been continuously promoting the developments of cryptography. This process never ends and we will continue to find more interesting and profound analysis and construct better cryptographic primitives.

Bibliography

- [1] 3GPP. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3, Document 1, 128-EEA3 and 128-EIA3 specification. The 3rd Generation Partnership Project (3GPP), 2010.
- [2] Martin Ågren, Martin Hell, Thomas Johansson, and Willi Meier. Grain-128a: A New Version of Grain-128 with Optional Authentication. In *International Journal of Wireless and Mobile Computing*, Vol. 5, No. 1, pages 48–59. Inderscience Publishers, 2011.
- [3] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. NIST AES Proposal, 1998.
- [4] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Qingju Wang, and Kan Yasuda. PRIMATES v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [5] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. AES-COPA v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.

- [6] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [7] Steve Babbage, Christophe De Cannière, Joseph Lano, Bart Preneel, and Joos Vandewalle. Cryptanalysis of Sober-t32. In Thomas Johansson, editor, *Fast Software Encryption*, volume 2887 of *Lecture Notes in Computer Science*, pages 111–128. Springer Berlin Heidelberg, 2003.
- [8] Steve Babbage and Matthew Dodd. The MICKEY stream ciphers. In *New Stream Cipher Designs*, pages 191–209. Springer, 2008.
- [9] Guy Barwell. Forgery on Stateless CMCC. Cryptology ePrint Archive, Report 2014/251, 2014. <http://eprint.iacr.org/>.
- [10] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.
- [11] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *Advances in Cryptology – CRYPTO’96*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.
- [12] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated Encryption in SSH: Provably Fixing the SSH Binary Packet Protocol. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS ’02*, pages 1–11, New York, NY, USA, 2002. ACM.
- [13] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Advances in Cryptology–ASIACRYPT 2000*, pages 531–545, 2000.

BIBLIOGRAPHY

- [14] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX mode of operation. In *Fast Software Encryption*, pages 389–407. Springer, 2004.
- [15] Steven M Bellovin. Frank Miller: Inventor of the One-Time Pad. *Cryptologia*, 35(3):203–222, 2011.
- [16] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, et al. Sosemanuk, A Fast Software-Oriented Stream Cipher. In *New Stream Cipher Designs*, pages 98–118. Springer, 2008.
- [17] Daniel J Bernstein. CAESAR call for submissions, final (2014.01.27). <http://competitions.cr.yp.to/caesar-call.html>.
- [18] Daniel J Bernstein. Failures of Secret-Key Cryptography. Invited talk in FSE 2013, available from <http://cr.yp.to/talks/2013.03.12/slides.pdf>.
- [19] Daniel J Bernstein. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs*, pages 84–97. Springer, 2008.
- [20] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. Ecrypt Hash Workshop, May 2007.
- [21] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak Sponge Function Family Main Document. Submission to NIST (Round 2), 2009.
- [22] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer Berlin Heidelberg, 2012.

- [23] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Ketje v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [24] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Keyak v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [25] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the Middle Attacks on IDEA and Khufu. In *Fast Software Encryption*, pages 124–138. Springer, 1999.
- [26] Eli Biham, Orr Dunkelman, and Nathan Keller. Enhancing Differential-Linear Cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology–ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 254–266. Springer Berlin Heidelberg, 2002.
- [27] Eli Biham, Orr Dunkelman, and Nathan Keller. Differential-Linear Cryptanalysis of Serpent. In Thomas Johansson, editor, *Fast Software Encryption*, volume 2887 of *Lecture Notes in Computer Science*, pages 9–21. Springer Berlin Heidelberg, 2003.
- [28] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.
- [29] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, London, UK, UK, 1993.
- [30] Begül Bilgin, Andrey Bogdanov, Miroslav Knežević, Florian Mendel, and Qingju Wang. Fides: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware. In *Cryptographic Hardware and Embedded Systems–CHES 2013*, pages 142–158. Springer, 2013.

BIBLIOGRAPHY

- [31] Alex Biryukov. A New 128-bit Key Stream Cipher LEX. *eSTREAM, ECRYPT Stream Cipher Project, Report*, 13:2005, 2005.
- [32] Alex Biryukov, Christophe De Canniere, and Michaël Quisquater. On Multiple Linear Approximations. In *Advances in Cryptology – CRYPTO 2004*, pages 1–22. Springer, 2004.
- [33] Alex Biryukov and Dmitry Khovratovich. PAEQ v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [34] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology – CRYPTO’99*, volume 1666 of *LNCS*, pages 216–233. Springer, 1999.
- [35] J. Black and P. Rogaway. A Block-Cipher Mode of Operation for Parallelizable Message Authentication. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 384–397. Springer, 2002.
- [36] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: A New High-Performance Stream Cipher. In *Fast Software Encryption*, pages 307–329. Springer, 2003.
- [37] Andrey Bogdanov, Florian Mendel, Francesco Regazzoni, Vincent Rijmen, and Elmar Tischhauser. ALE: AES-Based Lightweight Authenticated Encryption. In *Fast Software Encryption*, 2013.
- [38] Andrey Bogdanov and Vincent Rijmen. Zero-Correlation Linear Cryptanalysis on Block Cipher. *Des. Codes Cryptogr*, 70(3), 2011.
- [39] Carolynn Burwick, Don Coppersmith, Edward D’Avignon, Rosario Genaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr, Luke O’Connor, Mohammad Peyravian, Jr. Luke, O’connor Mohammad Peyravian, David

- Stafford, and Nevenko Zunic. MARS – A Candidate Cipher for AES. NIST AES Proposal, 1999.
- [40] CAESAR. Competition for Authenticated Encryption: Security, Applicability, and Robustness. <http://competitions.cr.yp.to/caesar.html>.
- [41] Simon Cogliani, Diana Ștefania Maimuț, David Naccache, Rodrigo Portella do Canto, Reza Reyhanitabar, Serge Vaudenay, and Damian Vizár. Offset Merkle-Damgård (OMD) version 1.0: A CAESAR Proposal. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [42] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications*, volume 1820 of *LNCS*, pages 277–284. Springer Berlin Heidelberg, 2000.
- [43] Joan Daemen and Vincent Rijmen. The Wide Trail Design Strategy. In Bahram Honary, editor, *Cryptography and Coding*, volume 2260 of *LNCS*, pages 222–238. Springer Berlin Heidelberg, 2001.
- [44] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES—the Advanced Encryption Standard*. Springer, 2002.
- [45] Joan Daemen and Vincent Rijmen. Refinements of the ALRED construction and MAC security claims. *IET information security*, 4(3):149–157, 2010.
- [46] Nilanjan Datta and Mridul Nandi. ELmD v1.0. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.

BIBLIOGRAPHY

- [47] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Information Security*, pages 171–186. Springer, 2006.
- [48] Tim Dierks and Eric Rescorla. Rfc 5246: The Transport Layer Security (TLS) Protocol. *The Internet Engineering Task Force*, 2008.
- [49] Whitfield Diffie and Martin E Hellman. New Directions in Cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
- [50] Itai Dinur and Jérémy. Cryptanalysis of FIDES. Cryptology ePrint Archive, Report 2014/058, 2014. <http://eprint.iacr.org/>.
- [51] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schlfer. Ascon v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [52] Orr Dunkelman, Sebastiaan Indesteege, and Nathan Keller. A Differential-Linear Attack on 12-Round Serpent. In DipanwitaRoy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 308–321. Springer Berlin Heidelberg, 2008.
- [53] Orr Dunkelman and Nathan Keller. A New Attack on the LEX Stream Cipher. In *ASIACRYPT 2008*, pages 539–556. Springer, 2008.
- [54] Orr Dunkelman and Nathan Keller. Cryptanalysis of CTC2. In Marc Fischlin, editor, *Topics in Cryptology CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 226–239. Springer Berlin Heidelberg, 2009.
- [55] Orr Dunkelman and Nathan Keller. Cryptanalysis of the Stream Cipher LEX. *Designs, Codes and Cryptography*, 67(3):357–373, 2013.

- [56] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST special publication 800–38C, 2004.
- [57] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode(GCM) and GMAC. NIST Special Publication 800–38D, 2007.
- [58] ECRYPT Stream Cipher Project. Call for Stream Cipher Primitives,<http://www.ecrypt.eu.org/stream/call/>, 2004-2008.
- [59] ECRYPT Stream Cipher Project. <http://www.ecrypt.eu.org/stream/>, 2004-2008.
- [60] Daniel Engels, Markku-Juhani O Saarinen, Peter Schweitzer, and Eric M Smith. The Hummingbird-2 Lightweight Authenticated Encryption Algorithm. In *RFIDSec 2011*, pages 19–31. Springer, 2011.
- [61] Xiutao Feng, Jun Liu, Zhaocun Zhou, Chuankun Wu, and Dengguo Feng. A Byte-based Guess and Determine Attack on SOSEMANUK. In *Advances in Cryptology–ASIACRYPT 2010*, pages 146–157. Springer, 2010.
- [62] Xiutao Feng and Fan Zhang. A practical state recovery attack on the stream cipher Sablier v1.
- [63] Niels Ferguson. NIST Public Comments for symmetric key block ciphers: Collision attacks on OCB. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/General_Comments/papers/Ferguson.pdf, 2002.
- [64] Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, and Tadayoshi Kohno. Helix, Fast Encryption and Authentication in a Single Cryptographic Primitive. In *Fast Software Encryption – FSE 2003*, pages 330–346. Springer, 2003.

BIBLIOGRAPHY

- [65] Ewan Fleischmann, Christian Forler, and Stefan Lucks. McOE: A Family of Almost Foolproof On-Line Authenticated Encryption Schemes. In *Fast Software Encryption–FSE 2012*, pages 196–215. Springer, 2012.
- [66] Christian Forler, David McGrew, Stefan Lucks, and Jakob Wenzel. COFFE: Ciphertext Output Feedback Faithful Encryption. *Early Symetric Crypto ESC 2013*, page 6, 2013.
- [67] Peter Freeman. The Zimmermann Telegram Revisited: A Reconciliation of the Primary Sources. *Cryptologia*, 30(2):98–150, 2006.
- [68] Wes Freeman, Geoff Sullivan, and Frode Weierud. Purple Revealed: Simulation and Computer-Aided Cryptanalysis of Angooki Taipu B. *Cryptologia*, 27(1):1–43, 2003.
- [69] Kris Gaj and Arkadiusz Orłowski. Facts and Myths of Enigma: Breaking Stereotypes. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 106–122. Springer Berlin Heidelberg, 2003.
- [70] Danilo Gligoroski, Hristina Mihajloska, Simona Samardjiska, Håkon Jacobsen, Mohamed El-Hadedy, and Rune Erlend Jensen. π -Cipher v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [71] Danilo Gligoroski, Hristina Mihajloska, Simona Samardjiska, Håkon Jacobsen, Mohamed El-Hadedy, and Rune Erlend Jensen. π -Cipher v2. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [72] Shay Gueron. AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR. DIAC 2013: Directions in Authenticated Ciphers, August 2013.

- [73] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. Intel® sha extensions: New instructions supporting the secure hash algorithm on intel® architecture processors, jul 2013. available from <https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf>.
- [74] Jian Guo, Jérémy Jean, Thomas Peyrin, and Wang Lei. Breaking POET Authentication with a Single Query. *Cryptology ePrint Archive*, Report 2014/197, 2014. <http://eprint.iacr.org/>.
- [75] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED Block Cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems—CHES 2011*, volume 6917 of *LNCS*, pages 326–341. Springer Berlin Heidelberg, 2011.
- [76] Philip Hawkes and Gregory G Rose. Guess-and-Determine Attacks on SNOW. In *Selected Areas in Cryptography*, pages 37–46. Springer, 2003.
- [77] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. *New Stream Cipher Designs*, pages 179–190, 2008.
- [78] Martin E Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401, 1980.
- [79] Miia Hermelin, Joo Yeon Cho, and Kaisa Nyberg. Multidimensional Linear Cryptanalysis of Reduced Round Serpent. In *Information Security and Privacy*, pages 203–215. Springer, 2008.
- [80] Miia Hermelin, Joo Yeon Cho, and Kaisa Nyberg. Multidimensional Extension of Matsui’s Algorithm 2. In *Fast Software Encryption*, pages 209–227. Springer, 2009.

BIBLIOGRAPHY

- [81] ISO/IEC 19772:2009. *Information technology – Security techniques – Authenticated encryption*. ISO, Geneva, Switzerland, 2009.
- [82] Tetsu Iwata and Kaoru Kurosawa. OMAC: One-key CBC MAC. In *Fast Software Encryption*, volume 2887 of *LNCS*, pages 129–153. Springer, 2003.
- [83] Tetsu Iwata and Kan Yasuda. BTM: A Single-Key, Inverse-Cipher-Free Mode for Deterministic Authenticated Encryption. In *Selected Areas in Cryptography – SAC 2009*, pages 313–330. Springer, 2009.
- [84] Tetsu Iwata and Kan Yasuda. HBS: A Single-Key Mode of Operation for Deterministic Authenticated Encryption. In *Fast Software Encryption–FSE 2009*, pages 394–415. Springer, 2009.
- [85] Goce Jakimoski and Samant Khajuria. ASC-1: An Authenticated Encryption Stream Cipher. In *Selected Areas in Cryptography – SAC 2011*, pages 356–372. Springer, 2011.
- [86] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Joltik v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [87] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. KIASU v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [88] Charanjit S Jutla. Encryption Modes with Almost Free Message Integrity. In *Advances in Cryptology – EUROCRYPT 2001*, pages 529–544. Springer, 2001.
- [89] Burton S Kaliski Jr and Matthew JB Robshaw. Linear Cryptanalysis Using Multiple Approximations. In *Advances in Cryptology–Crypto’94*, pages 26–39. Springer, 1994.

- [90] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. Prøst v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [91] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. Prøst v1.1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [92] Richard F Kayser. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 72(212):62, 2007.
- [93] Dmitry Khovratovich and Christian Rechberger. The LOCAL attack: Cryptanalysis of the authenticated encryption scheme ALE. In *Selected Areas in Cryptography – SAC 2013*. Springer Berlin Heidelberg, 2013.
- [94] Lars R Knudsen. Truncated and Higher Order Differentials. In *Fast Software Encryption*, pages 196–211. Springer, 1995.
- [95] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A High-Performance Conventional Authenticated Encryption Mode. In *Fast Software Encryption – FSE 2004*, pages 408–426. Springer, 2004.
- [96] Hugo Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer Berlin Heidelberg, 2001.
- [97] Ted Krovetz and Phillip Rogaway. Authenticated-Encryption Software Performance: Comparison of CCM, GCM, and OCB. Available at <http://www.cs.ucdavis.edu/~rogaway/ocb/performance/>.

BIBLIOGRAPHY

- [98] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In *Fast Software Encryption – FSE 2011*, pages 306–327. Springer, 2011.
- [99] Watson Ladd. MCMAMBO V1: A New Kind Of Latin Dance. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [100] Xuejia Lai. Higher Order Derivatives and Differential Cryptanalysis. In *Communications and Cryptography*, pages 227–233. Springer, 1994.
- [101] SusanK. Langford and MartinE. Hellman. Differential-Linear Cryptanalysis. In YvoG. Desmedt, editor, *Advances in Cryptology–CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 17–25. Springer Berlin Heidelberg, 1994.
- [102] Gaëtan Leurent. Differential Forgery Attack against LAC. Presented at DIAC 2014, July 2014.
- [103] Gaëtan Leurent and Thomas Fuhr. Observation on PiCipher. Cryptographic Competitions Mailing List, 2014.
- [104] Jiqiang Lu. A methodology for differential-linear cryptanalysis and its applications. In Anne Canteaut, editor, *Fast Software Encryption*, volume 7549 of *Lecture Notes in Computer Science*, pages 69–89. Springer Berlin Heidelberg, 2012.
- [105] Mitsuru Matsui. Linear Cryptanalysis Method for DES cipher. In *Advances in Cryptology–EUROCRYPT’93*, pages 386–397. Springer, 1994.
- [106] Mitsuru Matsui and Atsuhiro Yamagishi. A New Method for Known Plaintext Attack of FEAL Cipher. In *Advances in Cryptology – EUROCRYPT’92*, pages 81–91. Springer, 1993.

- [107] David McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). <http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf>.
- [108] Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grstl. In *Fast Software Encryption*, pages 260–276. Springer, 2009.
- [109] Carl H Meyer and Michael Schilling. Secure Program Load with Manipulation Detection Code. In *Proc. Securicom*, volume 88, pages 111–130, 1988.
- [110] Chris J Mitchell. Cryptanalysis of the EPBC Authenticated Encryption Mode. In StevenD. Galbraith, editor, *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 118–128. Springer Berlin Heidelberg, 2007.
- [111] Chris J Mitchell. Analysing the IOBC Authenticated Encryption Mode. In Colin Boyd and Leonie Simpson, editors, *Information Security and Privacy*, volume 7959 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2013.
- [112] Pawe Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wjcik. ICEPOLE v1. submission to CAESAR competition, available from <http://competitions.cr.yp.to/round1/icepolev1.pdf>.
- [113] Pawe Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wjcik. ICE-POLE: High-Speed, Hardware-Oriented Authenticated Encryption. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems, CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 392–413. Springer Berlin Heidelberg, 2014.

BIBLIOGRAPHY

- [114] Mridul Nandi. Forging Attack on COBRA Mode. Cryptographic Competitions Mailing List, 2014.
- [115] Samuel Neves. McMambo Iterative Differential. Cryptographic Competitions Mailing List, 2014.
- [116] Kaisa Nyberg. Linear Approximation of Block Ciphers. In *Advances in Cryptology–EUROCRYPT’94*, pages 439–444. Springer, 1995.
- [117] National Institute of Standards and Technology. Block Cipher Modes. <http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html>.
- [118] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation. NIST special publication 800–38A, 2001 Edition.
- [119] National Institute of Standards and Technology. Request for Candidate Algorithm Nominations for the AES. Available from http://csrc.nist.gov/archive/aes/pre-round1/aes_9709.htm, 1997.
- [120] National Institute of Standards and Technology. Data Encryption Standard (DES). FIPS PUB 46-3, 1999.
- [121] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standard, FIPS-180-4, March 2012.
- [122] Erez Petrank and Charles Rackoff. CBC MAC for Real-Time Data Sources. *Journal of Cryptology*, 13:315–338, 1997.
- [123] Thomas Peyrin, Siang Meng Sim, Lei Wang, and Guoyan Zhang. Cryptanalysis of JAMBU. *Fast Software Encryption – FSE 2015*, 2015.
- [124] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *Advances in Cryptology–CRYPTO’93*, pages 368–378. Springer, 1994.

- [125] Bart Preneel and Paul C Van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Advances in Cryptology – CRYPTO’ 95*, volume 963 of *LNCS*, pages 1–14. Springer, 1995.
- [126] Bart Preneel and Paul C Van Oorschot. On the Security of Iterated Message Authentication Codes. *Information Theory, IEEE Transactions on*, 45(1):188–199, 1999.
- [127] Francisco Recacha. IOBC: Un nuevo modo de encadenamiento para cifrado en bloque. In *Proceedings: IV Reunion Espanola de Criptologia, Valladolid, September 1996*, pages 85–92, 1996. ISBN 84-7762-645-6.
- [128] Francisco Recacha. IOBC: A new authenticated encryption mode. Available from <http://inputoutputblockchaining.blogspot.com/>, January 2012.
- [129] Francisco Recacha. IOC: The Most Lightweight Authenticated Encryption Mode? . Available from <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ioc/ioc-spec.pdf>, 2013.
- [130] Francisco Recacha. Input Output Chaining (IOC) AE Mode Revisited. Available from <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ioc/ioc-spec-2014.pdf>, 2014.
- [131] Ronald L Rivest, MJB Robshaw, Ray Sidney, and Yiqun Lisa Yin. The RC6™ block cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.
- [132] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, 2002.
- [133] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology–ASIACRYPT 2004*, pages 16–31. Springer, 2004.

BIBLIOGRAPHY

- [134] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 196–205. ACM, 2001.
- [135] Allyn Romanow, Romanow. Media Access Control (MAC) Security. *IEEE 802.1 AE*, 2004.
- [136] Markku-Juhani O. Saarinen. The STRIBOBr1 Authenticated Encryption Algorithm. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [137] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. NIST AES Proposal, 1998.
- [138] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.
- [139] Yongsup Shin, Jongsung Kim, Guil Kim, Seokhie Hong, and Sangjin Lee. Differential-Linear Type Attacks on Reduced Rounds of SHACAL-2. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *Information Security and Privacy*, volume 3108 of *Lecture Notes in Computer Science*, pages 110–122. Springer Berlin Heidelberg, 2004.
- [140] Siang Meng Sim and Lei Wang. Practical Forgery Attacks on SCREAM and iSCREAM, 2014. <http://www1.spms.ntu.edu.sg/~syllab/m/images/b/b3/ForgeryAttackonSCREAM.pdf>.
- [141] ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.4; 30th July 2010.

- [142] ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.5; 4th January 2011.
- [143] ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.6; 28th June 2011.
- [144] John Viega and David A McGrew. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP), 2005.
- [145] David Wagner. The Boomerang Attack. In *Fast Software Encryption*, pages 156–170. Springer, 1999.
- [146] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). Available from <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmo-des/ccm/ccm.pdf>, 2003.
- [147] Doug Whiting, B. Schneier, S. Lucks, and F. Muller. Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive. eSTREAM, ECRYPT Stream Cipher Project Report 2005/027.
- [148] Michael J Wiener. The Full Cost of Cryptanalytic Attacks. *Journal of Cryptology*, 17(2):105–124, 2004.
- [149] Hongjun Wu. ACORN: A Lightweight Authenticated Cipher (v1). Submission to CAESAR. Available from <http://competitions.cr.yp.to/round1/acornv1.pdf>.
- [150] Hongjun Wu. The Stream Cipher HC-128. In *New Stream Cipher Designs*, pages 39–47. Springer, 2008.

BIBLIOGRAPHY

- [151] Hongjun Wu and Tao Huang. JAMBU Lightweight Authenticated Encryption Mode and AES-JAMBU (v1). Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [152] Hongjun Wu and Tao Huang. The Authenticated Cipher MORUS (v1). Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [153] Hongjun Wu, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. Cryptanalysis of the Stream Cipher ZUC in the 3GPP Confidentiality & Integrity Algorithms 128-EEA3 & 128-EIA3. Rump Session of Asiacrypt 2010.
- [154] Hongjun Wu and Bart Preneel. Resynchronization Attacks on WG and LEX. In *Fast Software Encryption – FSE 2006*, pages 422–432. Springer, 2006.
- [155] Hongjun Wu and Bart Preneel. Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy. *Advances in Cryptology – EUROCRYPT 2007*, pages 276–290, 2007.
- [156] Hongjun Wu and Bart Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. *Selected Areas in Cryptography – SAC 2013*, 2013.
- [157] Shengbao Wu, Hongjun Wu, Tao Huang, Mingsheng Wang, and Wenling Wu. Leaked-State-Forgery Attack against the Authenticated Encryption Algorithm ALE. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 377–404. Springer Berlin Heidelberg, 2013.
- [158] Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, and Xiaoyun Wang. New Birthday Attacks on Some MACs Based on Block Ciphers. In *Advances in Cryptology – CRYPTO 2009*, pages 209–230. Springer, 2009.
- [159] Gideon Yuval. How to Swindle Rabin. *Cryptologia*, 3(3):187–191, 1979.

BIBLIOGRAPHY

- [160] Bin Zhang, Zhenqing Shi, Chao Xu, Yuan Yao, and Zhenqi Li. Sablier v1. Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.

Appendix A

Session Key Generation Analysis of COFFE

Recall that here we are trying to investigate the session key generation step whether we can introduce a collision between two valid input such that one has 1– byte value for the domain value, named S_1 , and another with 2– bytes value for the domain value, S_2 . Here we need to consider the number of bytes for $\mathcal{L}_{K_1}$, $\mathcal{L}_{V_1}$, and whether zero padding exists in S_1 . We will assume that if a string has length that needs to be expressed as a 2-bytes value, $256a + b$, then $a \neq 0$ since otherwise it is not necessary to express the length as a 2-bytes value. We also can assume that for any secret keys considered, it cannot have length 0 since otherwise, there is no need to perform any attack here.

I. **S_1 has no zero padding.** Let $S_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathcal{L}_{K_1} \parallel \mathcal{L}_{V_1} \parallel [0]_1)$. We will divide this case into smaller sub-cases to handle independently.

I.1. Case I.1. $[\mathcal{L}_{K_1}] = [\mathcal{L}_{V_1}] = 1$. By observation 5.3.2, we have that $\mathcal{L}_{V_1} = 0$.

Let $1 \leq a \leq 255$ such that $\mathcal{L}_{K_1} = a$. Then we have:

$$S_1 = (\mathcal{K}_1 \parallel a \parallel [0]_2).$$

Now we consider the second string $\mathcal{S}_2$ with the domain value being expressed by 2-bytes string. Then we have $\mathcal{L}_{V_2} \geq \mathbf{a}$. Since we need at least 1 byte to represent $\mathcal{L}_{K_2}$ and a non-empty string of $\mathcal{K}_2$, we need at least $\mathbf{a} + 2$ bits in the remaining unused string in $\mathcal{S}_2$. However, since we want $\mathcal{S}_1 = \mathcal{S}_2$, we only have $\mathbf{a}$ bits left that is unused. Hence if $[\mathcal{L}_{K_1}] = [\mathcal{L}_{V_1}] = 1$, it is impossible for us to have a collision between $\mathcal{S}_1$ and $\mathcal{S}_2$.

- I.2. Case I.2. $[\mathcal{L}_{K_1}] = 2$ and $[\mathcal{L}_{V_1}] = 1$. Observation 5.3.2 implies that $\mathcal{L}_{V_1} = 0$. Let $\mathcal{L}_{K_1} = 256\mathbf{a} + \mathbf{b}$ for some $0 \leq \mathbf{a}, \mathbf{b} \leq 255$ where $\mathbf{a} \neq 0$. So we have:

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Now we consider the second string $\mathcal{S}_2$. We have different cases depending on the number of bytes for $\mathcal{L}_{K_2}$ and $\mathcal{L}_{V_2}$.

- Case I.2.a. $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 1$. Then we have that $\mathcal{L}_{K_2} = \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Now if $\mathcal{S}_2$ has no zero padding, then we have that $256\mathbf{a} + \mathbf{b} = \mathbf{a} + \mathbf{b}$ which implies that $\mathbf{a} = 0$ which contradicts the fact that $\mathcal{L}_{K_1}$ needing to be expressed as a 2-bytes value. So $\mathcal{S}_2$ must have zero padding. A simple comparison tells us that there will be $255\mathbf{a}$ bits of zero paddings in $\mathcal{S}_2$. However, remember that this zero padding comes from $\mathcal{K}_1$ which is not controllable by us the attacker. Hence we can just hope this is true with probability $2^{-255\mathbf{a}}$. Since $\mathbf{a} \geq 1$, this probability is too small to be feasible. So this case can be discarded from the possible $\mathcal{S}_1$ collision with $\mathcal{S}_2$.
- Case I.2.b. $[\mathcal{L}_{K_2}] = 2, [\mathcal{L}_{V_2}] = 1$. For simplicity, let $\mathcal{K}_1 = \mathcal{K}'_1 \parallel \mathbf{c}$ where $\mathbf{c}$ is the last 8 bits of $\mathcal{K}_1$. Then $\mathcal{L}_{K_2} = 256\mathbf{c} + \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Let t' be the number of bits of zero padding in $\mathcal{S}_2$. Then equating the length of $\mathcal{S}_1$ and $\mathcal{S}_2$, we get $256\mathbf{a} + \mathbf{b} = 256\mathbf{c} + \mathbf{a} + \mathbf{b} + t' + 8$ where the 8 comes from $\mathbf{c}$ being a part of $\mathcal{K}_1$ despite it not being a part of $\mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$.

This can be simplified to $255\mathbf{a} = 256\mathbf{c} + t' + 8$. So we define

$$\mathcal{F}_{(2,1),(2,1)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') : 0 \leq \mathbf{a}, \mathbf{b}, \mathbf{c} \leq 255, t' \geq 0, \mathbf{a}, \mathbf{c} \neq 0, 255\mathbf{a} = 256\mathbf{c} + t' + 8\}$$

Now we consider the feasibility of each quadruple $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(2,1),(2,1)}$.

Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(2,1),(2,1)}$. Since we have $\mathcal{K}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$, it is clear that $|\mathcal{K}_1| > |\mathcal{K}_2|$. Then we have $256\mathbf{a} + \mathbf{b} - (256\mathbf{c} + \mathbf{a}) = 255\mathbf{a} + \mathbf{b} - 256\mathbf{c}$ bits of $\mathcal{K}_1$ which needs to be known or fixed. Like before, we assume the bits of $\mathcal{K}_1$ which is in $\mathcal{K}_2$ coincide with probability one since we assume that $\mathcal{K}_1$ is the extension of $\mathcal{K}_2$ by appending it with some bits. So the probability of this happening is $2^{-(255\mathbf{a} + \mathbf{b} - 256\mathbf{c})}$. A simple calculation tells us that we can set $255\mathbf{a} + \mathbf{b} - 256\mathbf{c}$ to be as low as 8. Now for any integer $k \geq 8$, we define $\mathcal{F}_{(2,1),(2,1),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(2,1),(2,1)}, 255\mathbf{a} + \mathbf{b} - 256\mathbf{c} \geq k\}$. Then if we take the quadruple $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(2,1),(2,1),k}$, we know that the probability of collision to happen is at least 2^{-k} . We also need to note that since $\mathcal{K}_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathbf{c}$, we must have the last $t' + 8$ bits of $\mathcal{K}_1$ to be $0^{t'} \parallel \mathbf{c}$. So any collision we derive from this family, $\mathcal{K}_1$ cannot be chosen to be any keys. The collision will only happen in the subfamily of the $256\mathbf{a} + \mathbf{b}$ -bits string such that the last $t' + 8$ bits must be $0^{t'} \parallel \mathbf{c}$.

- Case I.2.c. $[\mathcal{L}_{V_2}] = 2$. Then $\mathcal{L}_{V_2} = 256\mathbf{a} + \mathbf{b}$. Note that here we again have $\mathcal{K}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^t$ where $\mathcal{K}'_1$ is a $256\mathbf{a} + \mathbf{b} - 8$ bits binary string. However, this is impossible since $\mathcal{V}_2$ itself should have $256\mathbf{a} + \mathbf{b}$ bits by itself. So this case is again impossible.

I.3. Case I.3. $[\mathcal{L}_{K_1}] = 1, [\mathcal{L}_{V_1}] = 2$. By observation 5.3.2, $\mathcal{L}_{V_1}$ is a multiple of 256. Let $1 \leq \mathbf{a}, \mathbf{b} \leq 255$ such that $\mathcal{L}_{K_1} = \mathbf{a}$ and $\mathcal{L}_{V_1} = 256\mathbf{b}$. Then we have

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Now we consider the possibilities of $\mathcal{S}_2$.

- Case I.3.a $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 1$. Then $\mathcal{L}_{K_2} = \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Then certainly $\mathcal{S}_2$ must have zero paddings. More precisely, we must have $\mathcal{K}_1 = \mathcal{K}_2$ and $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^{255\mathbf{b}}$. So for any choice of key and nonce, we must have the collision with probability one as long as $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^{255\mathbf{b}}$.
- Case I.3.b $[\mathcal{L}_{K_2}] = 2, [\mathcal{L}_{V_2}] = 1$. Let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{c}$. Then we have $\mathcal{L}_{K_2} = 256\mathbf{c} + \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Now let $\mathcal{S}_2$ to have t' -bits zero padding. Then equating the length of $\mathcal{S}_1$ and $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathbf{c}$. So we have $\mathbf{a} + 256\mathbf{b} = 256\mathbf{c} + \mathbf{a} + \mathbf{b} + t' + 8$. Then we have $255\mathbf{b} = 256\mathbf{c} + 8 + t'$. As before, we define a family

$$\mathcal{F}_{(1,2),(2,1)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') : 1 \leq \mathbf{a}, \mathbf{b}, \mathbf{c} \leq 255, t' \geq 0, 255\mathbf{b} = 256\mathbf{c} + 8 + t'\}.$$

Now we consider the feasibility of each quadruple $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(1,2),(2,1)}$. Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t') \in \mathcal{F}_{(1,2),(2,1)}$. Then the length of the bits that is in $\mathcal{K}_2$ but not in $\mathcal{K}_1$ is $256\mathbf{c}$. Now remember that since $\mathcal{L}_{K_2} = 256\mathbf{c} + \mathbf{a}$, $\mathbf{c} \neq 0$. So the probability of collision to happen is at most 2^{-256} which is infeasible. So we can disregard this case.

- Case I.3.c $[\mathcal{L}_{K_2}] = 1, [\mathcal{L}_{V_2}] = 2$. This case has been discussed in detail in the main section. So we will skip the discussion of this case here.
- Case I.3.d $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 2$. Let $\mathcal{V}_1 = \mathcal{V}''_1 \parallel \mathbf{d} \parallel \mathbf{c}$ where $\mathbf{d} \parallel \mathbf{c}$ is the last 2-bytes of $\mathcal{V}_1$. So $\mathcal{L}_{K_2} = 256\mathbf{d} + \mathbf{c}$ and $\mathcal{L}_{V_2} = 256\mathbf{a} + \mathbf{b}$. Similar as before, we let $\mathcal{S}_2$ to have t' -bits of zero padding. Now equating the length of $\mathcal{S}_1$ and $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}''_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathcal{L}_{K_2}$. In other words, $\mathbf{a} + 256\mathbf{b} = 256\mathbf{d} + \mathbf{c} + 256\mathbf{a} + \mathbf{b} + t' + 16$ or equivalently

$$255(\mathbf{b} - \mathbf{a}) = 256\mathbf{d} + \mathbf{c} + t' + 16.$$

Let $\mathcal{F}_{(1,2),(2,2)}$ be the family:

$$\mathcal{F}_{(1,2),(2,2)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t') : 1 \leq \mathbf{a}, \mathbf{b}, \mathbf{d} \leq 255, 0 \leq \mathbf{c} \leq 255, t' \geq 0,$$

$$255(\mathbf{b} - \mathbf{a}) = 256\mathbf{d} + \mathbf{c} + t' + 16\}.$$

Now we investigate each quintuple in this family. Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t') \in \mathcal{F}_{(1,2),(2,2)}$. Obviously, since $[\mathcal{K}_2] = 2$ and $[\mathcal{K}_1] = 1$, $|\mathcal{K}_2| > |\mathcal{K}_1|$. Note that the number of bits that is in $\mathcal{K}_2$ but not in $\mathcal{K}_1$, and hence needs to be guessed, is $256\mathbf{d} + \mathbf{c} - \mathbf{a}$. This gives us the success probability of the collision to be $2^{\mathbf{a}-256\mathbf{d}-\mathbf{c}}$. A simple calculation tells us that this can be as small as 3, for example when $\mathbf{a} = 253, \mathbf{b} = 255, \mathbf{c} = 0, \mathbf{d} = 1, t' = 238$. Now as before, for any integer $k \geq 3$, we define a subclass $\mathcal{F}_{(1,2),(2,2),k}$ of $\mathcal{F}_{(1,2),(2,2)}$ such that $\mathcal{F}_{(1,2),(2,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t) \in \mathcal{F}_{(1,2),(2,2)} : 256\mathbf{d} + \mathbf{c} - \mathbf{a} \geq k\}$. Note that here since $\mathcal{K}_2$ is the longer key and $\mathcal{K}_1 \parallel \mathcal{V}_1'' = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathcal{L}_{K_2}$, the extension of the $\mathcal{K}_2$ but not in $\mathcal{K}_1$ must be in $\mathcal{V}_1''$ which is controllable. So any choice of $\mathcal{K}_2$ is vulnerable to the collision.

- I.4. Case I.4. $[\mathcal{L}_{K_1}] = [\mathcal{L}_{V_1}] = 2$. By observation 5.3.2, we have $\mathcal{L}_{V_1}$ is a multiple of 256. Let $1 \leq \mathbf{a}, \mathbf{c} \leq 255, 0 \leq \mathbf{b} \leq 255$ such that $\mathcal{L}_{K_1} = 256\mathbf{a} + \mathbf{b}$ and $\mathcal{L}_{V_1} = 256\mathbf{c}$. Then we have:

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathbf{a} \parallel \mathbf{b} \parallel \mathbf{c} \parallel [0]_2).$$

Dividing the case further based on $\mathcal{S}_2$, we have the following cases:

- Case I.4.a $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 1$. Then $\mathcal{L}_{K_2} = \mathbf{b}$ and $\mathcal{L}_{V_2} = \mathbf{c}$. Now let $\mathcal{S}_2$ to have t' bits of zero padding. Now equating the length of $\mathcal{S}_1$ and $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathbf{a} = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. So we have $256\mathbf{a} + \mathbf{b} + 256\mathbf{c} + 8 = \mathbf{b} + \mathbf{c} + t'$. So we have $t' = 256\mathbf{a} + 255\mathbf{c} + 8$. Now considering the string $\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel \mathbf{a} = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$ from the right, we have that since $\mathbf{a} \neq 0$, we must have $t' \leq 7$. Since otherwise, this will force $\mathbf{a} = 0$. So we have $256\mathbf{a} + 255\mathbf{c} + 8 \leq 7$. However, this is clearly impossible since $1 \leq \mathbf{a}, \mathbf{c}$. So we can disregard this case.
- Case I.4.b $[\mathcal{L}_{K_2}] = 2, [\mathcal{L}_{V_2}] = 1$. Then $\mathcal{L}_{K_2} = 256\mathbf{a} + \mathbf{b}$ and $\mathcal{L}_{V_2} = \mathbf{c}$.

Letting $\mathcal{S}_2$ to have t -bits of zero padding, equating $\mathcal{S}_1$ to $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^t$. Now since $\mathcal{L}_{K_1} = \mathcal{L}_{K_2}$, we must have $\mathcal{K}_1 = \mathcal{K}_2$. So this leaves us with $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^t$ which implies $t = 255c$.

- Case I.4.c $[\mathcal{L}_{K_2}] = 1, [\mathcal{L}_{V_2}] = 2$. Then we have $\mathcal{L}_{K_2} = \mathbf{a}$ and $\mathcal{L}_{V_2} = 256\mathbf{b} + \mathbf{c}$. Now inspecting the first part of both $\mathcal{S}_1$ and $\mathcal{S}_2$, we must have at least $255\mathbf{a} + \mathbf{b}$ bits of $\mathcal{K}_1$ that is not in $\mathcal{K}_2$ which means it must be guessed. Now since $\mathbf{a} \neq 0$, we have the probability of success to be at most 2^{-255} which is already infeasible. So we can disregard this case.
- Case I.4.d $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 2$. For this, we let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{d}$ where $\mathbf{d}$ is the last byte of $\mathcal{V}_1$. Then we have $\mathcal{L}_{K_2} = 256\mathbf{d} + \mathbf{a}$ and $\mathcal{L}_{V_2} = 256\mathbf{b} + \mathbf{c}$. Let $\mathcal{S}_2$ have t -bits of zero padding. Then equating $\mathcal{S}_1$ to $\mathcal{S}_2$, we have $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^t$. So we have $256\mathbf{a} + \mathbf{b} + 256\mathbf{c} - 8 = 256\mathbf{d} + \mathbf{a} + 256\mathbf{b} + \mathbf{c} + t'$. Equivalently, we have

$$255(\mathbf{a} - \mathbf{b} + \mathbf{c}) = 256\mathbf{d} + t' + 8.$$

Let $\mathcal{F}_{(2,2),(2,2)}$ be the family:

$$\mathcal{F}_{(2,2),(2,2)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t') : 1 \leq \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \leq 255, t' \geq 0,$$

$$255(\mathbf{a} - \mathbf{b} + \mathbf{c}) = 256\mathbf{d} + t' + 8\}.$$

Now we investigate the feasibility of each quintuple in the family.

Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t') \in \mathcal{F}_{(2,2),(2,2)}$. Note that there are $|256\mathbf{a} + \mathbf{b} - (256\mathbf{d} + \mathbf{a})| =$

$|255\mathbf{a} + \mathbf{b} - 256\mathbf{d}|$ bits that is in one but not both secret keys. This means that the probability of success is at most $2^{-|255\mathbf{a} + \mathbf{b} - 256\mathbf{d}|}$. We

can perform a simple calculation to discover that the value $|255\mathbf{a} + \mathbf{b} - 256\mathbf{d}|$ can be made to be 0, for example, when $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t') = (1, 1, 2, 1, 246)$.

For this purpose, for any non-negative integers k , we define a subclass $\mathcal{F}_{(2,2),(2,2),k}$ of $\mathcal{F}_{(2,2),(2,2)}$ such that $\mathcal{F}_{(2,2),(2,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t) \in$

$\mathcal{F}_{(2,2),(2,2)} : |255\mathbf{a} + \mathbf{b} - 256\mathbf{d}| \leq k\}$. So for any quintuple extracted from $\mathcal{F}_{(2,2),(2,2),k}$, it can be used to guess the extension of at most k bits of the secret key with the success probability being at least 2^{-k} . Now we need to investigate the extension of the secret key. Recall that $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. Now if $\mathcal{K}_2$ is longer, then the extension will all be in $\mathcal{V}'_1$ considering the left term only has $\mathcal{V}'_1$ in addition to $\mathcal{K}_1$. So for this case, there is no restriction for either key. Now we check the case when $\mathcal{K}_1$ is longer. Note that if $\mathcal{K}_1$ is longer, then the extension is from $\mathcal{V}_2 \parallel 0^{t'}$. Now if $\mathcal{V}'_1$ is longer than t' , then since the two strings are equal, no bits from the extension of $\mathcal{K}_1$ can be from the zero padding. So if $t' < \mathcal{L}_{\mathcal{V}'_1} = \mathcal{L}_{\mathcal{V}_1} - 8$, then there are no restriction to $\mathcal{K}_1$. On the other hand, if $t' > \mathcal{L}_{\mathcal{V}_1} - 8$, then the last $t' - (\mathcal{L}_{\mathcal{V}_1} - 8)$ bits of $\mathcal{K}_1$ must be zero. So in other words, when $\mathcal{K}_1$ is longer, the last $\max(0, t' - (\mathcal{L}_{\mathcal{V}_1} - 8))$ bits of it must be 0.

II. $\mathcal{S}_1$ has a t -bits zero padding. So now we have $\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^t \parallel \mathcal{L}_{\mathcal{K}_1} \parallel \mathcal{L}_{\mathcal{V}_1} \parallel [0]_1)$.

We will again consider some small sub-cases to analyse.

II.1. Case II.1. $[\mathcal{L}_{\mathcal{K}_1}] = [\mathcal{L}_{\mathcal{V}_1}] = 1$ As before, Observation 5.3.2 tells us that $\mathcal{L}_{\mathcal{V}_1} = 0$. Let $1 \leq \mathbf{a} \leq 255$ such that $\mathcal{L}_{\mathcal{K}_1} = \mathbf{a}$. Then we have:

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel 0^t \parallel \mathbf{a} \parallel [0]_2).$$

Now we consider the second string $\mathcal{S}_2$. Note that here $\mathcal{L}_{\mathcal{V}_2} \geq \mathbf{a}$. So we will divide this to two cases:

- Case II.1.a. $\mathcal{L}_{\mathcal{V}_2} = \mathbf{a}$. We also have that $\mathcal{K}_1 \parallel 0^t = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathcal{L}_{\mathcal{K}_2}$ where t' is the length of the zero padding of $\mathcal{S}_2$. Now since we need the next 1 or 2 bytes in the left of $\mathcal{L}_{\mathcal{V}_2}$ to be $\mathcal{L}_{\mathcal{K}_2}$ and it cannot be 0, it implies that $t < 8$ if $[\mathcal{L}_{\mathcal{K}_2}] = 1$ and $t < 16$ if $[\mathcal{L}_{\mathcal{K}_2}] = 2$. Now note that here $\mathcal{K}_1$ has the same length as $\mathcal{V}_2$. So for the two strings to be

the same, we need the sum of the length of $\mathcal{K}_2, 0^{t'}$, and $\mathcal{L}_{K_2}$ must be equal to t , which must be strictly less than 16. This tells us that $\mathcal{L}_{K_2}$ cannot be a 2-bytes value.

So this gives us that $[\mathcal{L}_{K_2}] = 1$ and $t < 8$. Let $\mathbf{b}$ be the last $8 - t$ bits of $\mathcal{K}_1$ which must satisfy $\mathbf{b} \neq 0$. Then we have $\mathcal{L}_{K_2} = \mathbf{b}.2^t$. So equating the length here, we have that $\mathbf{a} + t = \mathbf{b}.2^t + \mathbf{a} + t' + 8$ or equivalently, $(t - 8) - \mathbf{b}.2^t = t - \mathbf{b}.2^t - 8 = t' \geq 0$. Now remember that $t - 8 < 0$ and $-\mathbf{b}.2^t < 0$. The sum of two negative numbers cannot be non-negative. So this case can be discarded.

- Case II.1.b. $\mathcal{L}_{V_2} > \mathbf{a}$. So $[\mathcal{L}_{V_2}] = 2$. Now if $t > 8$, we will have that the most significant byte of $\mathcal{L}_{V_2}$ is zero, which cannot be by the observation we made in the very beginning of this section. So we have $t < 8$. We again let $\mathbf{b}$ to be the last $8 - t$ bits of $\mathcal{K}_1$ and $\mathbf{b} \neq 0$. Then we have $\mathcal{L}_{V_2} = \mathbf{b}.2^{t+8} + \mathbf{a}$ and the remaining string to be $\mathcal{K}'_1$. Assuming $\mathcal{S}_2$ has t' -bits of zero padding, equating the strings, we have $\mathcal{K}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathcal{L}_{K_1}$. Recall that here $\mathcal{K}'_1$ must have length strictly less than 256 bits since $[\mathcal{L}_{K_1}] = 1$ and it has been truncated by $8 - t$ bits. However, by assumption, we must already have $\mathcal{V}_2$ itself to have length more than 256 bits. Obviously, this is impossible. So we can disregard this case.

II.2. Case II.2. $[\mathcal{L}_{K_1}] = 2, [\mathcal{L}_{V_1}] = 1$. By Observation 5.3.2, we have $\mathcal{L}_{V_1} = 0$. Let $0 \leq \mathbf{a}, \mathbf{b} \leq 255, \mathbf{a} \neq 0$ such that $\mathcal{L}_{K_1} = 256\mathbf{a} + \mathbf{b}$. Then we have

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel 0^t \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Here, considering $\mathcal{S}_2$, we again have two cases:

- Case II.2.a $\mathcal{L}_{V_2} = 256\mathbf{a} + \mathbf{b}$. Then equating the length of the two strings, we must have $\mathcal{K}_1 \parallel 0^t = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathcal{L}_{K_2}$ where t' is the length of zero padding of $\mathcal{S}_2$. Now since the length of $\mathcal{K}_1$ and $\mathcal{V}_2$ are

the same, we must have the sum of the lengths of $\mathcal{K}_2, 0^{t'}$, and $\mathcal{L}_{K_2}$ to be equal to t . Now note that by the same argument as before, t must be less than 8 if $[\mathcal{L}_{K_2}] = 1$ and $t < 16$ if $[\mathcal{L}_{K_2}] = 2$. Now if $\mathcal{L}_{K_2}$ is a 2-byte string, the length of the string in the right hand side must exceed 256 bits. However, the one on the left cannot even exceed 16. So we cannot have $\mathcal{L}_{K_2}$ to be a 2-byte string. So we have $[\mathcal{L}_{K_2}] = 1$. So we have $t < 8$. Furthermore, we have that $t = \mathcal{L}_{K_2} + t' + 8$ where the addition by 8 comes from the fact that $\mathcal{L}_{K_2}$ is an 8-bit string. However, this is impossible since all the terms here are non-negative and this causes the left hand side to be strictly less than 8 while the right hand side is at least 8. So we can again disregard this case.

- Case II.2.b. $\mathcal{L}_{V_2} = \mathbf{b}$. For this, we further can divide the case into two depending on $[\mathcal{L}_{K_2}]$. First of all, if $\mathcal{L}_{K_2}$ is a 1-byte value, then we have $\mathcal{L}_{K_2} = \mathbf{a}$. From here we see that there must be $255\mathbf{a} + \mathbf{b}$ bits that can be found in $\mathcal{K}_1$ but not in $\mathcal{K}_2$. Now since $\mathbf{a} \neq 0$, this means at least 255-bits of secret string that we need to guess. This implies that the success probability of this case is at most 2^{-255} which is already infeasible. So we need $[\mathcal{L}_{K_2}] = 2$. By the same observation as before, $t < 8$. Let $\mathcal{K}_1 = \mathcal{K}'_1 \parallel \mathbf{c}$ where $\mathbf{c}$ is the last $8 - t$ bits of $\mathcal{K}_1$. Then we have that $\mathcal{L}_{K_2} = \mathbf{c}.2^{8+t} + \mathbf{a}$. Now we let $\mathcal{S}_2$ have t' -bits of zero padding. Then we have $\mathcal{K}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. So equating the length, we have that $256\mathbf{a} + \mathbf{b} - (8 - t) = \mathbf{c}.2^{8+t} + \mathbf{a} + \mathbf{b} + t'$. Equivalently, we have that

$$255\mathbf{a} = \mathbf{c}.2^{8+t} + t' + 8 - t.$$

Note that this also tells us that $\mathcal{L}_{K_1} \geq \mathcal{L}_{K_2}$. As before, we define a family:

$$\mathcal{F}_{\mathbf{p},(2,1),(2,1)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') : 1 \leq \mathbf{a}, \mathbf{b} \leq 255, 1 \leq t < 8, 1 \leq \mathbf{c} \leq 2^{8-t} - 1, t' \geq 0, 255\mathbf{a} = \mathbf{c}.2^{8+t} + t' + 8 - t\}.$$

Now we consider the feasibility of each quadruple in the family.

Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') \in \mathcal{F}_{\mathbf{p},(2,1),(2,1)}$. Then the length of the bits in $\mathcal{K}_1$ but not in $\mathcal{K}_2$ is $256\mathbf{a} + \mathbf{b} - (\mathbf{c} \cdot 2^{8+t} + \mathbf{a}) = 255\mathbf{a} + \mathbf{b} - (\mathbf{c} \cdot 2^{8+t})$. Now a simple enumeration of the cases tells us that this value can be kept as low as 8. An example of such quintuple is $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') = (249, 1, 124, 1, 14)$. One more thing to note here is that since $\mathcal{K}_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'} \parallel \mathbf{c}$, we have that the last $t' + 8 - t$ bits of $\mathcal{K}_1$ is fixed to be $0^{t'} \parallel \mathbf{c}$. So here we have a restriction of the $\mathcal{K}_1$ that is vulnerable to the collision. The secret string must have its last $t' + 8 - t$ bits to be $0^{t'} \parallel \mathbf{c}$. For any integers $k \geq 8$, we define a subfamily $\mathcal{F}_{\mathbf{p},(2,1),(2,1),k}$ of $\mathcal{F}_{\mathbf{p},(2,1),(2,1)}$ satisfying:

$$\mathcal{F}_{\mathbf{p},(2,1),(2,1),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') \in \mathcal{F}_{\mathbf{p},(2,1),(2,1)} : 255\mathbf{a} + \mathbf{b} - (\mathbf{c} \cdot 2^{8+t}) \leq k\}.$$

Taking any quintuple of $\mathcal{F}_{\mathbf{p},(2,1),(2,1),k}$, we can use this to guess at most k -bits of key extension with success probability at least 2^{-k} .

II.3. Case II.3. $[\mathcal{L}_{K_1}] = 1, [\mathcal{L}_{V_1}] = 2$. By observation 5.3.2, we must have $\mathcal{L}_{V_1}$ to be divisible by 256. So we let $1 \leq \mathbf{a}, \mathbf{b} \leq 255$ such that $\mathcal{L}_{K_1} = \mathbf{a}$ and $\mathcal{L}_{V_1} = 256\mathbf{b}$. Then we have:

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^t \parallel \mathbf{a} \parallel \mathbf{b} \parallel [0]_2).$$

Now as before, we consider the number of bytes of $\mathcal{L}_{K_2}$ and $\mathcal{L}_{V_2}$.

- Case II.3.a. $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 1$. Then $\mathcal{L}_{K_2} = \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Now $\mathcal{L}_{K_1} = \mathcal{L}_{K_2}$ and they are the first $\mathbf{a}$ bits of the strings $\mathcal{S}_1 = \mathcal{S}_2$. So $\mathcal{K}_1 = \mathcal{K}_2$. This implies that $\mathcal{V}_2$ should be the first $\mathbf{b}$ bits of $\mathcal{V}_1$ and the next $255\mathbf{b}$ bits of $\mathcal{V}_1$ should be the zero padding so that it can be absorbed by the zero padding of $\mathcal{S}_2$.
- Case II.3.b. $[\mathcal{L}_{K_2}] = 2, [\mathcal{L}_{V_2}] = 1$. Using the same analysis as before, we have $t < 8$ and we can let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{c}$ where $\mathbf{c}$ is the last $8 - t$

bits of $\mathcal{V}_1$. Then we have $\mathcal{L}_{K_2} = \mathbf{c}.2^{8+t} + \mathbf{a}$ and $\mathcal{L}_{V_2} = \mathbf{b}$. Now this means that we have $\mathbf{c}.2^{8+t}$ bits that is in $\mathcal{K}_2$ but not in $\mathcal{K}_1$. Now since we have $t > 0$ and $\mathbf{c} \neq 0$, the success probability of the collision is at most 2^{-256} which is infeasible. So we can disregard this case.

- Case II.3.c. $[\mathcal{L}_{K_2}] = 1, [\mathcal{L}_{V_2}] = 2$. By the same analysis as before, we get that $t < 8$ and we can let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{c}$ where $\mathbf{c}$ is the last $8 - t$ bits of $\mathcal{V}_1$. So we have $\mathcal{L}_{V_2} = 256\mathbf{a} + \mathbf{b}$ and $\mathcal{L}_{K_2} = \mathbf{c}.2^t$. Assuming $\mathcal{S}_2$ to have t' bits of zero padding, we have $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. So we have $\mathbf{a} + 256\mathbf{b} - (8 - t) = \mathbf{c}.2^t + 256\mathbf{a} + \mathbf{b} + t'$. Simplifying this, we get:

$$255(\mathbf{b} - \mathbf{a}) = \mathbf{c}.2^t + 8 - t + t'.$$

So we can consider the family:

$$\mathcal{F}_{\mathbf{p},(1,2),(1,2)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') : 1 \leq \mathbf{a}, \mathbf{b} \leq 255, 0 < t < 8, 1 \leq \mathbf{c} \leq 2^{(8-t)} - 1, t' \geq 0,$$

$$255(\mathbf{b} - \mathbf{a}) = \mathbf{c}.2^t + 8 - t + t'\}.$$

Now we discuss the feasibility of each quintuple in this family. Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') \in \mathcal{F}_{\mathbf{p},(1,2),(1,2)}$. So there will be $|\mathbf{a} - \mathbf{c}.2^t|$ -bits of secret string that is in one but not in both of the secret keys. In other words, we will need to guess these bits. A simple calculation reveals that this value can be kept as low as 0. A really small example would be to set $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') = (2, 3, 1, 1, 260)$. For any non-negative integer k , we then can define a non-empty subfamily $\mathcal{F}_{\mathbf{p},(1,2),(1,2),k}$ such that:

$$\mathcal{F}_{\mathbf{p},(1,2),(1,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') \in \mathcal{F}_{\mathbf{p},(1,2),(1,2)} : |\mathbf{a} - \mathbf{c}.2^t| \leq k\}.$$

Having these subfamilies, we know that if we take a quintuple from $\mathcal{F}_{\mathbf{p},(1,2),(1,2),k}$, then it can help us in guessing the extension of at most k -bits of secret key with success probability at least 2^{-k} . Now we also still need to investigate the extension secret string. We again

equate the two strings to get $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. Now if $\mathcal{K}_2$ is the longer one, since the left term contains only $\mathcal{K}_1$ and $\mathcal{V}'_1$, the extension can only come from $\mathcal{V}'_1$. So if $\mathcal{K}_2$ is longer, we have no restriction on either $\mathcal{K}_1$ or $\mathcal{K}_2$. On the other hand, if $\mathcal{K}_1$ is the longer one, the same argument cannot be used since the right term contains $0^{t'}$. However, we again notice that $[\mathcal{L}_{K_1}] = [\mathcal{L}_{K_2}] = 1$. So $[\mathcal{L}_{K_1} - \mathcal{L}_{K_2}] = 1$. Now since $[\mathcal{L}_{V_2}] = 2$, certainly $\mathcal{L}_{K_1} - \mathcal{L}_{K_2} < \mathcal{L}_{V_2}$. So again, all the extension can only come from the controllable nonce.

So we proved that in either case, we have no restriction on the choice of the secret key. So any choice of key following the parameters in $\mathcal{F}_{p,(1,2),(1,2)}$ is vulnerable to the collision.

- Case II.3.d. $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 2$. Then we have that $\mathcal{L}_{V_2} = 256a + b$. Furthermore, by the same analysis as before, we have $t < 16$ and we can let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel c$ where c is the last $16 - t$ -bits of $\mathcal{V}_1$. We also have that if $t < 8$, we need $c \geq 2^{8-t}$ to ensure $\mathcal{L}_{K_2}$ to be a 2-bytes value. Then we have $\mathcal{L}_{K_2} = c \cdot 2^t$. Note that this gives us that $\mathcal{L}_{K_1} = a < 256 \leq c \cdot 2^t = \mathcal{L}_{K_2}$. So $\mathcal{L}_{K_1} < \mathcal{L}_{K_2}$. Now we have $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$ assuming $\mathcal{S}_2$ has t' -bits of zero padding. So equating the length of these two strings, we have $a + 256b - (16 - t) = c \cdot 2^t + 256a + b + t'$ or equivalently

$$255(b - a) = c \cdot 2^t + 16 - t + t'.$$

Now as usual, consider the family

$$\mathcal{F}_{p,(1,2),(2,2)} = \{(a, b, c, t, t') : 1 \leq a, b \leq 255, 1 \leq t \leq 15, \max(1, 2^{8-t}) \leq c \leq 2^{16-t},$$

$$t' \geq 0, 255(b - a) = c \cdot 2^t + 16 - t + t'\}.$$

Now we consider the feasibility of each quintuple. Let $(a, b, c, t, t') \in \mathcal{F}_{p,(1,2),(2,2)}$. Then we have $c \cdot 2^t - a$ bits that is in $\mathcal{K}_2$ that is not in $\mathcal{K}_1$

and hence needs to be guessed. A simple calculation tells us that this value can reach as low as 3, which is achievable, for instance, when $(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') = (253, 255, 128, 1, 269)$. So like before, we can define a subfamily of $\mathcal{F}_{\mathbf{p},(1,2),(2,2),k}$ for any integer $k \geq 3$:

$$\mathcal{F}_{\mathbf{p},(1,2),(2,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, t, t') \in \mathcal{F}_{\mathbf{p},(1,2),(2,2)}, \mathbf{c} \cdot 2^t - \mathbf{a} \leq k\}.$$

Then we have that for any quintuple taken from $\mathcal{F}_{\mathbf{p},(1,2),(2,2),k}$, the success probability of collision of the session key is at least 2^{-k} .

II.4. Case II.4. $[\mathcal{L}_{K_1}] = [\mathcal{L}_{V_1}] = 2$. Then we can have $0 \leq \mathbf{a}, \mathbf{b}, \mathbf{c} \leq 255, \mathbf{a}, \mathbf{c} \neq 0$ such that $\mathcal{L}_{K_1} = 256\mathbf{a} + \mathbf{b}$ and $\mathcal{L}_{V_1} = 256\mathbf{c}$. Here 256 divides $\mathcal{L}_{V_1}$ due to Observation 5.3.2. So we have:

$$\mathcal{S}_1 = (\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^t \parallel \mathbf{a} \parallel \mathbf{b} \parallel \mathbf{c} \parallel [0]_2).$$

As before, we further divide this case to 4 smaller subcases based on $[\mathcal{L}_{K_2}]$ and $[\mathcal{L}_{V_2}]$.

- Case II.4.a. $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 1$. Then we have $\mathcal{L}_{K_2} = \mathbf{b}$ and $\mathcal{L}_{V_2} = \mathbf{c}$. Now since $\mathbf{a} \neq 0$, if t' is the number of zero paddings in $\mathcal{S}_2$, we must have $t' < 8$. Then we have $\mathcal{K}_1 \parallel \mathcal{V}_1 \parallel 0^t \parallel \mathbf{a} = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. So equating the length, we have $256\mathbf{a} + \mathbf{b} + 256\mathbf{c} + t + 8 = \mathbf{b} + \mathbf{c} + t'$ or equivalently

$$256\mathbf{a} + 255\mathbf{c} + t = t' - 8 < 0.$$

Now note that $\mathbf{a}, \mathbf{c}, t > 0$. So the expression $256\mathbf{a} + 255\mathbf{c} + t$ must be a positive number. So the inequality cannot be satisfied for any value of the variables. Hence this case is impossible and can be disregarded.

- Case II.4.b. $[\mathcal{L}_{K_2}] = 2$ and $[\mathcal{L}_{V_2}] = 1$. Then we have $\mathcal{L}_{V_2} = \mathbf{c}$ and $\mathcal{L}_{K_2} = 256\mathbf{a} + \mathbf{b}$. As before, this implies that $\mathcal{K}_1 = \mathcal{K}_2$ and $\mathcal{V}_1 = \mathcal{V}_2 \parallel 0^{255\mathbf{c}}$.

- Case II.4.c. $[\mathcal{L}_{K_2}] = 1$ and $[\mathcal{L}_{V_2}] = 2$. Then we have $\mathcal{L}_{V_2} = 256\mathbf{b} + \mathbf{c}$ and $\mathcal{L}_{K_2} = \mathbf{a}$. So we have $255\mathbf{a} + \mathbf{b}$ bits of $\mathcal{K}_1$ that is not in $\mathcal{K}_2$ and hence needs to be guessed. Since $\mathbf{a} \neq 0$, this translates to a success probability of at most 2^{-255} which is infeasible. So we can again disregard this case.
- Case II.4.d. $[\mathcal{L}_{K_2}] = [\mathcal{L}_{V_2}] = 2$. Then we have $\mathcal{L}_{V_2} = 256\mathbf{b} + \mathbf{c}$ and by similar analysis as before, $t < 8$. Furthermore, if we let $\mathcal{V}_1 = \mathcal{V}'_1 \parallel \mathbf{d}$ where $\mathbf{d}$ is the last $8 - t$ -bits of $\mathcal{V}_1$, we have $\mathcal{L}_{K_2} = 2^{8+t}\mathbf{d} + \mathbf{a}$. Then we have $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$ where t' is the number of zero paddings in $\mathcal{S}_2$. So equating the length, we get $256\mathbf{a} + \mathbf{b} + 256\mathbf{c} - (8 - t) = \mathbf{d} \cdot 2^{8+t} + \mathbf{a} + 256\mathbf{b} + \mathbf{c} + t'$. Simplifying this, we have the equation:

$$255(\mathbf{a} - \mathbf{b} + \mathbf{c}) = \mathbf{d} \cdot 2^{8+t} + 8 - t + t'.$$

So we can define the family

$$\mathcal{F}_{\mathbf{p},(2,2),(2,2)} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t, t') : 1 \leq \mathbf{a}, \mathbf{b}, \mathbf{c} \leq 255, 1 \leq t \leq 7, 1 \leq \mathbf{d} \leq 2^{8-t}-1,$$

$$t' \geq 0, 255(\mathbf{a} - \mathbf{b} + \mathbf{c}) = \mathbf{d} \cdot 2^{8+t} + 8 - t + t'\}$$

Next we check the feasibility of each elements of this family. Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t, t') \in \mathcal{F}_{\mathbf{p},(2,2),(2,2)}$. Then we have $|256\mathbf{a} + \mathbf{b} - (\mathbf{d} \cdot 2^{8+t} + \mathbf{a})| = |255\mathbf{a} + \mathbf{b} - \mathbf{d} \cdot 2^{8+t}|$ secret bits that is in one secret key but not both. So this needs to be guessed. Now enumerating all the elements of the family, we see that this value can be as low as 0 which, for instance, can be realised by the element $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t, t') = (2, 2, 3, 1, 1, 260)$. Then as before, we can define a subfamily $\mathcal{F}_{\mathbf{p},(2,2),(2,2),k}$ for any non-negative integers such that

$$\mathcal{F}_{\mathbf{p},(2,2),(2,2),k} = \{(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, t, t') \in \mathcal{F}_{\mathbf{p},(2,2),(2,2)} : |255\mathbf{a} + \mathbf{b} - \mathbf{d} \cdot 2^{8+t}| \leq k\}.$$

Then taking any element from $\mathcal{F}_{\mathbf{p},(2,2),(2,2),k}$, the success probability of a collision is at least 2^{-k} . Lastly, we still need to investigate the

extension of the secret key. Recall that $\mathcal{K}_1 \parallel \mathcal{V}'_1 = \mathcal{K}_2 \parallel \mathcal{V}_2 \parallel 0^{t'}$. As we have discussed in Case I.4.d, if $\mathcal{K}_2$ is longer, no restriction is required for $\mathcal{K}_2$. However, if $\mathcal{K}_1$ is longer, we must have the last $\max(0, t' - \mathcal{L}_{V'_1})$ of $\mathcal{K}_1$ to be all zero. Now in here we have $\mathcal{L}_{V'_1} = \mathcal{L}_{V_1} - (8 - t)$. So this is the only difference of the restriction we have for this family from the restriction in case I.4.d.

Appendix B

The List of Possible v and v' for ZUC IV Collision

Index	v	v'	Δiv	Index	v	v'	Δiv	Index	v	v'	Δiv
1	0x3fff8000	0x40007fff	0xff	86	0x3ffd555	0x40002aaa	0x55	171	0x7ffa	0x5555	0xaa
2	0x3fff8101	0x40007efe	0xfd	87	0x3ffd656	0x400029a9	0x53	172	0x7ffbabab	0x5454	0xa8
3	0x3fff8202	0x40007dfd	0xfb	88	0x3ffd757	0x400028a8	0x51	173	0x7ffacac	0x5353	0xa6
4	0x3fff8303	0x40007cfc	0xf9	89	0x3ffd858	0x400027a7	0x4f	174	0x7ffadad	0x5252	0xa4
5	0x3fff8404	0x40007bfb	0xf7	90	0x3ffd959	0x400026a6	0x4d	175	0x7ffa	0x5151	0xa2
6	0x3fff8505	0x40007afa	0xf5	91	0x3ffdfa5a	0x400025a5	0x4b	176	0x7ffa	0x5050	0xa0
7	0x3fff8606	0x400079f9	0xf3	92	0x3ffddb5b	0x400024a4	0x49	177	0x7ffb0b0	0x4f4f	0x9e
8	0x3fff8707	0x400078f8	0xf1	93	0x3fffdc5c	0x400023a3	0x47	178	0x7ffb1b1	0x4e4e	0x9c
9	0x3fff8808	0x400077f7	0xef	94	0x3ffdd5d	0x400022a2	0x45	179	0x7ffb2b2	0x4d4d	0x9a
10	0x3fff8909	0x400076f6	0xed	95	0x3ffde5e	0x400021a1	0x43	180	0x7ffb3b3	0x4c4c	0x98
11	0x3fff8a0a	0x400075f5	0xeb	96	0x3ffdf5f	0x400020a0	0x41	181	0x7ffb4b4	0x4b4b	0x96
12	0x3fff8b0b	0x400074f4	0xe9	97	0x3fffe060	0x40001ff9	0x3f	182	0x7ffb5b5	0x4a4a	0x94
13	0x3fff8c0c	0x400073f3	0xe7	98	0x3fffe161	0x40001e9e	0x3d	183	0x7ffb6b6	0x4949	0x92
14	0x3fff8d0d	0x400072f2	0xe5	99	0x3fffe262	0x40001d9d	0x3b	184	0x7ffb7b7	0x4848	0x90
15	0x3fff8e0e	0x400071f1	0xe3	100	0x3fffe363	0x40001c9c	0x39	185	0x7ffb8b8	0x4747	0x8e
16	0x3fff8f0f	0x400070f0	0xe1	101	0x3fffe464	0x40001b9b	0x37	186	0x7ffb9b9	0x4646	0x8c
17	0x3fff9010	0x40006fef	0xdf	102	0x3fffe565	0x40001a9a	0x35	187	0x7ffbaba	0x4545	0x8a
18	0x3fff9111	0x40006eee	0xdd	103	0x3fffe666	0x40001999	0x33	188	0x7ffbbbb	0x4444	0x88
19	0x3fff9212	0x40006ded	0xdb	104	0x3fffe767	0x40001898	0x31	189	0x7ffbcbc	0x4343	0x86
20	0x3fff9313	0x40006cec	0xd9	105	0x3fffe868	0x40001797	0x2f	190	0x7ffbdbd	0x4242	0x84
21	0x3fff9414	0x40006beb	0xd7	106	0x3fffe969	0x40001696	0x2d	191	0x7ffbebe	0x4141	0x82
22	0x3fff9515	0x40006aea	0xd5	107	0x3fffea6a	0x40001595	0x2b	192	0x7fffbfbf	0x4040	0x80
23	0x3fff9616	0x400069e9	0xd3	108	0x3fffeb6b	0x40001494	0x29	193	0x7ffc0c0	0x3f3f	0x7e
24	0x3fff9717	0x400068e8	0xd1	109	0x3fffec6c	0x40001393	0x27	194	0x7ffc1c1	0x3e3e	0x7c
25	0x3fff9818	0x400067e7	0xcf	110	0x3fffed6d	0x40001292	0x25	195	0x7ffc2c2	0x3d3d	0x7a
26	0x3fff9919	0x400066e6	0xcd	111	0x3fffee6e	0x40001191	0x23	196	0x7ffc3c3	0x3c3c	0x78
27	0x3fff9a1a	0x400065e5	0xcb	112	0x3fffef6f	0x40001090	0x21	197	0x7ffc4c4	0x3b3b	0x76
28	0x3fff9b1b	0x400064e4	0xc9	113	0x3fffef070	0x40000f8f	0x1f	198	0x7ffc5c5	0x3a3a	0x74
29	0x3fff9c1c	0x400063e3	0xc7	114	0x3fffff171	0x40000e8e	0x1d	199	0x7ffc6c6	0x3939	0x72
30	0x3fff9d1d	0x400062e2	0xc5	115	0x3fffff272	0x40000d8d	0x1b	200	0x7ffc7c7	0x3838	0x70

31	0x3ffff9e1e	0x400061e1	0xc3	116	0x3ffff373	0x40000c8c	0x19	201	0x7fffc8c8	0x3737	0x6e
32	0x3ffff9f1f	0x400060e0	0xc1	117	0x3ffff474	0x40000b8b	0x17	202	0x7fffc9c9	0x3636	0x6c
33	0x3ffffa020	0x40005fdf	0xbf	118	0x3ffff575	0x40000a8a	0x15	203	0x7fffcaca	0x3535	0x6a
34	0x3ffffa121	0x40005ede	0xbd	119	0x3ffff676	0x40000989	0x13	204	0x7fffcbbcb	0x3434	0x68
35	0x3ffffa222	0x40005ddd	0xbb	120	0x3ffff777	0x40000888	0x11	205	0x7fffcccc	0x3333	0x66
36	0x3ffffa323	0x40005cdc	0xb9	121	0x3ffff878	0x40000787	0xf	206	0x7fffcddcd	0x3232	0x64
37	0x3ffffa424	0x40005bdb	0xb7	122	0x3ffff979	0x40000686	0xd	207	0x7fffccece	0x3131	0x62
38	0x3ffffa525	0x40005ada	0xb5	123	0x3ffffa7a	0x40000585	0xb	208	0x7fffcfcf	0x3030	0x60
39	0x3ffffa626	0x400059d9	0xb3	124	0x3ffffb7b	0x40000484	0x9	209	0x7fffd0d0	0x2f2f	0x5e
40	0x3ffffa727	0x400058d8	0xb1	125	0x3ffffc7c	0x40000383	0x7	210	0x7fffd1d1	0x2e2e	0x5c
41	0x3ffffa828	0x400057d7	0xaf	126	0x3ffffd7d	0x40000282	0x5	211	0x7fffd2d2	0x2d2d	0x5a
42	0x3ffffa929	0x400056d6	0xad	127	0x3ffffe7e	0x40000181	0x3	212	0x7fffd3d3	0x2c2c	0x58
43	0x3ffffaa2a	0x400055d5	0xab	128	0x3fffff7f	0x40000080	0x1	213	0x7fffd4d4	0x2b2b	0x56
44	0x3ffffab2b	0x400054d4	0xa9	129	0x7fff8080	0x7f7f	0xfe	214	0x7fffd5d5	0x2a2a	0x54
45	0x3ffffac2c	0x400053d3	0xa7	130	0x7fff8181	0x7e7e	0xfc	215	0x7fffd6d6	0x2929	0x52
46	0x3ffffad2d	0x400052d2	0xa5	131	0x7fff8282	0x7d7d	0xfa	216	0x7fffd7d7	0x2828	0x50
47	0x3ffffae2e	0x400051d1	0xa3	132	0x7fff8383	0x7c7c	0xf8	217	0x7fffd8d8	0x2727	0x4e
48	0x3ffffaf2f	0x400050d0	0xa1	133	0x7fff8484	0x7b7b	0xf6	218	0x7fffd9d9	0x2626	0x4c
49	0x3ffffb030	0x40004fcf	0x9f	134	0x7fff8585	0x7a7a	0xf4	219	0x7fffdada	0x2525	0x4a
50	0x3ffffb131	0x40004ece	0x9d	135	0x7fff8686	0x7979	0xf2	220	0x7fffdbdb	0x2424	0x48
51	0x3ffffb232	0x40004dcd	0x9b	136	0x7fff8787	0x7878	0xf0	221	0x7fffdcdc	0x2323	0x46
52	0x3ffffb333	0x40004ccc	0x99	137	0x7fff8888	0x7777	0xee	222	0x7fffdddd	0x2222	0x44
53	0x3ffffb434	0x40004bcb	0x97	138	0x7fff8989	0x7676	0xec	223	0x7fffdede	0x2121	0x42
54	0x3ffffb535	0x40004aca	0x95	139	0x7fff8a8a	0x7575	0xea	224	0x7fffdfdf	0x2020	0x40
55	0x3ffffb636	0x400049c9	0x93	140	0x7fff8b8b	0x7474	0xe8	225	0x7fffe0e0	0x1f1f	0x3e
56	0x3ffffb737	0x400048c8	0x91	141	0x7fff8c8c	0x7373	0xe6	226	0x7fffe1e1	0x1e1e	0x3c
57	0x3ffffb838	0x400047c7	0x8f	142	0x7fff8d8d	0x7272	0xe4	227	0x7fffe2e2	0x1d1d	0x3a
58	0x3ffffb939	0x400046c6	0x8d	143	0x7fff8e8e	0x7171	0xe2	228	0x7fffe3e3	0x1c1c	0x38
59	0x3ffffba3a	0x400045c5	0x8b	144	0x7fff8f8f	0x7070	0xe0	229	0x7fffe4e4	0x1b1b	0x36
60	0x3ffffbb3b	0x400044c4	0x89	145	0x7fff9090	0x6f6f	0xde	230	0x7fffe5e5	0x1a1a	0x34
61	0x3ffffbc3c	0x400043c3	0x87	146	0x7fff9191	0x6e6e	0xdc	231	0x7fffe6e6	0x1919	0x32
62	0x3ffffbd3d	0x400042c2	0x85	147	0x7fff9292	0x6d6d	0xda	232	0x7fffe7e7	0x1818	0x30
63	0x3ffffbe3e	0x400041c1	0x83	148	0x7fff9393	0x6c6c	0xd8	233	0x7fffe8e8	0x1717	0x2e
64	0x3ffffbf3f	0x400040c0	0x81	149	0x7fff9494	0x6b6b	0xd6	234	0x7fffe9e9	0x1616	0x2c
65	0x3fffc040	0x40003fbf	0x7f	150	0x7fff9595	0x6a6a	0xd4	235	0x7fffeaeae	0x1515	0x2a
66	0x3fffc141	0x40003ebe	0x7d	151	0x7fff9696	0x6969	0xd2	236	0x7fffebebe	0x1414	0x28
67	0x3fffc242	0x40003dbd	0x7b	152	0x7fff9797	0x6868	0xd0	237	0x7fffeccec	0x1313	0x26
68	0x3fffc343	0x40003cbc	0x79	153	0x7fff9898	0x6767	0xce	238	0x7fffedede	0x1212	0x24
69	0x3fffc444	0x40003bbb	0x77	154	0x7fff9999	0x6666	0xcc	239	0x7fffeeee	0x1111	0x22
70	0x3fffc545	0x40003aba	0x75	155	0x7fff9a9a	0x6565	0xca	240	0x7fffefef	0x1010	0x20
71	0x3fffc646	0x400039b9	0x73	156	0x7fff9b9b	0x6464	0xc8	241	0x7fffef0f0	0xf0f	0x1e
72	0x3fffc747	0x400038b8	0x71	157	0x7fff9c9c	0x6363	0xc6	242	0x7fffef1f1	0xe0e	0x1c
73	0x3fffc848	0x400037b7	0x6f	158	0x7fff9d9d	0x6262	0xc4	243	0x7fffef2f2	0xd0d	0x1a
74	0x3fffc949	0x400036b6	0x6d	159	0x7fff9e9e	0x6161	0xc2	244	0x7fffef3f3	0xc0c	0x18
75	0x3fffc9a4a	0x400035b5	0x6b	160	0x7fff9f9f	0x6060	0xc0	245	0x7fffef4f4	0xb0b	0x16
76	0x3fffc9b4b	0x400034b4	0x69	161	0x7fffa0a0	0x5f5f	0xbe	246	0x7fffef5f5	0xa0a	0x14
77	0x3fffc9c4c	0x400033b3	0x67	162	0x7fffa1a1	0x5e5e	0xbc	247	0x7fffef6f6	0x909	0x12
78	0x3fffc9d4d	0x400032b2	0x65	163	0x7fffa2a2	0x5d5d	0xba	248	0x7fffef7f7	0x808	0x10
79	0x3fffc9e4e	0x400031b1	0x63	164	0x7fffa3a3	0x5c5c	0xb8	249	0x7fffef8f8	0x707	0xe
80	0x3fffc9f4f	0x400030b0	0x61	165	0x7fffa4a4	0x5b5b	0xb6	250	0x7fffef9f9	0x606	0xc
81	0x3fffd050	0x40002faf	0x5f	166	0x7fffa5a5	0x5a5a	0xb4	251	0x7fffefa5a	0x505	0xa
82	0x3fffd151	0x40002eae	0x5d	167	0x7fffa6a6	0x5959	0xb2	252	0x7fffefbfb	0x404	0x8
83	0x3fffd252	0x40002dad	0x5b	168	0x7fffa7a7	0x5858	0xb0	253	0x7fffefcf	0x303	0x6
84	0x3fffd353	0x40002cac	0x59	169	0x7fffa8a8	0x5757	0xae	254	0x7fffefd4d	0x202	0x4
85	0x3fffd454	0x40002bab	0x57	170	0x7fffa9a9	0x5656	0xac	255	0x7fffefe5e	0x101	0x2

Appendix C

Procedure for Generating Identical Keystreams for Δiv_1 to attack ZUC

Here we describe the details of the procedure that is used to generate identical keystreams for the IV difference at iv_1 for ZUC:

1. Initialize $iv_0, iv_1, \dots, iv_{15}$ with 0. Set $iv_{13} = 64$.
2. Denote $(iv_4 + 8iv_{13} + 16iv_{10})$ as sum_1 and guess sum_1 with 1 of the 6376 possible values.
3. Guess $iv_2[1, 2]$, and compute v , until the condition $v[1..7] - (v \ll 8)[1..7] \leq 1$ is satisfied. If not possible, go to (2) .
4. Guess iv_7 and iv_{11} , and compute u , until $u[24..31] = 0\text{xff}$ is satisfied. We store the intermediate state s_{16} . If not possible, go to (3).
5. Guess iv_{15} and re-compute u , until $u[1..7] = u[9..15]$ and $u[8] = 0$ are satisfied. If not possible, go to (4).
6. Now we compare the current s_{16} with stored s_{16} to capture the change. By properly changing iv_2 and iv_{13} (this is the reason iv_{13} is initialized as 64), we

can always change the current s_{16} back to the saved value. Hence, $u[24..31]$ will remain.

7. Determine iv_1 as follows:

- If $v[8] \neq v[16]$, then if $u[1..16] < v[1..16]$ is satisfied, $iv_1 = 256 + u[1..16] - v[1..16]$ and update v , otherwise, go to (5).
- If $v[8] = v[16]$, then if $u[1..16] \geq v[1..16]$ is satisfied, $iv_1 = u[1..16] - v[1..16]$ and update v , otherwise, go to (5).

8. Guess iv_0, iv_5 and iv_{14} , compute v , until $v[16..31] = 0xffff$. If not possible, go to (5).

9. If $(u \oplus v)[1] = 1$, let $iv_2 = iv_2 \oplus 2$. Choose iv_3 properly to ensure $u[16..23] = 0xfff$. Check if we indeed have $v = u$, then output $iv_0, iv_1, \dots, iv_{15}$. Otherwise, go to (8).

In this algorithm, we restrict the forms of v and u to those starting with $0x7fff$ to reduce the search space.

Appendix D

Values of Leaked-Bytes in the ALE attack

The values of leaked bytes for the differential characteristic used in the basic LSFA in Sect. 4.3 are given in Table D.1. The index is the byte position in the keystream block. δ_{in} and δ_{out} are the input and output differences for the S-box. α and β can be arbitrary values extracted from the leaked bytes in Round 3. From the table, the total number of possible values at the active leaked bytes in first two rounds is $2 \times 2 \times 2 \times 2 \times 4 \times 2 = 128$.

The values of leaked bytes for the differential characteristic used in the optimized LSFA in Sect. 4.4.2 are given in Table D.2. The total number of possible values at the active leaked bytes in first two rounds is 2^8 .

Table D.1: Possible values of leaked bytes in hexadecimal for the basic LSFA. “-” indicates no difference. “★” indicates arbitrary values. α and β are values from the leaked bytes.

Index	δ_{in}	δ_{out}	Value(s)
0 – 1	-	-	★
2	E	42	11 or 1F
3	F3	C6	F, FC
4	59	FC	23, 7A
5	37	E5	19, 2E
6	6E	FC	0, 6E, 8C, E2
7	B2	E5	46, F4
8	-	-	★
9	81	$S(\alpha) \oplus S(81 \oplus \alpha)$	α
10	6C	$S(\beta) \oplus S(6C \oplus \beta)$	β
11 – 15	-	-	★

APPENDIX D. VALUES OF LEAKED-BYTES IN THE ALE ATTACK

Table D.2: Possible values of leaked bytes in hexadecimal for the optimized LSFA in Sect. 4.4.2. “-” indicates no difference. “★” indicates arbitrary values. α and β are values from the leaked bytes.

Index	δ_{in}	δ_{out}	Value(s)
0	49	84	1D or 54
1	CE	97	33, FD
2	87	35	44, C3
3	92	13	5E, CC
4	74	89	10, 64
5	57	73	B0, E7
6	A6	23	6D, CB
7	3A	13	08, 32
8 – 9	-	-	★
10	3D	$S(\alpha) \oplus S(3D \oplus \alpha)$	α
11	EE	$S(\beta) \oplus S(EE \oplus \beta)$	β
12 – 15	-	-	★

Appendix E

Experiments in the ALE Attack

E.1 Details of one Forgery in the “2–8–12–4” Experiment

The initial state is: $0x7745fe4fa948da9$.

APPENDIX E. EXPERIMENTS IN THE ALE ATTACK

Figure E.1: Differential Path of type “2–8–12–4”. The hexadecimal numbers indicate the difference values. The empty squares indicate there is no difference. The squares of leaked bytes are marked with gray color.

Table E.1: The values of round keys in the “2–8–12–4” case.

	Block 1	Block 2
Round 1	<i>0x27de69bc8bbc6a71</i>	<i>0xb9cacf23fb387dd8</i>
Round 2	<i>0x0eda00f69a70d28f</i>	<i>0xe9d293e0d9550016</i>
Round 3	<i>0xcaa2cab4fb3cf8a8</i>	<i>0x7537baeca8ed970e</i>
Round 4	<i>0x8034f88c57ed2766</i>	<i>0xe1c9150ac5564aad</i>

Table E.2: The forgery attack on the “2–8–12–4” differential characteristic.

	Block 1	Block 2
Plaintext	<i>0x37dc069161450099</i>	<i>0xb1469433d739a810</i>
Original ciphertext	<i>0x6c2b36071e45d85d</i>	<i>0x39d7ac987dd694a8</i>
Forged ciphertext	<i>0x6cbb36071e35d85d</i>	<i>0x53ba102c0d1b4435</i>
Colliding state	<i>0xb23d4f8eeb91a13e</i>	

E.2 Details of one Forgery in the “6–4–6–9” Experiment

The initial state is: $0x92304e6d9b7c7373$.

Figure E.2: Differential Path of type “6–4–6–9”. The hexadecimal numbers indicate the difference values. The empty squares indicate there is no difference. The squares of leaked bytes are marked with gray color.

Table E.3: The values of round keys in the “6–4–6–9” case.

	Block 1	Block 2
Round 1	$0x60ee23ea2d7054dd$	$0x36a5467dc8ebe9d2$
Round 2	$0xcf849ed86e6774c0$	$0xbe9da2b83ae39382$
Round 3	$0x569d49934b68af00$	$0x724461aa61be86e2$
Round 4	$0x64b01cb5561255c8$	$0xa396ceccaa9d57f6$

Table E.4: The forgery attack on the “6–4–6–9” differential characteristic.

	Block 1	Block 2
Plaintext	<i>0x182841a869f5e890</i>	<i>0x35bdb2a519a0818f</i>
Original ciphertext	<i>0x7bb0dce1e61d0d43</i>	<i>0xa3398abfcd7fcd1d</i>
Forged ciphertext	<i>0x0bc0d7e8361d0d41</i>	<i>0x646cac5a462f92a8</i>
Colliding state	<i>0xf134343fa5b20472</i>	

Appendix F

The specification of SIMON family of block ciphers

SIMON is a family of lightweight blocks cipher published by NSA in 2013. It specifies 10 block ciphers with block sizes 32, 48, 64, 96, 128 bits and key sizes 64, 72, 96, 128, 144, 192, 256 bits in the SIMON family. SIMON uses the Feistel network with simple round operations which are efficient in both hardware and software. Hence the state is denoted as Simon_{2n} for n -bit word. And $\text{SIMON}_{2n/mn}$ refers to $2n$ -bit internal state with mn -bit key. Note that the key size is always a multiple of the word size n and the value of m can be 2, 3, or 4. In this specification, we only describe $\text{SIMON}_{64/96}$, $\text{SIMON}_{96/96}$ and $\text{SIMON}_{128/128}$ which are used to instantiate JAMBU.

Round function:

SIMON_{2n} encryption makes use of the following n -bit operations:

1. bitwise XOR, $\oplus$,
2. bitwise AND, $\&$, and
3. left circular shift, S^j by j bits.

APPENDIX F. THE SPECIFICATION OF SIMON FAMILY OF BLOCK CIPHERS

For round key $k \in GF(2)^n$, the round function is the two-stage Feistel map $R_k : GF(2)^n \times GF(2)^n \rightarrow GF(2)^n \times GF(2)^n$ define by:

$$R_k(x, y) = (y \oplus f(x) \oplus k, x),$$

where $f(x) = (Sx \& S^8x) \oplus S^2x$.

Number of rounds:

For SIMON64/96, the number of rounds is 42.

For SIMON96/96, the number of rounds is 52.

For SIMON128/128, the number of rounds is 68.

Key schedule:

The SIMON key schedules take a key and from it generate a sequence of T key words $k_0, \dots, k_{T-1}$, where T is the number of rounds. The round key generation function depends on the value m for key size mn . The round key (only the cases used in this specification are provided here) can be computed as follows:

- For $k_0, \dots, k_{m-1}$, the original m key words are used.
- For k_{i+m} ($0 \leq i < T - m$),

$$k_{i+m} = c \oplus (z_2)_i \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+1}, \text{ if } m = 2,$$

$$k_{i+m} = c \oplus (z_2)_i \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+2}, \text{ if } m = 3,$$

where $c = 2^n - 4$ is a constant, I is the identity and $z_2(j)$ is the j^{th} bit of constant

$$z_2 = 10101111011100000011010010011000101000010001111110010110110011.$$

List of Publications

- [1] Ivan Tjuawinata, Tao Huang, and Hongjun Wu. Cryptanalysis of the Authenticated Encryption Algorithm COFFE. In *Selected Areas in Cryptography – SAC 2015*, to appear, 2015.
- [2] Tao Huang, Ivan Tjuawinata, and Hongjun Wu. Differential-Linear Cryptanalysis of ICEPOLE. In Gregor Leander, editor, *Fast Software Encryption – FSE 2015*, volume 9054 of *Lecture Notes in Computer Science*, pages 243–263. Springer Berlin Heidelberg, 2015.
- [3] Hongjun Wu and Tao Huang. JAMBU Lightweight Authenticated Encryption Mode and AES-JAMBU (v1). Submission to CAESAR. Available from: <http://competitions.cr.yp.to/caesar-submissions.html>, 2014.
- [4] Hongjun Wu and Tao Huang. The Authenticated Cipher MORUS. Submission to CAESAR. Available from <http://competitions.cr.yp.to/round1/morusv1.pdf>, 2014.
- [5] Shengbao Wu, Hongjun Wu, Tao Huang, Mingsheng Wang, and Wenling Wu. Leaked-State-Forgery Attack against the Authenticated Encryption Algorithm ALE. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 377–404. Springer Berlin Heidelberg, 2013.

- [6] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. Differential Attacks Against Stream Cipher ZUC. In *Advances in Cryptology – ASIACRYPT 2012*, pages 262–277. Springer, 2012.

My contributions

I played an active role in finding the results and writing the papers in [2-6], while [1] is mainly done by Ivan Tjuawinata.