Constructions and Analyses of Efficient Symmetric-Key Primitives for Authentication and Encryption

A thesis submitted to the Indian Statistical Institute
in partial fulfillment of the thesis requirements for the degree of
Doctor of Philosophy in Computer Science

Author:
Sebati Ghosh

Supervisor:
Prof. Palash Sarkar

Applied Statistics Unit
Indian Statistical Institute
203, B. T. Road,
Kolkata, India - 700108.

August, 2021.

To Baba, Maa
&
Sir
(Late Jayanta Roy)

Acknowledgements

I believe writing a PhD thesis is not only a journey of five or six years, but the preparation preceding this journey starts much earlier. Innumerable people contribute to this preparation and the journey itself in some way or the other. There is a saying - each person you meet in life either leaves a memory or a lesson. At this moment I really want to thank each and everyone of them (though it is difficult to mention everyone) for enriching me.

There are a few people to whose contribution towards this thesis I can never do justice by using any amount of words. At the very outset I must mention the name of my thesis supervisor Prof. Palash Sarkar. It is needless to say that this thesis could never happen without his tremendous inspiration and continuous encouragement; but I must mention also that this could have never started had I not been inspired to go for a PhD by seeing him. On the very first day when I met him as one of his PhD students, he told me something which immediately made it clear to me how lucky his students were. He told "no one works *under* me, rather they work *with* me." In the subsequent five and a half years, I have actually felt the veracity of those words. He has been not a supervisor to me, but a true "guide". None of his qualities as a PhD supervisor needs to be mentioned by me. Those are already well established. All I can say is that I consider myself as one of the exceptionally luckiest persons to have such an out-and-out honest and pure human being as my thesis supervisor. He has provided me the perfect balance of independence and guidance throughout. Additionally, I take this opportunity to thank him for supervising my M.Tech dissertation as well.

I am deeply indebted to Prof. Debrup Chakraborty who has been my co-author more than once. He has always been happily available for any academic discussion or guidance even for the works where he was not a co-author. He has played an important role in my training in Intel Intrinsics programming.

I would like to thank my another co-author Cuauhtemoc Mancillas López with whom I had several interesting academic discussions. He has been really nice each time I needed any assistance.

I would like to extend my gratitude to all the present and former members, I know, of the Cryptology Research Group in ISI, Kolkata. I thank Prof. Rana Barua, Prof. Bimal Kumar Roy, Prof. Kishan Chand Gupta, Prof. Subhamoy Maitra, Prof. Mridul Nandi, Prof. Sushmita Ruj, Prof. Anisur Molla and Prof. Goutam Paul for their constant support and encouragement. I gladly remember the several nice and motivating discussions I had with Prof. Arijit Chaudhuri, Prof. Swapan Parui, Prof. Tapas Samanta, Prof. Anup Dewanji, Prof. Mandar Mitra, Prof. Sarbani Palit and Prof. Mrinal Nandi.

The excellent time spent at our very own Turing Lab will always occupy a special place in my heart. Whenever life has been tough on me, I have taken solace at my little corner in the Turing Lab.

I fondly remember all of my colleagues, seniors and juniors who have made this journey special. I thank all of them - Abhinandan, Amit (Jana), Amit-da (Sharma), Anwesha, Anindita-di, Aniruddha, Ashwin, Atanu-da, Avijit, Avik-da, Avishek, Binanda-da, Biswajit, Butu-da, Diptendu, Indranil-da, Jyotirmoy, Karati-da (though he is a faculty in ISI now, I

like to thank him as a senior and a friend), Kaushik-da (Majumder), Kaushik-da (Nath), Laltu, Madhurima, Mostaf, Nayana, Nilanjan-da, Nishant, Prabal, Pritam, Rahul, Ritam, Samir, Sanjay-da, Shashank-da, Shion, Somindu-di, Sreyosi, Srimanta-da, Subhadip, Subhra, Suchana, Suchismita-di, Sumit-da, Suprita and Tapas-da.

I would like to thank Prof. Debrup Chakraborty, Prof. Mridul Nandi and Avijit for also providing comments on some of our works. I thank all the reviewers of our works for providing helpful comments and feedback. Thanks to Sanjay-da, Nilanjan-da, Shashank-da, Shion, Binanda-da, Indranil-da, Karati-da and Sumit-da for their academic suggestions and help in different technical matters also. I am thankful for all the academic and non-academic help, encouragement and fun moments that I have been gifted with in several occasions from many of my colleagues.

Both the reviewers of this thesis have provided kind comments and feedback, which helped in improving the work further. I would like to thank both of them.

Thanks to all the technical staff and office staff of ASU for being immensely helpful in the hour of need. Also the technical staff of CSSC laboratory deserve gratitude for providing all the necessary facilities.

In my personal circle, I will start with the persons who are not there in the physical world for almost a decade and a half now, but their memories have given me the exceptional strength and motivation whenever I needed them. I cannot explain the influence Sir (Late Jayanta Roy) has on my life and career. He is the person who for the first time taught me to aim something extra-ordinary. The immense belief he had in my abilities has always ignited me to push myself beyond my comfort zone. My another hidden source of strength has been my Grandmother - Thamma (Late Parul Rani Ghosh). The immense struggle (along with my father) she had bravely gone through throughout her life has probably been genetically transferred to me to a small extent. I believe that has really helped me to come out of the darkest phases of my life so far. Today these two persons would really have been immensely happy and proud of me.

In the beginning of my career I could come out of a corporate job and return to academics for the support and encouragement given by Maharaj (Swami Ritananda Maharaj of Ramakrishna Mission Order), Swami Sarvapriyananda Maharaj (of the same order), Jyathima (Ms. Leena Bhattacharya), Indranilda and Avik. I would like to thank Swami Sarvapriyananda Maharaj for understanding me and encouraging me to pursue higher studies. I consider myself to be very much fortunate for receiving unconditional affection, support and encouragement from Swami Ritananda Maharaj and Jyathima (and her family members). They have pulled me out of many of my low phases both during PhD and otherwise. I cannot thank them enough. Indranilda has played an instrumental role behind my apathy to a corporate job and instead pursuing something more "meaningful". Thanks to him (along with some of his family members and Sunipa Aunty) for guiding me in several other occasions. Avik has forced me to ultimately give up on my most "comfortable" dream of being a full-timer in home-making and to stand on my own feet. Though I hated that initially, today I must thank him. I would like to thank my elder Sister Sonali Ghosh Chakraborty for teaching me to have patience and considerate in the toughest of times.

I would like to thank Amit uncle (Dr. Chakraborty) and Dr. Uncle (Dr. Bandyopadhyay) for the affection and support they have given me. Thanks to Dr. Amit Ranjan Biswas for the

kindness he has shown, though till now we have never met in person. I thank Sreemoyee-di (Dr. Tarafder) for all the support and guidance she has given and for being so nice to me. Thanks to Master Ruma Roy Chowdhury (the Taekwondo trainer in ISI, Kolkata) for being so friendly and supportive. I thank my Dance teachers and all the members of the Dance class for renewing my energy each time I have met them.

Aditi, Arkadeep, Arnab, Arunima, Bhavana, Bhuvana, Bibek-da, Debarati, Diptyajit, Namrata, Nandini, Samikshan, Shibsankar, Sreedhar and some of my colleagues in ISI are people who have been there during many ups and downs in the journey. Thanks to all of them for bringing me out of my several low and dark phases. Among them, during these years, Nandini has voluntarily shifted from the position of a friend to a "parent" of a stubborn "child" like me. I could possibly never start writing this thesis ultimately if I had not been so much irritated with her nagging day in and day out. Really I cannot thank her enough for all of her contributions!

There is a bunch of little ones whose contribution in boosting me up is anything but little. Leju and her entire gang have been truly instrumental not only in my PhD period, but in many difficult phases of my life. Their soft touches have done magic each time I needed them. Mithi (my Taekwondo partner!), Rik (my nephew Aarush Chakraborty), Ritavori and Riddhyan (Indranilda's kids) deserve big thanks for just being there.

As I told in the beginning, I can do no justice to the contribution of a very few people towards this thesis by writing down any amount of words. The other two such persons are undoubtedly my Parents - Baba and Maa. Their immense contribution towards my life can never be expressed in words. I could not have been even one percent of what I am today if they were not there with me continuously. My Mother, Rubykana Ghosh, has always prioritised my well-being, my studies, my career even over her own physical health. Till now, when I feel the most vulnerable, Maa is the person who I ultimately run to for seeking the comfort. I can never thank her enough for all that she has done for me till now. I want to conclude this acknowledgement thanking my Father, Samir Kumar Ghosh, not because he has the least contribution towards this thesis, rather the exact opposite. Ultimately he is the person who has the most long lasting impact on my entire life so far. Besides being my childhood hero and the all-time role model, Baba is the person who had for the first time shown me the fun inside Mathematics. We have actually spent a lot of quality time over Mathematics. Besides that, I can not really list down all the efforts and tremendous sacrifices he has made so far just for my sake. The selfless care he has always taken to make me an independent woman is unparalleled. I will always admire the humane qualities inside him and that is without any bias. No amount of love and gratefulness will be enough for these two poles of my life, Baba and Maa.

List of Tables

4.1 Timing results for BRW128 and POLYVAL. 40
4.2 Timing results for BRW256. 40

5.1 Summary of the features of the basic scheme, Horner and BRW for hashing $\eta\ell$ blocks with $\eta = 2^{\mathfrak{a}+1} - 1$ for some $\mathfrak{a} \geq 1$. 45
5.2 Computation of vecHash2L. 48
5.3 Efficiency and AU bound for $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ over $\mathbb{F}_{2^n}$ with $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$. 52
5.4 Cycles per byte for computing Hash2L, GHASH and POLYVAL on Haswell. For both $n = 128$ and $n = 256$, Karatsuba gave better performance compared to the schoolbook method. 55
5.5 Cycles per byte for computing Hash2L, GHASH and POLYVAL on Skylake. For $n = 128$, schoolbook was faster than Karatsuba, while for $n = 256$, Karatsuba was faster. 55

6.1 For the schemes in (6.5) to (6.12), a summary of whether the input and/or the key of F and/or Hash depend on the tag length λ. 69
6.2 A secure and efficient nvMAC scheme from a random function. 76
6.3 A secure and efficient nvMAC scheme using a stream cipher supporting an initialisation vector. 85
6.4 A secure and efficient nvMAC scheme using a short output length PRF. . . 88

7.1 Encryption and decryption algorithms for FAST. 97
7.2 A two-round Feistel construction required in Table 7.1. 98
7.3 Computations of vecHorner and vecHash2L. The string 1^n denotes the element of $\mathbb{F}_{2^n}$ whose binary representation consists of the all-one string. Here η is a positive integer ≥ 3 and $d(\eta)$ denote the degree of $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$, where $m_1, \ldots, m_\eta \in \mathbb{F}_{2^n}$. 100
7.4 Game G_{real}. 111
7.5 Game G_{int}. 112
7.6 Game G_{rnd} 114
7.7 Comparison of different tweakable enciphering schemes according to computational efficiency. [BC] denotes the number of block cipher calls; [M] denotes the number of field multiplications; [D] denotes the number of doubling ('multiplication by x') operations; 123
7.8 Comparison of different tweakable enciphering schemes according to practical and implementation simplicity. [BCK] denotes the number of block cipher keys; and [HK] denotes the number of blocks in the hash key. 125
7.9 Comparison of the cycles per byte measure of FAST with those of XCB, EME2 and AEZ in the setting of Fx_{256}. 132
7.10 Report of cycles per byte measure for the setting of Gn for FAST[Gn, $\mathfrak{k}$, vecHorner] and FAST[Gn, $\mathfrak{k}$, 31, vecHash2L]. 133

List of Figures

5.1 The 31-block BRW tree. 53

7.1 The hash functions **H** and **G**. 98

8.1 Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus \chi_0\tau \oplus \chi_1\tau||m||m$ with tweak **e** under XCB. 141
8.2 Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus \chi_0\tau \oplus \chi_1\tau||m||m$ under TET. 143
8.3 Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus s||m||m$ under FAST. . 146
8.4 The hash functions **H** (left) and $\mathbf{G}'$ (right). 147
8.5 Enciphering a 3-block message $m||\chi_0||x$ or $m||\chi_1||x \oplus s$ under CMC. Correspondingly, $M = 2(P_1 \oplus P_3)$ and $M' = 2(P'_1 \oplus P'_3)$. 150
8.6 Enciphering a 3-block message $m||x||x$ or $m||x \oplus 6L||x \oplus 6L$ under EME. Correspondingly, $M = MQ \oplus E_K(MQ)$, where $MQ = E_K(m \oplus L) \oplus E_K(x \oplus 2L) \oplus E_K(x \oplus 4L) \oplus T$ and $M' = MQ' \oplus E_K(MQ')$ where $MQ' = E_K(m \oplus L) \oplus E_K(x \oplus 4L) \oplus E_K(x \oplus 2L) \oplus T$. 153

Contents

1 Introduction **1**
1.1 Overview of the Thesis . . . 4

2 A Brief Survey of the Literature **8**
2.1 Universal Hash Functions . . . 8
2.1.1 Efficiency Issues . . . 10
2.1.2 Collision and Differential Probabilities . . . 11
2.2 Message Authentication Code . . . 11
2.3 Tweakable Enciphering Scheme . . . 12
2.4 Post Quantum Cryptography . . . 14

3 Preliminaries and Background **15**
3.1 Universal Hashing . . . 16
3.1.1 Polynomial Hashing . . . 16
3.1.2 BRW Hashing . . . 17
3.2 Adversarial Model . . . 17
3.3 Pseudo-Random Function . . . 18
3.3.1 Counter Mode . . . 18
3.4 Message Authentication Code . . . 19
3.5 Tweakable Enciphering Scheme . . . 19

4 Evaluating Bernstein-Rabin-Winograd Polynomials **21**
4.1 Preliminaries . . . 21
4.2 Algorithm . . . 23
4.3 Correctness and Complexity . . . 29
4.3.1 Structural Properties of unreducedBRW . . . 29
4.3.2 Correctness of EvalBRW . . . 31
4.3.3 Complexity of EvalBRW . . . 34
4.4 Design of Hash Function . . . 36
4.5 Implementation . . . 38
4.5.1 Timings . . . 39
4.6 Summary . . . 40

5 Hash2L: A Fast Two-Level Universal Hash Function **41**
5.1 Combining BRW with Horner . . . 43
5.2 Two-Level Hash Function . . . 46
5.2.1 Hashing a Vector of Strings . . . 47
5.3 Implementations Based on `pclmulqdq` . . . 49
5.3.1 Field Multiplication . . . 50
5.3.2 Efficient Reduction . . . 50
5.3.3 Arithmetic Operations for Computing BRW . . . 52
5.3.4 Computing BRW Polynomials . . . 53

5.3.5 Decimated Horner . . . 54
5.3.6 Implementation of Hash2L . . . 54
5.4 Implementation Strategy Without Using pclmulqdq . . . 56
5.5 Message Authentication Code . . . 58
5.6 Comparison to Some Previous Works . . . 58
5.6.1 Comparison to Schemes Using Long Hash Keys . . . 58
5.6.2 Comparison to Schemes Using Short Hash Keys . . . 60
5.7 Summary . . . 60

6 Variants of Wegman-Carter Message Authentication Code Supporting Variable Tag Lengths 62
6.1 Definitions . . . 64
6.1.1 Variable Tag Length Nonce-Based Message Authentication Code . . . 64
6.2 Towards Building a Variable Tag Length MAC . . . 67
6.3 Secure and Efficient MAC Schemes with Variable Length Tag . . . 76
6.3.1 Reducing Key Size . . . 84
6.4 Summary . . . 90

7 FAST: Disk Encryption and Beyond 91
7.1 Preliminaries . . . 94
7.2 Construction . . . 95
7.3 Instantiations of FAST . . . 97
7.3.1 Hash Functions . . . 98
7.3.2 Specific Instantiations . . . 101
7.4 Security . . . 107
7.4.1 Pseudo-Random Function . . . 107
7.4.2 Tweakable Enciphering Scheme . . . 107
7.4.3 Security of FAST . . . 108
7.5 Comparison . . . 123
7.6 Software Implementation . . . 126
7.6.1 Implementation of the Hash Functions . . . 127
7.6.2 Implementation of FAST . . . 129
7.6.3 Timing Results . . . 131
7.7 Additional Material on Implementation of AEZ . . . 133
7.7.1 Software Implementation . . . 134
7.8 Summary . . . 135

8 Breaking Tweakable Enciphering Schemes using Simon's Algorithm 136
8.1 Preliminaries . . . 137
8.1.1 Tweakable Enciphering Scheme . . . 137
8.1.2 Simon's Algorithm with Spurious Collisions . . . 137
8.2 Outline of the Attacks . . . 138
8.3 Partial Key Recovery Attacks . . . 140
8.3.1 XCB . . . 140

8.3.2 TET 142
8.3.3 FAST 145
8.4 Distinguishing Attacks 149
8.4.1 CMC 149
8.4.2 EME 152
8.5 Summary 155

9 Future Research Possibilities **157**

Chapter 1

Introduction

In symmetric key cryptography there are two fundamental objectives, viz. 1. *confidentiality* or secrecy of message from unexpected party and 2. *authentication* of message which includes authenticating the source of the message as well as integrity of the message against any unwanted modification. Let us first concentrate on confidentiality. In classical *symmetric-key cryptography* two parties, say Alice and Bob, first secretly exchange a key-pair (e, d). Later, if Alice wishes to send a secret message $m \in \mathcal{M}$ to Bob, she computes $c = E_e(m)$ and transmits c to Bob. Upon receiving c, Bob computes $D_d(c) = m$ and gets the original message. Here either e and d are identical or can be derived from each other via a simple transformation. For authentication, if Alice wants to send a message $m \in \mathcal{M}$ to Bob, which is not secret but needs to be conveyed to Bob in its original form, she computes a tag $t = H_e(m)$ and sends the pair (m, t) to Bob. Upon receiving the pair, Bob runs a verification algorithm on it. He becomes sure that the message was indeed sent by Alice and has not been modified on the way if the pair passes the verification test. In this case Bob accepts it; otherwise he rejects the pair.

From the discussion above, it is clear that symmetric-key cryptography involves three sets, viz. *key-space*, which contains all possible secret keys for the transformations, *message-space*, which contains all possible messages or plaintexts to be conveyed to the other party and *output-space* which contains the ciphertexts in case of confidentiality and tags (or digests) in case of authentication. In case of confidentiality, the algorithms E and D are called encryption algorithms and decryption algorithms respectively. In case of authentication H is called tag-generation algorithm in general.

To construct these algorithms we use some basic cryptographic primitives in a way which will serve our purpose. Block ciphers, stream ciphers and universal hash functions are three of the fundamental primitives in symmetric key cryptography.

Let n be a fixed integer. Informally, a block cipher converts an n-bit plaintext into an n-bit ciphertext through an invertible mapping parameterised by a key K, taking values from the key-space. This is the encryption algorithm and the inverse mapping is the corresponding decryption algorithm, where n denotes the block size of the block cipher. The block size denotes the length of plaintext unit that can be encrypted per application of the block cipher.

Another important encryption primitive is a stream cipher, which takes a short secret random key and, in most cases, a unique initialisation vector (IV) as input and generates a long random looking keystream as output. Though there are stream ciphers, which do not take IV, almost all modern practical stream ciphers support that. Encryption is performed by bitwise XORing the output keystream, truncated at the appropriate length, to the message and decryption is done by doing the same to the ciphertext. The IV is a public quantity, but it needs to be unique for each encryption for secure use of all conventional stream ciphers.

Next comes the hash functions. An important primitive in cryptography which is particularly relevant to this thesis is a family of universal hash functions. Roughly speaking, it is an indexed family of functions satisfying certain conditions. For each index (normally called

the hash key) the function generally maps a string of arbitrary length into a short fixed length output. Two kinds of probabilities are associated with a universal function family. Here we will introduce them informally. Precise definitions are available in a subsequent chapter.

Collision probability: For a pair of distinct inputs, the collision probability of the hash function for the particular pair is defined to be the probability that the hash output on the two inputs are equal.

Differential probability: Suppose the hash outputs are from an additively written group. For a pair of distinct inputs and an element from the hash output field, the differential probability of the hash function for this triplet is the probability that the difference of the hash outputs on the two inputs equals that particular field element.

Both kinds of probabilities are taken over uniform random choices of the hash key. When a single value is mentioned for either of these probabilities for a hash function, in general that is the maximum value possible for the function for any input. Roughly speaking, for a hash function, if the collision probability for each pair of inputs is upper bounded by a quantity ε which is "sufficiently low", that hash function is called an ε-almost universal (ε-AU) function; similarly a hash function with differential probability bounded by "sufficiently low" ε, is called an ε-almost XOR universal (ε-AXU) hash function. The term "sufficiently low" has later been quantified concretely.

As is mentioned earlier in case of authentication, when Alice wants to send the message m to Bob, she computes a tag t and sends (m, t) to Bob. The tag is in general desired to be a short fixed length string, such that the authentication of the tag will ensure the authentication of the long message. Hence this short tag needs to depend both on the source of the message and on the message itself. One way to accomplish this using a universal hash function family is the following. Alice and Bob secretly exchange a hash key e and a function $F(\cdot)$. Let the universal hash family be denoted by h and let m be the n-th message sent by Alice to Bob. Alice computes the hash output $h_e(m)$ on the message for the key e. She masks this output with the random number (precise description is available in the subsequent chapters) $F(n)$, i.e. she computes $t = h_e(m) + F(n)$ ($+$ is assumed to be the corresponding field operation). Alice sends (m, t, n) to Bob. Upon receiving a tuple (m', t', n'), as Bob has the secret key e and the function $F(\cdot)$, he can recompute the tag corresponding to m' and n'. If the tag he computes equals t', he accepts the message; otherwise he rejects it. This idea was proposed by Wegman and Carter in [110] and has been described more formally later. It is to be noted that the tag t depends both on the message m and the origin of the message Alice (through the secret key and the secret function).

On the other hand, as (m, t, n) is sent through an insecure channel, the adversary has access to this tuple. So, it needs to be ensured that the probability of the adversary computing a valid tuple (m', t', n'), such that $(m, t, n) \neq (m', t', n')$ is low. Otherwise, if, for example, the adversary can easily come up with a (m', t', n') where $m' \neq m$ is any message of its choice and t' is a valid tag corresponding to m' and n', then it can replace (m, t, n) with (m', t', n'). Bob will accept it believing it to be a message sent by Alice. The provably low

collision and differential probabilities of the universal hash function family play important role, along with other features of the construction (e.g. the random mask), in preventing this possibility.

However, the above discussion implies that just with these stand-alone primitives, we can achieve cryptographic goals in essentially a very restricted domain. With block ciphers we can encrypt messages of a particular length only, whereas in real life messages to be encrypted may be and, in most cases, will be of different and variable lengths. Though with stream ciphers, this restriction on the length of the message is not there, stream cipher by itself does not give authentication. Also only a family of universal hash functions is not enough to achieve authentication. As we will see in later chapters, even the construction of a universal hash function family involves many subtleties. Thus in reality, it is not enough to have only these basic primitives to achieve the cryptographic goals in practical scenarios. Hence comes the necessity to build several *modes of operations* according to the requirements. While building a mode these issues are taken care of. First we need to construct the basic primitives with sufficient power to achieve our goals. Next, we need to modify them appropriately so that they can be used in real life for any possible input message without compromising on the issue of the related cryptographic security. In most of the cases more than one of these primitives need to be operated together to achieve the target.

Universal hash function is a primitive which has applications in many important modes of operations like message authentication codes, tweakable enciphering schemes etc. It was introduced almost four decades ago. As a result there have been many works in the literature around this primitive. Several constructions of universal hash functions have been proposed so far. One important category among them, which is particularly relevant to this thesis, is univariate polynomial based hash functions. The digest of this type of hash functions is obtained by evaluating a univariate polynomial over a finite field where the coefficients of the polynomial are the blocks of the input to the hash function; the polynomial is evaluated at the hash key, which is a single field element. Two important polynomial based universal hash functions are the following.

- **Honer's rule based hash:** The digest of this type of hash function is obtained by evaluating a usual univariate polynomial of degree ℓ for an input consisting of ℓ blocks. Using Horner's rule, this can be evaluated using $\ell - 1$ field multiplications.

- **Bernstein-Rabin-Winograd polynomial based hash:** Bernstein [15] built on a previous work by Rabin and Winograd [91] to propose a hash function using a class of univariate polynomials called the BRW polynomials [98]. The main advantage of this type of hashing over Horner's rule based hashing is that it requires half the number of multiplications than that required by Horner's rule based hashing. For hashing an input consisting of ℓ blocks using BRW polynomial based hash, the number of field multiplications required is $\lfloor \ell/2 \rfloor$ with an additional $\lfloor \lg \ell \rfloor$ squarings. On the other hand, there is a difficulty of BRW polynomials as far as efficient implementation is concerned. The definition of these polynomials is inherently recursive. As a result a straightforward implementation of these polynomials based hash for arbitrary input length suffers from recursive implementation overhead.

Formal definitions of these hash functions along with a more detail discussion on universal hashing in general are present in subsequent chapters.

An important mode of operation which is particularly important to this thesis, is tweakable enciphering scheme (TES). A TES is a tweak-based length preserving encryption, i.e. the length of the ciphertext is equal to the length of the plaintext. The tweak is an additional quantity, which determines the ciphertext, but is itself not encrypted. As a result of the dependency on the tweak, identical plaintexts while encrypted with same key but under different tweaks, give rise to ciphertexts that look unrelated to computationally bounded adversary. The most popular use of TES is in disk encryption algorithms. A disk is divided into sectors where each sector can store a fixed number of bytes. Each sector has a unique address. This sector address works as the tweak in case of disk encryption.

Another point to be noted is that, in many of the practical applications, we require not only secrecy or authentication separately, but we require both together. The related schemes are called *authenticated encryption* (AE). In some cases, there is some additional data (AD) along with the message, where the AD needs only authentication and message needs both confidentiality and authentication. Examples are network packets with header as associated data. The corresponding schemes are called *authenticated encryption with associated data* (AEAD). Normally AE or AEAD schemes take nonce as an input. *Deterministic authenticated encryption (with associated data)* (DAE(AD)) is an authenticated encryption (with associated data) scheme which does not use a nonce. Otherwise, such a scheme is almost the same as an AE(AD) scheme as far as syntax or security formalisation are concerned.

1.1 Overview of the Thesis

This thesis is in the areas of universal hash functions and modes of operations. It is structured as follows. Chapter 2 consists of a brief survey of the relevant works present in the literature. In Chapter 3 we set the notation and describe other prerequisite materials required for the rest of the thesis.

In Chapter 4, we describe an algorithm which can efficiently evaluate BRW polynomials constructed from m field elements without any restriction on m. As mentioned earlier, BRW polynomials are by definition recursive. Hence, in a scenario where m can vary, there can be a recursive implementation of BRW polynomials evaluated on m field elements; but that will not be efficient. To the best of our knowledge, till this algorithm was proposed, there had been no efficient algorithm for evaluating this polynomial on arbitrary number of elements. This is the first algorithm which non-recursively evaluates the polynomial on inputs of any length efficiently. Previously the best known complexity of evaluating a BRW polynomial on $m \geq 3$ field elements was $\lfloor m/2 \rfloor$ field multiplications. Typically, a field multiplication consists of a basic multiplication followed by a reduction. The new algorithm requires $\lfloor m/2 \rfloor$ basic multiplications and $1 + \lfloor m/4 \rfloor$ reductions. This is a significant practical speedup. As a practical contribution, we propose two new hash functions BRW128 and BRW256 with digest sizes 128 bits and 256 bits respectively, based on the new algorithm for evaluating these polynomials. The practicability of these hash functions is demonstrated by implementing them using instructions available on modern Intel processors. Timing results obtained from

the implementations suggest that the new hash function compares favourably to the highly optimised implementation by Gueron of Horner's rule based hash function.

Chapter 5 proposes a new universal hash function, which combines the advantages produced by two of its predecessors, viz. Horner's rule based polynomial hash and BRW polynomial based hash function. This hash function is the present state-of-the-art in terms of efficiency (CPU cycles required per byte of the digest/output) in high end Intel processors. It is a two-level hash function which employs BRW polynomial based hash in the lower level and Horner's rule based hash in the upper level. As mentioned earlier, BRW polynomial based hash requires half the number of multiplications required by Horner's rule based hash. On the other hand, Horner's rule based hash can efficiently handle an arbitrary length message which the inherently recursive BRW polynomial based hash cannot. Our new hash function Hash2L combines these two hashes such that the number of multiplications required is much lesser than that required by the Horner's rule based hash function and the difficulty of the BRW polynomial based hashing in handling a variable length message is avoided. Thus it combines the advantages produced by the two hashes and gives a new efficient two level universal hash. Though it is two-level, it has the attractive feature of using a single key. Here we propose two such hash functions, one for dealing a single binary string as a message and the other for dealing vector of binary strings as a message. We have provided concrete implementations of two instantiations of the first type of hash function using Intel Intrinsic and provided timing measurements for Haswell and Skylake processors. The two instantiations are Hash128, dealing with messages from 128 bit binary field $\mathsf{GF}(2^{128})$ and Hash256, dealing with messages from 256 bit binary field $\mathsf{GF}(2^{256})$. Comparative timing measurements are provided with respect to highly efficient implementations by Gueron of two Horner's rule based hashing and results show that Hash2L is significantly faster than both of them.

In Chapter 6, we study message authentication code (MAC) schemes supporting variable tag lengths. MAC is the cryptographic mechanism to ensure the authenticity of messages transmitted across a public channel. A MAC scheme typically appends a short length tag to the message which is then transmitted. At the receiving end, a verification algorithm is run on the message-tag pair to confirm the authenticity. In such a set-up, the sender and the receiver share a previously agreed upon secret key. Most MAC schemes in the literature specify a single value for the tag length. The question that we address in this chapter is the following. Is it possible to have MAC schemes where the tag length can vary? While the question seems to be a natural one, there does not appear to have been much discussion about this issue in the literature except a brief mention more than 15 years ago and the proposal of KMAC [69]. Variable tag lengths may be used with the same key due to "misuse and poorly engineered security systems" [92]. Also for resource constrained devices, variable tag lengths may be desirable where changing the key for every tag length may be infeasible due to limited bandwidth and low power. A more concrete example of this scenario is mentioned in the respective chapter.

We provide a formalisation of a variable tag length MAC scheme. Several variants of the classical Wegman-Carter MAC scheme are considered. Most of these are shown to be insecure by pointing out detailed attacks. One of these schemes is highlighted and proved to be secure. We further build on this scheme to obtain single-key variable tag length MAC schemes utilising either a stream cipher or a short-output pseudo-random function. These

schemes can be efficiently instantiated using practical well known primitives.

Chapter 7 introduces a new family of tweakable enciphering schemes (TES) called FAST. Several instantiations of FAST have been described. These are targeted towards use in the following two settings. The first one is the fixed length setting which is targeted towards disk encryption application. In this case, the tweak is a sector address and can be encoded using a short fixed length string; the messages are contents of a sector and so are fixed length strings equal to the size of the sector. The second one is the general setting suitable for a wide variety of practical applications. Here the message space consists of binary strings of different lengths; the tweak space can also consist of strings of different lengths or even vector of strings where the numbers of components in the vectors can vary. This richness in the tweak space provides considerable flexibility in applications where there is a message and an associated set of attributes. The message is to be encrypted while the attributes are to be in the clear but the ciphertext needs to be bound to the attributes. One possible application of such a functionality is the following. The message is a data packet that is to be stored at a destination node while the vector of attributes encode the path taken by the data packet to reach the destination node with the components of the vector identifying the intermediate nodes.

A major contribution here is to present detailed and careful software implementations of all of these instantiations and some other important TES schemes in the literature. These implementations are targeted towards modern Intel processors and are in Intel intrinsics using the specialised AES-NI instructions and the `pclmulqdq` instruction. The code for the software implementation of FAST is publicly available; the reference is available in the respective chapter. Comparative performance is measured on the Skylake and the Kabylake processors of Intel. For disk encryption, the results from the implementations show that FAST compares very favourably to the IEEE disk encryption standards XCB and EME2 as well as the more recent proposal AEZ.

FAST is built using a fixed input length pseudo-random function and an appropriate hash function. It uses a single-block key, is parallelisable and can be instantiated using only the encryption function of a block cipher. The property of not requiring the invertibility property of a block cipher has the advantage of requiring a smaller hardware in a hardware implementation, a smaller size code in a software implementation and having the security proof of FAST on a weaker assumption on the underlying block cipher (PRF rather than SPRP). The hash function can be instantiated using either the Horner's rule based usual polynomial hashing or hashing based on the more efficient Bernstein-Rabin-Winograd polynomials. Security of FAST has been rigorously analysed using the standard provable security approach and concrete security bounds have been derived.

Chapter 8 evaluates the post-quantum security of some tweakable enciphering schemes. The eventual availability of large-scale quantum computers appears to be a certainty. This will have major impact on cryptography. It seems that public key cryptography will be severely affected and some part of it will be completely broken by quantum algorithms like the one by Shor [102]. On the other hand, though already some devastating attacks have been shown on some symmetric key algorithms, this area is still not sufficiently understood. In past few years there have been some works to evaluate the security provided by symmetric key crypto-systems in the post-quantum world. Some of them have shown the applicability

of Simon's period finding quantum algorithm [104] to the cryptanalysis of certain modes of operation. In this chapter we continue this line of work. Our target modes of operations are several tweakable enciphering schemes, namely, CMC, EME, XCB, TET and FAST. For all of the five TESs, we show distinguishing attacks, while for XCB, TET and FAST, the attacks reveal portions of the secret keys. It shows that for post-quantum world, many of the existing TESs will not work and we need to search for suitable solution.

Chapter 9 concludes the thesis and provides a brief discussion on the future direction of the relevant research.

Publications: This thesis is based on the following published works.

1. Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. IACR Trans. Symmetric Cryptol., 2017(1):106–128, 2017.

2. Sebati Ghosh and Palash Sarkar. Evaluating Bernstein-Rabin-Winograd polynomials. Des. Codes Cryptogr., 87(2-3):527–546, 2019.

3. Debrup Chakraborty, Sebati Ghosh, Cuauhtemoc Mancillas López, and Palash Sarkar. FAST: disk encryption and beyond. Advances in Mathematics of Communications. `https://www.aimsciences.org/article/doi/10.3934/amc.2020108`, 2020.

4. Sebati Ghosh and Palash Sarkar. Variants of Wegman-Carter message authentication code supporting variable tag lengths. Des. Codes Cryptogr., doi: `https://doi.org/10.1007/s10623-020-00840-w`, 2021.

5. Sebati Ghosh and Palash Sarkar. Breaking tweakable enciphering schemes using Simon's algorithm. Preprint: IACR Cryptology ePrint Archive 2019: 724, 2019. Communicated.

Chapter 2

A Brief Survey of the Literature

This chapter consists of a brief survey of the literature mainly relevant to the works done in this thesis. The goal of this survey is in no way to give an exhaustive list of all the works available in the literature on a particular topic. Rather we aim to give the reader a more or less bird eye view of the overall development of the area.

As discussed in Chapter 1, the basis of the area are some fundamental stand-alone primitives which are capable of providing sufficient security and robustness. These are in turn appropriately modified and combined into suitable *modes* in order to accomplish a cryptographic target.

One of the most important basic primitives relevant to this thesis is universal hash function. A brief survey of the well known universal hash functions of the literature is provided in Section 2.1. Sections 2.2 and 2.3 give overviews of two of its important applications relevant to this thesis, viz. message authentication code schemes and tweakable enciphering schemes respectively. Section 2.4 provides a brief overview of some of the works done in the area of post-quantum cryptanalysis on certain modes.

2.1 Universal Hash Functions

A good survey on various constructions of universal hash functions can be found in each of [15, 101]. Here we do not provide all the details available there. On the other hand, here we mention some constructions which are subsequent to [15, 101] and hence are not present there.

Universal hash functions were introduced by Carter and Wegman [28] in the year of 1979. Here they exhibit three universal classes of functions which can be evaluated easily and give several examples of the use of those functions. A well known proposal to another universal hash function called Multilinear Map [54] was given by Gilbert, MacWilliams and Sloane. This requires l field multiplications to obtain the digest when the message consists of l field elements. This computational complexity can be reduced to $l/2$ field multiplication by using the pseudo-dot product construction proposed by Winograd [111]. One negative issue for both the multi-linear hash and the pseudo-dot product is that the key required for the hash function is as long as the message.

This problem can be avoided by using another well known approach, which has been briefly discussed in Chapter 1. Here we provide more details. In this case the digest is obtained by evaluating a univariate polynomial over a finite field. The coefficients of the polynomial are the message blocks and the point at which the polynomial is evaluated is the hash key. As a result, the hash key consists of a single field element. Using Horner's rule, a univariate polynomial of degree l can be evaluated using (l–1) field multiplications. This cost is about the same as that required for multilinear map based hash function. The

advantage is in terms of key length. Examples of Horner's rule based polynomial hashing are Poly1305 [13], PolyR [73], GHASH [85] and POLYVAL [60].

Poly1305 is part of several practical and important platforms including the Transport Layer Security protocol version 1.3 [5]. Here, the arithmetic is over the prime field $\mathbb{F}_p$ with $p = 2^{130} - 5$. Initially, clever use of floating point techniques were made to provide efficient implementation of Poly1305 in [13]. Later significant speedups in high end Intel processors have been reported in [55] and [18], none of which has used floating point techniques.

PolyR is a universal hash function family which hashes short messages faster than long ones. This is quite contrary to most of the hash functions as they do better as the length of the message increases. As most network traffic is short, this property of PolyR is desirable in authenticating network traffic. Due to this property, another important use of PolyR is in a multi-layer hashing construction which is used for message authentication. Here, a first layer of fast hashing is applied on the input to get an output of length much smaller than the input. Now on this short message PolyR is applied for fast authentication. Another attractive property of PolyR is that its key requires no preprocessing to achieve maximal efficiency.

GHASH is also an important universal hash function, where the arithmetic is over the field $\mathbb{F}_{2^{128}}$. As it forms a part of the NIST standard [45], there has been much research in efficient implementation of GHASH. In fact, one of the reasons for Intel to include the `pclmulqdq` instruction, which multiplies two degree 64 polynomials, is to be able to efficiently implement GHASH. The best known highly optimized implementation of GHASH using `pclmulqdq` is by Gueron [59].

POLYVAL is also a univariate polynomial hashing algorithm. More precisely, this is a byte-swapped version of GHASH, applied over byte-swapped message. In this case also, the arithmetic is over the field $\mathbb{F}_{2^{128}}$. Again, the highly optimised implementation of POLYVAL using `pclmulqdq` is due to Gueron [57].

Another significant univariate polynomial based universal hash function, which we have briefly discussed in Chapter 1, was proposed by Bernstein. Bernstein [15] built on a previous work by Rabin and Winograd [91] to design a family of polynomials which was later named the BRW polynomials [98]. The importance of such polynomials for constructing a universal family of hash functions has been discussed in [15]. The hash key is still a single element of the field. The main advantage of BRW polynomial based hashing is that the number of multiplications required for hashing a message consisting of $l \geq 3$ blocks is $\lfloor l/2 \rfloor$ with an additional $\lfloor \lg l \rfloor$ squarings. In fact, what the pseudo-dot product is to the multilinear hash, the BRW polynomials is to the Horner based hash.

There is, however, an obstacle in efficient implementation of BRW polynomials. These polynomials are inherently recursive. As a result a straightforward implementation of these polynomials based hash functions for arbitrary length messages has recursive implementation overhead. Possibly due to this reason, until recently there had been no efficient software implementation of BRW based hash function for arbitrary length inputs. On the other hand, hardware implementations for fixed length inputs are known [33]. A work [30] gives software implementations of BRW polynomials having $l \leq 31$ blocks over the fields $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$. Another work [52] describes the first non-recursive algorithm which can efficiently evaluate BRW polynomials for arbitrary length inputs. This algorithm also results in a

practical speedup in evaluating these polynomials. Based on the new algorithm two new hash functions BRW128 and BRW256 with digest sizes 128 bits and 256 bits respectively have been proposed, along with their implementation on modern Intel processors. These are, to the best of our knowledge, the first efficient software implementation of BRW polynomials based hash function for arbitrary length inputs.

Well known constructions of hash functions based on the pseudo-dot product are hashes used in UMAC [20] and VMAC [72]. The core of the MAC scheme UMAC is the hash function NH^{T} which is based on integer arithmetic. This hash function processes an l-block message with each block being w-bit long to produce a digest of size $2tw$ for some parameter $t \geq 1$. The hash key consists of $l + 2(t-1)w$-bit blocks. So, the length of the hash key is longer than the length of the message to be hashed. The core of VMAC is the hash function VHASH which is also based on integer arithmetic and requires a key which is longer than the message. A more recent pseudo-dot-product based construction is the hash function Hash256, the core of the MAC scheme Auth256 [16]. This hash function uses arithmetic over $\mathbb{F}_{2^{256}}$ and the key is as long as the message. The work [16] reports an implementation of Hash256 using a tower field representation and a new FFT-based algorithm for field multiplication. It does not use the `pclmulqdq` instruction on Intel processors. The construction of the hash function CLHASH, proposed in [77], is based on the idea of VHASH, although the computation is over $\mathbb{F}_{2^{64}}$. The hash function produces 64-bit outputs. We note that more than 10 years ago, Bernstein had commented [107] that a 64-bit digest provides inadequate security.

2.1.1 Efficiency Issues

As can be understood from the above discussion one continuous target in this area has been to construct a universal hashing requiring minimum number of field multiplications to evaluate it. The reason is field multiplication is a costly operation in the sense it requires more CPU time than other field operations. Typically, a field multiplication consists of a basic multiplication followed by a reduction operation. Gueron and Kounavis [58] described an efficient method for reduction over binary fields. Late, or, delayed reduction is a well known technique for speeding up a group of field multiplications. Essentially, the idea is to perform several polynomial multiplications, add the results and then perform a single reduction for the entire group of multiplication. The most recent implementations of both GHASH [59] and POLYVAL [57] by Gueron use delayed reduction through the use of pre-computed tables. The implementation of Hash2L [31] also uses delayed reduction to achieve efficiency. In [51], reduction has been delayed as far as possible to achieve an algorithm which reduces the practical CPU time required for computing BRW polynomials to a considerable extent.

In [87] Nandi has shown a lower bound on the number of multiplications required for secure hashing which shows that the pseudo-dot and BRW based hashing essentially require an optimal number of field multiplications.

2.1.2 Collision and Differential Probabilities

In Chapter 1 we have informally introduced collision and differential probabilities associated with a universal hash function family. In general, for a Horner's rule based polynomial hash, on the field $\mathbb{F}$, for messages consisting of at most l field elements the collision probability is $((l-1)/\#\mathbb{F})$ and the differential probability is $l/\#\mathbb{F}$, where $\#\mathbb{F}$ means the cardinality of the field $\mathbb{F}$. Let the degree of BRW polynomial on a message consisting of l field elements be denoted by $\mathsf{deg}(l)$. Then $\mathsf{deg}(l) = 2^{\lfloor \lg l \rfloor + 1} - 1$ [15]. As a consequence, for BRW hashing the collision probability is $\mathsf{deg}(l)/\#F$ and the differential probability is $\mathsf{deg}(l) + 1/\#F$, for messages consisting of at most l field elements.

For the pseudo-dot-product based hash function NH^{T}, which is the core of the UMAC, the collision probability is 2^{-tw}, where the parameters t and w are defined before. For Auth256 the collision probability is at most 2^{-256} and differential probability is at most 2^{-255}.

Universal hash function families have many applications in cryptography including constructions of message authentication code (MAC), authenticated encryption and disk encryption schemes among others.

2.2 Message Authentication Code

The notion of MAC is several decades old. So, there is an extensive literature on this topic. Most MAC schemes specify a single value for the tag length. In this thesis, we address the following question. Is it possible to have MAC schemes where the tag length can vary? Here we mainly discuss those works which are directly related to our work.

Wegman and Carter proposed [110] the following method of using an AXU hash function to construct a MAC scheme. Let $\{H_\tau\}_\tau$ be an AXU hash function family such that the output of H_τ is n-bit; let $\{F_K\}_K$ be a pseudo-random function family where F_K maps n-bit strings to n-bit strings. Roughly speaking, the function family $\{F_K\}_K$ is considered to be pseudo-random if a resource limited adversary is unable to distinguish it from a uniform random function from $\{0,1\}^n$ to $\{0,1\}^n$. Formal definition is provided in the subsequent chapter. Using H and F a construction of a MAC scheme is the following. Using the key (K, τ), a nonce-message pair (N, M) is mapped to the n-bit tag $F_K(N) \oplus H_\tau(M)$. Since then it has been used in several important and practical MAC schemes, such as UMAC [20] and Poly1305-AES [13]. From a theoretical point of view, the security of the Wegman-Carter scheme was later analysed by Shoup [103] and Bernstein [14]. Recently, the optimality of Bernstein's bound was established in [79, 88].

The problem of tag length variability has been briefly mentioned about 15 years ago [107] in the context of UMAC [71]. Since then, to the best of our knowledge, the only proposal of a variable tag length MAC is KMAC [69]. KMAC is based on a sponge function called KECCAK [17] and it does not use nonce.

The point that tag lengths can vary in MAC schemes depending on the application has been noted in [96] where the problem of determining an economically optimal tag length has been considered from a game theoretic point of view. This is completely different from the

problem considered in this thesis.

Other Types of MACs: Here we have considered MAC schemes based on universal hash functions. Note that there are several MAC schemes based on primitives other than universal hash functions. Some popular examples are HMAC [9] based on collision resistant hash functions, CMAC [44] based on block cipher mode of operation, KMAC [69] based on sponge functions. On many systems HMAC is expected to be faster than CMAC, as collision resistant hash functions are usually faster than block ciphers. On the other hand, CMAC can be preferred over HMAC for short length messages or if there is special hardware support for the underlying block cipher. In terms of efficiency the universal hash function based Wegman-Carter MAC is superior to HMAC or CMAC. As far as security guarantees are concerned, the security of HMAC can be proved assuming certain properties on the underlying hash function. The security of CMAC depends on the security provided by the underlying block cipher. For universal hash function based MACs concrete security proofs can be given depending on the low collision or differential probability bound of the underlying hash function. On the other hand, none of HMAC, CMAC or KMAC uses nonce, whereas Wegman-Carter MACs use nonce. This nonce can be misused in some cases to break the security of this type of MACs.

2.3 Tweakable Enciphering Scheme

Another important application of universal hash functions in cryptography, as mentioned earlier, is construction of disk encryption schemes. A disk encryption mechanism is an example of a length preserving encryption where the length of the ciphertext is equal to the length of the plaintext. Further, there is another quantity (which is generally the sector address in case of disk encryption) which determines the ciphertext but, is itself not encrypted. In the literature this quantity has been called a tweak. The functionality of a tweak-based length preserving encryption has been called a tweakable enciphering scheme (TES) [65].

The notion of a tweakable block cipher and its security was formalised by Liskov, Rivest and Wagner [78]. This was followed by a formalisation of the notion of a tweakable enciphering scheme by Halevi and Rogaway [65]. The paper also described a TES called CMC which is based on the CBC mode of operation. A subsequent work [66] by the same authors introduced a TES called EME which is a parallelisable mode of operation of a block cipher. EME was extended to handle arbitrary length messages by Halevi [62] and the resulting scheme was called EME*. The EME family of TESs does not require finite field multiplication. The main cost of encryption is roughly two block cipher calls per block of the message.

Construction of a TES using a counter based mode of operation of a block cipher and a Horner type hash function was first proposed by McGrew and Fluhrer [81]. This scheme was called XCB. A later variant [83] of XCB was proposed to improve efficiency and reduce key size. Various security problems for XCB have been pointed out [32].

There have been a number of works proposing different constructions of TESs. Examples are PEP [36], ABL [84], HCTR [109], HCH [37], TET [63] and HEH [98]. An improved security analysis of HCTR has been done later [35]. A generalisation of EME using a general masking scheme has been proposed [97].

Most of the TES proposals in the literature, including the ones that have been standardised are modes of operations of a block cipher and use both the encryption and the decryption functions of the underlying block cipher. The possibility of constructing a TES using only the encryption function of a block cipher has been suggested first in [100]. The work was more at a conceptual level using generic components and some unnecessary operations. It did not provide any specific instantiation or implementation. Subsequent to [100], the constructions AEZ [67] and FMix [19] proposed single key TESs using only the encryption function of the block cipher. FMix is a sequential scheme while AEZ is parallelisable.

The possibility of constructing TESs from stream ciphers has been considered [100]. Concrete proposals and detailed FPGA implementations of stream cipher based TESs have been described [34].

While disk encryption is a very important application of a TES, the full functionality of a TES is much more broader than just disk encryption. For the specific case of disk encryption, messages are contents of a sector and so are fixed length strings. A TES can have a more general message space consisting of binary strings of different lengths. Similarly, in the case of disk encryption, the tweak is a sector address and can be encoded using a short fixed length string. More generally, the tweak space in a TES can also consist of strings of different lengths or even consist of vectors of strings. The idea of having associated data to be a vector of strings was earlier proposed [95] in the context of deterministic authenticated encryption. AEZ [67] provides a conceptual level description of how to handle a vector of strings as tweak using an almost XOR universal hash function to process the vector. A generic security bound is provided in terms of the collision probability of the hash function. No concrete proposal for the hash function is provided.

Another line of investigation has been the construction of ciphers that can securely encipher their own keys [64, 10]. A generic method is known [10] which converts a conventional TES to one which can be proved to be secure even under the possibility of encrypting its own key. This generic method has been applied to EME2 [10]. We note that the method can equally well be applied to the construction FAST [29].

IEEE [3] has standardised two tweakable enciphering schemes, namely EME2 and XCB. Essentially, EME2 is the variant EME* [62] while the standardised version of XCB is a variant [83] of the original scheme [81]. Both EME2 and XCB are patented algorithms. To the best of our knowledge, till date there is no unpatented algorithm which has been standardised. An earlier IEEE standard is XTS [2] which has also been standardised [46] by NIST of USA. This is based on the XEX construction of Rogaway [94]. XTS is not a TES and the security provided by XTS is not adequate for disk encryption application. Rogaway [1] himself mentioned that XTS only provides light security and should be preferred only when there is an overriding concern for speed.

One of the problems considered in this thesis is on cryptanalysis of several TESs against a quantum adversary. Hence, in the next section we give a brief overview of some relevant previous works.

2.4 Post Quantum Cryptography

An important line of works involving a significant portion of the community at present is the post-quantum cryptography. As the availability of large-scale quantum computers appears to be only a matter of time now, cryptographers almost in every branch of cryptography are in search of primitives that will be able to resist quantum adversaries. Different quantum adversarial models and the notion of security against such adversaries have been formalised in [21, 22, 105, 40, 49, 25, 6]. Public-key cryptography based on factoring and the discrete logarithm problem will be completely broken by Shor's algorithm [102]. Sensing the massive impact quantum algorithms are going to have on public-key cryptography, NIST, back in 2017, had already started the process of inviting ideas and evaluating them in order to standardise quantum-safe public-key cryptosystems. The process is referred to as post-quantum cryptography standardization [4] and is currently in its third round.

For symmetric key ciphers, exhaustive key search will be speeded up by a quadratic factor using Grover's algorithm [56]. A series of works have shown the applicability of Simon's period finding quantum algorithm [104] to cryptanalysis of certain modes of operation. Whereas a 3-round Fiestel cipher with internal permutations is secure against any chosen plaintext attack on the classical computer, it can be distinguished efficiently from a random permutation by a polynomial quantum algorithm. This algorithm has been proposed in [75] and uses Simon's algorithm. Simon's algorithm has also been used to show [76] that the quantum version of the Even-Mansour cipher is insecure. Anand et al. have used [7] this period finding quantum algorithm in cyptanalysis of some modes of operation like CBC, CFB and XTS. The work [68] by Kaplan et al. has shown its applicability to cryptanalysis of several other symmetric cryptosystems. These include many widely used modes of operation for authentication and authenticated encryption, viz. CBC-MAC, PMAC, GMAC, GCM, OCB etc. A generalisation of Simon's algorithm has been proposed [23] and used to mount a key-recovery attack on the authenticated encryption algorithm AEZ [67] against a quantum adversary. This line of work on cryptanalysis of symmetric-key primitives has been taken forward in [42]. These attacks require quantum access to the cryptographic algorithm. More recently, there has been work [24] on developing attacks based on offline Simon's algorithm which do not need to make quantum queries.

Chapter 3

Preliminaries and Background

In this chapter, we fix the notation which are used in several parts of the thesis. Some notation are problem specific and have been set in the beginning of the corresponding chapter.

Throughout the thesis, $\mathbb{F}$ will denote a finite field. The two standard operations over $\mathbb{F}$ are multiplication and addition. For $\alpha, \beta \in \mathbb{F}$, the product (resp. sum) of α and β will be denoted as $\alpha\beta$ (resp. $\alpha + \beta$) as is conventional.

Throughout the thesis, n is a positive integer. In most of the cases we have considered the underlying field to be $GF(2^n)$, where $GF(2^n)$ is the finite field of 2^n elements. We will denote $GF(2^n)$ by $\mathbb{F}_{2^n}$. Using a fixed irreducible polynomial of degree n over $GF(2)$ to represent $\mathbb{F}_{2^n}$, the elements of $\mathbb{F}_{2^n}$ can be identified with the binary strings of length n. Viewed in this manner, an n-bit binary string will be considered to be an element of $\mathbb{F}_{2^n}$. $GF(2^n)$ is a field of characteristic two. The addition operation over $\mathbb{F}_{2^n}$ will be denoted by $\oplus$; The additive identity of $\mathbb{F}_{2^n}$ will be denoted as $\mathbf{0}$ and will be represented as 0^n; the multiplicative identity of $\mathbb{F}_{2^n}$ will be denoted as $\mathbf{1}$ and will be represented as $0^{n-1}1$.

For $n = 128$, let $\mathbb{F}_{2^n}$ be represented as $GF(2)[x]/\psi(x)$ where $\psi(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$. The 128-bit string α is considered to be a polynomial $\alpha(x) \in GF(2)[x]$. Let β be a 128-bit string representing the polynomial $\beta(x) = x\alpha(x) \bmod \psi(x)$. The string β can be obtained from the string α as $\beta = (\alpha \ll 1) \oplus (\mathsf{msb}(\alpha) \cdot 135)$. Over $\mathbb{F}_{2^n}$, this operation corresponds to the 'multiply by x' map and has been called a doubling operation [94]. For $n = 256$, the irreducible polynomial is $\psi(x) = x^{256} \oplus x^{10} \oplus x^5 \oplus x^2 \oplus 1$.

Let p be a prime. The finite field $GF(p^n)$ will be denoted as $\mathbb{F}_{p^n}$. More details about this field will be provided in the chapter where it is used.

- Let α and β be two strings from any finite field $\mathbb{F}$.
  - The length of α will be denoted as $\mathsf{len}(\alpha)$. If α is a binary string, then $\mathsf{len}(\alpha)$ is the number of bits in α.
  - The concatenation of α and β will be denoted as $\alpha||\beta$.
  - For an integer i with $0 \leq i < 2^n$, $\mathsf{bin}_n(i)$ denotes the n-bit binary representation of i.
- Let α be a non-empty binary string.
  - $\mathsf{msb}(\alpha)$ denotes the most significant bit of α.
  - For a non-negative integer λ, $\mathsf{msb}_\lambda(\alpha)$ denotes the λ most significant bits of α.
  - For $0 < i \leq \mathsf{len}(\alpha)$, $\mathsf{first}_i(\alpha)$ will denote the first (or, the most significant) i bits of α.

- $\mathsf{format}_n(\alpha)$ denotes $(\alpha_1, \alpha_2, \ldots, \alpha_{\mathfrak{m}})$ where $\alpha = \alpha_1 || \alpha_2 || \cdots || \alpha_{\mathfrak{m}}$, $\mathfrak{m} = \lceil \mathsf{len}(x)/n \rceil$, $\mathsf{len}(\alpha_i) = n$ for $1 \leq i \leq \mathfrak{m} - 1$ and $1 \leq \mathsf{len}(\alpha_{\mathfrak{m}}) \leq n$. In other words, $\mathsf{format}_n(\alpha)$ divides the string α into $\mathfrak{m} - 1$ n-bit blocks $\alpha_1, \ldots, \alpha_{\mathfrak{m}-1}$ and a possibly partial last block $\alpha_{\mathfrak{m}}$.

- For any polynomial $q(x)$, the degree of it is represented by $\mathsf{deg}(q)$.

3.1 Universal Hashing

Let $\mathcal{M}$, $\mathcal{G}$ and T be finite non-empty sets. Let $\{H_\tau\}_{\tau \in \mathsf{T}}$ be an indexed family of functions such that for each $\tau \in \mathsf{T}$, $H_\tau : \mathcal{M} \to \mathcal{G}$. The index set T is considered to be the set of all keys and a particular τ from T is considered to be the key for H_τ. The sets $\mathcal{M}$ and $\mathcal{G}$ are respectively the message and the digest spaces. We define two kinds of probabilities associated with such a function family.

Collision probability: For distinct $m, m' \in \mathcal{M}$, the collision probability of $\{H_\tau\}_{\tau \in \mathsf{T}}$ for the pair (m, m') is defined to be $\Pr_\tau[H_\tau(m) = H_\tau(m')]$.

Differential probability: Suppose $\mathcal{G}$ is an additively written group. For distinct $m, m' \in \mathcal{M}$ and any $g \in \mathcal{G}$, the differential probability of $\{H_\tau\}_{\tau \in \mathsf{T}}$ for the triplet (m, m', g) is defined to be $\Pr_\tau[H_\tau(m) - H_\tau(m') = g]$.

In the above, the probabilities are taken over uniform random choices of τ from T.

These probabilities may depend on the lengths of m and m'. Suppose L is the maximum of the lengths of the elements in $\mathcal{M}$. Let $\varepsilon_c : \{0, \ldots, L\}^2 \to [0, 1]$ be a function such that the collision probability for any (m, m') is at most $\varepsilon_c(\mathsf{len}(m), \mathsf{len}(m'))$. Then the family $\{H_\tau\}_{\tau \in \Theta}$ is said to be ε_c-**almost universal** (ε_c-**AU**). Similarly, let $\varepsilon_d : \{0, \ldots, L\}^2 \to [0, 1]$ be a function such that the differential probability for any (m, m', g) is at most $\varepsilon_d(\mathsf{len}(m), \mathsf{len}(m'))$. Then the family $\{H_\tau\}_{\tau \in \Theta}$ is said to be ε_d-**almost XOR universal** (ε_d-**AXU**).

3.1.1 Polynomial Hashing

For $\ell \geq 0$, the polynomial $\mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell)$ in the variable τ with $m_1, \ldots, m_\ell \in \mathbb{F}$ is defined as follows:

If $\ell = 0$, then $\mathsf{Horner}_\tau() = 0$; and for $\ell > 0$,

$$\left.\begin{array}{l} \mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell) \\ \quad = \; m_1\tau^{\ell-1} + m_2\tau^{\ell-2} + \cdots + m_{\ell-1}\tau + m_\ell \\ \quad = \; (((m_1\tau + m_2)\tau + m_3)\tau + \cdots + m_{\ell-1})\tau + m_\ell. \end{array}\right\} \tag{3.1}$$

Note that computing Horner on ℓ field elements requires $\ell - 1$ additions and $\ell - 1$ multiplications.

It is well known that $\{\mathsf{Horner}_\tau\}_{\tau \in \mathbb{F}}$, is $((\ell - 1)/\#\mathbb{F})$-AU. Further, the hash function $\{\tau\mathsf{Horner}_\tau\}_{\tau \in \mathbb{F}}$ is $(\ell/\#\mathbb{F})$-AXU. These bounds on the collision and differential probabilities are under the assumption that the maximum length of any input message can be at most ℓ.

3.1.2 BRW Hashing

In [15], Bernstein defined a family of polynomials based on previous work by Rabin and Winograd [91], later called the BRW polynomials in [98]. For $\ell \geq 0$, $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\ell)$ with $m_1, \ldots, m_\ell \in \mathbb{F}$ is a polynomial in the variable τ and is defined as follows:

- $\mathsf{BRW}_\tau() = 0$;
- $\mathsf{BRW}_\tau(m_1) = m_1$;
- $\mathsf{BRW}_\tau(m_1, m_2) = m_1\tau + m_2$;
- $\mathsf{BRW}_\tau(m_1, m_2, m_3) = (\tau + m_1)(\tau^2 + m_2) + m_3$;
- $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\ell)$
  $= (\tau^k + m_k) \times \mathsf{BRW}_\tau(m_1, \cdots, m_{k-1}) + \mathsf{BRW}_\tau(m_{k+1}, \cdots, m_\ell)$;
  if $k \in \{4, 8, 16, 32, \cdots\}$ and $k \leq \ell < 2k$.

Suppose $\ell \geq 3$. Following [15], it can be shown that $\mathsf{BRW}_\tau(m_1, \ldots, m_\ell)$ can be computed using $\lfloor \ell/2 \rfloor$ multiplications and $\lfloor \lg \ell \rfloor$ additional squarings to compute $\tau^2, \tau^4, \ldots$.

Let $d(\ell)$ denote the degree of $\mathsf{BRW}_\tau(m_1, \ldots, m_\ell)$. Then $d(\ell) = 2^{\lfloor \lg \ell \rfloor + 1} - 1$ [15] and so $d(\ell) \leq 2\ell - 1$ where the bound is achieved if and only if $\ell = 2^{\mathfrak{a}}$ for some $\mathfrak{a} \geq 2$ and $d(\ell) = \ell$ if and only if $\ell = 2^{\mathfrak{a}+1} - 1$ for some $\mathfrak{a} \geq 1$.

It has been proved in [15] that the map from $\mathbb{F}^\ell$ to $\mathbb{F}[\tau]$ given by

$$(m_1, \ldots, m_\ell) \longmapsto \mathsf{BRW}_\tau(m_1, \ldots, m_\ell)$$

is injective. As a consequence, the hash function $\{\mathsf{BRW}_\tau\}_{\tau \in \mathbb{F}}$, $\mathsf{BRW}_\tau : (m_1, \ldots, m_\ell) \mapsto \mathsf{BRW}_\tau(m_1, \ldots, m_\ell)$ is $(d(\ell)/\#\mathbb{F})$-AU. Further, the hash function $\{\tau\mathsf{BRW}_\tau\}_{\tau \in \mathbb{F}}$, $\tau\mathsf{BRW}_\tau : (m_1, \ldots, m_\ell) \mapsto \tau \cdot \mathsf{BRW}_\tau(m_1, \ldots, m_\ell)$ is $(\frac{d(\ell)+1}{\#\mathbb{F}})$-AXU. Again, these bounds are under the assumption that the maximum length of any input message can be at most ℓ. As $d(\ell) \leq 2\ell - 1$, we can also say that the the hash function $\{\mathsf{BRW}_\tau\}$ is $((2\ell-1)/\#\mathbb{F})$-AU and the hash function $\{\tau\mathsf{BRW}_\tau\}$ is $(\frac{2\ell}{\#\mathbb{F}})$-AXU.

3.2 Adversarial Model

In this thesis, the security analysis of some of the schemes uses the following adversarial model. An adversary $\mathcal{A}$ is a possibly probabilistic algorithm with access to one or more oracles. The output of an adversary is a single bit. The notation $\mathcal{A}^{\mathcal{O}_1, \mathcal{O}_2, \cdots} \Rightarrow 1$ denotes the fact that $\mathcal{A}$ outputs the bit 1 after interacting with the oracles $\mathcal{O}_1, \mathcal{O}_2, \ldots$. The interaction of $\mathcal{A}$ with its oracles is allowed to be adaptive, i.e., the adversary is allowed to choose an oracle and a query to be made to this oracle based on the responses it has received to its previous queries.

The important parameters of an adversary are its running time $\mathfrak{T}$, the number of queries q that it makes to all its oracles and its query complexity σ. The definition query complexity may vary from case to case.

The bulk of the actual security analysis will be in the information theoretic sense which in particular means that there is no restriction on the resources of the adversary. For such analysis, it is sufficient to consider the adversary to be a deterministic algorithm.

3.3 Pseudo-Random Function

Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{K}$ be finite non-empty sets. Let $\{F_K\}_{K\in\mathcal{K}}$ be a keyed family of functions where for each $K \in \mathcal{K}$, $F_K : \mathcal{D} \to \mathcal{R}$. Here $\mathcal{K}$ is the keyspace, $\mathcal{D}$ is the domain and $\mathcal{R}$ is the range.

Informally speaking, the function family $\{F_K\}_{K\in\mathcal{K}}$ is considered to be pseudo-random if a resource limited adversary is unable to distinguish it from a uniform random function from $\mathcal{D}$ to $\mathcal{R}$. In other words, for a randomly chosen $K \in \mathcal{K}$, on distinct inputs, the outputs of $F_K(\cdot)$ appear independent and uniformly distributed to a computationally bounded adversary. This is formalised in the following manner.

We consider an adversary $\mathcal{A}$ which has access to an oracle $\mathcal{O}$, which is written as $\mathcal{A}^{\mathcal{O}}$. $\mathcal{A}$ adaptively sends queries to $\mathcal{O}$ and receives appropriate responses. We will assume that $\mathcal{A}$ does not repeat a query. At the end of the interaction, $\mathcal{A}$ outputs a bit. The adversary is allowed to perform computations and also has access to private random bits.

Let $(K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{F_K(\cdot)} \Rightarrow 1)$ denote the event that K is chosen uniformly at random from $\mathcal{K}$ and the adversary produces 1 after interacting with the oracle $F_K(\cdot)$. Let $\$(\cdot)$ be a function chosen uniformly at random from the set of all functions from $\mathcal{D}$ to $\mathcal{R}$. An equivalent and more convenient view of $\$$ is the following. For distinct elements $X_1, \ldots, X_q$ from $\mathcal{D}$, the elements $\$(X_1), \ldots, \$(X_q)$ are independent and uniformly distributed elements of $\mathcal{R}$. Let $(\mathcal{A}^{\$(\cdot)} \Rightarrow 1)$ denote the event that the adversary produces 1 after interacting with the oracle $\$(\cdot)$.

The advantage of $\mathcal{A}$ in breaking the pseudo-randomness of $\{F_K\}_{K\in\mathcal{K}}$ is defined as follows.

$$\mathsf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{F_K(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot)} \Rightarrow 1\right]. \tag{3.2}$$

The probabilities are over the randomness of $\mathcal{A}$, the choice of K and the randomness of $\$(\cdot)$.

Suppose that $\mathcal{A}$ makes a total of q queries sending a total of σ bits in all the queries. By $\mathsf{Adv}_F^{\mathrm{prf}}(\mathfrak{T}, q, \sigma)$ we denote the maximum of $\mathsf{Adv}_F^{\mathrm{prf}}(\mathcal{A})$ over all adversaries $\mathcal{A}$ taking time at most $\mathfrak{T}$, making at most q queries and sending at most σ bits in all its queries. The function family $\{F_K\}_{K\in\mathcal{K}}$ (or, more simply F) is said to be a $(\mathfrak{T}, q, \sigma, \varepsilon)$-PRF if $\mathsf{Adv}_F^{\mathrm{prf}}(\mathfrak{T}, q, \sigma) \leq \varepsilon$.

Some of our constructions require pseudo-random function mapping n-bit strings to n-bit strings. More precisely, $\{F_K\}_{K\in\mathcal{K}}$ needs to be a family of functions, where for $K \in \mathcal{K}$, $F_K : \{0,1\}^n \to \{0,1\}^n$. In this case, it is possible to instantiate F using the encryption (or the decryption) function of a block cipher. In particular, one may use the encryption function of AES to instantiate F. This, however, is an overkill, since the invertibility property of the block cipher is not required for being a pseudo-random function.

3.3.1 Counter Mode

Generally, the PRF F can handle only fixed length strings. The manner in which longer strings are handled is shown for the PRF $\{F_K\}_{K\in\mathcal{K}}$, where for $K \in \mathcal{K}$, $F_K : \{0,1\}^n \to \{0,1\}^n$. The method can be extended to other PRFs with suitable mapping of the underlying input field into binary extension field.

Let α be a non-empty binary string. For $K \in \mathcal{K}$ and $s \in \{0,1\}^n$, we define $\mathsf{Ctr}_{K,s}(\alpha)$ in the following manner.

$$\mathsf{Ctr}_{K,s}(\alpha) \;=\; (s_1 \oplus \alpha_1, \ldots, s_{\mathtt{m}-1} \oplus \alpha_{\mathtt{m}-1}, \mathsf{first}_r(s_{\mathtt{m}}) \oplus \alpha_{\mathtt{m}}) \tag{3.3}$$

where $(\alpha_1, \ldots, \alpha_{\mathtt{m}}) \leftarrow \mathsf{format}_n(\alpha)$, $\mathsf{len}(\alpha_{\mathtt{m}}) = r$ and $s_i = F_K(s \oplus \mathsf{bin}_n(i))$. This variant of the counter mode was originally used in HCTR [109]. Note that the PRF F is used to define the counter mode, but, the counter mode itself as defined here is not a PRF.

3.4 Message Authentication Code

Here we consider nonce-based message authentication code (MAC) algorithms. Such a MAC scheme is parameterised by a secret key $K \in \mathcal{K}$, which is shared between the sender and the receiver. The sender runs a tag generation algorithm which takes as input a nonce $N \in \mathcal{N}$ and a message $M \in \mathcal{M}$ and generates a output $\mathsf{tag} \in \mathcal{T}$. The sender sends the tuple (N, M, tag) to the receiver. Upon receiving it, the receiver runs a verification algorithm on this tuple, which checks the authenticity of the message-tag pair. If the tuple passes this verification test, then the receiver accepts it; otherwise he rejects it.

Formally a nonce-based MAC scheme is a pair $\mathsf{MAC} = (\mathsf{MAC.Gen}, \mathsf{MAC.Verify})$ where

- $\mathsf{MAC.Gen} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \to \mathcal{T}$ and
- $\mathsf{MAC.Verify} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \times \mathcal{T} \to \{\mathsf{true}, \mathsf{false}\}$;

where $\mathcal{K}$, $\mathcal{N}$, $\mathcal{M}$ are finite non-empty sets and $\mathcal{T}$ is the set containing all valid tags. The set $\mathcal{K}$ is called the key space, $\mathcal{N}$ is called the nonce space, $\mathcal{M}$ is called the message space and $\mathcal{T}$ is called the tag space. The output of $\mathsf{MAC.Verify}$ is either true indicating that the input is accepted or false indicating that the input is rejected.

In Chapter 6, we have added a new feature to this framework and have discussed the corresponding security notion.

3.5 Tweakable Enciphering Scheme

A tweakable enciphering scheme is a pair $\mathsf{TES} = (\mathsf{TES.Encrypt}, \mathsf{TES.Decrypt})$ where

$$\mathsf{TES.Encrypt}, \mathsf{TES.Decrypt} : \mathcal{K} \times \mathcal{T} \times \mathcal{P} \to \mathcal{P}$$

for finite non-empty sets $\mathcal{K}, \mathcal{T}$ and $\mathcal{P}$. The set $\mathcal{K}$ is called the key space, $\mathcal{T}$ is called the tweak space and $\mathcal{P}$ is called the message/ciphertext space. We write $\mathsf{TES.Encrypt}_K(\cdot,\cdot)$ (resp. $\mathsf{TES.Decrypt}_K(\cdot,\cdot)$) to denote $\mathsf{TES.Encrypt}(K,\cdot,\cdot)$ (resp. $\mathsf{TES.Decrypt}(K,\cdot,\cdot)$). The functions $\mathsf{TES.Encrypt}$ and $\mathsf{TES.Decrypt}$ satisfy the following two properties. For $K \in \mathcal{K}$, $T \in \mathcal{T}$ and $P \in \mathcal{P}$,

1. $\mathsf{TES.Decrypt}_K(T, \mathsf{TES.Encrypt}_K(T, P)) = P$;

2. $\mathsf{len}\left(\mathsf{TES.Encrypt}_K(T, P)\right) = \mathsf{len}(P)$.

The first property states that the encryption and the decryption functions are inverses of each other while the second property states that the length of the ciphertext is equal to the length of the plaintext. In other words, $\mathsf{TES.Encrypt}_K(T, \cdot)$ is a length preserving permutation of $\mathcal{P}$.

The corresponding security notion is discussed in detail in Chapter 7.

Chapter 4

Evaluating Bernstein-Rabin-Winograd Polynomials

The Bernstein-Rabin-Winograd (BRW) polynomials were introduced by Bernstein in [15], as an important stepping stone towards constructing an efficient AXU family of hash functions. We note in particular that the evaluation of BRW polynomials requires about half the number of multiplications required for evaluating usual polynomial based AXU functions. So, designing AXU families based on BRW polynomials is particularly attractive from a practical point of view.

A BRW polynomial is constructed from $\ell \geq 0$ field elements. It is a univariate polynomial and evaluating it for ℓ field elements requires $\lfloor \ell/2 \rfloor$ field multiplications.

The definition of BRW polynomials is recursive. A recursive implementation is possible, but, will not be efficient. To the best of our knowledge, till date no algorithm has been proposed for efficiently evaluating BRW polynomials where ℓ can vary.

In this chapter, we present an efficient non-recursive algorithm for evaluating BRW polynomials constructed from ℓ field elements without any restriction on ℓ. The algorithm processes its input in a left-to-right fashion and maintains a set of partial results computed from the elements that have been processed. For a fixed $\mathfrak{t} \geq 2$, the algorithm reads the next $2^{\mathfrak{t}}$ elements and updates the partial results. The subtlety of the algorithm is in the manner in which the partial results are maintained and updated.

For $\ell \geq 3$, Bernstein [15] showed that a BRW polynomial defined using ℓ field elements can be evaluated using $\lfloor \ell/2 \rfloor$ field multiplications. Typically, a field multiplication consists of a basic multiplication followed by a reduction operation. We show that a BRW polynomial defined using $\ell \geq 3$ field elements can be evaluated using $\lfloor \ell/2 \rfloor$ basic multiplications and $1 + \lfloor \ell/4 \rfloor$ reduction operations. This is a significant reduction in the number of operations.

As a practical contribution, we propose two new hash functions, namely BRW128 and BRW256 which are based on BRW polynomials over the fields $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ respectively. These hash functions have been implemented using the Intel intrinsics instruction set available on modern Intel processors. Timing results for BRW128 compare favourably to those of a highly optimised implementation by Gueron [60] of Horner's rule based hash function.

This chapter is based on the work [52].

4.1 Preliminaries

Here we consider $GF(p^n)$, the finite field of p^n elements. As mentioned earlier it is denoted here by $\mathbb{F}_{p^n}$.

For $X, Y \in \mathbb{F}_{p^n}$, let $\mathsf{mult}(X, Y)$ denote the operation used to compute the product XY. The operation $\mathsf{mult}(X, Y)$ consists of two distinct steps. The first step consists of a basic or

an unreduced multiplication $\mathsf{unreducedMult}(X, Y)$ which returns a value Z and the second step consists of an operation $\mathsf{reduce}(Z)$, i.e.,

$$\mathsf{mult}(X, Y) \quad = \quad \mathsf{reduce}(\mathsf{unreducedMult}(X, Y)). \tag{4.1}$$

Depending on the value of n, there are two scenarios.

1. Case $n = 1$. In this case, the corresponding field is $\mathbb{F}_p$ and its elements are represented by the integers $0, \ldots, p-1$. Given two integers X and Y in $\{0, \ldots, p-1\}$, the operation $\mathsf{unreducedMult}(X, Y)$ performs the integer multiplication of the integers X and Y. The result Z is then at most $(p-1)^2$ and the operation $\mathsf{reduce}(Z)$ returns the element $W \in \{0, \ldots, p-1\}$ such that $W \equiv Z \bmod p$.

2. Case $n > 1$. In this case, using a fixed irreducible polynomial $\psi(x)$ of degree n over $\mathbb{F}_p$, the elements of $\mathbb{F}_{p^n}$ can be identified with the polynomials over $\mathbb{F}_p$ of degrees less than n. Given two such polynomials $X = X(x)$ and $Y = Y(x)$, the operation $\mathsf{unreducedMult}(X, Y)$ performs the polynomial multiplication of $X(x)$ and $Y(x)$ and returns the result $Z = Z(x)$ which is a polynomial of degree at most $2(n-1)$ over $\mathbb{F}_p$. The operation $\mathsf{reduce}(Z)$ returns $W = W(x)$ such that the degree of $W(x)$ is less than n and $W(x) \equiv Z(x) \bmod \psi(x)$.

Though BRW polynomials have been introduced in detail in Chapter 3, here we repeat them with reference to the particular field $\mathbb{F}_{p^n}$. This is because these polynomials are at the center of this chapter and hence all specific details about them need to be emphasised.

BRW polynomials: For $\ell \geq 0$, let $\mathsf{BRW} : \mathbb{F}_{p^n} \times (\mathbb{F}_{p^n})^\ell \to \mathbb{F}_{p^n}$ be the function defined below, where, we write $\mathsf{BRW}_z(\cdots)$ to denote $\mathsf{BRW}(z, \cdots)$.

- $\mathsf{BRW}_z() = 0$;
- $\mathsf{BRW}_z(M_1) = M_1$;
- $\mathsf{BRW}_z(M_1, M_2) = M_1 z + M_2$;
- $\mathsf{BRW}_z(M_1, M_2, M_3) = (z + M_1)(z^2 + M_2) + M_3$;
- $\mathsf{BRW}_z(M_1, M_2, \ldots, M_\ell) = (z^k + M_k) \times \mathsf{BRW}_z(M_1, \ldots, M_{k-1}) + \mathsf{BRW}_z(M_{k+1}, \ldots, M_\ell)$;
  if $k \in \{4, 8, 16, 32, \ldots\}$ and $k \leq \ell < 2k$, i.e. k is the largest power of 2 such that $\ell \geq k$.

$\mathsf{BRW}_z(M_1, M_2, \ldots, M_\ell)$ is a polynomial in z whose coefficients are defined from $M_1, \ldots, M_\ell$. We will use the convention that $\mathsf{BRW}_z(M_1, M_2, \ldots, M_\ell)$ denotes a polynomial in the indeterminate z while $\mathsf{BRW}_\tau(M_1, M_2, \ldots, M_\ell)$ is a field element obtained by substituting the field element τ for z in $\mathsf{BRW}_z(M_1, M_2, \ldots, M_\ell)$. We consider the following problem:

$$\text{Given } \tau, M_1, \ldots, M_\ell \in \mathbb{F}_{p^n} \text{ compute } \mathsf{BRW}_\tau(M_1, M_2, \ldots, M_\ell).$$

Informally, we will say that the evaluation of $\mathsf{BRW}_\tau(M_1, M_2, \ldots, M_\ell)$ is an *ℓ-block* BRW *computation.*

The following facts have been proven in [15].

1. For $\ell \geq 3$, $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ can be computed using $\lfloor \ell/2 \rfloor$ field multiplications and $\lfloor \lg \ell \rfloor$ additional field squarings to compute $\tau^2, \tau^4, \ldots$.

2. *Let $d(\ell)$ denote the degree of* $\mathsf{BRW}_z(M_1, \ldots, M_\ell)$. For $\ell \geq 3$, $d(\ell) = 2^{\lfloor \lg \ell \rfloor + 1} - 1$ and so $d(\ell) \leq 2\ell - 1$; the bound is achieved if and only if $\ell = 2^{\mathfrak{a}}$; and $d(\ell) = \ell$ if and only if $\ell = 2^{\mathfrak{a}} - 1$; for some integer $\mathfrak{a} \geq 2$.

3. The map from $(\mathbb{F}_{p^n})^\ell$ to $\mathbb{F}_{p^n}[z]$ given by $(M_1, \ldots, M_\ell) \longmapsto \mathsf{BRW}_z(M_1, \ldots, M_\ell)$ is injective. Consequently, for $(M_1, \ldots, M_\ell), (M'_1, \ldots, M'_\ell) \in (\mathbb{F}_{p^n})^\ell$, $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$, and a uniform random τ from $\mathbb{F}_{p^n}$,

$$\Pr[\mathsf{BRW}_\tau(M_1, \ldots, M_\ell) = \mathsf{BRW}_\tau(M'_1, \ldots, M'_\ell)] \;\leq\; \frac{d(\ell)}{p^n} \leq \frac{2\ell - 1}{p^n}. \tag{4.2}$$

It follows from (4.2) that for a fixed value of ℓ, $\{\mathsf{BRW}_\tau\}$ is $((2\ell-1)/p^n)$-AU. In a similar manner, it can be shown that for a fixed value of ℓ, the family $\{\tau \cdot \mathsf{BRW}_\tau\}$ is $(2\ell/p^n)$-AXU. In Section 4.4 we show how to build an AXU family from BRW polynomials whose domain is the set of all bit strings of lengths less than 2^n.

4.2 Algorithm

The definition of BRW polynomials is recursive. It is easy to write a recursive program which takes as inputs τ and $M_1, \ldots, M_\ell$, $\ell \geq 0$ and produces as output $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$. The function will make two calls to itself on inputs of smaller sizes leading to a binary recursion tree. Such a recursive program, however, will have substantial overhead of stack maintenance and will not lead to a fast implementation. Due to this reason, we do not consider a recursive implementation of BRW.

Suppose ℓ is a fixed integer. For $\ell = 1, 2$ or 3, the evaluation of $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ is given by a simple formula. For $\ell = 4, 5, 6, 7$ and 8, the definitions of $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ are the following:

$$\left.\begin{array}{rcl}
\mathsf{BRW}_\tau(M_1, \ldots, M_4) &=& (\tau^4 + M_4) \times \mathsf{BRW}_\tau(M_1, M_2, M_3); \\
\mathsf{BRW}_\tau(M_1, \ldots, M_5) &=& (\tau^4 + M_4) \times \mathsf{BRW}_\tau(M_1, M_2, M_3) + M_5; \\
\mathsf{BRW}_\tau(M_1, \ldots, M_6) &=& (\tau^4 + M_4) \times \mathsf{BRW}_\tau(M_1, M_2, M_3) + M_5\tau + M_6; \\
\mathsf{BRW}_\tau(M_1, \ldots, M_7) &=& (\tau^4 + M_4) \times \mathsf{BRW}_\tau(M_1, M_2, M_3) + (\tau + M_5)(\tau^2 + M_6) + M_7; \\
\mathsf{BRW}_\tau(M_1, \ldots, M_8) &=& (\tau^8 + M_8) \times \mathsf{BRW}_\tau(M_1, \ldots, M_7).
\end{array}\right\} \tag{4.3}$$

Again, it is easy to write a sequence of field operations (additions and multiplications) to evaluate $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ in each of the above cases. Continuing, this process can be carried out for any ℓ. This process, however, is not general as separate code is required for each value of ℓ. In [30], implementations of this approach over $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ have been reported for $\ell = 1, \ldots, 31$.

The goal is to obtain a non-recursive algorithm to evaluate $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ which works for any ℓ. Suppose the input blocks are $M_1, \ldots, M_\ell$. From a practical point of view as well as for an efficient implementation, it is desirable to process the blocks in a left-to-right manner. A typical left-to-right algorithm would maintain a partial result X obtained by processing blocks $M_1, \ldots, M_i$ and would read the next block M_{i+1} and update X. We

discuss the difficulties faced when trying to use this approach to design an algorithm to evaluate BRW.

Suppose that i blocks have been processed and $i \equiv 0 \bmod 4$ and the partial result computed so far is X. If there are exactly $i+1$ blocks to be processed, then the final result is $X + M_{i+1}$; if there are exactly $i+2$ blocks to be processed, then the final result is $X + M_{i+1}\tau + M_{i+2}$; and if there are exactly $i+3$ blocks to be processed, then the final result is $X + (\tau + M_{i+1})(\tau^2 + M_{i+2}) + M_{i+3}$. This shows that reading only the next unprocessed block, i.e., one block look-ahead is not sufficient. So, the question arises as to how many look-ahead blocks are needed?

There is another difficulty. Suppose, there are exactly $i+4$ blocks to be processed. Since i is a multiple of 4, so is $i+4$. If $i+4$ is a power of two, then the final result is $(X + (M_{i+1} + \tau)(M_{i+2} + \tau^2) + M_{i+3})(M_{i+4} + \tau^{i+4})$. On the other hand, if $i+4$ is not a power of two, then to obtain the final result, $(M_{i+4} + \tau^{i+4})$ is not to be multiplied with $(X + (M_{i+1} + \tau)(M_{i+2} + \tau^2) + M_{i+3})$; instead, it has to be multiplied with the result Y corresponding to the last $(i+4-k)$ blocks where $k \in \{4, 8, 16, 32, \cdots\}$ and $k \leq i+4 < 2k$. Since only X is kept as the partial result of processing the blocks $M_1, \ldots, M_i$, Y is not available. This indicates that more than one partial result has to be maintained. The corresponding question is how many partial results need to be maintained and how are these to be updated?

We develop a non-recursive algorithm to compute $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ for any $\ell \geq 1$. The number of look-ahead blocks can be $2^{\mathfrak{t}}$ for any $\mathfrak{t} \geq 2$. Relevant partial results are stored in an array. The subtlety of the algorithm arises from the maintenance and updation of the partial results. In particular, we note that the number of partial results to be stored is not monotonically increasing with ℓ.

Evaluation of BRW polynomials requires field multiplications. As discussed in Section 8.1, a field multiplication is a composition of an unreducedMult operation followed by a reduce operation. The number of unreducedMult operations required in evaluating BRW is necessarily equal to the number of field multiplications. On the other hand, it is possible to reduce the number of reduce operations. To be able to do this, we define a modification of BRW polynomials where the final result is not reduced.

- $\mathsf{unreducedBRW}_z() = 0$;
- $\mathsf{unreducedBRW}_z(M_1) = M_1$;
- $\mathsf{unreducedBRW}_z(M_1, M_2) = \mathsf{unreducedMult}(M_1, z) + M_2$;
- $\mathsf{unreducedBRW}_z(M_1, M_2, M_3) = \mathsf{unreducedMult}((z + M_1), (z^2 + M_2)) + M_3$;
- $\mathsf{unreducedBRW}_z(M_1, M_2, \ldots, M_k)$
  $= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}_z(M_1, \ldots, M_{k-1})), (z^k + M_k))$,
  if $k \in \{4, 8, 16, 32, \ldots\}$;
- $\mathsf{unreducedBRW}_z(M_1, M_2, \ldots, M_\ell)$
  $= \mathsf{unreducedBRW}_z(M_1, \ldots, M_k) + \mathsf{unreducedBRW}_z(M_{k+1}, \ldots, M_\ell)$,
  if $k \in \{4, 8, 16, 32, \ldots\}$ and $k < \ell < 2k$.

EvalBRW given in Algorithm 1 shows how to compute $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ for any $\ell \geq 1$. The following parameters and data structures are used in the algorithm.

t: an integer ≥ 2 which is a parameter to the algorithm;
$\mathsf{isDef}[0, \ldots]$: a bit array;
$\mathsf{res}[0, \ldots]$: an array where partial results are stored;
$\mathsf{keyPow}[0, \ldots]$: the j-th location stores τ^{2^j}.

The interpretation of the two arrays isDef and res is as follows: $\mathsf{isDef}[j] = 1$ if and only if $\mathsf{res}[j]$ holds a valid partial result.

The array isDef can be implemented using a b-bit unsigned integer: the initialisation in Steps 8 to 13 can be done simply as $\mathsf{isDef} \leftarrow 0$; the value of the j-th position can be obtained as $((\mathsf{isDef} \gg j)$ `and` $1)$ (required in Steps 10 and 30); the value of the j-th bit can be set to 1 using $\mathsf{isDef} \leftarrow (\mathsf{isDef}$ `or` $(1 \ll j))$ (required in Step 22); the j least significant bits of isDef can be set to 0 using $\mathsf{isDef} \leftarrow (\mathsf{isDef}$ `and` $(1^b \ll j))$ (required in Steps 15 to 17).

At Step 8, EvalBRW calls unreducedBRW on $2^{\mathsf{t}} - 1$ blocks while at Step 28, EvalBRW calls unreducedBRW on η blocks where $0 \leq \eta < 2^{\mathsf{t}}$. The algorithm assumes that there is a separate subroutine which returns the evaluation of unreducedBRW on η blocks for $0 \leq \eta < 2^{\mathsf{t}}$. Since t is a fixed parameter of the algorithm, the computation of unreducedBRW on η blocks can be done by a fixed sequence of field operations without any loop or branch statement (essentially as a straight line program). For $\eta = 1, 2, 3$, the definition of $\mathsf{unreducedBRW}_\tau(X_1, \ldots, X_\eta)$ is simple and the code to directly evaluate the expression is also quite simple. If $\mathsf{t} = 2$, then the code to compute unreducedBRW on η blocks for $\eta = 0, 1, 2$ and 3 is sufficient. For $\mathsf{t} \geq 3$, similar code for direct computation of unreducedBRW on η blocks can be worked out from the definition. For $\mathbb{F}_{2^n}$ with $n = 128$ or $n = 256$, such implementations have been reported in [30] for $\mathsf{t} = 5$ and correspondingly $1 \leq \eta \leq 31$.

For $\ell \geq 4$ and $\ell \equiv 0 \bmod 4$, evaluating $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ requires τ and τ^2 along with various other powers of τ. These powers appear in a particular sequence as shown in the following figure.

ℓ	powers of τ
4	4
8	$4, 8$
12	$4, 8, 4$
16	$4, 8, 4, 16$
20	$4, 8, 4, 16, 4$
24	$4, 8, 4, 16, 4, 8$
28	$4, 8, 4, 16, 4, 8, 4$
32	$4, 8, 4, 16, 4, 8, 4, 32$
36	$4, 8, 4, 16, 4, 8, 4, 32, 4$
40	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8$
44	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4$
48	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4, 16$
52	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4, 16, 4$
56	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4, 16, 4, 8$
60	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4, 16, 4, 8, 4$
64	$4, 8, 4, 16, 4, 8, 4, 32, 4, 8, 4, 16, 4, 8, 4, 64$

Algorithm 1 Evaluation of $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$, $\ell \geq 1$.

```
1: function EvalBRW(τ, M_1, ..., M_ℓ)
2:     keyPow[0] ← τ;
3:     if ℓ > 2 then
4:         for j = 1 to ⌊lg ℓ⌋ do
5:             keyPow[j] ← keyPow[j − 1]^2;
6:         end for;
7:     end if;
8:     isDef[0] ← 0;
9:     if ℓ ≥ 2^t then
10:        for j = 1 to ⌊lg ℓ⌋ − t + 1 do
11:            isDef[j] ← 0;
12:        end for;
13:    end if;
14:    for i = 1 to ⌊ℓ/2^t⌋ do
15:        res[0] ← unreducedBRW_τ(M_{2^t·i−(2^t−1)}, ..., M_{2^t·i−1});
16:        j ← 1; tmp ← res[0];
17:        while (isDef[j] = 1) do
18:            tmp ← tmp + res[j];
19:            j ← j + 1;
20:        end while;
21:        res[j] ← unreducedMult(reduce(tmp), M_{2^t·i} + keyPow[j + t − 1]);
22:        isDef[j] ← 1;
23:        for k = 0 to j − 1 do
24:            isDef[k] ← 0;
25:        end for;
26:    end for;
27:    r = ℓ mod 2^t;
28:    tmp ← unreducedBRW_τ(M_{ℓ−r+1}, ..., M_ℓ);
29:    for j = 1 to ⌊lg ℓ⌋ − t + 1 do
30:        if isDef[j] = 1 then
31:            tmp ← tmp + res[j];
32:        end if;
33:    end for;
34:    return reduce(tmp);
35: end function.
```

The diagonal of the above array shows a regular structure[1]. Further, each row is equal to the prefix of the diagonal upto that row. It may be helpful to keep the above picture in mind while going through Algorithm EvalBRW.

Example: We provide an example of the execution of Algorithm EvalBRW. The purpose is to show how the array res is used to store partial results and how it is updated. For simplicity of understanding the basic flow of the algorithm, in the example, we ignore the efficiency issue of unreduced multiplications and instead consider all multiplications (and BRW computations) to be reduced. Let $\mathfrak{t} = 2$ and $\ell = 16$. Then the powers of τ that are required are τ, τ^2, τ^4, τ^8 and τ^{16}. These powers are computed in Steps 2 to 7 and stored in appropriate locations of keyPow. Steps 8 to 13 initialise the required locations of the array isDef to 0. With $\ell = 16$ and $\mathfrak{t} = 2$, the loop in Steps 14 to 26 runs for $i = 1, \ldots, 4$. The value of r computed at Step 27 is 0 and so Steps 28 to 33 have no effect on the final result. So, the main computation is done in Steps 14 to 26. These steps successively compute partial results and store them in various locations in res and also set the corresponding locations of isDef to 1. The details are as follows.

i	operations	Step(s)
1	$\mathsf{res}[0] \leftarrow \mathsf{BRW}_\tau(M_1, M_2, M_3)$	15
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	16
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	17-20
	$\mathsf{res}[1] \leftarrow \mathsf{tmp} \cdot (\tau^4 + M_4)$	21
	$\mathsf{isDef}[1] \leftarrow 1$	22
	$\mathsf{isDef}[0] \leftarrow 0$	23-25
2	$\mathsf{res}[0] \leftarrow \mathsf{BRW}_\tau(M_5, M_6, M_7)$	15
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	16
	$\mathsf{tmp} \leftarrow \mathsf{res}[0] + \mathsf{res}[1]$	17-20
	$\mathsf{res}[2] \leftarrow \mathsf{tmp} \cdot (\tau^8 + M_8)$	21
	$\mathsf{isDef}[2] \leftarrow 1$	22
	$\mathsf{isDef}[0] \leftarrow 0,\ \mathsf{isDef}[1] \leftarrow 0$	23-25
3	$\mathsf{res}[0] \leftarrow \mathsf{BRW}_\tau(M_9, M_{10}, M_{11})$	15
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	16
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	17-20
	$\mathsf{res}[1] \leftarrow \mathsf{tmp} \cdot (\tau^4 + M_{12})$	21
	$\mathsf{isDef}[1] \leftarrow 1$	22
	$\mathsf{isDef}[0] \leftarrow 0$	23-25
4	$\mathsf{res}[0] \leftarrow \mathsf{BRW}_\tau(M_{13}, M_{14}, M_{15})$	15
	$\mathsf{tmp} \leftarrow \mathsf{res}[0]$	16
	$\mathsf{tmp} \leftarrow \mathsf{res}[0] + \mathsf{res}[1] + \mathsf{res}[2]$	17-20
	$\mathsf{res}[3] \leftarrow \mathsf{tmp} \cdot (\tau^{16} + M_{16})$	21
	$\mathsf{isDef}[3] \leftarrow 1$	22
	$\mathsf{isDef}[0] \leftarrow 0,\ \mathsf{isDef}[1] \leftarrow 0,\ \mathsf{isDef}[2] \leftarrow 0$	23-25

[1]One of the reviewers has observed that this forms a fractal.

The various assignments to different locations of res have the following effects.

- For $i = 1$, the assignment $\mathsf{res}[1] \leftarrow \mathsf{tmp} \cdot (\tau^4 + M_4)$ ensures that $\mathsf{res}[1]$ stores $\mathsf{BRW}_\tau(M_1, \ldots, M_4)$.
- For $i = 2$, the assignment $\mathsf{tmp} \leftarrow \mathsf{res}[0] + \mathsf{res}[1]$ ensures that tmp stores $\mathsf{BRW}_\tau(M_5, M_6, M_7) + \mathsf{BRW}_\tau(M_1, \ldots, M_4)$. Subsequently, the assignment $\mathsf{res}[2] \leftarrow \mathsf{tmp} \cdot (\tau^8 + M_8)$ ensures that $\mathsf{res}[2]$ stores $\mathsf{BRW}_\tau(M_1, \ldots, M_8)$.
- For $i = 3$, the assignment $\mathsf{res}[1] \leftarrow \mathsf{tmp} \cdot (\tau^4 + M_{12})$ ensures that $\mathsf{res}[1]$ stores $\mathsf{BRW}_\tau(M_9, \ldots, M_{12})$.
- For $i = 4$, the assignment $\mathsf{tmp} \leftarrow \mathsf{res}[0] + \mathsf{res}[1] + \mathsf{res}[2]$ ensures that tmp stores $\mathsf{BRW}_\tau(M_{13}, M_{14}, M_{15}) + \mathsf{BRW}_\tau(M_9, \ldots, M_{12}) + \mathsf{BRW}_\tau(M_1, \ldots, M_8)$. Subsequently, the assignment $\mathsf{res}[3] \leftarrow \mathsf{tmp} \cdot (\tau^{16} + M_{16})$ ensures that $\mathsf{res}[3]$ stores $\mathsf{BRW}_\tau(M_1, \ldots, M_{16})$.

As mentioned earlier, the above explanation does not take into account the efficiency improvements obtained by separating a field multiplication into an unreduced multiplication and a reduction operation. It is only intended to help the reader to get an idea of how the algorithm proceeds with the computation. Detailed proofs of correctness and complexity of the complete algorithm are provided later.

Remarks:

1. At each iteration of the loop from Steps 14 to 26, not all the contents of the array res store useful information. In the above example, for $i = 2$, $\mathsf{res}[2]$ contains useful information, but, $\mathsf{res}[1]$ is undefined (since $\mathsf{isDef}[1] = 0$). Similarly, for $i = 4$, $\mathsf{res}[3]$ contains useful information while $\mathsf{res}[1]$ and $\mathsf{res}[2]$ are undefined. This suggests that it may be possible to obtain a more compressed representation of res which avoids having undefined locations. One of the reviewers has outlined a method which implements res using a stack and tentatively reduces the overall size of res by half. The trade-off is that some additional conditional statements are required which may lead to a somewhat slowdown of the resulting code. A possible future work can explore the details of this idea.

2. In EvalBRW, the number of blocks ℓ is assumed to be known. The value of ℓ is used to determine the maximum value of the loop counter at Step 7, to compute the value of r at Step 27 and in the computation of unreducedBRW at Step 28. It is possible to modify the algorithm to work in the case where the number of blocks is not known at the beginning. The idea is the following. While the buffer is not empty, attempt to read the next 2^{t} blocks from the buffer. If 2^{t} blocks are indeed retrieved, then these blocks are processed in the same manner as in EvalBRW; if less than 2^{t} blocks are retrieved, then these are the last blocks in the buffer and the "wrapping up" procedure is executed in a manner also similar to that of EvalBRW. With this idea, it is straightforward to write out the details of an algorithm which does not require to know the value of ℓ at the outset. Hence, we do not provide an explicit description of such an algorithm.

4.3 Correctness and Complexity

The material in this section is divided into three parts. In the first part, we prove some results on the structure of unreducedBRW. These results are required in the proofs of correctness and complexity of EvalBRW. The second part proves the correctness of EvalBRW while the third part proves the complexity of EvalBRW.

4.3.1 Structural Properties of unreducedBRW

We start with the following simple result.

Lemma 1. *For $\ell \geq 0$,* $\mathsf{BRW}_z(M_1, \ldots, M_\ell) = \mathsf{reduce}(\mathsf{unreducedBRW}_z(M_1, \ldots, M_\ell))$.

Proof. From the definition of BRW, we obtain the following.

1. For $k \in \{4, 8, 16, 32, \ldots\}$, $\mathsf{BRW}_z(M_1, \ldots, M_k) = (z^k + M_k) \times \mathsf{BRW}_z(M_1, \ldots, M_{k-1})$.
2. For $k \in \{4, 8, 16, 32, \ldots\}$ and $k < \ell < 2k$, $\mathsf{BRW}_z(M_1, \ldots, M_\ell) = \mathsf{BRW}_z(M_1, \ldots, M_k) + \mathsf{BRW}_z(M_{k+1}, \ldots, M_\ell)$.

Using these two facts and (4.1), the result follows from the definition of unreducedBRW by induction on ℓ. □

The next result is more complicated and forms the intuition behind the correctness of EvalBRW.

Lemma 2. *Let $\mathtt{t} \geq 2$ be an integer. For any $\ell \geq 2^{\mathtt{t}}$, write*

$$\left\lfloor \frac{\ell}{2^{\mathtt{t}}} \right\rfloor \;=\; 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}, \tag{4.4}$$

where $k_1, \ldots, k_s$ are integers such that $k_1 > k_2 > \cdots > k_s \geq 0$. Let $K_0 = 0$, $K_1 = 2^{\mathtt{t}+k_1}$, $K_2 = 2^{\mathtt{t}+k_1} + 2^{\mathtt{t}+k_2}, \ldots,$ $K_s = 2^{\mathtt{t}+k_1} + \cdots + 2^{\mathtt{t}+k_s}$. Then

$$\begin{aligned} &\mathsf{unreducedBRW}_z(M_1, \ldots, M_\ell) \\ &\quad = \mathsf{unreducedBRW}_z(M_{K_0+1}, \ldots, M_{K_1}) + \mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_{K_2}) \\ &\qquad + \cdots + \mathsf{unreducedBRW}_z(M_{K_s+1}, \ldots, M_\ell). \end{aligned} \tag{4.5}$$

Proof. Let $r = \ell \bmod 2^{\mathtt{t}}$. For $1 \leq i \leq s$, $K_i - K_{i-1} = 2^{\mathtt{t}+k_i}$ and $\ell - K_s = \ell - 2^{\mathtt{t}}(2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}) = r$. So, the first s terms on the right hand side of (4.5) consist of unreducedBRW on $2^{\mathtt{t}+k_1}, 2^{\mathtt{t}+k_2}, \ldots, 2^{\mathtt{t}+k_s}$ blocks respectively while the last term on the right hand side of (4.5) consists of unreducedBRW on r blocks. If $r = 0$, then the last term is not present. As a result, the number of terms on the right hand side of (4.5) equals s or $s + 1$ depending on whether $2^{\mathtt{t}}$ divides ℓ or not.

The proof of (4.5) is by induction on $s \geq 1$.

Base step: For $s = 1$, $\lfloor \ell/2^{\mathfrak{t}} \rfloor = 2^{k_1}$ and $K_1 = 2^{\mathfrak{t}+k_1}$. So, $2^{\mathfrak{t}+k_1} \leq \ell < 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}} \leq 2^{\mathfrak{t}+k_1+1}$ and we can write

$$\begin{aligned}
&\mathsf{unreducedBRW}_z(M_1, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_z(M_1, \ldots, M_{2^{\mathfrak{t}+k_1}}) + \mathsf{unreducedBRW}_z(M_{2^{\mathfrak{t}+k_1}+1}, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_z(M_{K_0+1}, \ldots, M_{K_1}) + \mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_\ell).
\end{aligned}$$

This proves the base case.

Induction step: Fix $s > 1$ and suppose that (4.5) holds for all $\ell \geq 2^{\mathfrak{t}}$ such that $\lfloor \ell/2^{\mathfrak{t}} \rfloor$ is the sum of $s-1$ powers of two.

Now consider ℓ such that $\lfloor \ell/2^{\mathfrak{t}} \rfloor = 2^{k_1} + \cdots + 2^{k_s}$ with $k_1 > k_2 > \cdots > k_s \geq 0$. Since $s > 1$, $\ell > 2^{\mathfrak{t}+k_1} = K_1$. From $k_1 > k_2 > \cdots > k_s \geq 0$, it follows that $k_i \leq k_1 - i + 1$ for $i \geq 2$; and $k_1 \geq s - 1$. So,

$$\begin{aligned}
\ell &= 2^{\mathfrak{t}}(2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}) + r \\
&< 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}+k_2} + 2^{\mathfrak{t}+k_3} + \cdots + 2^{\mathfrak{t}+k_s} + 2^{\mathfrak{t}} \qquad (\text{as } r = \ell \bmod 2^{\mathfrak{t}}) \\
&\leq 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}+k_1-1} + 2^{\mathfrak{t}+k_1-2} + \cdots + 2^{\mathfrak{t}+k_1-(s-1)} + 2^{\mathfrak{t}} \\
&= 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}+k_1-s+1}(2^{s-2} + 2^{s-3} + \cdots + 2^1 + 1) + 2^{\mathfrak{t}} \\
&= 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}+k_1-s+1}(2^{s-1} - 1) + 2^{\mathfrak{t}} \\
&\leq 2^{\mathfrak{t}+k_1} + 2^{\mathfrak{t}+k_1} - 2^{\mathfrak{t}} + 2^{\mathfrak{t}} \qquad (\text{as } k_1 - s + 1 \geq 0) \\
&= 2^{\mathfrak{t}+k_1+1}.
\end{aligned}$$

So, we have $K_1 = 2^{\mathfrak{t}+k_1} < \ell < 2^{\mathfrak{t}+k_1+1} = 2K_1$. Since $\mathfrak{t} \geq 2$ and $\ell \geq 2^{\mathfrak{t}} \geq 4$, it follows from the definition of $\mathsf{unreducedBRW}$ that

$$\begin{aligned}
&\mathsf{unreducedBRW}_z(M_1, M_2, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_z(M_1, \ldots, M_{K_1}) + \mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_\ell). \qquad (4.6)
\end{aligned}$$

Let $\ell' = \ell - K_1 = 2^{\mathfrak{t}+k_2} + \cdots + 2^{\mathfrak{t}+k_s} + r$ and note that $(M_{K_1+1}, \ldots, M_\ell)$ consists of ℓ' blocks. Also, $\lfloor \ell'/2^{\mathfrak{t}} \rfloor = 2^{k_2} + \cdots + 2^{k_s}$, i.e., $\lfloor \ell'/2^{\mathfrak{t}} \rfloor$ can be written as sum of $s-1$ powers of two. Since $s > 1$, we have $s - 1 \geq 1$ which implies that $\ell' \geq 2^{\mathfrak{t}}$. So, we can apply the induction hypothesis to $\mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_\ell)$ to obtain

$$\begin{aligned}
&\mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_z(M_{K_1+1}, \ldots, M_{K_2}) + \cdots + \mathsf{unreducedBRW}_z(M_{K_s+1}, \ldots, M_\ell). \qquad (4.7)
\end{aligned}$$

Combining (4.7) with (4.6) gives the desired result. □

The next result determines the number of $\mathsf{unreducedMult}$ and reduce operations required in the evaluation of $\mathsf{unreducedBRW}$. These counts are independent of $\mathsf{EvalBRW}$ and are obtained from the recursive definition of $\mathsf{unreducedBRW}$.

Lemma 3. *For $\ell \geq 1$, evaluating $\mathsf{unreducedBRW}_\tau(M_1, \ldots, M_\ell)$ requires $\lfloor \ell/2 \rfloor$ $\mathsf{unreducedMult}$ operations and $\lfloor \ell/4 \rfloor$ reduce operations. Additionally, for $\ell > 2$, $\lfloor \lg \ell \rfloor$ squarings are required to compute the relevant powers of τ.*

Proof. From the definition of unreducedBRW, the statement is clearly true for $\ell = 1, 2, 3$.

For $\ell \geq 4$, the proof follows by induction on ℓ.

If $\ell = 2^{\mathfrak{t}}$, then from the definition of unreducedBRW the number of unreducedMult (resp. reduce) operations is $1 + \lfloor (2^{\mathfrak{t}} - 1)/2 \rfloor = 2^{\mathfrak{t}-1}$ (resp. $1 + \lfloor (2^{\mathfrak{t}} - 1)/4 \rfloor = 2^{\mathfrak{t}-2}$).

If ℓ is not a power of two, then ℓ can be written as $\ell = 2^{\mathfrak{t}} + \ell_1$ with $\ell_1 < 2^{\mathfrak{t}}$. In this case, from the definition of unreducedBRW the number of unreducedMult (resp. reduce) operations is $2^{\mathfrak{t}-1} + \lfloor \ell_1/2 \rfloor = \lfloor \ell/2 \rfloor$ (resp. $2^{\mathfrak{t}-2} + \lfloor \ell_1/4 \rfloor = \lfloor \ell/4 \rfloor$). □

4.3.2 Correctness of EvalBRW

For the correctness proof of EvalBRW some preliminary results are required.

Lemma 4. *For $\ell > 2$, Steps 3 to 7 of* EvalBRW *ensure that* $\mathsf{keyPow}[j] = \tau^{2^j}$ *for* $j = 1, \ldots, \lfloor \lg \ell \rfloor$.

Lemma 5. *Let $\mathfrak{t} \geq 2$ and $\ell \geq 2^{\mathfrak{t}}$. Let the loop counter $i \in \{1, \ldots, i_{\max}\}$, with $i_{\max} = \lfloor \ell/2^{\mathfrak{t}} \rfloor$, in Step 7 of* EvalBRW *be written as*

$$i \;=\; 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \tag{4.8}$$

where $k_{i,1} > k_{i,2} > \cdots > k_{i,s_i} \geq 0$. Let $K_{i,0} = 0$, $K_{i,1} = 2^{\mathfrak{t}+k_{i,1}}$, $K_{i,2} = 2^{\mathfrak{t}+k_{i,1}} + 2^{\mathfrak{t}+k_{i,2}}, \ldots, K_{i,s_i} = 2^{\mathfrak{t}+k_{i,1}} + \cdots + 2^{\mathfrak{t}+k_{i,s_i}}$. After i iterations of the loop given by Steps 7 to 26, the following properties hold. For $l \in \{1, \ldots, s_i\}$,

$$\begin{aligned} \mathsf{isDef}[j] &= \begin{cases} 1 & \text{if } j = 1 + k_{i,l}; \\ 0 & \text{otherwise.} \end{cases} \\ \mathsf{res}[1 + k_{i,l}] &= \mathsf{unreducedBRW}_{\tau}(M_{K_{i,l-1}+1}, \ldots, M_{K_{i,l}}). \end{aligned}$$

Proof. The proof is by induction on $i \geq 1$.

Base step: For $i = 1$, $s_i = 1$, $k_{i,1} = 0$ and $K_{i,1} = 2^{\mathfrak{t}}$. The entries of the array isDef are set to 0 before Step 7. So, the **while** loop given by Steps 10 to 13 is not executed and in Step 22, j has the value it was assigned in Step 9 which is 1. As a result, $\mathsf{isDef}[1] = \mathsf{isDef}[1 + k_{i,1}]$ is set to 1 in Step 22 and all other entries of isDef remain 0.

In Step 8, res[0] is set to $\mathsf{unreducedBRW}_{\tau}(M_1, \cdots, M_{2^{\mathfrak{t}}-1})$ and in Step 9, tmp is set to res[0]. In Step 14, the value of j is 1 and $\mathsf{res}[1] = \mathsf{res}[1 + k_{i,1}]$ is set to the value $\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{tmp}), M_{2^{\mathfrak{t}}} + \mathsf{keyPow}[\mathfrak{t}])$. The correctness of this value is seen from the following simple computation.

$$\begin{aligned} &\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{tmp}), M_{2^{\mathfrak{t}}} + \mathsf{keyPow}[\mathfrak{t}]) \\ &\quad= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{res}[0]), M_{2^{\mathfrak{t}}} + \tau^{2^{\mathfrak{t}}}) \quad \text{(from Lemma 4)} \\ &\quad= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}_{\tau}(M_1, \ldots, M_{2^{\mathfrak{t}}-1})), M_{2^{\mathfrak{t}}} + \tau^{2^{\mathfrak{t}}}) \\ &\quad= \mathsf{unreducedBRW}_{\tau}(M_1, \ldots, M_{2^{\mathfrak{t}}}) \quad \text{(from the definition of unreducedBRW)} \\ &\quad= \mathsf{unreducedBRW}_{\tau}(M_{K_{i,0}+1}, \ldots, M_{K_{i,1}}). \end{aligned}$$

Inductive step: Suppose that the lemma holds for $i = 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \geq 1$. We have to show that it holds for $i+1$. Note that

$$\begin{aligned} i+1 &= 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} + 1 \\ &= 2^{k_{i+1,1}} + 2^{k_{i+1,2}} + \cdots + 2^{k_{i+1,s_{i+1}}}. \end{aligned}$$

Below we derive the expressions for $k_{i+1,1}, \ldots, k_{i+1,s_{i+1}}$ in terms of $k_{i,1}, \ldots, k_{i,s_i}$.

By the induction hypothesis, after i iterations, for $l = 1, \ldots, s_i$, $\mathsf{isDef}[j] = 1$ if and only if $j = 1 + k_{i,l}$ and $\mathsf{res}[1 + k_{i,l}] = \mathsf{unreducedBRW}_\tau(M_{K_{i,l-1}+1}, \ldots, M_{K_{i,l}})$.

There are two cases.

Case i is even: In this case $k_{i,s_i} > 0$ and so $s_{i+1} = s_i + 1$, $k_{i+1,1} = k_{i,1}, \ldots, k_{i+1,s_i} = k_{i,s_i}$ and $k_{i+1,s_{i+1}} = 0$ resulting in $K_{i+1,l} = K_{i,l}$ for $l = 1, \ldots, s_i$ and $K_{i+1,s_{i+1}} = K_{i,s_i} + 2^{\mathsf{t}}$.

Since $k_{i,s_i} > 0$, at the end of the i-th iteration, $\mathsf{isDef}[1] = 0$ and so in the $(i+1)$-st iteration, the **while** loop in Steps 10 to 13 is not executed. As a result, in Step 22, $\mathsf{isDef}[1] = \mathsf{isDef}[1 + k_{i+1,s_{i+1}}]$ is set to 1. No other value of isDef is changed. So, the stated conditions on isDef after $i+1$ iterations hold.

By a similar reasoning, at the end of the $(i+1)$st iteration, $\mathsf{res}[1] = \mathsf{res}[1 + k_{i+1,s_{i+1}}]$ gets set to $\mathsf{unreducedBRW}_\tau(M_{K_{i+1,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}})$. No other value of res changes and so the stated conditions on res after $i+1$ iterations hold.

Case i is odd: In this case $k_{i,s_i} = 0$. Let $\xi \in \{1, \ldots, s_i\}$ be the minimum value such that $k_{i,\xi} = s_i - \xi$. Since $k_{i,\xi} > k_{i,\xi+1} > \cdots > k_{i,s_i}$, it follows that for $l = \xi, \ldots, s_i - 1$, $k_{i,l} = s_i - l$. Further, if $\xi > 1$, then $k_{i,\xi-1} > k_{i,\xi} + 1 = s_i - \xi + 1$. So,

$$i = 2^{k_{i,1}} + \cdots + 2^{k_{i,\xi-1}} + 2^{s_i-\xi+1} - 1 \quad \text{and} \quad i+1 = 2^{k_{i,1}} + \cdots + 2^{k_{i,\xi-1}} + 2^{s_i-\xi+1}.$$

Consequently, $s_{i+1} = \xi$ and $k_{i+1,l} = k_{i,l}$ for $l = 1, \ldots, \xi - 1$ and $k_{i+1,\xi} = s_i - \xi + 1$.

From the induction hypothesis, after the ith iteration, $\mathsf{isDef}[1 + k_{i,l}] = 1$ for $l = 1, \ldots, s_i$ and 0 elsewhere. This in particular means that $\mathsf{isDef}[1] = \mathsf{isDef}[2] = \cdots = \mathsf{isDef}[1+s_i-\xi] = 1$ and $\mathsf{isDef}[2 + s_i - \xi] = 0$ after the ith iteration. So, during the $(i+1)$st iteration, at the end of the **while** loop given by Steps 10 to 13, the value of j is $2 + s_i - \xi$. This results in setting $\mathsf{isDef}[1 + k_{i+1,\xi}] = \mathsf{isDef}[2 + s_i - \xi] = 1$. The **for** loop given by Steps 15 to 17 results in setting the values of $\mathsf{isDef}[0], \ldots, \mathsf{isDef}[1 + s_i - \xi]$ to 0. No other value of isDef is changed. As a result, at the end of the $(i+1)$st iteration, $\mathsf{isDef}[1 + k_{i+1,l}] = \mathsf{isDef}[1 + k_{i,l}] = 1$ for $l = 1, \ldots, \xi - 1$; $\mathsf{isDef}[1 + k_{i+1,\xi}] = 1$ and all other positions of isDef contain 0. This shows that the stated conditions on isDef after $i+1$ iterations hold.

From the values of $k_{i+1,1}$ to $k_{i+1,\xi}$, it follows that $K_{i+1,0} = K_{i,0} = 0, K_{i+1,1} = K_{i,1}, \ldots, K_{i+1,\xi-1} = K_{i,\xi-1}$ and $K_{i+1,\xi} = K_{i+1,\xi-1} + 2^{s_i-\xi+\mathsf{t}+1}$. As a result, for $l = 1, \ldots, \xi - 1$,

$$\begin{aligned} \mathsf{res}[1 + k_{i+1,l}] &= \mathsf{res}[1 + k_{i,l}] \\ &= \mathsf{unreducedBRW}_\tau(M_{K_{i,l-1}+1}, \ldots, M_{K_{i,l}}) \\ &= \mathsf{unreducedBRW}_\tau(M_{K_{i+1,l-1}+1}, \ldots, M_{K_{i+1,l}}). \end{aligned}$$

So, we only need to argue that $\mathsf{res}[1+k_{i+1,\xi}]$ contains $\mathsf{unreducedBRW}_\tau(M_{K_{i+1,\xi-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}})$ which is an application of $\mathsf{unreducedBRW}$ on $2^{s_i-\xi+\mathsf{t}+1}$ blocks.

Since the total number of blocks processed up to and including the ith iteration is $i \cdot 2^{\mathsf{t}}$, we have $K_{i,s_i} = i \cdot 2^{\mathsf{t}}$ and similarly, $K_{i+1,s_{i+1}} = (i+1)2^{\mathsf{t}}$. After Step 8, in $(i+1)$st iteration,

$$\begin{aligned} \mathsf{res}[0] &= \mathsf{unreducedBRW}_\tau(M_{2^{\mathsf{t}} \cdot i+1}, \ldots, M_{2^{\mathsf{t}}(i+1)-1}) \\ &= \mathsf{unreducedBRW}_\tau(M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}). \end{aligned}$$

Note that

$$\begin{aligned} K_{i,\xi} &= K_{i,\xi-1} + 2^{\mathsf{t}+k_{i,\xi}} = K_{i,\xi-1} + 2^{\mathsf{t}+s_i-\xi} \\ K_{i,\xi+1} &= K_{i,\xi} + 2^{\mathsf{t}+k_{i,\xi+1}} = K_{i,\xi} + 2^{\mathsf{t}+s_i-\xi-1} \\ \cdots &\ . \ \cdots \\ K_{i,s_i} &= K_{i,s_i-1} + 2^{\mathsf{t}+k_{i,s_i}} = K_{i,s_i-1} + 2^{\mathsf{t}}. \end{aligned}$$

From the induction hypothesis, at the end of the ith step

$$\begin{aligned} \mathsf{res}[1+s_i-\xi] &= \mathsf{res}[1+k_{i,\xi}] = \mathsf{unreducedBRW}_\tau(M_{K_{i,\xi-1}+1}, \ldots, M_{K_{i,\xi}}) \\ \mathsf{res}[1+s_i-\xi-1] &= \mathsf{res}[1+k_{i,\xi+1}] = \mathsf{unreducedBRW}_\tau(M_{K_{i,\xi}+1}, \ldots, M_{K_{i,\xi+1}}) \\ \cdots &\ . \ \cdots \\ \mathsf{res}[1+s_i-s_i] &= \mathsf{res}[1+k_{i,s_i}] = \mathsf{unreducedBRW}_\tau(M_{K_{i,s_i-1}+1}, \ldots, M_{K_{i,s_i}}). \end{aligned}$$

In the $(i+1)$st iteration, at the end of the **while** loop given by Steps 10 to 13, the variable tmp contains the sum $\mathsf{res}[0] + \cdots + \mathsf{res}[1+s_i-\xi]$; in Step 14, $\mathsf{reduce}(\mathsf{tmp})$ is multiplied (without reduction) to $(M_{2^{\mathsf{t}}(i+1)} + \tau^{2^{1+s_i-\xi+\mathsf{t}}})$ and the value is assigned to $\mathsf{res}[2+s_i-\xi]$. We have

$$\begin{aligned} &\mathsf{res}[0] + \mathsf{res}[1] + \cdots + \mathsf{res}[1+s_i-\xi] \\ &\quad = \mathsf{unreducedBRW}_\tau(M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}) \\ &\qquad + \mathsf{unreducedBRW}_\tau(M_{K_{i,s_i-1}+1}, \ldots, M_{K_{i,s_i}}) \\ &\qquad + \cdots + \mathsf{unreducedBRW}_\tau(M_{K_{i,\xi-1}+1}, \ldots, M_{K_{i,\xi}}) \\ &\quad = \mathsf{unreducedBRW}_\tau(M_{K_{i,\xi-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}) \quad \text{(from Lemma 2).} \end{aligned}$$

So, at the end of $(i+1)$st iteration,

$$\begin{aligned} \mathsf{res}[1+k_{i+1,\xi}] &= \mathsf{res}[2+s_i-\xi] \\ &= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{res}[0] + \cdots + \mathsf{res}[1+s_i-\xi]), M_{2^{\mathsf{t}}(i+1)} + \tau^{2^{1+s_i-\xi+\mathsf{t}}}) \\ &= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}_\tau(M_{K_{i,\xi-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1})), \\ &\qquad M_{K_{i+1,s_{i+1}}} + \tau^{2^{1+s_i-\xi+\mathsf{t}}}) \\ &\overset{(\mathfrak{s})}{=} \mathsf{unreducedBRW}_\tau(M_{K_{i,\xi-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}}) \\ &= \mathsf{unreducedBRW}_\tau(M_{K_{i+1,\xi-1}+1}, \ldots, M_{K_{i+1,\xi}}). \end{aligned}$$

The equality in Step $(\mathfrak{s})$ above follows from the definition of $\mathsf{unreducedBRW}$.

This completes the induction step and the proof. □

Next we prove the correctness of EvalBRW.

Theorem 1. *For any* $\mathtt{t} \geq 2$ *and any* $\ell \geq 1$, $\mathsf{EvalBRW}(\tau, M_1, \ldots, M_\ell)$ *correctly computes* $\mathsf{BRW}_\tau(M_1, \cdots, M_\ell)$.

Proof. In Step 34, EvalBRW returns reduce(tmp). From Lemma 1, it follows that it is sufficient to show that tmp in Step 34 is equal to $\mathsf{unreducedBRW}_\tau(M_1, \cdots, M_\ell)$.

If $\ell < 2^{\mathtt{t}}$, then the **for** loop from Step 7 to 26 is not executed. In Step 28, tmp gets assigned to $\mathsf{unreducedBRW}_\tau(M_1, \ldots, M_\ell)$ which remains unchanged till Step 34. This proves the result for $\ell < 2^{\mathtt{t}}$.

So, suppose $\ell \geq 2^{\mathtt{t}}$ and as in Lemma 2, let $\lfloor \ell/2^{\mathtt{t}} \rfloor$ be written as $\lfloor \ell/2^{\mathtt{t}} \rfloor = 2^{k_1} + \cdots + 2^{k_s}$ and $K_0 = 0$, $K_1 = 2^{\mathtt{t}+k_1}$, $K_2 = 2^{\mathtt{t}+k_1} + 2^{\mathtt{t}+k_2}, \ldots,$ $K_s = 2^{\mathtt{t}+k_1} + \cdots + 2^{\mathtt{t}+k_s}$. Let $r = \ell \bmod 2^{\mathtt{t}}$ so that $\ell = 2^{\mathtt{t}}(2^{k_1} + \cdots + 2^{k_s}) + r$ implying $K_s = \ell - r$. From Lemma 2, we can write

$$\begin{aligned}
&\mathsf{unreducedBRW}_\tau(M_1, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_\tau(M_{K_0+1}, \ldots, M_{K_1}) + \cdots + \mathsf{unreducedBRW}_\tau(M_{K_{s-1}+1}, \ldots, M_{K_s}) \\
&\qquad + \mathsf{unreducedBRW}_\tau(M_{K_s+1}, \ldots, M_\ell) \\
&\quad = \mathsf{unreducedBRW}_\tau(M_{K_0+1}, \ldots, M_{K_1}) + \cdots + \mathsf{unreducedBRW}_\tau(M_{K_{s-1}+1}, \ldots, M_{K_s}) \\
&\qquad + \mathsf{unreducedBRW}_\tau(M_{\ell-r+1}, \ldots, M_\ell). \qquad (4.9)
\end{aligned}$$

The loop counter i of the **for** loop in Step 7 runs from 1 to $i_{\max} = \lfloor \ell/2^{\mathtt{t}} \rfloor$. Applying Lemma 5 to $i_{\max}$, we obtain that after $i_{\max}$ iterations, for $l \in \{1, \ldots, s\}$,

$$\begin{aligned}
\mathsf{isDef}[j] &= \begin{cases} 1 & \text{if } j = 1 + k_l; \\ 0 & \text{otherwise.} \end{cases} \\
\mathsf{res}[1 + k_l] &= \mathsf{unreducedBRW}_\tau(M_{K_{l-1}+1}, \ldots, M_{K_l}).
\end{aligned}$$

From $\ell = 2^{\mathtt{t}}(2^{k_1} + \cdots + 2^{k_s}) + r$ and $0 \leq r < 2^{\mathtt{t}}$, we have $2^{\mathtt{t}}(2^{k_1} + \cdots + 2^{k_s}) \leq \ell < 2^{\mathtt{t}}(2^{k_1} + \cdots + 2^{k_s} + 1)$. Since $k_1 > k_2 > \cdots > k_s$, we obtain $2^{\mathtt{t}+k_1} \leq \ell < 2^{\mathtt{t}+k_1+1}$ and so $k_1 = \lfloor \lg \ell \rfloor - \mathtt{t}$. As a result, we get that the maximum positions of isDef and res that are accessed by the algorithm are both equal to $1 + \lfloor \lg \ell \rfloor - \mathtt{t}$. This justifies the upper bound on the loop counter of the **for** loops in Steps 10 and 29.

In Step 28, tmp is initialised to $\mathsf{unreducedBRW}_\tau(M_{\ell-r+1}, \ldots, M_\ell)$. For $j = 1, \ldots, 1 + \lfloor \lg \ell \rfloor - \mathtt{t}$, the **for** loop in Steps 29 to 33 adds res[j] to tmp if and only if isDef[j] $= 1$. As a result, after this **for** loop

$$\begin{aligned}
\mathsf{tmp} &= \mathsf{unreducedBRW}_\tau(M_{K_0+1}, \ldots, M_{K_1}) + \cdots + \mathsf{unreducedBRW}_\tau(M_{K_{s-1}+1}, \ldots, M_{K_s}) \\
&\quad + \mathsf{unreducedBRW}_\tau(M_{\ell-r+1}, \ldots, M_\ell).
\end{aligned}$$

From (4.9), we have that in Step 34, $\mathsf{tmp} = \mathsf{unreducedBRW}_\tau(M_1, \ldots, M_\ell)$ as desired. □

4.3.3 Complexity of EvalBRW

The space complexity of EvalBRW is determined by the maximum sizes of the arrays res, isDef and keyPow (plus a constant number of variables). The sizes of these arrays are determined in the proof of Theorem 1 and we record these in the following result.

Proposition 1. *Let* $\mathsf{t} \geq 2$. *For correctly computing* $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$, *it is sufficient for the arrays* isDef *and* res *to have length* $\lfloor \lg \ell \rfloor - \mathsf{t} + 2$. *Further, for* $\ell > 2$, keyPow *stores* $1 + \lfloor \lg \ell \rfloor$ *elements of* $\mathbb{F}_{p^n}$.

Proof. The proof of Theorem 1 shows that the maximum positions of isDef and res that are accessed are both equal to $\lfloor \lg \ell \rfloor - \mathsf{t} + 1$. Since both the arrays start from position 0, the length is $\lfloor \lg \ell \rfloor - \mathsf{t} + 2$. □

We now consider the time complexity of $\mathsf{EvalBRW}(\tau, M_1, \ldots, M_\ell)$. For this, we separately count the number of unreducedMult and reduce operations that are required.

Theorem 2. *For* $\ell \geq 2$, $\mathsf{EvalBRW}(\tau, M_1, \ldots, M_\ell)$ *requires* $\lfloor \ell/2 \rfloor$ unreducedMult *operations and* $1 + \lfloor \ell/4 \rfloor$ reduce *operations. Additionally, for* $\ell > 2$, $\lfloor \lg \ell \rfloor$ *squarings are required to compute the relevant powers of* τ.

Proof. The final output returned by $\mathsf{EvalBRW}(\tau, M_1, \ldots, M_\ell)$ at Step 34 is reduce(tmp). So, it is sufficient to show that computing tmp up to Step 34 requires $\lfloor \ell/2 \rfloor$ unreducedMult operations and $\lfloor \ell/4 \rfloor$ reduce operations.

The call to unreducedBRW in Step 8 is on $2^{\mathsf{t}} - 1$ blocks while the call to unreducedBRW in Step 28 is on r blocks. From Lemma 3, we have that for any fixed $\mathsf{t} \geq 2$, a straight line code to compute $\mathsf{unreducedBRW}_\tau$ in Step 8 requires $2^{\mathsf{t}-1} - 1$ unreducedMult operations and $2^{\mathsf{t}-2} - 1$ reduce operations. Similarly, the call to $\mathsf{unreducedBRW}_\tau$ in Step 28 requires $\lfloor r/2 \rfloor$ unreducedMult operations and $\lfloor r/4 \rfloor$ reduce operations.

Let $i_{\max} = \lfloor \ell/2^{\mathsf{t}} \rfloor$. The counter of the **for** loop in Step 7 runs up to $i_{\max}$. Then $\ell = i_{\max} \cdot 2^{\mathsf{t}} + r$, where $r = \ell \bmod 2^{\mathsf{t}}$ is computed in Step 27. Since $\mathsf{t} \geq 2$, we have $(\ell - r)/2 = i_{\max} \cdot 2^{\mathsf{t}-1}$ and $(\ell - r)/4 = i_{\max} \cdot 2^{\mathsf{t}-2}$ to be both integers.

First consider the number of unreducedMult operations required for computing tmp at Step 34. Each iteration of the **for** loop given by Steps 7 to 26 makes one call to unreducedBRW on $2^{\mathsf{t}} - 1$ blocks and one unreducedMult operation. The call to unreducedBRW operations requires $2^{\mathsf{t}-1} - 1$ unreducedMult operations. So, the number of unreducedMult operations required during the entire **for** loop given by Steps 7 to 26 is $i_{\max}(1 + 2^{\mathsf{t}-1} - 1)$. Additionally, $\lfloor r/2 \rfloor$ unreducedMult operations are required by the call to unreducedBRW in Step 28. As a result, the total number of unreducedMult operations required to compute tmp required at Step 34 is

$$i_{\max} \cdot 2^{\mathsf{t}-1} + \lfloor r/2 \rfloor \;=\; (\ell - r)/2 + \lfloor r/2 \rfloor = \lfloor \ell/2 \rfloor.$$

Since $(\ell - r)/2$ is an integer, ℓ and r are either both even or both odd. If both are even, then the last equality is immediate while if they are both odd, then writing $(\ell - r)/2 = (\ell - 1)/2 - (r - 1)/2$ shows the last equality.

For the number of reduce operations required for computing tmp at Step 34, a reasoning similar to above shows that the required number is

$$i_{\max} \cdot 2^{\mathsf{t}-2} + \lfloor r/4 \rfloor \;=\; (\ell - r)/4 + \lfloor r/4 \rfloor = \lfloor \ell/4 \rfloor.$$

In this case, for the last equality, we have to use the fact that $(\ell - r)/4$ is an integer implies that $\ell \equiv r \bmod 4$ so that if $\ell \bmod 4 = a = r \bmod 4$, then $(\ell - r)/4 + \lfloor r/4 \rfloor = (\ell - a)/4 - (r - a)/4 + (r - a)/4 = (\ell - a)/4 = \lfloor \ell/4 \rfloor$. □

In Algorithm EvalBRW, the main intuition for reducing the number of reduce operations is a lazy strategy, i.e. do not reduce immediately after each multiplication and instead apply the reduce operation to an unreduced quantity only when this quantity is an operand in a multiplication operation. Theorem 2 provides the formal argument that for an ℓ-block computation, $1 + \lfloor \ell/4 \rfloor$ reduce operations are sufficient for all values of $\mathsf{t} \geq 2$.

One of the reviewers has suggested the following example to see why $\lfloor \ell/4 \rfloor$ reduce operations are sufficient for ℓ blocks. Suppose $\mathsf{t} = 3$ so that blocks are processed in groups of 8. Let $A_1, \ldots, A_8$ be the next 8 blocks. Let $B_1 = \tau + A_1$, $B_2 = \tau^2 + A_2$, $B_4 = \tau^4 + A_4$, $B_5 = \tau + A_5$ and $B_6 = \tau^2 + A_6$. Then the computation associated with these 8 blocks is either

$$X+((B_1 \cdot B_2+A_3)B_4+B_5 \cdot B_6+A_7)(\tau^8+A_8), \text{ or } (X+(B_1 \cdot B_2+A_3)B_4+B_5 \cdot B_6+A_7)(\tau^{2^k}+A_8),$$

where X is an unreduced partial result and in the second expression τ^{2^k} is an appropriate power of τ. For the above computation, one reduce operation is required for $B_1 \cdot B_2$ and another for the either the quantity $((B_1 \cdot B_2 + A_3)B_4 + B_5 \cdot B_6 + A_7)$ or the quantity $(X + (B_1 \cdot B_2 + A_3)B_4 + B_5 \cdot B_6 + A_7)$ before the multiplication to $(\tau^8 + A_8)$ or $(\tau^{2^k} + A_8)$ respectively. The result is kept in unreduced form. So, for processing 8 blocks, only 2 reduce operations are sufficient.

The role of the parameter t: Note that from Theorem 2, the number of operations required by EvalBRW does not depend on t. The parameter t determines the number of blocks in the unreducedBRW call at Step 8. As mentioned earlier, it is assumed that there is a subroutine for this computation and the subroutine performs this computation as a straight line code without any loop. In other words, the value of t determines the extent of loop unrolling. To some extent, using a greater amount of loop unrolling leads to improved efficiency as indicated by the results of our implementations.

4.4 Design of Hash Function

We propose a hash function based on BRW polynomials. The description uses the terminology $\mathsf{pad}_n(\cdot)$, which is defined as follows.

$\mathsf{pad}_n(X)$: For a binary string X and $n > 0$, if X is the empty string, then $\mathsf{pad}_n(X)$ will denote the string 0^n; while if X is non-empty, then $\mathsf{pad}_n(X)$ will denote $X||0^i$, where $i \geq 0$ is the minimum integer such that n divides $\mathsf{len}(X||0^i)$.

Note that the notations $\mathsf{bin}_n(\cdot)$ and $\mathsf{format}_n(\cdot)$, used in Algorithm 2 are defined in Chapter 3.

The hash function that we define uses BRW computation over $\mathbb{F}_{2^n}$. Using a fixed irreducible polynomial over $GF(2)$ of degree n, it is possible to identify the elements of $\mathbb{F}_{2^n}$ with the elements of $\{0,1\}^n$. In the following, we will implicitly assume this identification.

For a positive integer n, let

$$\mathcal{D} = \bigcup_{i=0}^{2^n-1} \{0,1\}^i. \tag{4.10}$$

We define the hash function $\mathsf{BRW}n$ in the following manner.

$$\mathsf{BRW}n : \{0,1\}^n \times \mathcal{D} \to \{0,1\}^n. \tag{4.11}$$

Here $\mathcal{D}$ is the message space of $\mathsf{BRW}n$. The computation of $\mathsf{BRW}n$ is shown in Algorithm 2.

Algorithm 2 Computation of BRW based hash function.

1: **function** $\mathsf{BRW}n(\tau, X)$
2: $(X_1, \ldots, X_\ell) \leftarrow \mathsf{format}_n(\mathsf{pad}_n(X))$;
3: $Y \leftarrow \mathsf{EvalBRW}(\tau, X_1, \ldots, X_\ell)$;
4: $Z \leftarrow \tau(\tau Y \oplus \mathsf{bin}_n(\mathsf{len}(X)))$;
5: return Z;
6: **end function**.

We will write $\mathsf{BRW}n_\tau(\cdot)$ to denote $\mathsf{BRW}n(\tau, \cdot)$.

The following result shows that the differential probability of $\mathsf{BRW}n$ is small.

Proposition 2. *Let $X, X' \in \mathcal{D}$, $X \neq X'$ and $\alpha \in \mathbb{F}_{2^n}$. Then for a uniform random $\tau \in \mathbb{F}_{2^n}$,*

$$\Pr[\mathsf{BRW}n_\tau(X) \oplus \mathsf{BRW}n_\tau(X') = \alpha] \leq \frac{2\max(\ell, \ell') + 1}{2^n} \tag{4.12}$$

where ℓ (resp. ℓ') denotes the number of blocks in the output of $\mathsf{format}_n(\mathsf{pad}_n(X))$ (resp. $\mathsf{format}_n(\mathsf{pad}_n(X'))$).

Proof. Let $(X_1, \ldots, X_\ell) = \mathsf{format}_n(\mathsf{pad}_n(X))$, $Y = \mathsf{EvalBRW}(\tau, X_1, \ldots, X_\ell)$, $\mathsf{BRW}n_\tau(X) = Z = \tau(\tau Y \oplus \mathsf{bin}_n(\mathsf{len}(X)))$. Similarly, for X', let $(X'_1, \ldots, X'_{\ell'}) = \mathsf{format}_n(\mathsf{pad}_n(X'))$, $Y' = \mathsf{EvalBRW}(\tau, X'_1, \ldots, X'_{\ell'})$ and $\mathsf{BRW}n_\tau(X') = Z' = \tau(\tau Y' \oplus \mathsf{bin}_n(\mathsf{len}(X')))$.

Y (resp. Y') is a polynomial in τ of degree $d(\ell)$ (resp. $d(\ell')$). Consequently, Z (resp. Z') is a polynomial in τ of degree $d(\ell) + 2$ (resp. $d(\ell') + 2$). Assume without loss of generality $\ell \geq \ell'$, so that $\max(\ell, \ell') = \ell$.

Suppose that X and X' are of different lengths. Then the coefficients of τ in Z and Z' are different. Consequently $Z \oplus Z' \oplus \alpha$ is a non-zero polynomial of degree at most $d(\ell) + 2$.

So, suppose that X and X' have the same length. Then $\ell = \ell'$. Since $X \neq X'$, it follows that $(X_1, \ldots, X_\ell) \neq (X'_1, \ldots, X'_\ell)$. By the injectivity of BRW (see Section 8.1), it follows that the polynomials Y and Y' are distinct. Consequently, $Z \oplus Z' \oplus \alpha$ is a non-zero polynomial of degree at most $d(\ell) + 2$.

In both cases, we have $Z \oplus Z' \oplus \alpha$ to be a non-zero polynomial of degree at most $d(\ell) + 2$. The probability that a uniform random τ from $\mathbb{F}_{2^n}$ is a root of this polynomial is at most $(d(\ell) + 2)/2^n \leq (2\ell + 1)/2^n$. The last inequality follows from the fact that $d(\ell) \leq 2\ell - 1$ (see Section 8.1). □

Remark: Proposition 2 guarantees low differential probability of $\{\mathsf{BRW}n_\tau\}$ under the condition that the key τ is chosen uniformly at random from $\mathbb{F}_{2^n}$. As mentioned in previous chapters, one popular method of using an AXU hash function family $\{H_\tau\}_\tau$ to construct

a MAC scheme is the following. Using the key (K, τ), a nonce-message pair (N, M) is mapped to the n-bit tag $F_K(N) \oplus H_\tau(M)$; here let the digest size of $\{H_\tau\}_\tau$ be n-bit and $\{F_K\}_K$ be either a pseudo-random function family or a pseudo-random permutation family, mapping n-bit strings to n-bit strings. Consider this MAC scheme with the hash function instantiated by $\mathsf{BRW}n$, i.e. a nonce-message pair (N, M) is mapped using a key (K, τ) to $F_K(N) \oplus \mathsf{BRW}n_\tau(M)$. Here K and τ are independent and this scheme can be analysed under the condition that τ is uniformly distributed. On the other hand, there could be applications where the hash key τ is obtained as the output of some other cryptographic primitive. In this case, the uniform distribution assumption on τ would not be justified.

The above issue is not particular to $\{\mathsf{BRW}n_\tau\}$ and is relevant for any AXU hash function family. We consider this issue in some more detail. Suppose $\mathfrak{X}$ is some cryptographic functionality built using several components one of which is an AXU family $\{H_\tau\}_\tau$ where the key τ of H_τ is derived using a pseudo-random function (PRF) F_K. A typical reductionist security proof for $\mathfrak{X}$ would provide an upper bound on the advantage of an adversary in defeating the security of $\mathfrak{X}$, in terms of the security bounds of the individual components. The security bound for $\mathfrak{X}$ would consist of at least two components – an appropriate security bound for the PRF F and a bound on the differential probability of H. The security bound for F will cover the adversary's advantage in distinguishing the actual distribution of τ from a uniform random string of the same length. Consequently, this will allow analysing the differential probability of $\{H_\tau\}_\tau$ for a uniform random τ.

4.5 Implementation

We report implementations of BRW128 and BRW256. These require the implementations of EvalBRW over $\mathbb{F}_{2^n}$ in the two cases of $n = 128$ and $n = 256$.

Our target platform for the implementation was the Intel Skylake processor, which supports the Intel Intrinsics instruction set[2]. The instruction of particular interest for our implementation was `pclmulqdq`, which takes as input two polynomials over $GF(2)$ of degrees at most 63 each and returns their product which is a polynomial over $GF(2)$ of degree at most 126. The two input polynomials fit into 64-bit words while the output polynomial fits into a 128-bit word.

A field multiplication in $\mathbb{F}_{2^n}$ consists of a unreducedMult followed by a reduce operation.

1. For each unreducedMult over $\mathbb{F}_{2^{128}}$, we need to compute the polynomial multiplication of two polynomials of degrees at most 127 each. Using the schoolbook method this requires 4 `pclmulqdq` instructions while using Karatsuba's algorithm it requires 3 `pclmulqdq` instructions and some additional XOR instructions. It has been reported in [60] that on the Skylake processor, the schoolbook method is faster and so we have used this method.

2. For each unreducedMult over $\mathbb{F}_{2^{256}}$, we need to compute the polynomial multiplication of two polynomials of degrees at most 255 each. For this, the schoolbook and Karat-

[2] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#`

suba's methods require 16 and 9 `pclmulqdq` instructions respectively. In this case, Karatsuba's method gives better performance [30] and so we have used this method.

For the reduce operation over $\mathbb{F}_{2^{128}}$, following the procedure described in [58] the reduction modulo $\psi(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$ can be done using 2 `pclmulqdq` instructions along with some XORs and shifts. An extension of this procedure [30] for $\mathbb{F}_{2^{256}}$ shows that the reduction modulo $\psi(x) = x^{256} \oplus x^{10} \oplus x^5 \oplus x^2 \oplus 1$ requires 4 `pclmulqdq` instructions along with some XORs and shifts.

EvalBRW has the parameter $\mathfrak{t}$. For the implementation, we have considered $\mathfrak{t} = 2, 3, 4$ and 5. This requires implementation of unreducedBRW as a straight line program for ℓ blocks with $1 \leq \ell \leq 31$. Implementations of BRW for 1 to 31 blocks have been reported in [30]. Here we use these implementations with the modification that the final reduction is not applied so that unreducedBRW is computed instead of BRW.

The code for our implementations of BRW128 and BRW256 is publicly available[3].

4.5.1 Timings

For measuring time, we followed the strategy of [74], which we briefly describe below. On x86 machines, there is a time-stamp counter (TSC) which increments once per CPU cycle. The difference in the values of TSC before and after a group of operations provides the number of cycles required to execute this group of operations. In the present context, the following strategy has been used. Fix a value λ and consider M sets of N messages of length λ bytes. Using the TSC, find the number of cycles required to process each set of N messages. This provides a set of M values and the median of these values is taken. Recording the median avoids outliers which arise from possible system interrupts. Dividing this median by N, represents the number of cycles $\mathfrak{c}$ required to process a message of length λ. Further, dividing $\mathfrak{c}$ by λ, the cycles per byte measure is obtained. Note that before recording timings, the code to be timed is executed a sufficient number of times so that both the code and the data are present in the Level 1 cache of the machine. This ensures that the timing results are not much affected by the time required to access the memory.

The timing measurements were taken on a single core of a machine with Intel Core i7-6500U Skylake @ 2.5GHz. The operating system was 64-bit Ubuntu-14.04-LTS and the C codes were complied using GCC version 4.8.4. The corresponding timing results that were obtained are shown in Tables 4.1 and 4.2. The column headers of the first row provide the message size in bytes. From the second row onwards, the rows are for different values of $\mathfrak{t}$. The entries in these rows are in cycles per byte.

For $n = 128$, we provide the timings of POLYVAL [60] for the purpose of comparison. The code for POLYVAL is essentially a highly optimised implementation of Horner's rule based polynomial hash. In particular, it performs a single reduction for every eight polynomial multiplications and the ordering of the instructions seems to have been done very carefully so as to minimise the cycle counts.

In contrast, we would like to clarify that we do not claim to have provided the best possible implementations of BRW128 and BRW256. We have considered the possible algorithmic

[3] `https://github.com/sebatighosh/BRW`

Table 4.1: Timing results for BRW128 and POLYVAL.

	512	1024	4096	8192	16384	30000
$\mathfrak{t} = 2$	0.819	0.611	0.425	0.388	0.368	0.356
$\mathfrak{t} = 3$	0.826	0.623	0.444	0.407	0.389	0.379
$\mathfrak{t} = 4$	0.787	0.583	0.401	0.364	0.344	0.336
$\mathfrak{t} = 5$	0.776	0.552	0.348	0.309	0.287	0.278
POLYVAL	0.786	0.549	0.376	0.347	0.333	0.328

Table 4.2: Timing results for BRW256.

	512	1024	4096	8192	16384	30000
$\mathfrak{t} = 2$	1.162	0.909	0.675	0.628	0.603	0.587
$\mathfrak{t} = 3$	1.118	0.864	0.629	0.581	0.559	0.539
$\mathfrak{t} = 4$	1.099	0.841	0.607	0.558	0.533	0.519
$\mathfrak{t} = 5$	1.095	0.862	0.619	0.569	0.544	0.529

improvements and the corresponding implementation in Intel intrinsics. For concrete speed-ups one also needs to consider details of instruction level pipelining issues and also possibly carry out an implementation in assembly. Since the main goal of our implementation was to show the practicability of Algorithm EvalBRW, we have not tried to aggresively optimise the code. Future implementation efforts may attempt such work.

4.6 Summary

In this chapter, we have given an efficient non-recursive algorithm for evaluating BRW polynomials. This is the first non-recursive algorithm to evaluate this polynomial for input of any length. As recursive implementation has significant overhead, this algorithm, for the first time gives efficient implementation of BRW polynomials. Besides, this algorithm improves the so far established complexity of evaluating these polynomials by reducing the number of field reductions required by a significant number. Actual implementation of a BRW based hash function where BRW polynomial has been evaluated by this new algorithm shows its performance superiority over a highly efficient implementation of POLYVAL by Gueron in high end Intel processors.

Chapter 5

Hash2L: A Fast Two-Level Universal Hash Function

Two well known universal hash functions based on univariate polynomials are Horner's rule based polynomial hashing and BRW (Bernstein-Rabin-Winograd) polynomial based hash function. Though the computational complexity of BRW polynomial based hashing is significantly smaller than Horner's rule based hashing, implementation of BRW polynomials for variable length messages present significant difficulties.

The definition of BRW polynomial is inherently recursive and the computation for an ℓ-block message requires two recursive calls on messages consisting of smaller number of blocks. In principle, the recursion can be simulated in a bottom-up fashion. The major problematic issue is that even the first recursive call cannot be made unless the length of the whole message is available. The whole message has to be buffered before even the first message block can be processed. A second problem is that at each point of the computation, it is quite complicated to figure out the operands that are to be multiplied. Again, in principle this can be done, but, actually determining the operands requires additional time.

In this chapter, we investigate the possibility of harnessing the speed of BRW polynomial based hashing without the associated difficulties in implementation. To this end, our first observation is that if the number of blocks in a message is a small fixed number, then the above mentioned difficulties disappear. Making effective use of this observation leads us to consider a two-level hash design. Suppose BRW is to be applied when the number of blocks is η. Let us call an η-block message to be a super-block. The input message blocks are divided into super-blocks and BRW is applied to each super-block. The outputs of these BRW calls are then combined using a Horner based hashing.

The number of multiplications required for a message consisting of ℓ super-blocks is about $\ell\eta/2 + \ell - 1$ (the precise count is provided later). Applying Horner to such a message will require $\ell\eta - 1$ multiplications while BRW will require $\ell\eta/2$ multiplications. By choosing a suitable value of η, the number of multiplications required by the new hash function can be made quite close to that of BRW. Such a two-level strategy has the advantage that it avoids the difficulties associated with implementing BRW on variable length messages.

The idea of two-level (or, multi-level) hashing is not new and has been proposed in the literature [106, 89, 99]. Two-level hashing in general requires independent keys for each level. So, applied directly, the hash key will consist of two field elements. For many applications, it is desirable to have only a single field element as the overall hash key.

An important aspect of our construction is the fact that the hash key consists of a single field element. Suppose the hash key for the BRW layer is τ. We show that it is possible to use a suitable power of τ as the key to the Horner layer. Moreover, if η is one less than some power of two, then the required power of τ can be computed using only one extra squaring over and above the computations required by BRW.

The underlying field for the new hash function can be any field. In particular, this field

can be a suitable binary field or, it can be a field of large characteristic such as $\mathbb{F}_{2^{130}-5}$, the field which has been used in Poly1305.

To make the ideas concrete, we instantiate the two-level hash construction for the binary fields $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$. For implementing the new hash functions, the basic requirement is efficient implementation of multiplication over $\mathbb{F}_{2^n}$ for n equal to either 128 or 256. Being a binary field, it is possible to utilise the instruction `pclmulqdq` available in modern Intel processors for field multiplication. The instruction `pclmulqdq` multiplies two 64-bit polynomials and returns the 128-bit polynomial as the product. Our implementations for both $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ are based on the `pclmulqdq` instruction.

A field multiplication in $\mathbb{F}_{2^n}$ consists of a polynomial multiplication followed by a reduction modulo the irreducible polynomial representing the field. Late, or, delayed reduction is a well known technique for speeding up a group of field multiplications. Essentially, the idea is to perform several polynomial multiplications, add the results and then perform a single reduction for the entire group of multiplication. This technique cannot always be applied. We carefully analyse the structure of BRW and identify the groups of multiplications for which a single reduction suffices. Our implementations of the hash function for $n = 128$ and $n = 256$ make use of delayed reduction to achieve efficiency improvement.

Several other concrete efficiency issues for BRW have been identified and implemented. One of these is to perform independent multiplications together so that all the `pclmulqdq` instructions for these multiplications can be placed together.

This permits possible utilisation of instruction level pipelining. For $n = 128$, the implementation of the new hash function is faster than the implementation of GHASH by Gueron [59]; on the Haswell processor of Intel, we obtain speed improvements of about 23% to 49%, while on the Skylake processor, the speed improvements are about 25% to 53%. In [61], Gueron and Lindell have proposed a new nonce misuse-resistant AEAD scheme called GCM-SIV. This scheme uses a polynomial based hash function called POLYVAL, which has a highly optimised implementation by Gueron [57]. The implementation of the new hash function for $n = 128$ is faster than the aforementioned implementation of POLYVAL by 15% to 19% on Haswell processors and by 10% to 15% on Skylake processors.

The work by Bernstein and Chou [16] reports the implementation of a pseudo-dot product based hash function over $\mathbb{F}_{2^{256}}$. This implementation does not use the `pclmulqdq` instruction and is instead based on the Fast Fourier Transform (FFT) algorithm. The work shows that the hash function can be computed at the cost of 29 bit operations per bit of the digest. There is, however, a considerable hidden cost of generating the hash key which is as long as the message. This cost is not accounted for in the 29 bit operations per bit measure given in [16].

The FFT based multiplication algorithm can also be used with the new hash construction that we propose. The code for the multiplication algorithm described in [16] is not publicly available and so we could not carry out a concrete implementation. Instead, we used the operation counts for direct and inverse FFT, pointwise multiplication and the reduction algorithm reported in [16], to derive an expression for the number of bit operations per bit of the digest for the new hash function. For $\eta = 31$, this cost is at most about 46, while for $\eta = 63$ or 127, the cost is lower. The cost of 46 bit operations per bit is higher than the cost of 29 bit operations per bit reported in [16]. On the other hand, unlike [16],

in our case there is no hidden cost of generating the hash key. Securely generating a long hash key will have a significant cost and if this cost is taken into account, then we expect the total cost in [16] to be significantly more than the 46 bit operations per bit that we obtain.

This chapter is based on the work [30].

5.1 Combining BRW with Horner

Both $\{\mathsf{Horner}_\tau\}$ and $\{\mathsf{BRW}_\tau\}$ use a single key $\tau \in \mathbb{F}$. The number of multiplications in $\mathbb{F}$ required to evaluate the two functions, though, are different. For a message consisting of ℓ field elements, Horner can be evaluated using $\ell - 1$ multiplications, while for $\ell \geq 3$, BRW requires $\lfloor \ell/2 \rfloor$ multiplications plus $\lfloor \log_2 \ell \rfloor$ squarings. In theory, this difference makes BRW much faster than Horner.

The problem, however, is that the definition of BRW is recursive. It is possible to have a recursive implementation of BRW. The overhead of such an implementation will nullify the benefit of lesser number of multiplications. On the other hand, if ℓ is a fixed integer, then it is possible to have a very fast non-recursive implementation of BRW.

Horner on the other hand can handle arbitrary values of ℓ quite easily. So, it makes sense to try and combine BRW and Horner so that the benefits of both the approaches can be obtained. One top-level strategy for doing this is the following. Suppose the message is a bit string which is formatted into a sequence of blocks where each block is an element of the field $\mathbb{F}$. Divide the sequence of field elements into groups of η blocks (assuming that η divides the number of blocks in the message). Each such group will be called a super-block.

We fix the value of η. The function BRW is used to process each super-block. Each invocation of BRW on a superblock produces a field element. These field elements are processed using Horner. So, there are two levels of the hash function. At the lower level, the message is formatted into super-blocks and BRW is used to process the super-blocks, while at the upper level, Horner is used to process the outputs of the BRW invocations. Since the number of blocks in a super-block is fixed, a fast non-recursive implementation of BRW can be used to process the super-blocks. A fast implementation of Horner can be used to combine the outputs of BRW calls. The number of multiplications required by this approach is a little greater than that of BRW and is significantly smaller than that of Horner.

An important issue that needs to be properly tackled is the size of the key for the hash function. Generic approaches to multi-level hash [106, 89, 99] require the key to have independent components for each level of the hash. For a two-level hash, this would normally require two independent field elements as the key. It is, however, desirable to use a single field element as the key. We show how this can be done.

Proposition 3. *Let η, ℓ be positive integers. For $M \in \mathbb{F}^{\eta\ell}$ write $M = (M_1, \ldots, M_\ell)$ where*

each $M_i \in \mathbb{F}^\eta$. We define $G_\tau(M_1, \ldots, M_\ell)$ to be a polynomial in τ in the following manner.

$$\begin{aligned} & G_\tau(M_1, \ldots, M_\ell) \\ &= \mathsf{Horner}_{\tau^{d(\eta)+1}}\left(\mathsf{BRW}_\tau(M_1), \ldots, \mathsf{BRW}_\tau(M_\ell)\right) \\ &= \tau^{(d(\eta)+1)(\ell-1)}\mathsf{BRW}_\tau(M_1) + \tau^{(d(\eta)+1)(\ell-2)}\mathsf{BRW}_\tau(M_2) + \\ &\qquad \cdots + \tau^{(d(\eta)+1)}\mathsf{BRW}_\tau(M_{\ell-1}) + \mathsf{BRW}_\tau(M_\ell). \end{aligned} \tag{5.1}$$

The following hold for the function G given by (5.1).

1. *The degree of G in τ is $\ell d(\eta) + \ell - 1$.*
2. *G injectively maps $\mathbb{F}^{\eta\ell}$ to $\mathbb{F}[\tau]$.*

Consequently, the hash family $\{G_\tau\}_{\tau\in\mathbb{F}}$ is $((\ell d(\eta) + \ell - 1)/\#\mathbb{F})$-AU and the hash family $\{\tau\, G_\tau\}_{\tau\in\mathbb{F}}$ is $((\ell d(\eta) + \ell)/\#\mathbb{F})$-AXU.

Proof. Since each $M_i \in \mathbb{F}^\eta$, the degree of $\mathsf{BRW}_\tau(M_i)$ is $d(\eta)$ and so the degree of the polynomial G is $(d(\eta) + 1)(\ell - 1) + d(\eta) = \ell d(\eta) + \ell - 1$. This proves the first point.

Each $M_i \in \mathbb{F}^\eta$ and so for all i, $\mathsf{BRW}_\tau(M_i)$ has degree $d(\eta)$. Let

$$\mathsf{BRW}_\tau(M_i) = \tau^{d(\eta)}c_{i,d(\eta)} + \tau^{d(\eta)-1}c_{i,d(\eta)-1} + \cdots + \tau c_{i,1} + c_{i,0}$$

for some $c_{i,d(\eta)}, \ldots, c_{i,1}, c_{i,0} \in \mathbb{F}$ which depend on M_i. Using this, we write

$$\begin{aligned} & G_\tau(M_1, \ldots, M_\ell) \\ &= \tau^{(d(\eta)+1)(\ell-1)+d(\eta)}c_{1,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-1)+1}c_{1,1} + \tau^{(d(\eta)+1)(\ell-1)}c_{1,0} \\ &\quad + \tau^{(d(\eta)+1)(\ell-2)+d(\eta)}c_{2,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-2)+1}c_{2,1} + \tau^{(d(\eta)+1)(\ell-2)}c_{2,0} \\ &\quad + \cdots + \\ &\quad + \tau^{(d(\eta)+1)(\ell-i)+d(\eta)}c_{i,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-i)+1}c_{i,1} + \tau^{(d(\eta)+1)(\ell-i)}c_{i,0} \\ &\quad + \cdots + \\ &\quad + \tau^{2d(\eta)+1}c_{\ell-1,d(\eta)} + \cdots + \tau^{d(\eta)+2}c_{\ell-1,1} + \tau^{d(\eta)+1}c_{\ell-1,0} \\ &\quad + \tau^{d(\eta)}c_{\ell,d(\eta)} + \cdots + \tau c_{\ell,1} + c_{\ell,0}. \end{aligned}$$

Considered as a polynomial in τ, the coefficients of $G_\tau(M_1, \ldots, M_\ell)$ are $c_{i,j}$ with $1 \le i \le \ell$ and $0 \le j \le d(\eta)$. Due to the choice of the key for Horner to be $\tau^{d(\eta)+1}$, each $c_{i,j}$ is associated with a unique power of τ.

Let $M, M' \in \mathbb{F}^{\eta\ell}$ with $M \neq M'$. Write $M' = (M'_1, \ldots, M'_\ell)$ with each $M'_i \in \mathbb{F}^\eta$. Let $c'_{i,j}$ be the coefficients of the polynomial $G_\tau(M'_1, \ldots, M'_\ell)$. Since $M \neq M'$, there is an i such that $M_i \neq M'_i$. From the injectivity property of BRW, it follows that $\mathsf{BRW}_\tau(M_i) \neq \mathsf{BRW}_\tau(M'_i)$ and so there is a $j \in \{0, 1, \ldots, d(\eta)\}$ such that $c_{i,j} \neq c'_{i,j}$. From this it follows that $G_\tau(M_1, \ldots, M_\ell) \neq G_\tau(M'_1, \ldots, M'_\ell)$. This shows the second point.

Since for distinct M and M', $G_\tau(M_1, \ldots, M_\ell)$ and $G_\tau(M'_1, \ldots, M'_\ell)$ are distinct and have the same degree, it follows that $G_\tau(M_1, \ldots, M_\ell) - G_\tau(M'_1, \ldots, M'_\ell)$ is a non-zero polynomial of degree at most $\ell d(\eta) + \ell - 1$. Consequently, the probability that $G_\tau(M_1, \ldots, M_\ell)$ is

Table 5.1: Summary of the features of the basic scheme, Horner and BRW for hashing $\eta\ell$ blocks with $\eta = 2^{\mathfrak{a}+1} - 1$ for some $\mathfrak{a} \geq 1$.

scheme	# sqr	# mult	AU bound
Horner	–	$\eta\ell - 1$	$(\eta\ell - 1)/\#\mathbb{F}$
BRW	$\lfloor \lg \eta\ell \rfloor$	$\lfloor \eta\ell/2 \rfloor$	$d(\eta\ell)/\#\mathbb{F}$
G_τ	$\mathfrak{a} + 1$	$\ell(\eta+1)/2 - 1$	$(\eta\ell + \ell - 1)/\#\mathbb{F}$

equal to $G_\tau(M'_1, \ldots, M'_\ell)$ is the probability that τ is a root of the non-zero polynomial $G_\tau(M_1, \ldots, M_\ell) - G_\tau(M'_1, \ldots, M'_\ell)$. The number of distinct roots of a non-zero polynomial over a field is at most its degree from which it follows that the required probability is at most $(\ell d(\eta) + \ell - 1)/\#\mathbb{F}$. This shows the AU property.

For the AXU property, we note that the degree of $\tau\, G_\tau(M_1, \ldots, M_\ell)$ is $\ell d(\eta) + \ell$. The rest of the argument is similar to that of the AU propery. □ □

A crucial point in the above construction and the proof is the choice of the appropriate power of τ as the key for Horner so that the injectivity of G_τ follows directly from the injectivity of BRW_τ. The key for BRW_τ is τ and the degree of BRW_τ in τ is $d(\eta)$. Based on this, the key for Horner is chosen to be $\tau^{d(\eta)+1}$. This ensures that during the computation of Horner, the BRW polynomials arising from distinct super-blocks do not "overlap".

For $\eta \geq 3$ suppose $\mathfrak{a} \geq 1$ is such that $2^{\mathfrak{a}} \leq \eta < 2^{\mathfrak{a}+1}$. Then $d(\eta) = 2^{\mathfrak{a}+1} - 1$. The number of multiplications required in evaluating (5.1) is given by the number of multiplications required to evaluate all the BRW invocations and the number of multiplications required to evaluate the single Horner invocation. Each BRW requires $\lfloor \eta/2 \rfloor$ multiplications and Horner requires $\ell - 1$ multiplications for a total of $\ell\lfloor \eta/2 \rfloor + \ell - 1$ multiplications. Additionally, $\lfloor \lg \eta \rfloor = \mathfrak{a}$ squarings are required to compute the powers $\tau^2, \ldots, \tau^{2^{\mathfrak{a}}}$ which are used for evaluating BRW; an additional squaring is required to compute the power $\tau^{d(\eta)+1} = \tau^{2^{\mathfrak{a}+1}}$ which is used as a key to Horner. So a total of $\lfloor \lg \eta \rfloor + 1$ squarings are required to compute all the required powers of τ.

For $\eta = 2^{\mathfrak{a}+1} - 1$ with $\mathfrak{a} \geq 1$, Table 5.1 compares the efficiency and security of G_τ with that of Horner and BRW. The ratio of the number of multiplications required by G_τ to that required by Horner is $(\ell(\eta+1) - 2)/(2(\ell\eta - 1))$ and the ratio of the number of multiplications required by BRW to that required by G_τ is $2\lfloor \eta\ell/2 \rfloor/(\ell(\eta+1) - 2)$. Suppose $\eta = 31$: the first ratio is $(16\ell - 1)/(31\ell - 1)$ which equals $1/2$ for $\ell = 1$ and has the limiting upper bound of $16/31 \approx 0.52$; the second ratio is $\lfloor 31\ell/2 \rfloor/(16\ell - 1)$ which equals 1 for $\ell = 1$ and decreases to about 0.97 as ℓ increases. So, for $\eta = 31$, the number of multiplications required by G_τ is about 50% to 52% of that required by Horner while the number of multiplications required by BRW is about 97% to 100% of that required by G_τ.

For $\eta = 31$, the AU bound for Horner is $(31\ell - 1)/\#\mathbb{F}$; the AU bound for G_τ is $(32\ell - 1)/\#\mathbb{F}$; and the AU bound for BRW is $d(31\ell)/\#\mathbb{F} = (2^{\lfloor \lg(31\ell) \rfloor + 1} - 1)/\#\mathbb{F}$. The AU bound for BRW is in general higher than the AU bound for G_τ. The two bounds can be equal, e.g. for $\ell = 1, 2, 4, 8, \ldots$. On the other hand, the AU bound for BRW can be about twice as large as the AU bound for G_τ, e.g. for $\ell = 9$, the bound for G_τ is $287/\#\mathbb{F}$ and the bound for BRW

is $511/\#\mathbb{F}$.

Overall, G_τ allows a range of efficiency/security trade-offs between BRW and Horner. By choosing an appropriate value for η, it is possible to attain speed nearly equal to that of BRW with AU bound not too larger than Horner.

Multi-level hashing: The idea of using BRW at the lower level and Horner at the upper level can be extended to more than one level. The critical issue is to choose an appropriate power of τ as the key for each level. While this can be done, extending to more than two levels results in a rather complicated construction which would mainly be of theoretical, rather than any practical, interest. So, we did not pursue the idea of multi-level hashing.

5.2 Two-Level Hash Function

For practical applications, it is required to handle variable length strings. We show how to modify the construction in Proposition 3 to be able to do this. For concreteness, *in the rest of this chapter we will fix the finite field* $\mathbb{F}$ *to be* $\mathbb{F}_{2^n}$ *for some positive integer* n. The ideas, on the other hand, are quite general and can be adapted to work with other finite fields.

The following notation will be used in addition to the general notation already set.

- Given a positive integer n and a binary string S, let $\mathsf{pad}_n(S)$ denote $S||0^i$ where i is the minimum non-negative integer such that $\mathsf{len}(S)+i$ is divisible by n.

Let

$$\mathcal{M} \;=\; \bigcup_{i=0}^{2^n-1} \{0,1\}^i. \tag{5.2}$$

The reason for the bound 2^n-1 on the length of the strings in $\mathcal{M}$ is that we require the binary representation of the length of any string in $\mathcal{M}$ to fit into an n-bit string. For $M \in \mathcal{M}$, we define a function $\mathsf{superBlks}_{n,\eta}(M)$ as follows. Consider $\mathsf{pad}_n(M)$ to be formatted into a sequence of n-bit blocks. Let ℓ be such that

$$\frac{\mathsf{len}(\mathsf{pad}_n(M))}{n} = \eta(\ell-1)+\lambda \tag{5.3}$$

for some $\lambda \in \{1,\ldots,\eta\}$. Then $\mathsf{pad}_n(M)$ consists of $\ell-1$ full super-blocks and one possibly partial super-block. Let $\mathsf{superBlks}_{n,\eta}(M)$ denote these super-blocks and we write

$$\mathsf{superBlks}_{n,\eta}(M) = (M_1,\ldots,M_\ell)$$

where $M_1,\ldots,M_{\ell-1}$ are full super-blocks (consisting of exactly η n-bit blocks each) and M_ℓ is a possibly partial superblock (consisting of at most η n-bit blocks).

Theorem 3. *Let* $\mathcal{M}$ *be as given in (5.2). Define a hash family* $\{\mathsf{Hash2L}_\tau\}_{\tau\in\mathbb{F}_{2^n}}$ *where* $\mathsf{Hash2L}_\tau : \mathcal{M} \to \{0,1\}^n$ *such that for* $M \in \mathcal{M}$,

$$\begin{aligned}&\mathsf{Hash2L}_\tau(M)\\ &\quad= \;\tau^2\mathsf{Horner}_{\tau^{d(\eta)+1}}(\mathsf{BRW}_\tau(M_1),\ldots,\mathsf{BRW}_\tau(M_\ell)) \oplus \tau\mathsf{bin}_n(\mathsf{len}(M))\end{aligned} \tag{5.4}$$

where $(M_1, \ldots, M_\ell) = \mathsf{superBlks}_{n,\eta}(M)$.

Let M *and* M' *be distinct elements of* $\mathcal{M}$ *with* $\mathsf{len}(M) \geq \mathsf{len}(M')$. *For a uniform random* $\tau \in \mathbb{F}_{2^n}$ *and any* $\beta \in \mathbb{F}_{2^n}$

$$\Pr_{\tau}\left[\mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') = \beta\right] \;\leq\; \frac{\ell(d(\eta)+1)+1}{2^n} \tag{5.5}$$

where ℓ *is the number of super-blocks in* M.

Proof. The proof follows if we can show that $\mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') \oplus \beta$ is a non-zero polynomial in τ of degree at most $\ell(d(\eta)+1)+1$. The maximum degree of $\mathsf{Hash2L}_\tau(M)$ as a polynomial in τ is $\ell d(\eta) + \ell - 1 + 2 = \ell(d(\eta)+1)+1$. So, we only need to argue that $P = \mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') \oplus \beta$ is a non-zero polynomial.

First suppose that $\mathsf{len}(M) \neq \mathsf{len}(M')$. The coefficient of τ in $\mathsf{Hash2L}_\tau(M)$ is $\mathsf{bin}_n(\mathsf{len}(M))$ and the coefficient of τ in $\mathsf{Hash2L}_\tau(M')$ is $\mathsf{bin}_n(\mathsf{len}(M')) \neq \mathsf{bin}_n(\mathsf{len}(M))$. So, P is a non-zero polynomial in τ.

So, suppose that $\mathsf{len}(M) = \mathsf{len}(M')$. Then $\ell = \ell'$ and an argument similar to that provided for Proposition 3 shows that P is a non-zero polynomial in τ. The only difference with the argument in Proposition 3 is that the last super-blocks M_ℓ and M'_ℓ may be partial. This, however, does not affect the argument, since the property that $M_\ell \neq M'_\ell$ implies $\mathsf{BRW}_\tau(M_\ell) \neq \mathsf{BRW}_\tau(M'_\ell)$ is preserved. □

Remark: The manner in which $\mathsf{Hash2L}_\tau(M)$ has been defined ensures the AXU property. If only the AU property is desired, then one can define $\mathsf{Hash2L}_\tau(M)$ to be

$$\tau\mathsf{Horner}_{\tau^{d(\eta)+1}}(\mathsf{BRW}_\tau(M_1), \ldots, \mathsf{BRW}_\tau(M_\ell)) \oplus \mathsf{bin}_n(\mathsf{len}(M)).$$

This requires one less multiplication.

5.2.1 Hashing a Vector of Strings

The hash family Hash2L handles a single binary string. We show how to extend it to handle a vector where each component is a binary string.

The parameters n and η are defined as in the case of Hash2L. We define the hash family

$$\left\{\mathsf{vecHash2L}_\tau\right\}_{\tau \in \mathbb{F}_{2^n}} \text{ such that } \mathsf{vecHash2L}_\tau : \mathcal{VD} \to \mathbb{F}_{2^n}. \tag{5.6}$$

The domain $\mathcal{VD}$ consists of variable length vectors of binary strings. Formally,

$$\mathcal{VD} \;=\; \bigcup_{k=0}^{255} \left\{(M_1, \ldots, M_k) : 0 \leq \mathsf{len}(M_i) \leq 2^{n-16} - 1\right\}. \tag{5.7}$$

The reason for the bound $k \leq 255$ is that we require the binary representation of k to fit into a byte. Similarly, the reason for the bound $\mathsf{len}(M_i) \leq 2^{n-16} - 1$ is that we require the binary representation of the length of any M_i to fit into $n-16$ bits. If $k = 0$, then the input is the empty list. Note that this input is different from the input where $k = 1$ and M_1 is the empty string.

The computation of the output of $\mathsf{vecHash2L}_\tau$ is shown in Table 5.2.

Table 5.2: Computation of vecHash2L.

$\mathsf{vecHash2L}_\tau(M_1, \ldots, M_k)$
  if $k == 0$ return $1^n\tau$;
  $\mathsf{digest} = 0^n$;
  for $i = 1, \ldots, k-1$ do
    $(M_{i,1}, \ldots, M_{i,\ell_i}) = \mathsf{superBlks}_{n,\eta}(M_i)$;
    $L_i = \mathsf{bin}_n(\mathsf{len}(M_i))$;
    for $j = 1, \ldots, \ell_i$ do
      $\mathsf{digest} = \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{i,j})$;
    end for;
    $\mathsf{digest} = \tau\mathsf{digest} \oplus L_i$;
  end for;
  $(M_{k,1}, \ldots, M_{k,\ell_k}) = \mathsf{superBlks}_{n,\eta}(M_k)$;
  $L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k))$;
  for $j = 1, \ldots, \ell_k$ do
    $\mathsf{digest} = \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{k,j})$;
  end for;
  $\mathsf{digest} = \tau\mathsf{digest} \oplus L_k$;
  $\mathsf{digest} = \tau\mathsf{digest}$;
  return digest.

Theorem 4. *Let $k \geq k' \geq 0$; $\mathbf{M} = (M_1, \ldots, M_k)$ and $\mathbf{M}' = (M'_1, \ldots, M'_{k'})$ be two distinct vectors in $\mathcal{VD}$. For a uniform random $\tau \in \mathbb{F}_{2^n}$ and for any $\beta \in \mathbb{F}_{2^n}$,*

$$\Pr_\tau\left[\mathsf{vecHash2L}_\tau(\mathbf{M}) \oplus \mathsf{vecHash2L}_\tau(\mathbf{M}') = \beta\right] \leq \frac{\max\left(k + (d(\eta)+1)\Lambda, k' + (d(\eta)+1)\Lambda'\right)}{2^n} \quad (5.8)$$

where $\Lambda = \sum_{i=1}^{k} \ell_i$ and $\Lambda' = \sum_{i=1}^{k'} \ell'_i$.

Proof. Quantities corresponding to M' will have the superscript $'$.

For $i = 1, \ldots, k$ and $1 \leq j \leq \ell_i$, the degree of $\mathsf{BRW}_\tau(M_{i,j})$ is $d(\eta)$ for $1 \leq j < \ell_i$ and it is at most $d(\eta)$ for $j = \ell_i$. Write

$$\mathsf{BRW}_\tau(M_{i,j}) = c_{i,j,0} \oplus c_{i,j,1}\tau \oplus \cdots \oplus c_{i,j,d(\eta)}\tau^{d(\eta)}$$

where the c's depend on the M_i's. So, each M_i contributes at most $\ell_i(d(\eta)+1)+1$ coefficients

$$c_{i,1,0}, \ldots, c_{i,1,d(\eta)}, \ldots, c_{i,\ell_i,0}, \ldots, c_{i,\ell_i,d(\eta)}, L_i$$

to $\mathsf{vecHash2L}_\tau(\mathbf{M})$. The total number of coefficients is at most $k + (d(\eta)+1)\Lambda$. The last step in the digest computation increases the degree by one and so the maximum degree of $\mathsf{vecHash2L}_\tau(\mathbf{M})$ is equal to the maximum number of coefficients in $\mathsf{vecHash2L}_\tau(\mathbf{M})$. The degree of

$$P = \mathsf{vecHash2L}_\tau(\mathbf{M}) \oplus \mathsf{vecHash2L}_\tau(\mathbf{M}') \oplus \beta$$

is $\max\left(k + (d(\eta)+1)\Lambda, k' + (d(\eta)+1)\Lambda'\right)$. The result follows if we can show that P is a non-zero polynomial in τ. The detailed proof is divided into several cases.

Case $k' = 0$: Since $\mathbf{M} \neq \mathbf{M}'$, it follows that $k > 0$. $\mathsf{vecHash2L}_\tau(M')$ equals $1^n\tau$. Since $k > 0$, $\mathsf{vecHash2L}_\tau(M)$ is of the form $L_k\tau \oplus \tau^2(\cdots)$ where

$$L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k)) \neq 1^n.$$

So, P is a non-zero polynomial.

Case $k > k' > 0$: In this case, we have $\mathsf{vecHash2L}_\tau(M)$ of the form $L_k\tau \oplus \tau^2(\cdots)$ and $\mathsf{vecHash2L}_\tau(M')$ of the form $L'_{k'}\tau \oplus \tau^2(\cdots)$ where

$$L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k)) \neq \mathsf{bin}_8(k')||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M'_{k'})) = L'_{k'}.$$

So, again P is a non-zero polynomial.

Case $k = k' > 0$: There are two subcases to consider.

Sub-case (a): There is some i such that $\mathsf{len}(M_i) \neq \mathsf{len}(M'_i)$. Let i be the maximum such value and so, $\mathsf{len}(M_j) = \mathsf{len}(M'_j)$ for $j = i+1, \ldots, k$. Since $\mathsf{len}(M_i) \neq \mathsf{len}(M'_i)$, it follows that $L_i \neq L'_i$. Let $s = 1 + (k-i) + \sum_{j=i+1}^{k} \ell_j(d(\eta)+1) = 1 + (k'-i) + \sum_{j=i+1}^{k'} \ell'_j(d(\eta)+1)$. Then the coefficient of τ^s in P is $L_i \oplus L'_i \neq 0$. So, again P is a non-zero polynomial.

Sub-case (b): In this case, $\mathsf{len}(M_i) = \mathsf{len}(M'_i)$ for $i = 1, \ldots, k$. As a result, in this case the number of components and the length of all the components in $\mathbf{M}$ and $\mathbf{M}'$ are equal. Since $\mathbf{M} \neq \mathbf{M}'$, it follows that there must be some s such that $M_s \neq M'_s$.

Since $\mathsf{len}(M_s) = \mathsf{len}(M'_s)$, it follows that the number of superblocks of M_s and M'_s are equal, i.e., $\ell_s = \ell'_s$. The super-blocks corresponding to M_s are $M_{s,1}, \ldots, M_{s,\ell_s}$ while the super-blocks corresponding to M'_s are $M'_{s,1}, \ldots, M'_{s,\ell_s}$. Since $M_s \neq M'_s$, at least one of the superblocks must be unequal. Let $t \in \{1, \ldots, \ell_s\}$ be such that $M_{s,t} \neq M'_{s,t}$. By the injectivity of BRW, it follows that $\mathsf{BRW}_\tau(M_{s,t}) \neq \mathsf{BRW}_\tau(M'_{s,t})$ and so there is a $k \in \{0, \ldots, d(\eta)\}$ such that $c_{s,t,k} \neq c'_{s,t,k}$. As a result, P is a non-zero polynomial.

This completes all the cases and the proof. □

5.3 Implementations Based on `pclmulqdq`

Our target platform were the Intel processors which support the `pclmulqdq` instruction. This instruction takes as input two degree 64 polynomials over $GF(2)$ (represented as two 64-bit words) and returns as output the degree 128 polynomial which is the product of the two input polynomials. The implementation was done using Intel intrinsics.

We report implementations for $n = 128$ and $n = 256$. As mentioned in Chapter 3, for $n = 128$, $\mathbb{F}_{2^{128}}$ was represented using the irreducible polynomial $\psi(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$ and for $n = 256$, $\mathbb{F}_{2^{256}}$ was represented using the irreducible polynomial $\psi(x) = x^{256} \oplus x^{10} \oplus x^5 \oplus x^2 \oplus 1$. In both cases, $\psi(x)$ is of the form $x^n \oplus g_0(x)$ where $g_0(x)$ is a polynomial of degree less than $n/2$ having exactly 4 non-zero coefficients.

We report timings on two different machines. For the timing measurements, we followed the strategy of [74]. The first timing measurements were taken on a single core of a machine with Intel Core i7-4790 Haswell @ 3.60GHz. The second timing measurements were taken on a single core of a machine with Intel Core i7-6500U Skylake @ 2.5GHz. In both cases, the operating system was 64-bit Ubuntu-14.04-LTS and the C code was complied using GCC version 4.8.4. The code is publicly available[1].

5.3.1 Field Multiplication

The multiplication of two 128-bit polynomials using the schoolbook method requires 4 `pclmulqdq` calls and using Karatsuba's algorithm requires 3 `pclmulqdq` calls. The multiplication of two 256-bit polynomials using the schoolbook method requires 16 `pclmulqdq` calls and using Karatsuba's algorithm requires 9 `pclmulqdq` calls. The reduction step can also be computed using `pclmulqdq` calls. From the work of Gueron and Kounavis [58] one obtains that for $n = 128$, 2 `pclmulqdq` calls are sufficient for the reduction while for $n = 256$, 4 `pclmulqdq` calls are sufficient. Details are provided in Section 5.3.2 below.

Batch multiplications: Suppose $\mathfrak{m}$ independent multiplications are to be computed. The code can be arranged such that the `pclmulqdq` instructions for these multiplications can be grouped together. This may help the instruction scheduler to utilise instruction pipelining to speed up the computation. We have experimented with values of $\mathfrak{m} \leq 4$ and have found some speed improvements. In theory, the speed improvement should continue as $\mathfrak{m}$ increases. In practice, however, this does not always happen.

5.3.2 Efficient Reduction

Let n be a positive even integer and $\mathbb{F}_{2^n}$ be represented by an irreducible polynomial $\psi(x)$ of degree n over $GF(2)$. Elements of $\mathbb{F}_{2^n}$ are represented using polynomials over $GF(2)$ of degrees less than n. Let $\alpha = \alpha(x)$ and $\beta = \beta(x)$ be two elements of $\mathbb{F}_{2^n}$. The computation of $\alpha\beta = \alpha(x)\beta(x) \bmod \psi(x)$ consists of two steps. In the first step, $\alpha(x)$ and $\beta(x)$ are multiplied together to obtain a result $e(x)$ of degree less than $2n-1$ and then $e(x)$ is reduced modulo $\psi(x)$ to obtain the desired result. Let $e(x) = \alpha(x)\beta(x)$ and write $e(x) = d(x) \oplus c(x)x^n$ where $c(x)$ and $d(x)$ have degrees less than n. The essential task is to compute $c(x)x^n \bmod \psi(x)$. A method for doing this was described by Gueron and Kounavis [58]. We review their method and determine the number of `pclmulqdq` instructions required for the reduction for both $n = 128$ and $n = 256$.

Let $q(x)$ and $h(x)$ be such that $c(x)x^n = q(x)\psi(x) \oplus h(x)$ with $\deg(q), \deg(h) \leq n-1$. The goal is to find $h(x)$. Write $\psi(x) = x^n \oplus \psi^*(x)$, where $\deg(\psi^*) \leq n-1$. The equation $c(x)x^n = q(x)\psi(x) \oplus h(x)$ becomes $c(x)x^n = q(x)x^n \oplus q(x)\psi^*(x) \oplus h(x)$ and so $h(x) = q(x)\psi^*(x) \bmod x^n$. So, finding $q(x)$ is sufficient for obtaining $h(x)$.

[1] `https://github.com/sebatighosh/HASH2L.git`

Let $q^+(x)$ and $h^+(x)$ be such that $x^{2n} = q^+(x)\psi(x) \oplus h^+(x)$ with $\deg(h^+) \leq n-1$ and $\deg(q^+) = n$. So,

$$\begin{array}{rrcl} & c(x)x^{2n} & = & q(x)\psi(x)x^n \oplus h(x)x^n \\ \Rightarrow & c(x)(q^+(x)\psi(x) \oplus h^+(x)) & = & q(x)\psi(x)x^n \oplus h(x)x^n \\ \Rightarrow & c(x)q^+(x)\psi(x) \oplus c(x)h^+(x) & = & q(x)\psi(x)x^n \oplus h(x)x^n \\ \Rightarrow & \left\lfloor \frac{c(x)q^+(x)\psi(x) \oplus c(x)h^+(x)}{x^{2n}} \right\rfloor & = & \left\lfloor \frac{q(x)\psi(x)x^n \oplus h(x)x^n}{x^{2n}} \right\rfloor \\ \Rightarrow & \left\lfloor \frac{c(x)q^+(x)\psi(x)}{x^{2n}} \right\rfloor & = & \left\lfloor \frac{q(x)\psi(x)}{x^n} \right\rfloor. \end{array}$$

In the above the following two facts have been used: $\deg(ch^+) \leq 2n-2$ and $\deg(h) \leq n-1$. Let $u(x)$ be of degree at most $n-1$, $v_1(x)$ of degree at most $2n-1$ and $v_2(x)$ of degree at most $n-1$ such that $c(x)q^+(x)\psi(x) = u(x)x^{2n} \oplus v_1(x)$ and $q(x)\psi(x) = u(x)x^n \oplus v_2(x)$. From this we obtain $c(x)q^+(x)\psi(x)/x^n = u(x)x^n \oplus v_1(x)/x^n = q(x)\psi(x) \oplus v_2(x) \oplus v_1(x)/x^n$. This is re-written as $c(x)q^+(x)/x^n = q(x) \oplus v_2(x)/\psi(x) \oplus v_1(x)/\psi(x)x^n$. Since $\deg(v_2) \leq n-1$, $\lfloor v_2(x)/\psi(x) \rfloor = 0$ and since $\deg(v_1) \leq 2n-1$, $\lfloor v_1(x)/(\psi(x)x^n) \rfloor = 0$. So, we obtain $q(x) = \lfloor c(x)q^+(x)/x^n \rfloor$.

Further simplifications: Suppose n is even, and $\psi^*(x) = g_0(x)$ with $\deg(g_0) < n/2$. So, $\psi(x) = x^n \oplus g_0(x)$. We have $x^n = \psi(x) \oplus g_0(x)$ and so $x^{2n} = \psi^2(x) \oplus g_0(x^2)$. Since $\deg(g_0) \leq n/2 - 1$, $\deg(g_0(x^2)) \leq n-2$. So, $q^+(x) = \psi(x)$ and $h^+(x) = g_0(x^2)$. Write $c(x) = c_1(x)x^{n/2} \oplus c_0(x)$ where $\deg(c_1), \deg(c_0) < n/2$. Consider the product

$$\begin{array}{rcl} c(x)q^+(x) & = & c(x)\psi(x) = c(x)(x^n \oplus g_0(x)) = c(x)x^n \oplus c(x)g_0(x) \\ & = & c(x)x^n \oplus (c_1(x)x^{n/2} \oplus c_0(x))g_0(x) = c(x)x^n \oplus x^{n/2}c_1(x)g_0(x) \oplus c_0(x)g_0(x). \end{array}$$

Since $\deg(c_0g_0) \leq n-2$, $\lfloor (c_0g_0)/x^n \rfloor = 0$ and we have

$$q(x) = \left\lfloor \frac{c(x)q^+(x)}{x^n} \right\rfloor = c(x) \oplus \left\lfloor \frac{x^{n/2}c_1(x)g_0(x)}{x^n} \right\rfloor = c(x) \oplus \left\lfloor \frac{c_1(x)g_0(x)}{x^{n/2}} \right\rfloor.$$

Write $q(x) = q_1(x)x^{n/2} \oplus q_0(x)$ with $\deg(q_1), \deg(q_0) < n/2$. Since $\deg(c_1), \deg(g_0) < n/2$, it follows that $\deg(c_1g_0) < n-1$ and so $\lfloor (c_1(x)g_0(x))/x^{n/2} \rfloor$ is a polynomial of degree less than $n/2$. So, we have $q(x) = c_1(x)x^{n/2} \oplus c_0(x) \oplus \left\lfloor (c_1(x)g_0(x))/x^{n/2} \right\rfloor$. In effect, $q_1(x) = c_1(x)$ and $q_0(x) = c_0(x) \oplus \lfloor (c_1(x)g_0(x))/x^{n/2} \rfloor$. Computing $q(x)$ requires computing $c_1(x)g_0(x)$ which accounts for one $n/2$-bit polynomial multiplication.

Given the quantity $q(x)$, $h(x)$ is obtained as $h(x) = q(x)\psi^*(x) \bmod x^n = q(x)g_0(x) \bmod x^n = q_1(x)g_0(x)x^{n/2} \oplus q_0(x)g_0(x) \bmod x^n = c_1(x)g_0(x)x^{n/2} \oplus q_0(x)g_0(x) \bmod x^n$. The product $c_1(x)g_0(x)$ has already been computed. So, computing $h(x)$ requires another additional $n/2$-bit polynomial multiplication, namely $q_0(x)g_0(x)$. So, the entire reduction can be carried out using 2 $n/2$-bit polynomial multiplications. For $n = 128$, $n/2 = 64$ and the two $n/2$-bit polynomial multiplications can be computed using 2 `pclmulqdq` calls. The entire reduction $e(x) \bmod \psi(x)$ requires a total of 7 instructions. For $n = 256$, $n/2 = 128$ and an $n/2$-bit polynomial multiplication is a 128-bit polynomial multiplication. We choose $g_0(x)$ to have degree less than 64. Since $c_1(x)$ is a polynomial of degree less than 128, the product $c_1(x)g_0(x)$

# sqr	# n-bit XORs	# poly mult	# red	AU bnd
$\mathfrak{a}$	$14 \cdot 2^{\mathfrak{a}-2} - 4$	$2^{\mathfrak{a}} - 1$	$2^{\mathfrak{a}-1}$	$\eta/2^n$

Table 5.3: Efficiency and AU bound for $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ over $\mathbb{F}_{2^n}$ with $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$.

can be computed using 2 `pclmulqdq` instructions. Similarly, the product $q_0(x)g_0(x)$ can also be computed using 2 `pclmulqdq` instructions. So, 4 `pclmulqdq` instructions are sufficient for the reduction and the code for computing $e(x)$ mod $\psi(x)$ requires a total of 14 instructions.

5.3.3 Arithmetic Operations for Computing BRW

Let $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$. Suppose $A_{\mathfrak{a}+1}$ is the number of field additions required to evaluate $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$. Then $A_{\mathfrak{a}+1} = 2+2A_{\mathfrak{a}}$, $\mathfrak{a} \geq 2$ and using $A_2 = 3$, we have $A_{\mathfrak{a}+1} = 5 \cdot 2^{\mathfrak{a}-1} - 2$ for $\mathfrak{a} \geq 1$.

The number of multiplications for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ is $\lfloor \eta/2 \rfloor = 2^{\mathfrak{a}} - 1$. Two field elements β and γ are represented using polynomials over $GF(2)$ of degrees less than n. Let us denote these polynomials as $\beta(x)$ and $\gamma(x)$. As described above, the computation of $\beta\gamma$ is done in two steps, namely a polynomial multiplication followed by a reduction.

For computing BRW, it is possible to reduce the number of reductions. We describe this with respect to $\mathbb{F}_{2^n}$, but, the general idea also applies to other fields. While computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$, the product of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^{\mathfrak{a}}-1})$ and $(\tau^{2^{\mathfrak{a}}} + m_{2^{\mathfrak{a}}})$ is added to $\mathsf{BRW}_\tau(m_{2^{\mathfrak{a}}+1}, \ldots, m_{2^{\mathfrak{a}+1}-1})$. This involves a reduction step in the computation of the product $(\tau^{2^{\mathfrak{a}}} + m_{2^{\mathfrak{a}}}) \times \mathsf{BRW}_\tau(m_1, \ldots, m_{2^{\mathfrak{a}}-1})$ and a reduction step in the computation of the output of $\mathsf{BRW}_\tau(m_{2^{\mathfrak{a}}+1}, \ldots, m_{2^{\mathfrak{a}+1}-1})$. These two reductions can be combined into a single reduction in the following manner. Perform the polynomial multiplication of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^{\mathfrak{a}}-1})$ and $(\tau^{2^{\mathfrak{a}}} + m_{2^{\mathfrak{a}}})$; compute $\mathsf{BRW}_\tau(m_{2^{\mathfrak{a}}+1}, \ldots, m_{2^{\mathfrak{a}+1}-1})$ without the final reduction; add the two polynomials; then perform a reduction on the resulting polynomial.

For $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$, let $R_{\mathfrak{a}+1}$ be the number of reductions required to compute $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $R_2 = 1$. The computation of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^{\mathfrak{a}}-1})$ requires $R_{\mathfrak{a}}$ reductions; the computation of $\mathsf{BRW}_\tau(m_{2^{\mathfrak{a}}+1}, \ldots, m_{2^{\mathfrak{a}+1}-1})$ without the final reduction requires $R_{\mathfrak{a}} - 1$ reductions; and there is a final reduction. So, $R_{\mathfrak{a}+1} = R_{\mathfrak{a}} + (R_{\mathfrak{a}} - 1) + 1 = 2R_{\mathfrak{a}}$ for $\mathfrak{a} \geq 2$, $R_2 = 1$ and we obtain $R_{\mathfrak{a}+1} = 2^{\mathfrak{a}-1}$.

A field addition in $\mathbb{F}_{2^n}$ is XOR of two n-bit strings. In trying to reduce the number of reductions, the number of n-bit XORs go up. Unreduced quantities are $2n$-bit polynomials and adding together two such polynomials require 2 n-bit XORs. Further, the cross product terms of the different multiplications are first added together and then shifted. This requires an extra n-bit XOR per delayed reduction. Let $N_{\mathfrak{a}+1}$ be the number of n-bit XORs required to evaluate $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$. Then $N_{\mathfrak{a}+1} = 2N_{\mathfrak{a}} + 4$ for $\mathfrak{a} \geq 2$ with $N_2 = 3$ so that $N_{\mathfrak{a}+1} = 14 \cdot 2^{\mathfrak{a}-2} - 4$.

The relevant parameters for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ along with the AU bound are summarised in Table 5.3.

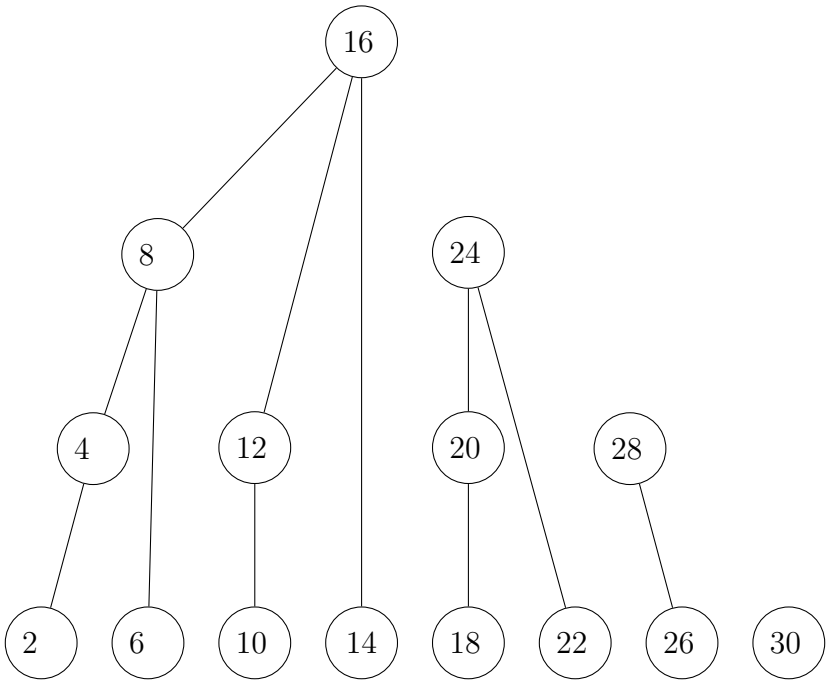
5.3.4 Computing BRW Polynomials

For the actual implementation, for both $n = 128$ and $n = 256$, we set $\eta = 31$, i.e., the number of n-bit blocks in a super-block is 31. For the two-level hash function, the last super-block can be partial. So, we did separate implementations of BRW for handling number of blocks from 1 to 31. Below we provide the details for the BRW implementation for 31-block inputs.

On a 31-block input, BRW requires a total of 15 n-bit multiplications. There is some amount of parallelism in these multiplications. A convenient way to bring out this parallelism is to represent the BRW computation using a tree as has been done in [33]. Such a tree depicts the dependencies among the multiplications required for BRW computation. We omit the details of how the tree is constructed as these details are not directly relevant to the present work.

The relevant part of a 31-block BRW tree is shown in Figure 5.1. Each node is marked by an even number between 2 and 30 corresponding to the 15 multiplications that are required. (For the reason why the node labels are 2 to 30 instead of 1 to 15, we refer to [33].) If there is an edge from a lower marked node to a higher marked node, then the multiplication corresponding to the lower marked node has to be computed before the multiplication corresponding to the higher marked node. So, for example, the multiplication corresponding to node 2 has to be computed before the multiplication corresponding to node 4 can be computed and the multiplications corresponding to nodes 8, 12 and 14 have to be computed before the multiplication corresponding to node 16 can be computed.

Figure 5.1: The 31-block BRW tree.



Nodes which are not connected by an edge are independent and can be computed in

parallel. For example, the eight multiplications at the lowest level are independent; the four multiplications at the next level are independent; and so on. There are, however, other ways to group the independent multiplications. Such groupings allow using batch multiplications to speed up the computations. Using batch size 3 as given below is particularly nice since the 15 multiplications can be cleanly grouped into 5 batches of 3 multiplications each.
Batch size 3: $\{2, 6, 10\}, \{14, 18, 22\}, \{26, 30, 4\}, \{20, 12, 8\}, \{28, 24, 16\}$.

In conjunction with the above, we also implemented the delayed reduction strategy described in Section 5.3.3. From Table 5.3, for $\eta = 31$, 15 multiplications of n-bit polynomials, 8 reductions and 52 n-bit XORs are required.

5.3.5 Decimated Horner

Given a sequence of n-bit blocks $m_1, \ldots, m_\ell$ and a positive integer $d \geq 1$, $\mathsf{Horner}_\tau(m_1, \ldots, m_\ell)$ can be computed as

$$\begin{aligned}\mathsf{Horner}_\tau(m_1, \ldots, m_\ell) \;=\; & \tau^{\rho-1}\mathsf{Horner}_{\tau^d}(m_1, m_{d+1}, m_{2d+1}, \ldots) \\ & \oplus \cdots \\ & \oplus\ \tau^{\rho-\rho}\mathsf{Horner}_{\tau^d}(m_\rho, m_{d+\rho}, m_{2d+\rho}, \ldots) \\ & \oplus\ \tau^{d-1}\mathsf{Horner}_{\tau^d}(m_{\rho+1}, m_{d+\rho+1}, m_{2d+\rho+1}, \ldots) \\ & \oplus\ \cdots \\ & \oplus\ \tau^{d-(d-\rho)}\mathsf{Horner}_{\tau^d}(m_d, m_{2d}, m_{3d}, \ldots) \end{aligned} \tag{5.9}$$

where $\rho = \ell \bmod d$. We call this d-decimated Horner computation. In (5.9), the d calls to Horner are independent leading to d independent multiplications at each step with the boundary conditions appropriately handled. These d independent multiplications can be computed as a batch multiplication. After the individual Horner calls are completed, the outputs are multiplied by $\tau^{\rho-1}, \ldots, 1, \tau^{d-1}, \ldots, \tau^\rho$ which can be done as a batch multiplication with batch size $d - 1$ (since one multiplication is by 1).

5.3.6 Implementation of Hash2L

During implementation, there is a choice of batch size for BRW. For $n = 128$, we have found that choosing the batch size to be 3 provides slightly better speed compared to choosing the batch size to be 1. So, for $n = 128$, we implemented BRW using batch size 3 and 3-decimated Horner. For $n = 256$, however, there does not seem to be any noticeable improvement in speed by choosing the batch size to be greater than 1. So, in this case, we implemented both BRW and Horner using batch size 1.

Timing results on the Haswell and the Skylake processors are presented in Tables 5.4 and 5.5 respectively. The percentage figures indicate the percentage of speed improvement obtained by our implementation of Hash2L over the publicly available implementations of GHASH and POLYVAL. The implementation of GHASH is by Gueron and has been taken from [59]. The implementation uses a delayed reduction strategy whereby a single reduction is done per four polynomial multiplications. This strategy requires pre-computing a table

	128-bit				256-bit			
	length of message in bytes				length of message in bytes			
	512	1024	4096	8192	512	1024	4096	8192
Hash2L	0.88	0.687	0.498	0.463	1.4	0.95	0.718	0.67
GHASH [59]	1.15	1.02	0.93	0.91	–	–	–	–
	(23.5%)	(32.6%)	(46.5%)	(49.1%)	–	–	–	–
POLYVAL [57]	1.09	0.81	0.602	0.567	–	–	–	–
	(19.3%)	(15.2%)	(17.3 %)	(18.3%)	–	–	–	–

Table 5.4: Cycles per byte for computing Hash2L, GHASH and POLYVAL on Haswell. For both $n = 128$ and $n = 256$, Karatsuba gave better performance compared to the schoolbook method.

	128-bit				256-bit			
	length of message in bytes				length of message in bytes			
	512	1024	4096	8192	512	1024	4096	8192
Hash2L	0.667	0.468	0.33	0.301	1.11	0.758	0.562	0.525
GHASH [59]	0.89	0.77	0.67	0.65	–	–	–	–
	(25.1%)	(39.2%)	(50.7%)	(53.7%)	–	–	–	–
POLYVAL [57]	0.79	0.55	0.369	0.339	–	–	–	–
	(15.6%)	(14.9%)	(10.6%)	(11.2%)	–	–	–	–

Table 5.5: Cycles per byte for computing Hash2L, GHASH and POLYVAL on Skylake. For $n = 128$, schoolbook was faster than Karatsuba, while for $n = 256$, Karatsuba was faster.

consisting of 4 consecutive powers of the hash key. The implementation of POLYVAL is also by Gueron and has been taken from [57]. This implementation also uses a delayed reduction strategy, but here a single reduction is done per eight polynomial multiplications. Hence, it requires a pre-computed table consisting of 8 consecutive powers of the hash key. Thus, the implementation of POLYVAL in [57] requires half the number of reductions required by the implementation of GHASH in [59] which leads to significant speed up. We note that an implementation of GHASH using one reduction per 8 polynomial multiplications will have the same performance as that of the implementation of POLYVAL in [57].

For POLYVAL, both intrinsics and assembly codes are provided and it is mentioned that the performance of both the codes are similar. Since, we have implemented in intrinsics, we chose to compare to the intrinsics implementation in [57]. We measured the time required by the intrinsics implementation of GHASH in [59] and of POLYVAL in [57] on the same machine where we measured the time required by Hash2L.

For timing each of Hash2L, GHASH and POLYVAL, the hash key was updated in every iteration. This ensured that the timing measurements included the time for pre-computing the powers of τ in case of Hash2L and the pre-computed tables in Gueron's implementations of both GHASH and POLYVAL.

From the results in Tables 5.4 and 5.5 we find that Hash2L is about 23% to 49% faster than GHASH and about 15% to 19% faster than POLYVAL on the Haswell processor. On Skylake processor, these figures are about 25% to 53% and 10% to 15% respectively.

In theory, the number of multiplications required by Hash2L is slightly more than half the number of multiplications required by GHASH or POLYVAL. This, however, does not directly turn into a roughly two times speed improvement for the following reasons. First, the strategy of delayed reduction used in GHASH and POLYVAL to some extent mitigates the effect of requiring about two times as many multiplications for short messages. Second, the code for GHASH is much more simpler and smaller than that for Hash2L and this has an effect on the overall speed.

To summarise, the speed improvements that we are able to achieve are indicative of the algorithmic superiority of Hash2L over Horner based hash computations such as GHASH and POLYVAL. We do not claim that our code provides the fastest possible timing for Hash2L. Experts on intrinsics and assembly programming should be able to tune the code to achieve even higher speeds. Further, we have considered only $\eta = 31$ for implementation. It would be interesting to explore the speed achievable using other values of η. We leave these as interesting work for the future.

5.4 Implementation Strategy Without Using `pclmulqdq`

For $n = 256$, Bernstein and Chou [16] have provided a description of how to implement binary field arithmetic using the Fast Fourier Transform (FFT) algorithm. The method does not require the `pclmulqdq` instruction. The following counts of number of bit operations are provided in [16]. Forward Fourier transform: $4068 - 656 = 3412$ bit operations without radix conversions; pointwise multiplications: $64 \cdot 110$ bit operations; inverse Fourier transform: 5996 bit operations; reduction: 992 bit operations.

In the FFT based polynomial multiplication, the inverse Fourier transform is applied to the pointwise product. As pointed out in [16], to compute an expression of the type $\alpha_1\alpha_2 + \beta_1\beta_2$, it is equivalent to compute the pointwise multiplications for α_1, α_2 and β_1, β_2; add the vectors; and then perform a single inverse Fourier transform. So, whenever a sum of products of polynomials is to be computed, a single inverse Fourier transform suffices. In the present context, this means that the number of inverse Fourier transforms to be computed is equal to the number of reductions.

We consider the use of this strategy for computing Hash2L. The polynomial multiplication and reduction procedures used in [16] can be directly considered in the present context.

Suppose $\eta = 2^{\mathfrak{a}+1} - 1 \geq 3$. From Table 5.3, computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ requires $14 \cdot 2^{\mathfrak{a}-2} - 4$ 256-bit XORs; $2(2^{\mathfrak{a}} - 1)$ forward Fourier transforms (each polynomial multiplication requires two forward Fourier transforms); $2^{\mathfrak{a}} - 1$ pointwise multiplications; $2^{\mathfrak{a}-1}$ inverse Fourier transforms; and $2^{\mathfrak{a}-1}$ reductions. The total number of bit operations for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ comes to

$$256(14 \cdot 2^{\mathfrak{a}-2} - 4) + 2 \cdot 3412 \cdot (2^{\mathfrak{a}} - 1) + (64 \cdot 110)(2^{\mathfrak{a}} - 1) + 5996 \cdot 2^{\mathfrak{a}-1} + 992 \cdot 2^{\mathfrak{a}-1}$$
$$= 18254 \cdot 2^{\mathfrak{a}} - 14888. \quad (5.10)$$

The number of bits in $(m_1, \ldots, m_\eta)$ is $256\eta = 256 \cdot (2^{\mathfrak{a}+1} - 1)$ and so the number of bit operations per bit for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ is

$$\mathfrak{B}_{\mathfrak{a}+1} = \frac{18254 \cdot 2^{\mathfrak{a}} - 14888}{256 \cdot (2^{\mathfrak{a}+1} - 1)}.$$

We have $\mathfrak{B}_2 \approx 28.2$, $\mathfrak{B}_3 \approx 32.4$, $\mathfrak{B}_4 \approx 34.2$, $\mathfrak{B}_5 \approx 34.9$, $\mathfrak{B}_6 \approx 35.3$, $\mathfrak{B}_7 \approx 35.5$.

For Hash2L having $\eta\ell$ 256-bit blocks, there are ℓ super-blocks consisting of η 256-bit blocks each. Processing these super-blocks require $\mathfrak{B}_{\mathfrak{a}+1}$ bit operations per block. Additionally, the ℓ blocks which are produced as the output of the ℓ BRW invocations are processed using Horner. For achieving AXU, this requires ℓ field multiplications. In the multiplications of Horner computation, one of the operands is always $\tau^{2^{\mathfrak{a}+1}}$ and so the number of forward Fourier transforms is $\ell+1$ (one transform for each of the ℓ blocks, plus a transform for $\tau^{d(\eta)+1}$) instead of 2ℓ. In addition to these, there are ℓ pointwise multiplications; ℓ inverse Fourier transforms; ℓ reductions; and ℓ 256-bit XORs. Plugging in the number of bit operations for each of the aforementioned operations shows that a total of $17696\ell + 3412$ bit operations are required for evaluating Horner. Since there are a total of $\eta\ell$ 256-bit blocks, the number of bit operations per bit for evaluating Horner is $(17696\ell + 3412)/(256 \cdot \eta\ell) = 69.125/(2^{\mathfrak{a}+1} - 1) + 13.22/(\ell(2^{\mathfrak{a}+1} - 1))$.

There is an additional cost for computing the powers $\tau^2, \tau^4, \ldots, \tau^{2^{\mathfrak{a}+1}}$. Each of these is a squaring and requires 17440 bit operations for a total of $17440 \cdot \mathfrak{a}$ bit operations to compute all the powers. Amortised over the entire computation, the cost per bit for computing the powers is $(17440 \cdot \mathfrak{a})/(256 \cdot \eta\ell)$.

So, the total number of bit operations per bit for computing Hash2L on $\eta\ell$ 256-bit blocks with $\eta = 2^{\mathfrak{a}+1} - 1$ is

$$\mathfrak{C}_{\mathfrak{a}+1} = \mathfrak{B}_{\mathfrak{a}+1} + 69.125/(2^{\mathfrak{a}+1} - 1) + (68.125 \cdot \mathfrak{a} + 13.22)/(\ell(2^{\mathfrak{a}+1} - 1)).$$

We have $\mathfrak{C}_2 \approx 51.2 + 27.1/\ell$, $\mathfrak{C}_3 \approx 42.3 + 21.4/\ell$, $\mathfrak{C}_4 \approx 38.8 + 14.5/\ell$, $\mathfrak{C}_5 \approx 37.2 + 9.2/\ell$, $\mathfrak{C}_6 \approx 36.4 + 5.6/\ell$, $\mathfrak{C}_7 \approx 36.0 + 3.3/\ell$.

Choosing $\eta = 31 = 2^5 - 1$ shows that the number of bit operations per bit for computing Hash2L is $37.2 + 9.2/\ell \leq 46.4$ By choosing $\eta = 63$ or 127, it is possible to lower the number of bit operations per bit though this is still greater than the 29 bit operations per bit required for the pseudo-dot product [16]. The main reason behind the cost of Hash2L being higher than that of the pseudo-dot product is that in the later case, there is a single inverse Fourier transform and a single reduction for the entire computation. The problem with the pseudo-dot product, however, is that the hash key is as long as the message. The cost of securely generating this key will be significant and has not been considered in [16].

Remark: The complete Hash2L requires an additional multiplication to process the block containing the message length. The above cost measure does not include this multiplication. The reason is that a complete hash function based on the pseudo-dot product will also require such a multiplication and this is not covered by the figure of 29 bit operations per bit reported in [16].

5.5 Message Authentication Code

As mentioned in Chapter 2, a well known method for constructing a nonce-based MAC scheme from a hash function is the following [110]. Let $F : \mathcal{K} \times \mathcal{N} \rightarrow \{0,1\}^n$ be a mapping and $\{H_\tau\}_{\tau \in \mathsf{T}}$ with $H_\tau : \mathcal{M} \rightarrow \{0,1\}^n$ be a hash family. The key space for the MAC scheme is $\mathcal{K} \times \mathsf{T}$, the nonce space is $\mathcal{N}$ and the message space is $\mathcal{M}$. Given a nonce N and a message M, the output of the MAC scheme under a key (K, τ) is

$$(N, M) \xrightarrow{(K,\tau)} F_K(N) \oplus H_\tau(M). \tag{5.11}$$

It is possible to instantiate the hash function H using Hash2L. In this case, the message M is a binary string. More generally, it is possible to instantiate H using vecHash2L in which case the message M is a vector where each component is a binary string. The function F_K can be either a block cipher or a stream cipher.

Analysis of this scheme under the assumption that F is either a pseudo-random function (PRF) or a pseudo-random permutation (PRP) has a long history starting from [110] with the best known bounds appearing in [14]. If F is instantiated using a stream cipher, then security is based on the assumption that F is a PRF while if F is instantiated using a block cipher, then security is based on the assumption that F is a PRP. The overall security bound for the MAC scheme is obtained from the security assumption on F and the AXU bound on H_τ. These bounds are derived in [14] and so we do not repeat them here.

Instantiation at the 128-bit security level: It is possible to use a 128-bit block cipher such as AES to instantiate F_K. The size of K could be any of the options allowed for AES and the size of N will be 128 bits. It is also possible to instantiate F using a stream cipher whose key size is at least 128 bits.

Instantiation at the 256-bit security level: A 128-bit block cipher such as AES cannot be directly used to instantiate F at the 256-bit security level. Instead, a stream cipher supporting a 256-bit key can be directly used to instantiate F.

5.6 Comparison to Some Previous Works

We consider some of the important universal hash functions and corresponding MAC schemes that have been proposed. The discussion is divided into two parts. In the first part, we consider schemes for which the keys to the hash function are long and in the second part, we consider schemes for which the keys to the hash functions are short.

5.6.1 Comparison to Schemes Using Long Hash Keys

Some important and popular hash functions which fall under this category are NH^{T} used in UMAC, VHASH used in VMAC, Hash256 used in Auth256 and CLHASH. More details about these are already discussed in Chapter 2.

For hash functions using long keys, in practice, the key has to be generated using either a stream cipher or a block cipher mode of operation. This leads to both efficiency and security issues as mentioned below.

Efficiency: Generation of the key can be either done on the fly, or, it could be pre-computed and stored. Both the approaches have problems. Generating the key on the fly requires significant additional time which should be included in the total time for hashing. However, the above mentioned schemes do not report this time. On the other hand, pre-computing and storing a large key has its own problems. To quote Bernstein [14], the large key "creates a huge speed penalty: cache misses become much more common and much more expensive."

Security: The analysis of the scheme given in (5.11) is well known when K and τ are chosen independently and uniformly at random with the best known bounds appearing in [14]. However, if τ is generated using a mode of operation of a block or a stream cipher, then there are two issues. If the key for the mode of operation used to generate τ is the same as that of F, then the independence condition is violated. Even if the key for the mode of operation is independent of the key for F, using a mode of operation to generate τ violates the uniform distribution property of τ. Consequently, if τ is generated using a mode of operation, the analysis and the bounds provided in [14] do not direcly apply and a fresh analysis and security bound need to be worked out. In fact, there has been a lengthy discussion on this issue [107] in the context of UMAC where Bernstein had strongly argued for the necessity of precise security statement and proof for UMAC. By the same reasoning, a precise security statement and proof is also required for Auth256 which is not available in [16].

While the above issues are relevant for hash functions which use long keys, we note below two issues which are particularly relevant to Hash256 and Auth256.

1. Hash256 avoids using `pclmulqdq` under the rationale that not all processors provide this instruction. Consider this issue in conjunction with the requirement of generating the hash key using AES in counter mode. Processors which do not provide an instruction similar to `pclmulqdq` are unlikely to provide support in the instruction set for AES. So, on such processors, the generation of the hash key will take significantly more time than the actual hashing. This time is neither reported nor considered in [16].

2. The digest size of Hash256 is 256 bits and so the goal of Auth256 is the 256-bit security level. It is suggested in [16] that the hash key can be generated using counter-mode AES. Since AES is a 128-bit block cipher, a direct use of counter-mode AES will not provide security at the 256-bit level. So, a combination of Hash256 with counter-mode AES is unlikely to provide 256-bit security. A further issue is that of instantiating F in (5.11) using AES. The output of F is required to be 256 bits long and since AES is a 128-bit cipher, it cannot be directly used to instantiate F. Since [16] does not provide a clear description of how the hash key for Hash256 is to be generated and how F is to be instantiated, the acutal security claim of Auth256 at the 256-bit level is unclear.

In terms of efficiency, [16] reports a cost of 29 bit operations per bit for computing Hash256 along with a hidded cost of generating the hash key. Any secure method for generating the long hash key will have a significant cost. In Section 5.4, we have shown that choosing $\eta = 31$ leads to a cost of at most 46.4 bit operations per bit. There is, however, no associated hidden cost of generating the hash key. The cost can be made lower by choosing a higher value of η. While the comparison in terms of bit operation counts is indicative, it would have been better to obtain the actual speed measurements. Since the code for Hash256 is not (yet) publicly available, we were unable to do this.

5.6.2 Comparison to Schemes Using Short Hash Keys

Some important and popular hash functions which fall under this category are Poly1305, GHASH and POLYVAL. More details about these hash functions are provided in Chapter 2.

All three of Poly1305, GHASH and POLYVAL are computed using Horner and hence, require $\ell-1$ multiplications for evaluating an ℓ-block message. The design approach proposed here, on the other hand, requires a little more than $\ell/2$ multiplications. So, inherently this approach is faster than each of those hash schemes. We have instantiated this approach over binary fields to develop Hash2L. On the other hand, if one wishes to work over prime fields, it is equally possible to instantiate the approach over any appropriate field such as $\mathbb{F}_{2^{130}-5}$.

The hash key for Poly1305, GHASH, POLYVAL and also Hash2L is a single field element. So, in terms of key agility, there is no difference between these four algorithms. The collision probabilities for Poly1305, GHASH and POLYVAL are those obtained from the usual Horner style hash and hence are only slightly lower than that of Hash2L. See Table 5.1 for more details.

Poly1305, GHASH and POLYVAL are designed for the 128-bit security level. The speeds of GHASH, POLYVAL and Hash2L for both Haswell and Skylake processors have been stated earlier. For Poly1305, in Haswell, the best reported speed we could find is 0.65 cycles/byte using 64-bit AVX2 instructions[2]. The instantiation of Hash2L at the 128-bit security level turns out to be faster than all of these functions on Haswell processor of Intel; it is faster than GHASH and POLYVAL on Skylake; and we were unable to locate a speed report for Poly1305 on Skylake. We expect the 128-bit version of Hash2L to be faster than GHASH and POLYVAL on any platform and to be faster than Poly1305 on any processor which supports the `pclmulqdq` instruction. The comparison of Hash2L to Poly1305 on processors which do not provide support for `pclmulqdq` cannot be determined without getting into the details of a particular processor.

5.7 Summary

In this chapter, we have proposed a single-key two-level universal hash function which combines the advantages given by Horner's rule based hash and BRW polynomial based hash function. One important aspect here is an efficient software implementation for high-end

[2] `https://www.openssl.org/blog/blog/2016/02/15/poly1305-revised/`

Intel processors. Comparative timing measurements show that this hash function is significantly faster than highly efficient implementations of GHASH and POLYVAL by Gueron. This indicates that this hash function is current state-of-the-art universal hash as far as the efficiency in high end processors is concerned. The security bound is same as given by the standard universal hashes.

Chapter 6

Variants of Wegman-Carter Message Authentication Code Supporting Variable Tag Lengths

As already mentioned in Chapter 2, most MAC schemes, in the literature so far, specify a single value for the tag length. The question that we explore here is the following. Is it possible to have a variable tag length MAC scheme? While the question seems to be a natural one, there does not appear to have been much discussion about this issue in the literature. There is an almost 15-year old CFRG [107] discussion pertaining to different tag lengths suggested for the MAC scheme UMAC [71]. This scheme had the possibility of using 32-bit, 64-bit, 96-bit and 128-bit tags. Finney [47], crediting "Dan Bernstein's poly1305-aes mailing list", had pointed out that this feature would allow forging a 64-bit tag using about 2^{33} queries. A later post [48] explains the issue further and suggests how a valid 128-bit tag can be obtained with only about 2^{34} queries. Wagner [108] supporting the issue raised by Finney, had mentioned that to fix the problem "it suffices to ensure that the tag length is a parameter that is immutably bound to the key and never changed. In other words, never use the same key with different parameter sizes." Following this suggestion, Section 6.5 of the UMAC specification [71] states that a "UMAC key (or session) must have an associated and immutable tag length". Another suggestion put forward by Finney [48] to handle the issue requires "stealing two bits of input into the block cipher from the nonce and using them to encode tag size". More recently a variable tag-length MAC scheme KMAC [69] has been proposed by Kelsey at al., based on the sponge function KECCAK [17].

The question of variable tag length received some attention in the past few years in the context of authenticated encryption (AE) schemes and the CAESAR [26] competition. Manger [80] pointed out that for the AE scheme OCB, 64-bit, 96-bit and 128-bit tags are defined where the "64-bit and 96-bit tags are simply truncated 128-bit tags". This leads to simple truncation attacks on the scheme. An earlier paper by Rogaway and Wagner [93] had also discussed the problem of variable tag lengths in the context of the AE scheme CCM. A formal treatment of variable tag length AE schemes has been given by Reyhanitabar, Vaudenay and Vizár [92].

Two concrete motivations are provided in [92] as to why a variable tag length AE scheme may indeed be desirable in practice. The first mentions that variable tag lengths may be used with the same key due to "misuse and poorly engineered security systems". The second reason is that for resource constrained devices, variable tag lengths may be desirable though changing the key for every tag length may be infeasible due to limited bandwidth and low power.

While the above two reasons have been put forward in the context of AE schemes, they are equally valid for MAC schemes. More generally, the issue of "mis-implementation" (also called "footguns") [90] of cryptographic primitives has been extensively discussed as part of

the discussion forum on post-quantum cryptography.

More concretely, Auth256 [16] is a Wegman-Carter type construction targeted at the 256-bit security level. Similarly, a 256-bit secure universal hash function has been proposed in [30], which can be mated to a 256-bit secure PRF using the Wegman-Carter template to obtain a 256-bit secure MAC. Such MAC schemes would be appropriate for high-security applications, or, for a post-quantum world. On the other hand, bandwidth limited applications would require shorter tags. Also, the possibility of mis-implementation using tag truncation remains. So, the question of designing a MAC scheme which can support various tag lengths up to 256 bits is of practical interest.

To summarise, the problem of constructing variable tag length MAC schemes did not gain much attention, though, it is of contemporary and future practical interest.

In this chapter, we provide a formalisation of the notion of security for a variable tag length MAC scheme. For the same key, the desired tag length is to be provided as part of the input to the tag generation algorithm. Consequently, in the security model, we allow the adversary to control the tag length as well as the message. This is an extension of the usual security model for MAC schemes.

We consider the problem of obtaining secure variable tag length MAC schemes. The Wegman-Carter [110] scheme is the classical nonce-based MAC scheme. A naive approach to obtain a variable tag length MAC scheme is to truncate tags produced by the Wegman-Carter scheme. We show an easy attack on such a truncation scheme. Next, we consider eight possible "natural" variants that arise from the Wegman-Carter MAC scheme. We show attacks on six of these schemes. These attacks do not repeat nonces for tag generation queries. Among the attacked schemes is the scheme obtained by nonce stealing following the suggestion of Finney [47] as mentioned in Chapter 2. One of the eight schemes is generically secure since it uses independent keys for different tag lengths. The last of the eight schemes is proved to be secure. This scheme uses nonce stealing *but*, for different tag lengths, it uses independent keys for the universal hash function component of the Wegman-Carter scheme.

From a practical point of view, it is desirable to have a scheme which uses a single key. The key for the hash function is then derived from the key of the scheme and the tag length. The manner in which such derivation is made depends upon the primitive used to derive the hash key. We show two methods of deriving the hash key. The first method uses a stream cipher while the second method uses a short output length pseudo-random function (PRF). So, in effect, we obtain two constructions of single key variable tag length MAC scheme.

All the schemes that we describe can be instantiated by readily available concrete cryptographic primitives. For example, either of the 256-bit secure universal hash functions in [16, 30] can be combined with Salsa20 [12] to obtain nonce-based MAC schemes supporting variable tag lengths up to 256 bits. So, here we provide templates for designing efficient and practical MAC schemes which support variable tag lengths.

This chapter is based on the work [53].

Relation to the work of Reyhanitabar et al. [92]: The notion of authenticated encryption with associated data (AEAD) which can support variable tag lengths was introduced

in [92]. An AEAD scheme has two algorithms, namely encryption and decryption. The encryption algorithm takes as input a nonce, a plaintext, an associated data and a tag length and returns the corresponding ciphertext; while the decryption algorithm takes as input a nonce, a ciphertext, an associated data and a tag length and either returns $\perp$ indicating that the input is improper, or, returns the corresponding plaintext. Such an AEAD scheme can be considered to be a nonce-based MAC scheme where the plaintext is always fixed to the empty string and the message to be authenticated is provided as the associated data. With this modification, the formalisation of the authenticity of the AEAD scheme in [92] turns out to be the same as the formalisation of the variable tag length nonce-based MAC scheme introduced in this chapter. The difference between our formalisation and that of [92] is in the treatment of adversarial resources. We have considered the notion of query profile, while the usual notion of query complexity has been considered in [92]. In terms of construction, the contribution of [92] is different from ours. A variant of OCB [74] is considered in [92], while we describe variants of the Wegman-Carter scheme.

6.1 Definitions

Throughout this chapter, n is a fixed positive integer.

We consider an ε-AXU family of hash functions $\{H_\tau\}_{\tau\in\mathsf{T}}$, where for each $\tau \in \mathsf{T}$, $H_\tau : \mathcal{M} \to \{0,1\}^n$. Typically, a message is a binary string of some maximum length.

For the pseudo-random function (PRF) $\{F_K\}_{K\in\mathcal{K}}$, $F_K : \mathcal{D} \to \mathcal{R}$ as well, we consider $\mathcal{D}$ and $\mathcal{R}$ to be finite non-empty sets of binary strings.

6.1.1 Variable Tag Length Nonce-Based Message Authentication Code

A MAC scheme has two algorithms, namely, the tag generation algorithm and the verification algorithm. Typically, in a MAC scheme, tags are binary strings of some fixed length. The definition of MAC schemes, however, does not require tags to have the same length. So, it is possible to consider variable length tags within the ambit of the currently used definition of MAC schemes.

Our goal, on the other hand, is different. We would like the tag length to be provided as part of the input to the tag generation and verification algorithms. So, for the same message, by providing different values of the tag length, it is possible to generate tags of different lengths. This feature is not covered by the presently used definition of MAC schemes. We extend the syntax of MAC schemes and the definition of security to incorporate this feature.

A nonce-based MAC scheme is given by the message space $\mathcal{M}$, the nonce space $\mathcal{N}$, the key space $\mathcal{K}$, the allowed set $\mathcal{L}$ of tag lengths, the tag space $\mathcal{T}$; and two algorithms $\mathsf{nvMAC.Gen}(K, N, m, \lambda)$ and $\mathsf{nvMAC.Verify}(K, N, m, \mathsf{tag}, \lambda)$, where $K \in \mathcal{K}$, $N \in \mathcal{N}$, $m \in \mathcal{M}$, $\lambda \in \mathcal{L}$ and $\mathsf{tag} \in \mathcal{T}$. We consider $\mathcal{M}$, $\mathcal{N}$, $\mathcal{K}$ and $\mathcal{L}$ to be finite non-empty sets and $\mathcal{T}$ to be equal to $\cup_{i\in\mathcal{L}}\{0,1\}^i$. We write $\mathsf{nvMAC.Gen}_K(N, m, \lambda)$ to denote $\mathsf{nvMAC.Gen}(K, N, m, \lambda)$, and similarly $\mathsf{nvMAC.Verify}_K(N, m, \mathsf{tag}, \lambda)$ to denote $\mathsf{nvMAC.Verify}(K, N, m, \mathsf{tag}, \lambda)$.

The inputs and outputs of $\mathsf{nvMAC.Gen}_K(N, m, \lambda)$ and $\mathsf{nvMAC.Verify}_K(N, m, \mathsf{tag}, \lambda)$ are as follows.

- $\mathsf{nvMAC.Gen}_K(N, m, \lambda)$:
  input: $K \in \mathcal{K}$; $N \in \mathcal{N}$; $m \in \mathcal{M}$; and $\lambda \in \mathcal{L}$.
  output: $\mathsf{tag} \in \mathcal{T}$ is a binary string of length λ.

- $\mathsf{nvMAC.Verify}_K(N, m, \mathsf{tag}, \lambda)$:
  input: $K \in \mathcal{K}$; $N \in \mathcal{N}$; $m \in \mathcal{M}$; $\mathsf{tag} \in \mathcal{T}$; and $\lambda \in \mathcal{L}$ such that tag is of length λ. Note that for a correct implementation it must be checked that the tag provided is actually of length λ.
  output: an element from the set $\{\mathsf{true}, \mathsf{false}\}$. The value true indicates that the input is accepted while the value false indicates that the input is rejected.

The following correctness condition must hold.

$$\mathsf{nvMAC.Verify}_K(N, m, \mathsf{nvMAC.Gen}_K(N, m, \lambda), \lambda) \;=\; \mathsf{true}.$$

Security: The security for a (nonce-based) MAC scheme against an adversary $\mathcal{A}$ is modelled as follows. Suppose K is chosen uniformly at random from $\mathcal{K}$ and the tag generation and verification algorithms are instantiated with K. $\mathcal{A}$ is given oracle access to the tag generation and the verification algorithms. $\mathcal{A}$ makes a total of q_g queries to the tag generation oracle and a total of q_v queries to the verification oracle. The queries are made adaptively and queries to the tag generation oracle can be interleaved with those to the verification oracle.

Let the queries to the tag generation oracle be

$$\left(N_g^{(1)}, m_g^{(1)}, \lambda_g^{(1)}\right), \ldots, \left(N_g^{(q_g)}, m_g^{(q_g)}, \lambda_g^{(q_g)}\right)$$

and the corresponding responses be $\mathsf{tag}_g^{(1)}, \ldots, \mathsf{tag}_g^{(q_g)}$ respectively. Similarly, let the queries to the verification oracle be

$$\left(N_v^{(1)}, m_v^{(1)}, \mathsf{tag}_v^{(1)}, \lambda_v^{(1)}\right), \ldots, \left(N_v^{(q_v)}, m_v^{(q_v)}, \mathsf{tag}_v^{(q_v)}, \lambda_v^{(q_v)}\right)$$

and the corresponding responses be $\mathsf{xxx}_v^{(1)}, \ldots, \mathsf{xxx}_v^{(q_v)}$ respectively, where for $1 \leq j \leq q_v$, $\mathsf{xxx}_v^{(j)}$ is either true or false. The query profile of $\mathcal{A}$ is the list

$$\begin{aligned}\mathfrak{C} \;=\; (q_g, q_v, (\mathfrak{n}_g^{(1)}, \mathfrak{m}_g^{(1)}, \lambda_g^{(1)}), \ldots, (\mathfrak{n}_g^{(q_g)}, \mathfrak{m}_g^{(q_g)}, \lambda_g^{(q_g)}), (\mathfrak{n}_v^{(1)}, \mathfrak{m}_v^{(1)}, \lambda_v^{(1)}),\\ \ldots, (\mathfrak{n}_v^{(q_v)}, \mathfrak{m}_v^{(q_v)}, \lambda_v^{(q_v)}))\end{aligned} \tag{6.1}$$

where for $1 \leq s \leq q_g$, $\mathfrak{n}_g^{(s)} = \mathsf{len}(N_g^{(s)}), \mathfrak{m}_g^{(s)} = \mathsf{len}(m_g^{(s)})$ and for $1 \leq s \leq q_v$, $\mathfrak{n}_v^{(s)} = \mathsf{len}(N_v^{(s)}), \mathfrak{m}_v^{(s)} = \mathsf{len}(m_v^{(s)})$.

There are two restrictions on the adversary. The first is a weaker form of nonce-respecting behaviour, namely, $\left(N_g^{(i)}, \lambda_g^{(i)}\right) \neq \left(N_g^{(j)}, \lambda_g^{(j)}\right)$ for $1 \leq i < j \leq q_g$. Note that the adversary is allowed to repeat (nonce, tag-length) pair for verification queries and it is also allowed to

re-use a (nonce, tag-length) pair used in a tag generation query in one or more verification queries. Usual nonce-respecting behaviour requires the nonces in the tag generation queries to be distinct. By relaxing this condition, we provide the adversary with more power. So, a scheme proved secure against the weaker form of nonce-respecting behaviour maintains security even if nonces are repeated in tag generation queries as long as the (nonce, tag-length) pairs are distinct. The second restriction on the adversary is that it should not make any useless query. A query is useless if its response can be computed by the adversary. This means that the adversary should not repeat a query to the tag generation oracle or the verification oracle; and it should not query the verification oracle with $\left(N_g^{(i)}, m_g^{(i)}, \mathsf{tag}_g^{(i)}, \lambda_g^{(i)}\right)$ for any i in $\{1, \ldots, q_g\}$, where the query made to the tag generation oracle should be prior to the current verification query.

The adversary makes a number of verification queries. The tag lengths of these queries could be different. There is no restriction on the adversary to choose a target tag length before making the queries to its oracles. For any tag length λ, the adversary is successful if a verification query for this tag length returns true. So, for any value of the tag length, there is a corresponding event that the adversary is successful for a particular tag length. Formally, for $\lambda \in \mathcal{L}$, let $\mathsf{succ}_{\mathcal{A}}(\lambda)$ be the event that there is some $j \in \{1, \ldots, q_v\}$ such that $\lambda_v^{(j)} = \lambda$ and $\mathsf{nvMAC.Verify}_K\left(N_v^{(j)}, m_v^{(j)}, \mathsf{tag}_v^{(j)}, \lambda_v^{(j)}\right)$ returns true. For each $\lambda \in \mathcal{L}$, the adversary's advantage in breaking the authenticity of nvMAC is defined to be $\Pr[\mathsf{succ}_{\mathcal{A}}(\lambda)]$. This is written as follows.

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathcal{A}) \;=\; \Pr\left[\mathsf{succ}_{\mathcal{A}}(\lambda)\right]. \tag{6.2}$$

The above probability is taken over the uniform random choice of K from $\mathcal{K}$ and over the possible internal randomness of the adversary $\mathcal{A}$.

Given a query profile $\mathfrak{C}$, $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathfrak{T}, \mathfrak{C})$ is the maximum of $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathcal{A})$ taken over all adversaries running in time $\mathfrak{T}$ and having query profile $\mathfrak{C}$.

Remark: The security model allows nonces to be repeated with different tag lengths. As explained above, this provides the adversary with more power. We further note that allowing nonces to be reused with different tag lengths permits generation of fewer nonces which may be of interest in some resource-constrained applications. At this point though, we are unable to provide a concrete example.

Security in terms of query complexity: The query complexity is the total number of bits sent by the adversary in all its queries. For tag generation queries, this consists of the number of bits sent as part of the nonces, the messages and the λ_g's; for verification queries, this consists of the number of bits sent as part of the nonces, the messages, the tags and the λ_v's. Let the q_g tag generation queries require a total of σ_g bits and the q_v verification queries require a total of σ_v bits. So, $\sigma_g = \sum_{1 \le i \le q_g}(\mathsf{len}(N_g^{(i)}) + \mathsf{len}(m_g^{(i)}) + \mathsf{len}(\lambda_g^{(i)})) = \sum_{1 \le i \le q_g}(\mathfrak{n}_g^{(i)} + \mathfrak{m}_g^{(i)} + \mathsf{len}(\lambda_g^{(i)}))$ and $\sigma_v = \sum_{1 \le i \le q_v}(\mathsf{len}(N_v^{(i)}) + \mathsf{len}(m_v^{(i)}) + \mathsf{len}(\mathsf{tag}_v^{(i)}) + \mathsf{len}(\lambda_v^{(i)})) = \sum_{1 \le i \le q_v}(\mathfrak{n}_v^{(i)} + \mathfrak{m}_v^{(i)} + \lambda_v^{(i)} + \mathsf{len}(\lambda_v^{(i)}))$, as $\mathsf{len}(\mathsf{tag}_v^{(i)}) = \lambda_v^{(i)}$. If the elements of $\mathcal{L}$ are expressed as $\mathfrak{t}$-bit binary strings, then $\sigma_g = \sum_{1 \le i \le q_g}(\mathfrak{n}_g^{(i)} + \mathfrak{m}_g^{(i)}) + q_g\mathfrak{t}$ and $\sigma_v = \sum_{1 \le i \le q_v}(\mathfrak{n}_v^{(i)} + \mathfrak{m}_v^{(i)} + \lambda_v^{(i)}) + q_v\mathfrak{t}$.

Given query complexity (σ_g, σ_v), $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathfrak{T}, \sigma_g, \sigma_v)$ is the maximum of $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathcal{A})$ taken over all adversaries $\mathcal{A}$ running in time up to $\mathfrak{T}$ and having query complexity (σ_g, σ_v).

Given a query profile $\mathfrak{C}$ of any adversary $\mathcal{A}$ the corresponding query complexity (σ_g, σ_v) can be readily derived in the above manner. On the other hand, it is to be noted that, various query profiles can have the same query complexity. Hence, in the security definition above in terms of query complexity, when one maximises over query complexity, the value obtained is the maximum over all possible query profiles which have that same query complexity. This gives us the following relation.

Definition 5. *Let us fix a query complexity (σ_g, σ_v) and let $\mathcal{C}_{(\sigma_g,\sigma_v)}$ be the set of all query profiles having query complexity (σ_g, σ_v), i.e.,*

$$\mathcal{C}_{(\sigma_g,\sigma_v)} := \{\mathfrak{C} : \textit{the query complexity of } \mathfrak{C} \textit{ is } (\sigma_g, \sigma_v)\}.$$

Then,

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathfrak{T}, \sigma_g, \sigma_v) = \mathsf{max}_{\mathfrak{C}\in\mathcal{C}_{(\sigma_g,\sigma_v)}} \mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathfrak{T}, \mathfrak{C}). \tag{6.3}$$

Later we explain the rationale for considering query profiles.

Information theoretic security: This consists of analysing the security of a MAC scheme against a computationally unbounded adversary. In other words, the probability in (6.2) is considered for an adversary $\mathcal{A}$ without any reference to the run time of $\mathcal{A}$. For such a computationally unbounded adversary $\mathcal{A}$, without loss of generality, we may assume $\mathcal{A}$ to be deterministic. In the context of information theoretic security, given a query profile $\mathfrak{C}$, $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathfrak{C})$ is the maximum of $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda](\mathcal{A})$ taken over all adversaries $\mathcal{A}$ having query profile $\mathfrak{C}$.

6.2 Towards Building a Variable Tag Length MAC

It may appear that a variable tag length nonce-based MAC scheme can be obtained simply by truncating the output of the Wegman-Carter MAC algorithm. This, however, does not work as we show in this section. We further consider several "natural" extensions of the Wegman-Carter MAC algorithm and show that most of them are insecure. Only two of these extensions are secure: one of them is a generic construction, while we prove the security of the other in the next section. Overall, the discussion in the present section may be considered as showing the subtlety involved in constructing a variable tag length nonce-based MAC scheme.

Let $\mathcal{N}$ be the nonce space and $\mathcal{M}$ be the message space. Let $\{F_K\}_{K\in\mathcal{K}}$ be a PRF such that $F_K : \mathcal{N} \to \{0,1\}^n$; let $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ be an AXU hash function such that $\mathsf{Hash}_\tau : \mathcal{M} \to \{0,1\}^n$. Given $\{F_K\}_{K\in\mathcal{K}}$ and $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$, the Wegman-Carter MAC [110] is the following. A nonce-message pair (N, m) is mapped under a key (K, τ) to $F_K(N) \oplus \mathsf{Hash}_\tau(m)$, i.e.,

$$\mathsf{WC\text{-}nvMAC} : (N, m) \xrightarrow{(K,\tau)} F_K(N) \oplus \mathsf{Hash}_\tau(m). \tag{6.4}$$

Below we argue that several natural extensions of WC-nvMAC are not secure. We assume that binary representation of tag lengths fit within a byte. The attacks are shown for the following specific choice of the hash function. Under a fixed representation of the elements of the finite field $\mathbb{F}_{2^n}$, we identify the elements of $\mathbb{F}_{2^n}$ with the set $\{0,1\}^n$. The specific hash function that we consider is $\mathsf{Hash}_\tau(m) = \tau m$, i.e., the output of $\mathsf{Hash}_\tau(m)$ is the n-bit string representing the product of τ and m considered as elements of $\mathbb{F}_{2^n}$. This hash function is known to be AXU. Attacks on schemes built using this specific hash function is sufficient to show that the schemes described below are not secure for an arbitrary AXU hash function. The choice of the hash function fixes the key space of the hash function to be $\mathsf{T} = \mathbb{F}_{2^n}$ and the message space $\mathcal{M}$ to be either $\mathbb{F}_{2^n}$ or $\mathbb{F}_{2^{n-8}}$, depending on the scheme.

We will use the following simple fact about the specific hash function that we consider.

Proposition 4. *Consider the AXU hash function* $\{\mathsf{Hash}_\tau\}_{\tau\in\mathbb{F}_{2^n}}$ *where* $\mathsf{Hash}_\tau(m) = \tau m$. *Let* m_1 *and* m_2 *be distinct elements of* $\mathbb{F}_{2^n}$ *and* c *be such that* $\mathsf{Hash}_\tau(m_1) \oplus \mathsf{Hash}_\tau(m_2) = c$, *then* $\tau = c(m_1 \oplus m_2)^{-1}$.

The most obvious approach to obtain a variable tag length scheme from (6.4) is to truncate the output, i.e.,

$$\mathsf{trunc}: \ (N, m, \lambda) \xrightarrow{(K,\tau)} \mathsf{msb}_\lambda(\mathsf{WC\text{-}nvMAC}_{K,\tau}(N, m)) = \mathsf{msb}_\lambda(F_K(N) \oplus \mathsf{Hash}_\tau(m)).$$

The scheme trunc is not secure as can be seen from the following attacks. Note that in this case the message space is $\mathbb{F}_{2^n}$.

Attack 1 on trunc: Let m be a message and N be a nonce. The adversary makes a tag generation query (N, m, n) and gets in response t. Now the adversary makes a verification query $(N, m, \mathsf{msb}_{n-1}(\mathsf{t}), n-1)$ and it is successful with probability 1. Thus the adversary makes a successful forgery with only one tag generation query.

Attack 2 on trunc: Another attack which repeats nonces in tag generation queries and reveals more information is the following. Let m_1, m_2 and m_3 be distinct messages and N be a nonce. The adversary makes two tag generation queries (N, m_1, n) and $(N, m_2, n-1)$ and gets in response t_1 and t_2 respectively. So, we have the following relations: $F_K(N) \oplus \mathsf{Hash}_\tau(m_1) = \mathsf{t}_1$ and $\mathsf{msb}_{n-1}(F_K(N) \oplus \mathsf{Hash}_\tau(m_2)) = \mathsf{t}_2$. From the second relation, it follows that either $F_K(N) \oplus \mathsf{Hash}_\tau(m_2) = \mathsf{t}_2||0$ or $F_K(N) \oplus \mathsf{Hash}_\tau(m_2) = \mathsf{t}_2||1$. Using Proposition 4, the adversary solves the equations $\mathsf{Hash}_\tau(m_1) \oplus \mathsf{Hash}_\tau(m_2) = \mathsf{t}_1 \oplus (\mathsf{t}_2||0)$ and $\mathsf{Hash}_\tau(m_1) \oplus \mathsf{Hash}_\tau(m_2) = \mathsf{t}_1 \oplus (\mathsf{t}_2||1)$ for τ to obtain the solutions τ_0 and τ_1 respectively. As $F_K(N) \oplus \mathsf{Hash}_\tau(m_2)$ takes exactly one of the two values $\mathsf{t}_2||0$ or $\mathsf{t}_2||1$, τ takes exactly one of the two values τ_0 or τ_1. Let $y_0 = \mathsf{t}_1 \oplus \mathsf{Hash}_{\tau_0}(m_1)$. The adversary makes a verification query $(N, m_3, y_0 \oplus \mathsf{Hash}_{\tau_0}(m_3), n)$. If the verification query is successful then τ_0 is the correct value of τ. If the verification query fails, then τ_1 is the correct value of τ. Thus the adversary recovers the hash key with two tag generation and one verification queries.

The first attack shows that a simple truncation of the Wegman-Carter MAC scheme does not work while the second attack shows that by repeating nonces in tag generation queries

Table 6.1: For the schemes in (6.5) to (6.12), a summary of whether the input and/or the key of F and/or Hash depend on the tag length λ.

scheme	F		Hash		secure?
	i/p	key	i/p	key	
nvMAC-t1	yes	no	no	no	no
nvMAC-t2	no	no	yes	no	no
nvMAC-t3	yes	no	yes	no	no
nvMAC-Generic	no	yes	no	yes	yes
nvMAC-t4	no	yes	no	no	no
nvMAC-t5	no	yes	yes	no	no
nvMAC-t6	no	no	no	yes	no
nvMAC	yes	no	no	yes	yes

the hash key can be obtained. One possibility of modifying trunc is to apply F_K a second time before applying truncation, i.e., the tag is obtained as $\mathsf{msb}_\lambda(F_K(F_K(N) \oplus \mathsf{Hash}_\tau(m)))$. The resulting scheme is also not secure. The first simple attack on trunc also works for this modified scheme.

In the scheme trunc, the output of neither F nor Hash depends on λ. To rectify this situation, one may introduce λ as part of the input of one or both of F and Hash. Another possibility is to have one or both of the keys K and τ to depend on λ. Key dependencies are achieved by using a family of independent keys $\{K_\lambda\}_{\lambda\in\mathcal{L}}$ and/or a family of independent keys $\{\tau_\lambda\}_{\lambda\in\mathcal{L}}$. The various schemes that arise from such considerations are as follows.

$$\mathsf{nvMAC\text{-}t1}_{K,\tau} : (N,m,\lambda) \xrightarrow{(K,\tau)} \mathsf{msb}_\lambda(F_K(\mathsf{bin}_8(\lambda)||N) \oplus \mathsf{Hash}_\tau(m)). \tag{6.5}$$

$$\mathsf{nvMAC\text{-}t2}_{K,\tau} : (N,m,\lambda) \xrightarrow{(K,\tau)} \mathsf{msb}_\lambda(F_K(N) \oplus \mathsf{Hash}_\tau(\mathsf{bin}_8(\lambda)||m)). \tag{6.6}$$

$$\mathsf{nvMAC\text{-}t3}_{K,\tau} : (N,m,\lambda) \xrightarrow{(K,\tau)} \mathsf{msb}_\lambda(F_K(\mathsf{bin}_8(\lambda)||N) \oplus \mathsf{Hash}_\tau(\mathsf{bin}_8(\lambda)||m)). \tag{6.7}$$

$$\mathsf{nvMAC\text{-}Generic}_{(K_\lambda,\tau_\lambda)_{\lambda\in\mathcal{L}}} : (N,m,\lambda) \xrightarrow{(K_\lambda,\tau_\lambda)} \mathsf{msb}_\lambda(F_{K_\lambda}(N) \oplus \mathsf{Hash}_{\tau_\lambda}(m)). \tag{6.8}$$

$$\mathsf{nvMAC\text{-}t4}_{(K_\lambda,\tau)_{\lambda\in\mathcal{L}}} : (N,m,\lambda) \xrightarrow{(K_\lambda,\tau)} \mathsf{msb}_\lambda(F_{K_\lambda}(N) \oplus \mathsf{Hash}_\tau(m)). \tag{6.9}$$

$$\mathsf{nvMAC\text{-}t5}_{(K_\lambda,\tau)_{\lambda\in\mathcal{L}}} : (N,m,\lambda) \xrightarrow{(K_\lambda,\tau)} \mathsf{msb}_\lambda(F_{K_\lambda}(N) \oplus \mathsf{Hash}_\tau(\mathsf{bin}_8(\lambda)||m)). \tag{6.10}$$

$$\mathsf{nvMAC\text{-}t6}_{(K,\tau_\lambda)_{\lambda\in\mathcal{L}}} : (N,m,\lambda) \xrightarrow{(K,\tau_\lambda)} \mathsf{msb}_\lambda(F_K(N) \oplus \mathsf{Hash}_{\tau_\lambda}(m)). \tag{6.11}$$

$$\mathsf{nvMAC}_{(K,\tau_\lambda)_{\lambda\in\mathcal{L}}} : (N,m,\lambda) \xrightarrow{(K,\tau_\lambda)} \mathsf{msb}_\lambda(F_K(\mathsf{bin}_8(\lambda)||N) \oplus \mathsf{Hash}_{\tau_\lambda}(m)). \tag{6.12}$$

Dependencies of input and/or key on λ for the above schemes are summarised in Table 6.1.

Nonce stealing: Finney [47] had suggested that the nonce may be reduced by a few bits and a binary encoding of the tag length be inserted. In the present context, this refers to letting the input of F depend on the tag length. From Table 6.1, we see that the schemes

nvMAC-t1, nvMAC-t3 and nvMAC use nonce stealing. While nvMAC is secure (as proved later), schemes nvMAC-t1 and nvMAC-t3 are insecure. So, nonce stealing by itself does not guarantee security.

For the ensuing discussion, we will consider the message space for the schemes nvMAC-t1, nvMAC-Generic, nvMAC-t4 and nvMAC-t6 to be $\mathbb{F}_{2^n}$, and that for the schemes nvMAC-t2, nvMAC-t3 and nvMAC-t5 to be $\mathbb{F}_{2^{n-8}}$.

Algorithm 3 describes an attack on nvMAC-t1 which uses findTag as a subroutine. In the attack, the tag generation and verification oracles are denoted by $\mathcal{O}_g$ and $\mathcal{O}_v$ respectively. On being supplied with input (N, m, λ), the function $\mathsf{findTag}(N, m, \lambda)$ finds tag such that $(N, m, \mathsf{tag}, \lambda)$ passes the test by the verification oracle. To do this, findTag repeatedly queries the verification oracle, until a suitable tag is obtained. The expected number of queries made by $\mathsf{findTag}(N, m, \lambda)$ is 2^λ. Algorithm 3 invokes findTag with values of the tag length which are less than the target tag length.

The intuition behind the attack in Algorithm 3 is the following. The key (K, τ) of the scheme does not depend on λ. In particular, as the hash key τ does not depend on λ, the attack retrieves τ using a smaller value of λ and uses it for the forgery with the target λ successfully. Retrieving τ using a smaller value of λ requires significantly less number of oracle queries than that required for an attack by exhaustive search for the target λ. The analysis of the attack is given in Proposition 5. This divide-and-conquer attack strategy of using shorter tag length to learn information, with low cost, which is useful for longer tag lengths has previously been used in the context of AE [41, 92].

Algorithm 3 Attack on nvMAC-t1 for $\lambda = n$.

set $\lambda \leftarrow n$;
choose $\lambda_1 \in \mathcal{L}$, such that $\lambda_1 < \lambda$;
choose distinct $N_1, N_2 \in \mathcal{N}$ and distinct $m_1, m_2, m_3, m_4 \in \mathcal{M}$;
$\mathsf{tag}^{(1)} \leftarrow \mathcal{O}_g(N_1, m_1, \lambda_1)$;
$\mathsf{tag}^{(2)} \leftarrow \mathsf{findTag}(N_1, m_2, \lambda_1)$;
set $\mathcal{C} \leftarrow \{\}$;
do
    choose $c \leftarrow \{0,1\}^{n-\lambda_1} \setminus \mathcal{C}$;
    set $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$;
    using Proposition 4 solve $\mathsf{Hash}_\tau(m_1) \oplus \mathsf{Hash}_\tau(m_2) = (\mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)})||c$
        for τ and let the solution be τ_c;
    set $m_c \leftarrow \mathsf{tag}^{(1)} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_c}(m_1))$;
    $\mathcal{R}_v^{(3)} \leftarrow \mathcal{O}_v(N_1, m_3, m_c \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_c}(m_3)), \lambda_1)$;
while $\mathcal{R}_v^{(3)} = \mathsf{false}$;
$\mathsf{tag}^{(4)} \leftarrow \mathcal{O}_g(N_2, m_4, \lambda)$;
choose any $m \in \mathcal{M} \setminus \{m_4\}$;
return $(N_2, m, \mathsf{Hash}_{\tau_c}(m) \oplus \mathsf{Hash}_{\tau_c}(m_4) \oplus \mathsf{tag}^{(4)}, \lambda)$.

Proposition 5. *The attack given in Algorithm 3 on the scheme* nvMAC-t1 *given in (6.5) produces a forgery for tag length λ which is correct with probability 1. It requires one tag*

$\mathsf{findTag}(N, m, \lambda)$

1: set $\mathcal{D} \leftarrow \{\}$;
2: **do**
3: choose $\mathsf{tag} \leftarrow \{0,1\}^{\lambda} \setminus \mathcal{D}$;
4: set $\mathcal{D} \leftarrow \mathcal{D} \cup \mathsf{tag}$;
5: $\mathcal{R}_v \leftarrow \mathcal{O}_v(N, m, \mathsf{tag}, \lambda)$;
6: **while** $\mathcal{R}_v = \mathsf{false}$
7: return tag.

---

generation query and at most $2^{\lambda_1} + 2^{n-\lambda_1}$ *verification queries on tag length* λ_1 *and one tag generation query and one verification query on tag length* λ.

Proof. That the attack mentioned in Algorithm 3 forges with probability 1 is proved if it can be shown that the forgery returned by the attack in Step 17 is accepted, i.e. the corresponding response from $\mathcal{O}_v$ is true.
From Step 4 we get,

$$\mathsf{msb}_{\lambda_1}(F_K(\mathsf{bin}_8(\lambda_1)||N_1) \oplus \mathsf{Hash}_{\tau}(m_1)) = \mathsf{tag}^{(1)}. \tag{6.13}$$

The $\mathsf{tag}^{(2)}$ returned by Step 5 satisfies

$$\mathsf{msb}_{\lambda_1}(F_K(\mathsf{bin}_8(\lambda_1)||N_1) \oplus \mathsf{Hash}_{\tau}(m_2)) = \mathsf{tag}^{(2)}. \tag{6.14}$$

So, from (6.13) and (6.14) we get,

$$\mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau}(m_1) \oplus \mathsf{Hash}_{\tau}(m_2)) = \mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)}. \tag{6.15}$$

Here $\mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)}$ is a λ_1-bit binary string. Following Proposition 4, for each choice of c in the do-while loop in Steps 7 to 14, the equation in Step 10 can be solved to get τ_c and m_c. The fact that $\mathsf{Hash}_{\tau}(m_1) \oplus \mathsf{Hash}_{\tau}(m_2) \in \{0,1\}^n$ and (6.15) suggest that there is a correct c, such that the equation in Step 10 holds and we consider that iteration of the do-while loop which deals with this particular c. The τ_c obtained in this iteration is the actual hash key used in the scheme. So,

$$\begin{aligned}
&\mathsf{nvMAC\text{-}t1}(N_1, m_3, \lambda_1) \\
&= \mathsf{msb}_{\lambda_1}(F_K(\mathsf{bin}_8(\lambda_1)||N_1) \oplus \mathsf{Hash}_{\tau_c}(m_3)) \\
&= \mathsf{tag}^{(1)} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_c}(m_1)) \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_c}(m_3)) \qquad (6.16) \\
&= m_c \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_c}(m_3)). \qquad (6.17)
\end{aligned}$$

The expression in (6.16) comes from (6.13) and that in (6.17) comes from Step 12 in Algorithm 3. Hence, in this particular iteration of the do-while loop, $\mathcal{R}_v^{(3)} = \mathsf{true}$ and the loop terminates.

Since $\lambda = n$, from Step 15 we obtain $F_K(\mathsf{bin}_8(\lambda)||N_2) = \mathsf{Hash}_{\tau_c}(m_4) \oplus \mathsf{tag}^{(4)}$. For the choice of m in Step 16, i.e., $m \in \mathcal{M} \setminus \{m_4\}$ we have

$$\begin{aligned}
\mathsf{nvMAC\text{-}t1}(N_2, m, \lambda) &= F_K(\mathsf{bin}_8(\lambda)||N_2) \oplus \mathsf{Hash}_{\tau_c}(m) \\
&= \mathsf{Hash}_{\tau_c}(m_4) \oplus \mathsf{tag}^{(4)} \oplus \mathsf{Hash}_{\tau_c}(m), \qquad (6.18)
\end{aligned}$$

which is returned as the tag for (N_2, m, λ) in the forgery and hence, the corresponding response from $\mathcal{O}_v$ is true with probability 1, which proves the first part of the result.

In the attack, there are 2 tag generation queries in Steps 4 and 15. The subroutine $\mathsf{findTag}$ makes a maximum of 2^{λ_1} verification queries on tags of lengths λ_1. The do-while loop in Steps 7 to 14 iterates at most $2^{n-\lambda_1}$ times for different values of c making a maximum of $2^{n-\lambda_1}$ verification queries on tags of lengths λ_1. The forgery returned in Step 17 is a verification query on a tag of length λ. Hence, the attack requires 2 tag generation queries and at most $2^{\lambda_1} + 2^{n-\lambda_1} + 1$ verification queries including the forgery. □

Remarks:

1. One may note that here we consider variable length tags. So, the adversary can make verification queries for a particular tag length and provide a forgery for another tag length. The attack given in Algorithm 3 on the scheme $\mathsf{nvMAC\text{-}t1}$, forges the scheme with an n-bit tag, i.e. the attack is for tag length n; whereas, as shown in Proposition 5, the attack requires 2 tag generation queries and $2^{\lambda_1} + 2^{n-\lambda_1} + 1$ verification queries including the forgery, where $\lambda_1 < \lambda$. Among these queries, 1 tag generation query and $2^{\lambda_1} + 2^{n-\lambda_1}$ verification queries are with tag length λ_1. For example, suppose $n = 128$, and let $\lambda_1 = 64$. So, the attack uses $2^{65} + 1 < 2^{128}$ verification queries and produces a forgery for tag length 128. This constitutes a valid attack for tag length 128.

2. The security model for variable length tag nonce-based MAC allows nonces in tag generation queries to be repeated as long as the tag lengths are distinct. The attack in Algorithm 3 does not repeat nonces in tag generation queries. So, the scheme $\mathsf{nvMAC\text{-}t1}$ is insecure even under the restriction that nonces in tag generation queries are distinct.

Insecurities of the schemes $\mathsf{nvMAC\text{-}t1}$ to $\mathsf{nvMAC\text{-}t5}$ follow from applications of Algorithm 3.

Attack on $\mathsf{nvMAC\text{-}t2}$: Algorithm 3 works with the only modification that the forgery is changed to $(N_2, m, \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m_4) \oplus \mathsf{tag}^{(4)} \oplus \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m), \lambda)$.

Attack on $\mathsf{nvMAC\text{-}t3}$: Algorithm 3 works with the only modification that the forgery is changed to $(N_2, m, \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m) \oplus \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m_4) \oplus \mathsf{tag}^{(4)}, \lambda)$.

Attack on $\mathsf{nvMAC\text{-}t4}$: Algorithm 3 works with the only modification that the forgery is changed to $(N_2, m, \mathsf{Hash}_{\tau_c}(m) \oplus \mathsf{Hash}_{\tau_c}(m_4) \oplus \mathsf{tag}^{(4)}, \lambda)$.

Attack on $\mathsf{nvMAC\text{-}t5}$: Algorithm 3 works with the only modification that the forgery is changed to $(N_2, m, \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m) \oplus \mathsf{Hash}_{\tau_c}(\mathsf{bin}_8(\lambda)||m_4) \oplus \mathsf{tag}^{(4)}, \lambda)$.

Algorithm 4 describes an attack on $\mathsf{nvMAC\text{-}t6}$ which also uses $\mathsf{findTag}$ as a subroutine; $\mathcal{O}_g$ and $\mathcal{O}_v$ are as described before.

Proposition 6. *The attack given in Algorithm 4 on the scheme* $\mathsf{nvMAC\text{-}t6}$ *produces a forgery for tag length* λ *which is correct with probability 1. It requires at most* $2^{\lambda_1+1} + 2^{n-\lambda_1}$ *verification queries on tag length* λ_1 *and one tag generation query and at most* $2^{n-\lambda_1}$ *verification queries on tag length* λ.

Algorithm 4 Attack on nvMAC-t6 for $\lambda = n$:

set $\lambda \leftarrow n$;
choose $\lambda_1 \in \mathcal{L}$, such that $\lambda_1 < \lambda$;
choose any $N_1 \in \mathcal{N}$ and distinct $m_1, m_2, m_3, m_4, m \in \mathcal{M}$;
$\mathsf{tag}^{(1)} \leftarrow \mathsf{findTag}(N_1, m_1, \lambda_1)$;
$\mathsf{tag}^{(2)} \leftarrow \mathsf{findTag}(N_1, m_2, \lambda_1)$;
set $\mathcal{C}_1 \leftarrow \{\}$;
do
    choose $c_1 \leftarrow \{0,1\}^{n-\lambda_1} \setminus \mathcal{C}_1$;
    set $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{c_1\}$;
    using Proposition 4 solve $\mathsf{Hash}_{\tau_{\lambda_1}}(m_1) \oplus \mathsf{Hash}_{\tau_{\lambda_1}}(m_2) = (\mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)})||c_1$
        for τ_{λ_1} and let the solution be τ_{c_1};
    set $m_{c_1} \leftarrow \mathsf{tag}^{(1)} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{c_1}}(m_1))$;
    $\mathcal{R}_v^{(3)} \leftarrow \mathcal{O}_v(N_1, m_3, m_{c_1} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{c_1}}(m_3)), \lambda_1)$;
while $\mathcal{R}_v^{(3)} = \mathsf{false}$;
$\mathsf{tag}^{(4)} \leftarrow \mathcal{O}_g(N_1, m_4, \lambda)$;
set $\mathcal{C}_2 \leftarrow \{\}$;
do
    choose $c_2 \leftarrow \{0,1\}^{n-\lambda_1} \setminus \mathcal{C}_2$;
    set $\mathcal{C}_2 \leftarrow \mathcal{C}_2 \cup \{c_2\}$;
    solve $\mathsf{Hash}_{\tau_\lambda}(m_4) = \mathsf{msb}_{\lambda_1}(\mathsf{tag}^{(4)}) \oplus m_{c_1}||c_2$
        for τ_λ and let the solution be τ_{c_2};
    set $m_{c_2} \leftarrow (\mathsf{msb}_{\lambda_1}(\mathsf{tag}^{(4)}) \oplus m_{c_1}||c_2) \oplus \mathsf{tag}^{(4)}$;
    $\mathcal{R}_v^{(5)} \leftarrow \mathcal{O}_v(N_1, m, m_{c_2} \oplus \mathsf{Hash}_{\tau_{c_2}}(m), \lambda)$;
while $\mathcal{R}_v^{(5)} = \mathsf{false}$.

Proof. That the attack mentioned in Algorithm 4 forges with probability 1 is proved if it can be shown that there is an iteration of the do-while loop in Steps 17 to 24 such that $\mathcal{R}_v^{(5)} = \mathsf{true}$, i.e. there is a verification query in Step 23 which succeeds.
From Steps 4 and 5, we get that

$$\mathsf{msb}_{\lambda_1}(F_K(N_1) \oplus \mathsf{Hash}_{\tau_{\lambda_1}}(m_1)) = \mathsf{tag}^{(1)}. \tag{6.19}$$

$$\mathsf{msb}_{\lambda_1}(F_K(N_1) \oplus \mathsf{Hash}_{\tau_{\lambda_1}}(m_2)) = \mathsf{tag}^{(2)}. \tag{6.20}$$

So,

$$\mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{\lambda_1}}(m_1) \oplus \mathsf{Hash}_{\tau_{\lambda_1}}(m_2)) = \mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)}. \tag{6.21}$$

Here $\mathsf{tag}^{(1)} \oplus \mathsf{tag}^{(2)}$ is a λ_1-bit binary string.

Following Proposition 4, for each choice of c_1 in the do-while loop in Steps 7 to 14, the equation in Step 10 can be solved to get τ_{c_1} and m_{c_1}. The fact that $\mathsf{Hash}_{\tau_{\lambda_1}}(m_1) \oplus \mathsf{Hash}_{\tau_{\lambda_1}}(m_2) \in \{0,1\}^n$ and (6.21) suggest that there is a correct c_1, such that the equation in Step 10 holds and we consider that iteration of the do-while loop which deals with this particular c_1. The τ_{c_1} obtained in this iteration is the actual hash key used in the scheme. So,

$$\begin{aligned}
&\mathsf{nvMAC\text{-}t6}(N_1, m_3, \lambda_1) \\
&= \mathsf{msb}_{\lambda_1}(F_K(N_1) \oplus \mathsf{Hash}_{\tau_{c_1}}(m_3)) \\
&= \mathsf{tag}^{(1)} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{c_1}}(m_1)) \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{c_1}}(m_3)) \qquad (6.22) \\
&= m_{c_1} \oplus \mathsf{msb}_{\lambda_1}(\mathsf{Hash}_{\tau_{c_1}}(m_3)). \qquad (6.23)
\end{aligned}$$

The expression in (6.22) comes from (6.19) and that in (6.23) comes from Step 12 in Algorithm 4. Hence, in this particular iteration of the do-while loop, $\mathcal{R}_v^{(3)} = \mathsf{true}$ and the loop terminates.

Noting that $\lambda = n$, from Step 15, we get

$$F_K(N_1) \oplus \mathsf{Hash}_{\tau_\lambda}(m_4) = \mathsf{tag}^{(4)} \;\Rightarrow\; \mathsf{Hash}_{\tau_\lambda}(m_4) = \mathsf{tag}^{(4)} \oplus F_K(N_1). \tag{6.24}$$

Here, the n bits of $\mathsf{tag}^{(4)}$ and $\mathsf{msb}_{\lambda_1}(\cdot)$ of $F_K(N_1)$, which is m_{c_1}, are known. As $\mathsf{Hash}_{\tau_\lambda}(m_4) \in \{0,1\}^n$, there is a $c_2 \in \{0,1\}^{n-\lambda_1}$, such that,

$$\mathsf{Hash}_{\tau_\lambda}(m_4) = \mathsf{msb}_{\lambda_1}(\mathsf{tag}^{(4)} \oplus F_K(N_1))||c_2 = (\mathsf{msb}_{\lambda_1}(\mathsf{tag}^{(4)}) \oplus m_{c_1})||c_2. \tag{6.25}$$

For the correct choice of c_2, the correct values of τ_{c_2} and m_{c_2} are obtained in Steps 21 and 22 respectively. For the correct c_2, from (6.24) and (6.25), we get,

$$F_K(N_1) = \mathsf{Hash}_{\tau_\lambda}(m_4) \oplus \mathsf{tag}^{(4)} = ((\mathsf{msb}_{\lambda_1}(\mathsf{tag}^{(4)}) \oplus m_{c_1})||c_2) \oplus \mathsf{tag}^{(4)}, \tag{6.26}$$

which equals m_{c_2} according to Step 22 in Algorithm 4. Hence,

$$\mathsf{nvMAC\text{-}t6}(N_1, m, \lambda) = F_K(N_1) \oplus \mathsf{Hash}_{\tau_{c_2}}(m) = m_{c_2} \oplus \mathsf{Hash}_{\tau_{c_2}}(m). \tag{6.27}$$

The last equality follows from (6.26). From (6.27), it is clear that for the iteration of the do-while loop in Steps 17 to 24, in which the correct c_2 is used, $\mathcal{R}_v^{(5)} = \mathsf{true}$ with probability 1, which proves the first part of the Lemma.

Steps 4 and 5 each require at most 2^{λ_1} verification queries for tag length λ_1. Step 13 requires at most $2^{n-\lambda_1}$ verification queries for tag length λ_1. A tag generation query for tag length λ is made in Step 15 and at most $2^{n-\lambda_1}$ verification queries are made for tag length λ in Step 23. This shows the complexity of the attack. □

Remarks:

1. With $\lambda = n$ suppose $\lambda_1 = n/2$. Then the adversary makes a maximum of $3 \cdot 2^{n/2}$ verification queries for tag length $n/2$, one tag generation query and at most $2^{n/2}$ verification queries for tag length n. It produces a forgery for tag length n which is correct with probability 1. So, this is a valid forgery attack for tag length n.

2. Algorithm 4 makes a single tag generation query. Hence, the issue of repeating nonces in tag generation queries does not arise.

The scheme nvMAC-Generic can be considered to be a collection of $\#\mathcal{L}$ independent WC-nvMAC schemes, one for each value of λ. Each of the individual schemes for fixed values of λ are already known to be secure, since the proof from [14] applies to the individual schemes where the values of λ are fixed. Since the keys of the various schemes are independent, it can be argued that the collection is also secure. The problem, however, is that size of the key increases by a factor of $\#\mathcal{L}$. So, nvMAC-Generic cannot be considered to be a practical solution to the problem of obtaining a variable tag length MAC scheme.

The first step towards reducing key size is taken in the scheme nvMAC which uses a single key K for F and independent keys τ_λ. In the next section, we prove nvMAC to be secure and also consider further variants with smaller keys.

Remark: Suppose nvMAC-t1 is modified to obtain a scheme nvMAC-t1′ in the following manner. The tag is obtained as $\mathsf{msb}_\lambda(F_K(F_K(\mathsf{bin}_8(\lambda)||N) \oplus \mathsf{Hash}_\tau(m)))$, i.e., a second application of F_K is made before truncating. It is not difficult to show that the scheme mapping (N, m, λ), under the key (K, τ), to the quantity $F_K(F_K(\mathsf{bin}_8(\lambda)||N) \oplus \mathsf{Hash}_\tau(m))$ is a PRF. It can be argued that nvMAC-t1′ is a secure variable tag length MAC scheme. However, the security bound for nvMAC-t1′ will be in the order of $q^2\varepsilon$, where the total number of queries is q and the hash function is ε-AXU. This bound is higher than the bounds obtained for the schemes that we consider. Hence, we do not consider nvMAC-t1′. In the above discussion, we have considered modification of nvMAC-t1 to nvMAC-t1′. The same comments apply to similar modifications of the other insecure schemes, namely nvMAC-t2 to nvMAC-t6.

6.3 Secure and Efficient MAC Schemes with Variable Length Tag

We start with the scheme nvMAC given in (6.12). We carry out an information theoretic analysis of this scheme. To this end, we consider the scheme obtained by replacing F_K with a random function $f : \{0,1\}^n \to \{0,1\}^n$. The tag generation algorithm for this scheme is shown in Table 6.2. We require a hash family $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$, where for each $\tau \in \mathsf{T}$, $\mathsf{Hash}_\tau : \mathcal{M} \to \{0,1\}^n$, with $\mathcal{M} = \cup_{i=0}^{L}\{0,1\}^i$ for some sufficiently large positive integer L.

The nonce space for the scheme nvMAC is $\mathcal{N} = \{0,1\}^{n-8}$ and the message space is $\mathcal{M}$. Let $\mathcal{L} \subseteq \{1,\ldots,\min(256,n)\}$ be the allowed set of tag lengths. Note that tag length equal to zero is not allowed and there are 256 possible values of the tag length that are supported. If λ is the tag length, then $\lambda - 1$ is at most 255 and consequently fits within a byte. So, instead of encoding λ, we encode $\lambda - 1$. This is a modification that we make to the scheme given in (6.12). Note that larger (or smaller) values of $\#\mathcal{L}$ can be considered by suitably adjusting the length of the nonces. From a practical point of view, however, it is difficult to think of any application which would require $\#\mathcal{L}$ to be more than 256.

The key space for nvMAC is $\mathsf{T}^{\#\mathcal{L}}$, i.e., a particular key is a tuple $(\tau_\lambda)_{\lambda\in\mathcal{L}}$. The key generation algorithm consists of choosing τ_λ independently and uniformly at random from T for each λ. The verification algorithm is as follows. Given $(N, m, \mathsf{tag}, \lambda)$, compute $\mathsf{tag}' = \mathsf{nvMAC.Gen}_{(\tau_\lambda)_{\lambda\in\mathcal{L}}}(N, m, \lambda)$; if $\mathsf{tag} = \mathsf{tag}'$ then return true, else return false.

Here f is a random function but, not necessarily a uniform random function. Given q pairs $(a_1, b_1), \ldots, (a_q, b_q)$, the q-interpolation probability [14] of f is defined to be $\Pr[f(a_1) = b_1, \ldots, f(a_q) = b_q]$. Following the analysis in [14], the security bound for the resulting scheme is obtained in terms of the interpolation probability of f. Known bounds on the interpolation probability of uniform random function and uniform random permutation provide the corresponding bounds on the security of the resulting nvMAC schemes.

Table 6.2: A secure and efficient nvMAC scheme from a random function.

$\mathsf{nvMAC.Gen}_{(\tau_\lambda)_{\lambda\in\mathcal{L}}}(N, m, \lambda)$
$Q = f(\mathsf{bin}_8(\lambda - 1)\|\|N);$
$R = Q \oplus \mathsf{Hash}_{\tau_\lambda}(m);$
$\mathsf{tag} = \mathsf{msb}_\lambda(R);$
return tag.

Theorem 6. *In the scheme* nvMAC *defined in Table 6.2, suppose that the hash function* $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ *is* ε*-AXU, where* $\varepsilon(\ell, \ell') \geq 1/2^n$ *for all* $\ell, \ell' \leq L$.

Fix a query profile $\mathfrak{C}$. *For* $\lambda \in \mathcal{L}$, *let* $q_{g,\lambda}$ *(resp.* $q_{v,\lambda}$*) be the number of tag generation (resp. verification) queries for* λ *which are in* $\mathfrak{C}$. *Let* λ *be such that* $q_{v,\lambda} \geq 1$ *and for* $1 \leq i \leq q_{v,\lambda}$, *let* $Q_{v,\lambda}^{(i)} = (N_{v,\lambda}^{(i)}, m_{v,\lambda}^{(i)}, \mathsf{tag}_{v,\lambda}^{(i)}, \lambda)$ *be the* i*-th verification query with tag length* λ. *Let* $\ell_{v,\lambda}^{(i)} = \mathsf{len}(m_{v,\lambda}^{(i)})$. *Corresponding to* $Q_{v,\lambda}^{(i)}$, *there is at most one tag generation query*

$Q^{(i^\star)}_{g,\lambda} = (N^{(i^\star)}_{g,\lambda}, m^{(i^\star)}_{g,\lambda}, \lambda)$ *such that* $N^{(i)}_{v,\lambda} = N^{(i^\star)}_{g,\lambda}$. *Let* $\ell^{(i^\star)}_{g,\lambda} = \mathsf{len}(m^{(i^\star)}_{g,\lambda})$ *if there is such a* $Q^{(i^\star)}_{g,\lambda}$, *otherwise* $\ell^{(i^\star)}_{g,\lambda}$ *is undefined.*

Fix $\lambda_0 \in \mathcal{L}$. *Let* $\mathcal{S}_{\lambda_0}$ *be the set of all queries made by the adversary other than the verification queries for tag length* λ_0. *Suppose that the queries in* $\mathcal{S}_{\lambda_0}$ *give rise to at most* q *distinct (nonce, tag-length) values. Further, suppose* δ_i *be such that the* i*-interpolation probability of* f *is at most* $\delta_i/(2^n)^i$. *Then*

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda_0](\mathfrak{T}, \mathfrak{C}) \;\leq\; \frac{1}{2^{\lambda_0}} \times \sum_{1 \leq i \leq q_{v,\lambda_0}} \gamma_i \tag{6.28}$$

where $\gamma_i = 2^n \delta_q \varepsilon \left(\ell^{(i)}_{v,\lambda_0}, \ell^{(i^\star)}_{g,\lambda_0}\right)$ *if there is a* $Q^{(i^\star)}_{g,\lambda_0}$ *corresponding to* $Q^{(i)}_{v,\lambda_0}$ *with* $N^{(i)}_{v,\lambda_0} = N^{(i^\star)}_{g,\lambda_0}$; *otherwise* $\gamma_i = \delta_{q+1}$.

Remark: It has been proved in [14], that for $1 \leq j \leq 2^n$, if f is a uniform random function, then $\delta_j = 1$, and if f is a uniform random permutation, then $\delta_j \leq (1 - (j-1)/2^n)^{-j/2}$.

Proof. The proof builds upon and generalises ideas used in the security proof of the Wegman-Carter nonce-based MAC scheme given in [14].

Let $\mathcal{A}$ be an adversary attacking the authenticity of nvMAC. The result concerns information theoretic security and so we consider the adversary to be deterministic. $\mathcal{A}$ makes a number of queries to its oracles and receives the appropriate responses. The interaction of $\mathcal{A}$ with its two oracles is given by a transcript $\mathcal{T}$ which is a list of the queries made by $\mathcal{A}$ and the responses it received in return. The adversary's view of the oracles is completely determined by the transcript $\mathcal{T}$. By $\mathcal{A}(\mathcal{T})$, we will denote the interaction of $\mathcal{A}$ with the oracles as given by the transcript $\mathcal{T}$. The responses to the queries made by $\mathcal{A}$ are computed using the random function f and hence are random variables. Since $\mathcal{A}$ is deterministic, the randomness in a transcript $\mathcal{T}$ arises only from these responses. By $\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)$ we will denote the event that the adversary $\mathcal{A}$ with transcript $\mathcal{T}$ makes a verification query for tag length λ_0 which returns true. So, if the transcript $\mathcal{T}$ corresponds to the query profile $\mathfrak{C}$, then $\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda_0](\mathfrak{T}, \mathfrak{C}) = \Pr[\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)]$.

The first reduction is to assume that $q_{v,\lambda_0} = 1$. If $q_{v,\lambda_0} = 0$, i.e., $\mathcal{A}$ does not make any verification query, then clearly, $\mathcal{A}$ has advantage 0 so that the theorem is trivially proved. So, suppose that $\mathcal{A}$ with transcript $\mathcal{T}$ makes $q_{v,\lambda_0} > 1$ verification queries for tag-length λ_0. Let $\mathcal{E}$ be the event that the first verification query for the tag length λ_0 is successful and $\mathcal{S}$ be the event that one of the later verification queries for the tag length λ_0 is successful. So,

$$\begin{aligned} \mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda_0](\mathcal{A}) = \Pr[\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)] = \Pr[\mathcal{E} \vee \mathcal{S}] &= \Pr[\mathcal{E} \vee (\overline{\mathcal{E}} \wedge \mathcal{S})] \\ &= \Pr[\mathcal{E}] + \Pr[\overline{\mathcal{E}} \wedge \mathcal{S}]. \end{aligned}$$

Given the adversary $\mathcal{A}$ and the transcript $\mathcal{T}$, we define two adversaries $\mathcal{A}'$ and $\mathcal{A}''$ and correspondingly two transcripts $\mathcal{T}'$ and $\mathcal{T}''$ in the following manner.

- Adversary $\mathcal{A}'$ is the same as $\mathcal{A}$ up to and including the first verification query for tag length λ_0; the transcript $\mathcal{T}'$ is obtained from $\mathcal{T}$ by dropping from $\mathcal{T}$ all queries after the first verification query for tag length λ_0. So, $\Pr[\mathsf{succ}(\mathcal{A}'(\mathcal{T}'), \lambda_0)] = \Pr[\mathcal{E}]$.

- Adversary $\mathcal{A}''$ is the same as $\mathcal{A}$ except for the first verification query for tag length λ_0. $\mathcal{A}''$ does not issue the first verification query for tag length λ_0. The transcript $\mathcal{T}''$ is the same as that of $\mathcal{T}$ except that in $\mathcal{T}''$, the answer to the first verification query for tag length λ_0 is set to be false[1]. The event $\overline{\mathcal{E}} \wedge \mathcal{S}$ captures the following situation for $\mathcal{A}$ on the transcript $\mathcal{T}$: the response to the first verification query for tag length λ_0 is false and $\mathcal{A}$ is successful on some later verification query for tag length λ_0. Note that this situation is exactly the event that $\mathcal{A}''$ is successful for tag length λ_0 on transcript $\mathcal{T}''$. So, $\Pr[\mathsf{succ}(\mathcal{A}''(\mathcal{T}''), \lambda_0)] = \Pr[\overline{\mathcal{E}} \wedge \mathcal{S}]$.

Note that $\mathcal{A}''$ makes $q_{v,\lambda_0} - 1$ verification queries for tag length λ_0. So, the problem of proving the result for q_{v,λ_0} verification queries has been reduced to the problem of proving the result for $q_{v,\lambda_0} - 1$ verification queries. Proceeding by induction, to prove the bound given in (6.28), it is sufficient to consider an adversary which makes exactly one verification query for tag length λ_0. Let the single verification query for tag length λ_0 be $(N, m, \mathsf{tag}, \lambda_0)$.

The second reduction is to ignore all queries in $\mathcal{T}$ after the verification query for tag length λ_0. Such queries have no effect on the success probability of the verification query for tag length λ_0.

The third reduction is the following. If the queries in $\mathcal{S}_{\lambda_0}$ give rise to less than q distinct (nonce, tag-length) values, then insert additional tag generation queries to the transcript with (nonce, tag-length) values not equal to (N, λ_0) such that the queries in the augmented $\mathcal{S}_{\lambda_0}$ give rise to exactly q distinct (nonce, tag-length) values. Such augmentation of the transcript does not decrease the adversary's advantage.

In view of the above reductions, it is sufficient to consider an adversary $\mathcal{A}$ with a transcript $\mathcal{T}$ where the last query is the verification query $(N, m, \mathsf{tag}, \lambda_0)$ for tag length λ_0 and the queries in $\mathcal{S}_{\lambda_0}$ give rise to exactly q distinct (nonce, tag-length) values. The transcript $\mathcal{T}$ can contain any number of tag generation queries for the tag length λ_0. However, by the restriction that among the tag generation queries, the (nonce, tag-length) pair cannot repeat, $\mathcal{T}$ can contain at most one tag generation query of the form (N, m', λ_0). For $\lambda \neq \lambda_0$, the transcript $\mathcal{T}$ can contain multiple verification queries with the same value for the (nonce, λ) pair. So, the total number of queries in $\mathcal{S}_{\lambda_0}$ can be greater than q.

Let $\mathfrak{N} = \mathsf{bin}_8(\lambda_0 - 1)||N$, $Q = f(\mathfrak{N})$ and $\tau_0 = \tau_{\lambda_0}$. Let the q distinct values of (nonce, tag-length) pairs arising from the queries in $\mathcal{S}_{\lambda_0}$ be $(N^{(1)}, \lambda^{(1)}), \ldots, (N^{(q)}, \lambda^{(q)})$. For $i = 1, \ldots, q$, let $\mathfrak{N}^{(i)} = \mathsf{bin}_8(\lambda^{(i)} - 1)||N^{(i)}$ and $Q^{(i)} = f(\mathfrak{N}^{(i)})$. Define $\mathbf{Q} = (Q^{(1)}, \ldots, Q^{(q)})$. Let q' be the number of distinct tag-length values arising from the queries in $\mathcal{S}_{\lambda_0}$ and let $\lambda^{(1)}, \ldots, \lambda^{(q')}$ be these tag lengths. For $i = 1, \ldots, q'$, define $\tau_i = \tau_{\lambda^{(i)}}$ and $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_{q'})$. The entire randomness in the transcript arises from $\mathbf{Q}$ and $\boldsymbol{\tau}$.

Consider the final verification query $(N, m, \mathsf{tag}, \lambda_0)$ and let $\ell = \mathsf{len}(m)$. Let $\ell^{(\star)} = \mathsf{len}(m^{(\star)})$ if there is a prior tag generation query $(N^{(\star)}, m^{(\star)}, \lambda^{(\star)})$ (with response $\mathsf{tag}^{(\star)}$) such

[1]Bernstein's proof in [14] for nonce-based MAC considers simulation of the first forgery attempt with the simulator returning true if the provided tag is equal to the tag returned by a previous tag generation query on the same nonce and message, and false otherwise. In our case, since we are disallowing useless queries, there could not have been a previous tag generation query for the tag length λ_0 with the same nonce and message as that of the first verification query for tag length λ_0. So, in our case, such a simulator would always return false.

that $N^{(\star)} = N$ and $\lambda^{(\star)} = \lambda_0$; otherwise, $\ell^{(\star)}$ is undefined. Let $\gamma = 2^n \delta_q \varepsilon(\ell, \ell^{(\star)})$ if $\ell^{(\star)}$ is defined, otherwise, $\gamma = \delta_{q+1}$. To prove the theorem, it is sufficient to show

$$\Pr[\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)] \;\leq\; \gamma / 2^{\lambda_0}. \tag{6.29}$$

The verification query is successful if $\mathsf{tag} = \mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m))$. So,

$$\Pr[\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)] \;=\; \Pr[\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m)) = \mathsf{tag}]. \tag{6.30}$$

We consider the probability on the right hand side of (6.30) under two cases.

The first case is when there is no tag generation query having (nonce, tag-length) pair to be equal to (N, λ_0) in $\mathcal{T}$. In this case, $\mathfrak{N}^{(1)}, \ldots, \mathfrak{N}^{(q)}, \mathfrak{N}$ are distinct values to which f is applied. Since the adversary is adaptive, the m and tag in the final verification query are functions of the earlier responses it received and in turn are functions of $\mathbf{Q}$ and $\boldsymbol{\tau}$. We write $m \equiv m(\mathbf{Q}, \boldsymbol{\tau})$ and $\mathsf{tag} \equiv \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau})$ to denote this functional dependence. We would like to emphasise that the adversary does not have access to $\mathbf{Q}$ and $\boldsymbol{\tau}$ and writing m and tag as functions of $\mathbf{Q}$ and $\boldsymbol{\tau}$ is only to help in the argument. Let α and $\mathbf{a}$ be arbitrary values of τ_0 and $\boldsymbol{\tau}$. Let $\beta_1, \ldots, \beta_q$ be arbitrary n-bit strings and let $\mathbf{b} = (\beta_1, \ldots, \beta_q)$. So,

$$\begin{aligned}
&\Pr[\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau}))) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau})] \\
&\quad= \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau})))] \\
&\quad= \sum_{\mathbf{a}, \alpha} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau}))) \wedge (\boldsymbol{\tau} = \mathbf{a}) \wedge (\tau_0 = \alpha)] \\
&\quad= \sum_{\mathbf{a}, \alpha} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m(\mathbf{Q}, \mathbf{a}))) \wedge (\boldsymbol{\tau} = \mathbf{a}) \wedge (\tau_0 = \alpha)] \\
&\quad= \sum_{\mathbf{a}, \alpha} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m(\mathbf{Q}, \mathbf{a})))] \Pr[(\boldsymbol{\tau} = \mathbf{a}) \wedge (\tau_0 = \alpha)].
\end{aligned} \tag{6.31}$$

Let c be an arbitrary $(n-\lambda_0)$-bit binary string. We consider

$$\begin{aligned}
&\Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q},\mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m(\mathbf{Q},\mathbf{a})))] \\
&= \sum_{\mathbf{b}} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q},\mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m(\mathbf{Q},\mathbf{a}))) \wedge (\mathbf{Q}=\mathbf{b})] \\
&= \sum_{\mathbf{b}} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{b},\mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m(\mathbf{b},\mathbf{a}))) \wedge (\mathbf{Q}=\mathbf{b})] \\
&= \sum_{\mathbf{b}} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \beta \wedge (\mathbf{Q}=\mathbf{b})] \\
&\qquad\qquad (\text{where } \beta = \mathsf{tag}(\mathbf{b},\mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m(\mathbf{b},\mathbf{a})))) \\
&= \sum_{\mathbf{b}} \Pr[\mathsf{msb}_{\lambda_0}(f(\mathfrak{N})) = \beta, f(\mathfrak{N}^{(1)}) = \beta_1, \ldots, f(\mathfrak{N}^{(q)}) = \beta_q] \\
&= \sum_{\mathbf{b}} \sum_{c} \Pr[f(\mathfrak{N}) = \beta || c, f(\mathfrak{N}^{(1)}) = \beta_1, \ldots, f(\mathfrak{N}^{(q)}) = \beta_q] \\
&\leq \sum_{\mathbf{b}} 2^{n-\lambda_0}\delta_{q+1}/(2^n)^{q+1} \\
&= 2^{n-\lambda_0}\delta_{q+1}/2^n = \gamma/2^{\lambda_0}. \qquad (6.32)
\end{aligned}$$

Combining (6.31) and (6.32), we have

$$\begin{aligned}
&\Pr[\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q},\boldsymbol{\tau}))) = \mathsf{tag}(\mathbf{Q},\boldsymbol{\tau})] \\
&= \sum_{\mathbf{a},\alpha} \Pr[\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q},\mathbf{a}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m(\mathbf{Q},\mathbf{a})))] \Pr[(\boldsymbol{\tau}=\mathbf{a}) \wedge (\tau_0 = \alpha)] \\
&\leq \gamma/2^{\lambda_0} \sum_{\mathbf{a},\alpha} \Pr[(\boldsymbol{\tau}=\mathbf{a}) \wedge (\tau_0=\alpha)] = \gamma/2^{\lambda_0}. \qquad (6.33)
\end{aligned}$$

This proves the first case.
In the second case, let the transcript $\mathcal{T}$ be such that there is a tag generation query $(N^{(\star)}, m^{(\star)}, \lambda^{(\star)})$ (with response $\mathsf{tag}^{(\star)}$) where $N^{(\star)} = N$ and $\lambda^{(\star)} = \lambda_0$. Note that by the query restriction on the adversary, $m^{(\star)} \neq m$. Let $\mathfrak{N}^{(\star)} = \mathsf{bin}_8(\lambda^{(\star)}-1)||N^{(\star)}$, $Q^{(\star)} = f(\mathfrak{N}^{(\star)})$ and $\tau_\star = \tau_{\lambda^{(\star)}}$. Then $Q^{(\star)} = Q$ and $\tau_\star = \tau_0$. Let $\mathbf{Q}$ be the vector consisting of $Q^{(1)}, \ldots, Q^{(q)}$ but, not containing $Q^{(\star)}$ and let $\boldsymbol{\tau}$ be the vector consisting of $\tau_1, \ldots, \tau_{q'}$ but, not containing $\tau_\star$. So, $\mathbf{Q}$ is a vector having $q-1$ components and $\boldsymbol{\tau}$ is a vector having $q'-1$ components. In this case, $m \equiv m(\mathbf{Q},\boldsymbol{\tau},\mathsf{tag}^{(\star)})$ and $\mathsf{tag} \equiv \mathsf{tag}(\mathbf{Q},\boldsymbol{\tau},\mathsf{tag}^{(\star)})$. As in the earlier argument, we highlight that the adversary does not have access to $\mathbf{Q}$ and $\boldsymbol{\tau}$ and writing m and tag as functions of $\mathbf{Q}$ and $\boldsymbol{\tau}$ (and also $\mathsf{tag}^{(\star)}$) is to help in the argument. Due to the adaptive nature of the adversary, $m^{(\star)}$ is also a function of portions of $\mathbf{Q}$ and $\boldsymbol{\tau}$ which corresponds to the queries earlier to $(N^{(\star)}, m^{(\star)}, \lambda^{(\star)})$. Hence, we write $m^{(\star)} \equiv m^{(\star)}(\mathbf{Q},\boldsymbol{\tau})$. Note that τ_0 is independent of $\boldsymbol{\tau}$.

Let $\mathbf{a}$ and t be arbitrary values for $\boldsymbol{\tau}$ and $\mathsf{tag}^{(\star)}$ respectively. Then

$$\begin{aligned}
&\Pr[\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)}))) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)})] \\
&= \sum_{\mathbf{a},\mathsf{t}} \Pr[(\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)}))) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)})) \wedge (\boldsymbol{\tau} = \mathbf{a}) \\
&\qquad \wedge(\mathsf{tag}^{(\star)} = \mathsf{t})] \\
&= \sum_{\mathbf{a},\mathsf{t}} \Pr[(\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t}))) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t})) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}(\mathbf{Q}, \mathbf{a}))) = \mathsf{t}) \wedge (\boldsymbol{\tau} = \mathbf{a})] \\
&= \sum_{\mathbf{a}} \Big(\sum_{\mathsf{t}} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t})) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}(\mathbf{Q}, \mathbf{a}))) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{t}) \\
&\quad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t}))))] \Big) \times \Pr[\boldsymbol{\tau} = \mathbf{a}]. \qquad (6.34)
\end{aligned}$$

Let $\mathbf{b}$ and α be an arbitrary value of $\mathbf{Q}$ and τ_0. Let c_1 and c_2 be arbitrary $(n - \lambda_0)$-bit strings. We consider

$$\begin{aligned}
&\Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t})) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}(\mathbf{Q}, \mathbf{a}))) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{t}) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t}))))] \\
&= \sum_{\mathbf{b}} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t})) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}(\mathbf{Q}, \mathbf{a}))) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{t}) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{Q}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \mathbf{a}, \mathsf{t})))) \wedge (\mathbf{Q} = \mathbf{b})] \\
&= \sum_{\mathbf{b}} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{b}, \mathbf{a}, \mathsf{t})) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}(\mathbf{b}, \mathbf{a}))) = \mathsf{tag}(\mathbf{b}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{t}) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag}(\mathbf{b}, \mathbf{a}, \mathsf{t}) \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m(\mathbf{b}, \mathbf{a}, \mathsf{t})))) \wedge (\mathbf{Q} = \mathbf{b})]
\end{aligned}$$

To simplify notation, we write $m(\mathbf{b}, \mathbf{a}, \mathsf{t})$ as m, $m^{\star}(\mathbf{b}, \mathbf{a})$ as $m^{\star}$ and $\mathsf{tag}(\mathbf{b}, \mathbf{a}, \mathsf{t})$ as tag. So, we have

$$\begin{aligned}
&\sum_{\mathbf{b}} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag} \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m))) \wedge (\mathbf{Q} = \mathbf{b})] \\
&= \sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m) \oplus \mathsf{Hash}_{\alpha}(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \\
&\qquad \wedge(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag} \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m))) \wedge (\mathbf{Q} = \mathbf{b}) \wedge (\tau_0 = \alpha)] \\
&= \sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m) \oplus \mathsf{Hash}_{\alpha}(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \wedge (\tau_0 = \alpha)] \\
&\qquad \times \Pr[(\mathsf{msb}_{\lambda_0}(Q) = \mathsf{tag} \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m))) \wedge (\mathbf{Q} = \mathbf{b})] \\
&= \sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m) \oplus \mathsf{Hash}_{\alpha}(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \wedge (\tau_0 = \alpha)] \\
&\qquad \times \left(\sum_{c_1} \Pr[(Q = (\mathsf{tag} \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\alpha}(m)))||c_1) \wedge (\mathbf{Q} = \mathbf{b})] \right)
\end{aligned}$$

Let $\beta = (\mathsf{tag} \oplus \mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m)))||c_1)$. Then $\Pr[(Q = \beta) \wedge (\mathbf{Q} = \mathbf{b})]$ is bounded from above by the q-interpolation probability of f. So, we have

$$\begin{aligned}
&\sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m) \oplus \mathsf{Hash}_\alpha(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \wedge (\tau_0 = \alpha)] \\
&\qquad\qquad \times \left(\sum_{c_1} \Pr[(Q = \beta) \wedge (\mathbf{Q} = \mathbf{b})] \right) \\
&\leq \sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m) \oplus \mathsf{Hash}_\alpha(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \wedge (\tau_0 = \alpha)] \times 2^{n-\lambda_0} \frac{\delta_q}{(2^n)^q} \\
&= 2^{n-\lambda_0} \frac{\delta_q}{(2^n)^q} \times \sum_{\mathbf{b},\alpha} \Pr[(\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_\alpha(m) \oplus \mathsf{Hash}_\alpha(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}) \wedge (\tau_0 = \alpha)] \\
&= 2^{n-\lambda_0} \delta_q/(2^n)^q \times \sum_{\mathbf{b}} \Pr[\mathsf{msb}_{\lambda_0}(\mathsf{Hash}_{\tau_0}(m) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)})) = \mathsf{tag} \oplus \mathsf{t}] \\
&= 2^{n-\lambda_0} \delta_q/(2^n)^q \times \sum_{\mathbf{b}} \sum_{c_2} \Pr[\mathsf{Hash}_{\tau_0}(m) \oplus \mathsf{Hash}_{\tau_0}(m^{(\star)}) = (\mathsf{tag} \oplus \mathsf{t})||c_2] \\
&\leq 2^{n-\lambda_0} \delta_q/(2^n)^q \times \sum_{\mathbf{b}} 2^{n-\lambda_0} \varepsilon(\ell, \ell^{(\star)}) \\
&= 2^{n-\lambda_0} \delta_q/(2^n)^q \times (2^n)^{q-1} \times 2^{n-\lambda_0} \varepsilon(\ell, \ell^{(\star)}) \\
&= 2^{n-2\lambda_0} \delta_q \varepsilon(\ell, \ell^{(\star)}). \qquad (6.35)
\end{aligned}$$

Combining (6.34) and (6.35), we have,

$$\begin{aligned}
&\Pr[\mathsf{msb}_{\lambda_0}(Q \oplus \mathsf{Hash}_{\tau_0}(m(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)}))) = \mathsf{tag}(\mathbf{Q}, \boldsymbol{\tau}, \mathsf{tag}^{(\star)})] \\
&\leq \sum_{\mathbf{a}} \Big(\sum_{\mathsf{t}} 2^{n-2\lambda_0} \delta_q \varepsilon(\ell, \ell^{(\star)}) \Big) \times \Pr[\boldsymbol{\tau} = \mathbf{a}] \\
&= \sum_{\mathsf{t}} 2^{n-2\lambda_0} \delta_q \varepsilon(\ell, \ell^{(\star)}) \times \sum_{\mathbf{a}} \Pr[\boldsymbol{\tau} = \mathbf{a}] \\
&= 2^{\lambda_0} 2^{n-2\lambda_0} \delta_q \varepsilon(\ell, \ell^{(\star)}) \\
&= 2^{n-\lambda_0} \varepsilon(\ell, \ell^{(\star)}) \delta_q = \gamma/2^{\lambda_0}. \qquad (6.36)
\end{aligned}$$

This proves the second case.

□

Tightness of the security bound: The scheme nvMAC is obtained as a variant of the Wegman-Carter scheme. The statement and proof of Theorem 6 follows the bound on the Wegman-Carter scheme established by Bernstein [14]. As mentioned earlier, Bernstein's bound has been proved to be tight [79, 88]. A natural question is to consider whether the bound of Theorem 6 is also tight. We have considered this question for nvMAC. It does not seem possible to use the proof approach used in [79, 88] to show the tightness of the bound in Theorem 6. In fact, the approach does not also seem to work for the generic scheme nvMAC-Generic.

The security bound of Theorem 6 in terms of query complexity: The statement of Theorem 6 and the security bound provided in it are in terms of query profile. If it is to be translated to terms of query complexity, the following point is to be noted. The hash function $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ may be such that, the differential probability of the hash function may depend on the lengths of the particular queries. For example, if $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ is a polynomial hash, the degree of the polynomial formed from the messages and hence the corresponding differential probability is a function of the lengths of the messages. The details of this variation in the query lengths are lost when we move from the notion of query profile to the notion of query complexity. As a result, the variability in the differential probability also cannot be captured when the security is considered in terms of query complexity. In this case, a uniformity is required in the probability and to attain that, the maximum of all the differential probabilities is considered. As a result, the security bound obtained in terms of query complexity is not precise and depending on the particular queries made by the adversary, it may be an over-estimation by a large margin. Hence, in the detailed security analysis we consider the notion of query profile and the security in terms of query complexity has been mentioned in respective corollaries.

The statement of Theorem 6 and the security bound provided in it look as follows in terms of query complexity.

Corollary 1. *In the scheme* nvMAC *defined in Table 6.2, suppose that the hash function* $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ *be such that for any distinct* $m, m' \in \mathcal{M}$ *and any* $y \in \{0,1\}^n$, $\Pr[\mathsf{Hash}_\tau(m) \oplus \mathsf{Hash}_\tau(m') = y]$ *is at most* $\varepsilon \geq 1/2^n$, *i.e.* ε *is the maximum of the differential probabilities for all combination of messages.*

For $\lambda \in \mathcal{L}$, *let* $q_{g,\lambda}$ *(resp.* $q_{v,\lambda}$*) be the number of tag generation (resp. verification) queries for* λ. *Let the total number of bits in the tag generation queries be* σ_g *and that in the verification queries be* σ_v. *Fix* $\lambda_0 \in \mathcal{L}$. *Let* $\mathcal{S}_{\lambda_0}$ *be the set of all queries made by the adversary other than the verification queries for tag length* λ_0. *Suppose that the queries in* $\mathcal{S}_{\lambda_0}$ *give rise to at most* q *distinct (nonce, tag-length) values. Further, suppose* δ_q *be such that the* q*-interpolation probability of* f *is at most* $\delta_q/(2^n)^q$ *and* $(q+1)$*-interpolation probability of* f *is at most* $(\delta_q\varepsilon)/(2^n)^q$. *Then*

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{nvMAC}}[\lambda_0](\mathfrak{T}, \sigma_g, \sigma_v) \leq 2^{n-\lambda_0} q_{v,\lambda_0} \delta_q \varepsilon. \tag{6.37}$$

Essentially, in this case the bound is similar to the bound given in the security proof of the Wegman-Carter nonce-based MAC scheme given in [14]. If in some case the actual queries are such that the corresponding differential probabilities are much less than the maximum value, then this bound becomes much higher than the actual advantage of the adversary, i.e. the bound becomes more loose. Let us consider a numerical example to illustrate this scenario.

In this example, we will consider Horner's rule based hash function and the underlying field to be $\mathbb{F}_{2^n}$. The differential probability of the Horner's rule based hash for two distinct messages of length ℓ and ℓ', where $\ell \geq \ell'$, is given by $\varepsilon(\ell, \ell') = \ell/2^n$. For ease of understanding, in this example let us consider $\delta_q = 1$, which is true for a uniform random function. Let $n = 128$, $\lambda_0 = 96$, $q_{v,\lambda_0} = 1$. Let us consider an (rather artificial) upper limit of 2^{20} n-bit

blocks on the length of the message the adversary can query on. We consider some scenarios and the corresponding query profile based advantages.

- **Scenario 1:** For tag length λ_0, let the adversary make 1 tag generation query and 1 verification query, each on a message containing 512 blocks. The differential probability reflected in the bound (6.28) is $\varepsilon(512, 512) = 2^9/2^{128}$ and the corresponding bound becomes 2^{-87}.
- **Scenario 2:** For tag length λ_0, let the adversary make 1023 tag generation queries and 1 verification query, each on a message containing 1 block. Let one of the tag generation queries have the same nonce as the verification query. Then, the differential probability reflected in the bound (6.28) is $\varepsilon(1, 1) = 1/2^{128}$ and the corresponding bound becomes 2^{-96}.
- **Scenario 3:** For tag length λ_0, let the adversary make one tag generation query and one verification query on messages having 2^{20} blocks and the same nonce. Then, the differential probability reflected in the bound (6.28) is $\varepsilon(2^{20}, 2^{20}) = 2^{20}/2^{128}$ and the corresponding bound becomes 2^{-76}.

Let us now consider the query complexity based advantage for the above scenarios. Looking at the bound in (6.37), we have no clue about which value of the differential probability to be used here. The reason is, in this case, we only have the information regarding the total query complexity, but we do not know the length of each message. As a result, we are forced to use the maximum value of the differential probability which is obtained for 2^{20}-block messages resulting in the differential probability to be $2^{20}/2^{128}$. The corresponding bound given by (6.37) in all three scenarios becomes 2^{-76}. So, we see that even though the query complexities in Scenarios 1 and 2 is 1024 blocks and the query complexity in Scenario 3 is 2^{21} blocks, the query complexity based advantage in all three cases are the same. This illustrates that compared to the query complexity based advantage, the query profile based advantage provides a more granular information about the advantage.

It is to be noted that, the bound given by Bernstein [14] in the security proof of the Wegman-Carter nonce-based MAC scheme is $q_{v,\lambda_0}\delta_q\varepsilon$. This bound also lacks the information of particular message lengths. Hence, the difficulty stated above in case of complexity based advantage is applicable for this bound as well.

We have highlighted the differences between query profile based and query complexity based advantages. Also, we have provided bounds for both kinds of advantages. Depending on the requirement, one may use the appropriate kind of advantage and the corresponding bound.

6.3.1 Reducing Key Size

In a practical instantiation of nvMAC, the random function f will be instantiated by a keyed function F_K. The key for the entire scheme will consist of the key K along with the $\#\mathcal{L}$ keys $(\tau_\lambda)_{\lambda\in\mathcal{L}}$ for the hash function Hash. Depending on the size of $\mathcal{L}$, for certain applications, the size of the key may be too large. Our next constructions show how to obtain nvMAC schemes with short keys.

The hash family $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$, the nonce space $\mathcal{N}$, the message space $\mathcal{M}$, the set of allowed tag lengths $\mathcal{L}$ and the tag space remain the same as in the case of nvMAC.

Our goal is to derive the key for the hash function by applying a PRF to the concatenation of the tag length and the nonce. Thus essentially here we intend to use a fixed input-length PRF to obtain a variable input-length MAC. Depending upon the actual choice of the hash function, the key could either be an n-bit string (or, a string of some fixed length which is at least n), or, it could be a variable length string which depends upon the length of the message. Typical examples of hash function where the key is a fixed length string is the polynomial hash or the BRW hash [13, 85, 15] while typical examples of hash function where the key depends upon the length of the message is either the multi-linear hash [54], or the pseudo-dot product [111], or the UMAC [20] construction.

We consider the key of the hash function to be a sequence of n-bit blocks with the last block possibly being a partial block. Given the hash function Hash and a message m, let $\mathfrak{b}(m)$ denote the number of n-bit blocks of key material required by Hash to process the message m. As mentioned above, depending upon the choice of Hash, $\mathfrak{b}(m)$ could be independent of m (i.e., Hash uses fixed length keys), or, it could depend upon m (i.e., Hash uses a key which depends upon the length of m).

We start by constructing a nonce-based MAC scheme from a stream cipher supporting an initialisation vector. The assumption on such a stream cipher is that it is a PRF [11]. Formally, we use the PRF $\{\mathsf{SC}_K\}_{K\in\mathcal{K}}$, where SC_K is a stream cipher which maps an n-bit string under the key K to an output keystream. We will assume that the output keystream is of some fixed length which is sufficiently big for all practical applications. An appropriate length prefix of the output keystream is used in a particular context. We denote the nvMAC scheme built from SC as SC-nvMAC. The tag generation algorithm for the SC-nvMAC scheme is shown in Table 6.3. The verification algorithm $\mathsf{SC\text{-}nvMAC.Verify}_K(N, m, \mathsf{tag}, \lambda)$ works as follows: compute $\mathsf{tag}' = \mathsf{SC\text{-}nvMAC.Gen}_K(N, m, \lambda)$; return true if $\mathsf{tag} = \mathsf{tag}'$, else return false.

The key space for SC-nvMAC is $\mathcal{K}$. The key generation algorithm consists of sampling K uniformly at random from $\mathcal{K}$.

Table 6.3: A secure and efficient nvMAC scheme using a stream cipher supporting an initialisation vector.

$\mathsf{SC\text{-}nvMAC.Gen}_K(N, m, \lambda)$
  $b = \mathfrak{b}(m)$;
  $(Q, \tau) = \mathsf{msb}_{(b+1)n}(\mathsf{SC}_K(\mathsf{bin}_8(\lambda - 1)||N))$;
  $R = Q \oplus \mathsf{Hash}_\tau(m)$;
  $\mathsf{tag} = \mathsf{msb}_\lambda(R)$;
return tag.

The security of SC-nvMAC is given by the following result.

Theorem 7. *In* SC-nvMAC *defined in Table 6.3, suppose that the hash function* $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ *is* ε*-AXU, where* $\varepsilon(\ell, \ell') \geq 1/2^n$ *for all* $\ell, \ell' \leq L$.

Fix a query profile $\mathfrak{C}$. For $\lambda \in \mathcal{L}$, let $q_{g,\lambda}$ (resp. $q_{v,\lambda}$) be the number of tag generation (resp. verification) queries for λ which are in $\mathfrak{C}$. Let $q_g = \sum_{\lambda\in\mathcal{L}} q_{g,\lambda}$ and $q_v = \sum_{\lambda\in\mathcal{L}} q_{v,\lambda}$. Let σ_g (resp. σ_v) be the total number of bits in all the tag generation (resp. verification) queries in $\mathfrak{C}$.
Let λ be such that $q_{v,\lambda} \geq 1$ and for $1 \leq i \leq q_{v,\lambda}$, let $Q^{(i)}_{v,\lambda} = (N^{(i)}_{v,\lambda}, m^{(i)}_{v,\lambda}, \mathsf{tag}^{(i)}_{v,\lambda}, \lambda)$ be the i-th verification query with tag length λ. Let $\ell^{(i)}_{v,\lambda} = \mathsf{len}(m^{(i)}_{v,\lambda})$. Corresponding to $Q^{(i)}_{v,\lambda}$, there is at most one tag generation query $Q^{(i^\star)}_{g,\lambda} = (N^{(i^\star)}_{g,\lambda}, m^{(i^\star)}_{g,\lambda}, \lambda)$ such that $N^{(i)}_{v,\lambda} = N^{(i^\star)}_{g,\lambda}$. Let $\ell^{(i^\star)}_{g,\lambda} = \mathsf{len}(m^{(i^\star)}_{g,\lambda})$ if there is such a $Q^{(i^\star)}_{g,\lambda}$, otherwise $\ell^{(i^\star)}_{g,\lambda}$ is undefined.
Fix $\lambda_0 \in \mathcal{L}$. Let $\mathcal{S}_{\lambda_0}$ be the set of all queries made by the adversary other than the verification queries for tag length λ_0. Suppose that the queries in $\mathcal{S}_{\lambda_0}$ give rise to at most q distinct (nonce, tag-length) values. Then

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{SC\text{-}nvMAC}}[\lambda_0](\mathfrak{T}, \mathfrak{C}) \;\leq\; \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(\mathfrak{T}+\mathfrak{T}', q_g+q_v, n(q_g+q_v)) + \frac{1}{2^{\lambda_0}} \times \sum_{1\leq i\leq q_{v,\lambda_0}} \gamma_i \tag{6.38}$$

where $\gamma_i = 2^n \varepsilon(\ell^{(i)}_{v,\lambda_0}, \ell^{(i^\star)}_{g,\lambda_0})$ if there is a $Q^{(i^\star)}_{g,\lambda_0}$ corresponding to $Q^{(i)}_{v,\lambda_0}$ with $N^{(i)}_{v,\lambda_0} = N^{(i^\star)}_{g,\lambda_0}$; otherwise $\gamma_i = 1$. Here $\mathfrak{T}'$ is the time required to hash $q_v + q_g$ messages of total length at most $\sigma_g + \sigma_v$, plus some bookkeeping time.

Proof. The proof is similar to the proof of Theorem 6. We mention the differences.

The first reduction is to replace SC_K by a uniform random function ρ from $\{0,1\}^n$ to $\{0,1\}^L$. The advantage of the adversary in detecting this change is captured by the term $\mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(\mathfrak{T}+\mathfrak{T}', q_g+q_v, n(q_g+q_v))$ in (6.38). Let the scheme resulting from the replacement be denoted as ρ-nvMAC.

Since SC_K has been taken care of, the ensuing analysis is information theoretic. Let $\mathcal{A}$ be a deterministic and computationally unbounded adversary attacking ρ-nvMAC and having query profile $\mathfrak{C}$. It is required to upper bound $\mathsf{Adv}^{\mathrm{auth}}_{\rho\text{-}\mathsf{nvMAC}}[\lambda_0](\mathcal{A})$.

As in the proof of Theorem 6, the task reduces to analysing the probability of the event $\mathsf{succ}(\mathcal{A}(\mathcal{T}), \lambda_0)$ for a transcript $\mathcal{T}$ whose query profile is $\mathfrak{C}$.

The second reduction is to assume that $q_{v,\lambda_0} = 1$; *the third reduction* is to assume that all queries after the single verification query for tag length λ_0 are discarded. These reductions are also used in the proof of Theorem 6 and the justifications for these reductions in the present context are the same as those described in the proof of Theorem 6. As in Theorem 6, consider the set $\mathcal{S}_{\lambda_0}$ which consists of all queries made by $\mathcal{A}$ other than the verification queries for λ_0. Further, similar to the proof of Theorem 6, insert queries to the transcript $\mathcal{T}$, to ensure that the number of distinct (nonce, tag-length) pairs arising from the queries in $\mathcal{S}_{\lambda_0}$ is q.

In view of the above reductions, it is sufficient to consider an adversary $\mathcal{A}$ with a transcript $\mathcal{T}$ where the last query is the verification query $(N, m, \mathsf{tag}, \lambda_0)$ for tag length λ_0. Also, let $(N^{(1)}, \lambda^{(1)}), \ldots, (N^{(q)}, \lambda^{(q)})$ be the distinct (nonce, tag-length) pairs arising from the queries in $\mathcal{S}_{\lambda_0}$. For $1 \leq i \leq q$, define $\mathfrak{N}^{(i)} = \mathsf{bin}_8(\lambda^{(i)} - 1)||N^{(i)}$, $(Q^{(i)}, \tau_i) = \rho(\mathfrak{N}^{(i)})$ (considering the full length output of ρ), $\mathbf{Q} = (Q^{(1)}, \ldots, Q^{(q)})$ and $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_q)$. The entire randomness in the transcript arises from $\mathbf{Q}$ and $\boldsymbol{\tau}$.

At this point, we would like to mention a small difference with the proof of Theorem 6. In the scheme nvMAC, the hash key depends upon the tag length, whereas in SC-nvMAC, the hash key is determined by (nonce, tag-length) pair. As a consequence, the vector $\boldsymbol{\tau}$ defined above has q components, while the vector $\boldsymbol{\tau}$ defined in the proof of Theorem 6 has q' components, where q' is the number of distinct tag lengths arising from the queries in $\mathcal{S}_{\lambda_0}$.

Modulo this small difference, the rest of the proof is the same as the proof of Theorem 6. In particular, the proof divides into two cases. The first case is where the adversary does not make any previous tag generation query with (nonce, tag-length) pair equal to (N, λ_0) and the second case is where the adversary does make such a query. The probability calculations for these two cases are almost the same as those in the proof of Theorem 6. The only difference is that in the present case, ρ is uniform random function and so $\delta_j = 1$. Using these values of δ_j, the calculations done in the two cases of the proof of Theorem 6 show the bound stated in (6.38).

□

The following corollary provides the translation of Theorem 7 in terms of query complexity.

Corollary 2. *In* SC-nvMAC *defined in Table 6.3, suppose that the hash function* $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ *be such that for any distinct* $m, m' \in \mathcal{M}$ *and any* $y \in \{0,1\}^n$, $\Pr[\mathsf{Hash}_\tau(m) \oplus \mathsf{Hash}_\tau(m') = y]$ *is at most* $\varepsilon \geq 1/2^n$*, i.e.* ε *is the maximum of the differential probabilities for all combination of messages.*

For $\lambda \in \mathcal{L}$*, let* $q_{g,\lambda}$ *(resp.* $q_{v,\lambda}$*) be the number of tag generation (resp. verification) queries for* λ*. Let* $q_g = \sum_{\lambda\in\mathcal{L}} q_{g,\lambda}$ *and* $q_v = \sum_{\lambda\in\mathcal{L}} q_{v,\lambda}$*. Let* σ_g *(resp.* σ_v*) be the total number of bits in all the tag generation (resp. verification) queries.*

Fix $\lambda_0 \in \mathcal{L}$*. Let* $\mathcal{S}_{\lambda_0}$ *be the set of all queries made by the adversary other than the verification queries for tag length* λ_0*. Suppose that the queries in* $\mathcal{S}_{\lambda_0}$ *give rise to at most* q *distinct (nonce, tag-length) values. Then*

$$\mathsf{Adv}^{\mathrm{auth}}_{\mathsf{SC\text{-}nvMAC}}[\lambda_0](\mathfrak{T}, \sigma_g, \sigma_v) \;\leq\; \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(\mathfrak{T} + \mathfrak{T}', q_g + q_v, n(q_g + q_v)) + 2^{n-\lambda_0} q_{v,\lambda_0}\varepsilon. \tag{6.39}$$

Here $\mathfrak{T}'$ *is the time required to hash* $q_v + q_g$ *messages of total length at most* $\sigma_g + \sigma_v$*, plus some bookkeeping time.*

In the scheme SC-nvMAC, the pair (Q, τ) is derived by applying the stream cipher to $\mathsf{bin}_8(\lambda-1)||N$. Since a stream cipher produces a long enough keystream, a single application of SC is sufficient to obtain the pair (Q, τ). Suppose that we wish to use a PRF F whose output is an n-bit string (or, a short fixed length string). Clearly, then a single invocation of F will not be sufficient to obtain the pair (Q, τ). The PRF F will have to be invoked repeatedly to obtain an output bit string of desired length from which the pair (Q, τ) can be obtained.

Formally, we use a PRF family $\{F_K\}_{K\in\mathcal{K}}$, where for each $K \in \mathcal{K}$, $F_K : \{0,1\}^n \to \{0,1\}^n$. Similar to the case of SC-nvMAC, the hash family $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$, the nonce space $\mathcal{N}$, the message space $\mathcal{M}$, the set of allowed tag lengths $\mathcal{L}$ and the tag space remain the same as

in the case of nvMAC. The key space for the scheme is $\mathcal{K}$. The key generation algorithm consists of sampling K uniformly at random from $\mathcal{K}$.

The tag generation algorithm of an nvMAC scheme built from the PRF F is shown in Table 6.4 and is denoted as F-nvMAC.Gen. The verification algorithm F-nvMAC.Verify$(N, m, \mathsf{tag}, \lambda)$ works as follows. Given $(N, m, \mathsf{tag}, \lambda)$, compute $\mathsf{tag}' = F\text{-nvMAC.Gen}_K(N, m, \lambda)$; if $\mathsf{tag} = \mathsf{tag}'$, return true, else return false. In Table 6.4, F is used in a counter type mode of operation which was proposed in [109].
Instantiation of F may be done by a fixed output length PRF such as Siphash [8]. Alternatively, it can also be done using the encryption function $E_K(\cdot)$ of a block cipher. Since E is a bijection, the PRF assumption on $E_K(\cdot)$ does not hold beyond the birthday bound. While using $E_K(\cdot)$, it would have been better to perform the analysis under the assumption that $E_K(\cdot)$ is a pseudo-random permutation (PRP). This, however, is problematic. The key τ to the hash function is derived by applying $E_K(\cdot)$. Under the assumption that $E_K(\cdot)$ is a PRP, it would not be possible to assume that τ is uniformly distributed. The differential probability determining the AXU property of the hash function is computed based on uniform random τ. So, if τ cannot be considered to be uniform random, the AXU property of the hash function cannot be invoked. As a result, the proof would not go through. On the other hand, up to the birthday bound, it is reasonable to assume that the encryption function of a secure block cipher behaves like a PRF.

Table 6.4: A secure and efficient nvMAC scheme using a short output length PRF.

F-nvMAC.Gen$_K(N, m, \lambda)$
$b = \mathfrak{b}(m)$;
$\mathsf{S} = F_K(\mathsf{bin}_8(\lambda - 1)||N)$;
$(Q, \tau) = F_K(\mathsf{S} \oplus \mathsf{bin}_n(1))|| \cdots ||F_K(\mathsf{S} \oplus \mathsf{bin}_n(b + 1))$;
$R = Q \oplus \mathsf{Hash}_\tau(m)$;
$\mathsf{tag} = \mathsf{msb}_\lambda(R)$;
return tag.

The security of F-nvMAC is given by the following result.

Theorem 8. *In F-nvMAC defined in Table 6.4, suppose that the hash function $\{\mathsf{Hash}_\tau\}_{\tau \in \mathsf{T}}$ is ε-AXU, where $\varepsilon(\ell, \ell') \geq 1/2^n$ for all $\ell, \ell' \leq L$.*

Fix a query profile $\mathfrak{C}$. For $\lambda \in \mathcal{L}$, let $q_{g,\lambda}$ (resp. $q_{v,\lambda}$) be the number of tag generation (resp. verification) queries for λ which are in $\mathfrak{C}$. Let $q_g = \sum_{\lambda \in \mathcal{L}} q_{g,\lambda}$ and $q_v = \sum_{\lambda \in \mathcal{L}} q_{v,\lambda}$. Let σ_g (resp. σ_v) be the total number of bits in all the tag generation (resp. verification) queries in $\mathfrak{C}$.
Let λ be such that $q_{v,\lambda} \geq 1$ and for $1 \leq i \leq q_{v,\lambda}$, let $Q^{(i)}_{v,\lambda} = (N^{(i)}_{v,\lambda}, m^{(i)}_{v,\lambda}, \mathsf{tag}^{(i)}_{v,\lambda}, \lambda)$ be the i-th verification query with tag length λ. Let $\ell^{(i)}_{v,\lambda} = \mathsf{len}(m^{(i)}_{v,\lambda})$. Corresponding to $Q^{(i)}_{v,\lambda}$, there is at most one tag generation query $Q^{(i^\star)}_{g,\lambda} = (N^{(i^\star)}_{g,\lambda}, m^{(i^\star)}_{g,\lambda}, \lambda)$ such that $N^{(i)}_{v,\lambda} = N^{(i^\star)}_{g,\lambda}$. Let $\ell^{(i^\star)}_{g,\lambda} = \mathsf{len}(m^{(i^\star)}_{g,\lambda})$ if there is such a $Q^{(i^\star)}_{g,\lambda}$, otherwise $\ell^{(i^\star)}_{g,\lambda}$ is undefined.

Fix $\lambda_0 \in \mathcal{L}$. Let $\mathcal{S}_{\lambda_0}$ be the set of all queries made by the adversary other than the verification queries for tag length λ_0. Suppose that the queries in $\mathcal{S}_{\lambda_0}$ give rise to at most q distinct (nonce, tag-length) values. Then

$$\begin{aligned}\mathsf{Adv}^{\mathrm{auth}}_{F\text{-}\mathsf{nvMAC}}[\lambda_0](\mathfrak{T},\mathfrak{C}) \;\leq\; & \mathsf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}', B_g+B_v, n(B_g+B_v)) \\ & + \frac{(B_g+B_v)^2}{2^n} + \frac{1}{2^{\lambda_0}} \times \sum_{1\leq i\leq q_{v,\lambda_0}} \gamma_i \end{aligned} \tag{6.40}$$

where

- *$\gamma_i = 2^n\varepsilon(\ell^{(i)}_{v,\lambda_0}, \ell^{(i^\star)}_{g,\lambda_0})$ if there is a $Q^{(i^\star)}_{g,\lambda_0}$ corresponding to $Q^{(i)}_{v,\lambda_0}$ with $N^{(i)}_{v,\lambda_0} = N^{(i^\star)}_{g,\lambda_0}$; otherwise $\gamma_i = 1$;*
- *$b^{(i)}_{v,\lambda} = \mathfrak{b}(m^{(i)}_{v,\lambda})$, $B_v = \sum_{\lambda} \sum_{1\leq i\leq q_{v,\lambda}} (b^{(i)}_{v,\lambda}+2)$;*
- *$b^{(i)}_{g,\lambda} = \mathfrak{b}(m^{(i)}_{g,\lambda})$, $B_g = \sum_{\lambda} \sum_{1\leq i\leq q_{g,\lambda}} (b^{(i)}_{g,\lambda}+2)$.*

Here $\mathfrak{T}'$ is the time required to hash q_v+q_g messages of total length at most $\sigma_g+\sigma_v$, plus some bookkeeping time.

Proof. The proof is very similar to the proofs of Theorems 6 and 7. We briefly discuss the differences. There are two differences in the bound.

The first difference is in the number of queries to the PRF F in the expression $\mathsf{Adv}^{\mathrm{prf}}_F$. In the present case, if a query requires $b+1$ n-bit blocks to obtain the pair (Q,τ), the number of times F is invoked is $b+2$. The rest of the analysis proceeds by replacing F with a uniform random function ρ from $\{0,1\}^n$ to $\{0,1\}^n$.

The main argument requires that for distinct values of (N,λ), the random variables (Q,τ) are independent and uniformly distributed. The pair (Q,τ) is derived by successively applying ρ to $\mathsf{S}\oplus\mathsf{bin}_n(1),\ldots,\mathsf{S}\oplus\mathsf{bin}_n(b+1)$ where S itself is obtained by applying ρ to $\mathsf{bin}_8(\lambda-1)||N$. If for distinct values of (N,λ), the quantities $\mathsf{S}, \mathsf{S}\oplus\mathsf{bin}_n(1),\ldots,\mathsf{S}\oplus\mathsf{bin}_n(b+1)$ are distinct, then the independent and uniform random distribution of (Q,τ) is ensured.

Let the q distinct values of (nonce, tag-length) pairs arising from the queries in $\mathcal{S}_{\lambda_0}$ be $(N^{(1)},\lambda^{(1)}),\ldots,(N^{(q)},\lambda^{(q)})$. Let $\mathcal{D}^{(i)} = \{\mathsf{S}^{(i)}, \mathsf{S}^{(i)}\oplus\mathsf{bin}_n(1),\ldots,\mathsf{S}^{(i)}\oplus\mathsf{bin}_n(b^{(i)}+1)\}$ be the set of random variables in the input of ρ corresponding to $(N^{(i)},\lambda^{(i)})$. Let $\mathcal{D} = \cup_{i=1}^{q}\mathcal{D}^{(i)}$ and so $\#\mathcal{D} \leq B_g+B_v$. Let bad be the event that any two of the variables in $\mathcal{D}$ are equal. Using the fact that ρ is a uniform random function, it is standard to see that $\Pr[\mathsf{bad}] \leq (B_g+B_v)^2/2^n$.

Let $\mathcal{A}$ be an adversary attacking the scheme where F is replaced with ρ. We assume that $\mathcal{A}$ is deterministic and computationally unbounded. Let $\mathsf{succ}(\mathcal{A})$ be the event that an adversary $\mathcal{A}$ is successful. Then

$$\begin{aligned}\Pr[\mathsf{succ}(\mathcal{A})] \;&\leq\; \Pr[\mathsf{bad}] + \Pr[\mathsf{succ}(\mathcal{A})|\overline{\mathsf{bad}}] \\ &\leq\; \frac{(B_g+B_v)^2}{2^n} + \Pr[\mathsf{succ}(\mathcal{A})|\overline{\mathsf{bad}}].\end{aligned}$$

Conditioned on the event $\overline{\mathsf{bad}}$, the pairs $(Q^{(i)}, \tau^{(i)})$ are independent and uniformly distributed. From this point onwards, the rest of the proof is exactly the same as the proof of Theorem 7 and provides the same bound. We skip these details. □

The following corollary provides the translation of Theorem 8 in terms of query complexity.

Corollary 3. *In F-nvMAC defined in Table 6.4, suppose that the hash function $\{\mathsf{Hash}_\tau\}_{\tau\in\mathsf{T}}$ be such that for any distinct $m, m' \in \mathcal{M}$ and any $y \in \{0,1\}^n$, $\Pr[\mathsf{Hash}_\tau(m) \oplus \mathsf{Hash}_\tau(m') = y]$ is at most $\varepsilon \geq 1/2^n$, i.e. ε is the maximum of the differential probabilities for all combination of messages.*

For $\lambda \in \mathcal{L}$, let $q_{g,\lambda}$ (resp. $q_{v,\lambda}$) be the number of tag generation (resp. verification) queries for λ. Let $q_g = \sum_{\lambda\in\mathcal{L}} q_{g,\lambda}$ and $q_v = \sum_{\lambda\in\mathcal{L}} q_{v,\lambda}$. Let σ_g (resp. σ_v) be the total number of bits in all the tag generation (resp. verification) queries.

Let λ be such that $q_{g,\lambda}, q_{v,\lambda} \geq 1$ and for $1 \leq i \leq q_{g,\lambda}$, let $Q^{(i)}_{g,\lambda} = (N^{(i)}_{g,\lambda}, m^{(i)}_{g,\lambda}, \lambda)$ be the i-th tag generation query with tag length λ; for $1 \leq i \leq q_{v,\lambda}$, let $Q^{(i)}_{v,\lambda} = (N^{(i)}_{v,\lambda}, m^{(i)}_{v,\lambda}, \mathsf{tag}^{(i)}_{v,\lambda}, \lambda)$ be the i-th verification query with tag length λ.

Fix $\lambda_0 \in \mathcal{L}$. Let $\mathcal{S}_{\lambda_0}$ be the set of all queries made by the adversary other than the verification queries for tag length λ_0. Suppose that the queries in $\mathcal{S}_{\lambda_0}$ give rise to at most q distinct (nonce, tag-length) values. Then

$$\begin{aligned}\mathsf{Adv}^{\mathrm{auth}}_{F\text{-}\mathsf{nvMAC}}[\lambda_0](\mathfrak{T}, \sigma_g, \sigma_v) \;\leq\; & \mathsf{Adv}^{\mathrm{prf}}_F(\mathfrak{T}+\mathfrak{T}', B_g + B_v, n(B_g+B_v)) \\ & + \frac{(B_g+B_v)^2}{2^n} + 2^{n-\lambda_0} \times q_{v,\lambda_0}\varepsilon, \qquad (6.41)\end{aligned}$$

where $b^{(i)}_{v,\lambda} = \mathfrak{b}(m^{(i)}_{v,\lambda})$, $b^{(i)}_{g,\lambda} = \mathfrak{b}(m^{(i)}_{g,\lambda})$, $B_v = \sum_\lambda \sum_{1\leq i\leq q_{v,\lambda}} (b^{(i)}_{v,\lambda}+2)$ and $B_g = \sum_\lambda \sum_{1\leq i\leq q_{g,\lambda}} (b^{(i)}_{g,\lambda}+2)$. Here $\mathfrak{T}'$ is the time required to hash $q_v + q_g$ messages of total length at most $\sigma_g + \sigma_v$, plus some bookkeeping time.

6.4 Summary

In this chapter, we have considered the problem of constructing MACs supporting variable tag lengths. It has several important practical motivations as mentioned earlier, but has not gained much attention in the literature till now. We have formalised the security notion of this type of MACs. Next, we have considered the problem of obtaining secure variable tag length MAC schemes. Several variants of classical Wegman-Carter MAC schemes are considered, most of which are shown to be insecure by giving detailed attacks. One of them is proved to be secure. We further build on this scheme to obtain single-key variable tag length MAC schemes utilising either a stream cipher or a short-output pseudo-random function. These schemes can also be efficiently instantiated using practical well known primitives.

Chapter 7

FAST: Disk Encryption and Beyond

There is a huge amount of data residing on various kinds of storage devices. For example, the Indian national repository of biometric data called Aadhaar runs into several petabytes[1]. In today's world, much of the data at rest are sensitive and require encryption to be protected from unwanted access or tampering. The solution is to use full disk encryption where the storage device holds the encryption of the data under a secret key. Reading from the disk requires decrypting the relevant portion of the disk, while writing to the disk requires encrypting the data and then storing it at an appropriate location on the disk. The tasks of encryption and decryption are performed using a disk encryption algorithm. To be useful in practice a disk encryption algorithm needs to be both secure and efficient. The goal of security is to ensure that unwanted access or tampering is indeed not feasible while the goal of efficiency is to ensure that there is no noticeable slowdown in the process of reading from or writing to the disk.

A logical level view of a hard disk and most other storage devices is as a collection of sectors where each sector can store a fixed number of bytes. For example, present day hard disks have 4096-byte sectors while some of the older disks had 512-byte sectors[2]. Each sector has a unique address. A read or write operation on a disk works at the granularity of sectors. A read operation will specify a bunch of sector addresses and the complete contents of those sectors will be returned. Similarly, a write operation will specify the data and a bunch of sector addresses and the contents of the corresponding sectors will be overwritten with the new data.

A disk encryption algorithm proceeds sector by sector. The content of a sector is encrypted using the secret key and stored in-place, i.e., the content of the sector is overwritten using the encrypted content. The original unencrypted content is not stored anywhere. Using only encryption is not sufficient for security as can be seen from the following simple attack. Suppose that the contents of two successive sectors s_1 and s_2 are C_1 and C_2 corresponding to plaintexts P_1 and P_2 respectively. An adversary may simply swap C_1 and C_2. Subsequent decryption will show s_1 containing P_2 and s_2 containing P_1 whereas decryption before the swap would have shown s_1 containing P_1 and s_2 containing P_2. If it turns out that s_1 containing P_2 and s_2 containing P_1 is meaningful data, then by a simple swap operation, the adversary has been able to alter the content of the disk to a meaningful data which was not originally stored on the disk.

To prevent the above possibility, the encryption of the content of a sector needs to be somehow tied to the sector address. Decryption of any adversarially modified content of a sector should result in a random looking string which is unlikely to be meaningful data.

Viewed in this manner, a disk encryption mechanism is a tweak-based length preserving encryption which has been called a tweakable enciphering scheme (TES) [65] in the literature. In a TES, the tweak is a quantity which determines the ciphertext but, is itself not encrypted.

[1] `https://www.cse.iitb.ac.in/~comad/2010/pdf/IndustrySessions/UID_Pramod_Varma.pdf`

[2] `https://en.wikipedia.org/wiki/Disk_sector`

In case of disk encryption, the sector address works as tweak.

While disk encryption is a very important application of a TES, the full functionality of a TES is much more broader than just disk encryption. For the specific case of disk encryption, messages are contents of a sector and so are fixed length strings. A TES can have a more general message space consisting of binary strings of different lengths. Similarly, in the case of disk encryption, the tweak is a sector address and can be encoded using a short fixed length string. More generally, the tweak space in a TES can also consist of strings of different lengths or even consist of vectors of strings.

This chapter describes a new family of tweakable enciphering schemes called FAST which is built using a pseudo-random function (PRF) and a hash function with provably low collision and differential probabilities. The domain and the range of the pseudo-random function are both equal to the set of all n-bit binary strings for an appropriately chosen n. The hash function is built using arithmetic over the finite field $\mathbb{F}_{2^n}$. Some of the salient aspects of FAST are described below.

Wide range of applications: FAST can be used in the following settings.

- *Fixed length setting:* This setting is targeted towards disk encryption application. It supports an n-bit tweak and messages whose lengths are a fixed multiple of the block size n.
- *General setting:* This setting is very general. As mentioned earlier, here messages are allowed to have different lengths and tweaks are allowed to be vectors of binary strings where the numbers of components in the vectors can vary. The richness of the tweak space provides considerable flexibility in applications where there is a message and an associated set of attributes. In Chapter 1, we have already mentioned one possible application for such a functionality. Here we mention another.
  - The message consists of biometric information while the attributes are date-time, gender and other related information. A possible application would be to the Aadhaar database mentioned earlier.

We note that the idea of having associated data to be a vector of strings was earlier proposed [95] in the context of deterministic authenticated encryption. AEZ [67] provides a conceptual level description of how to handle a vector of strings as tweak using an almost XOR universal hash function to process the vector. A generic security bound is provided in terms of the collision probability of the hash function. No concrete proposal for the hash function is provided. Consequently, the efficiency of processing the tweak cannot be determined and neither it is possible to obtain a concrete security bound. In contrast, following our objective of practical implementation we put forward several concrete designs for hashing a vector of strings with associated concrete secuity bounds and detailed software implementations.

Software implementations: A major objective here consists of rigorous software implementations of FAST and the most important TES schemes in the literature. The goal of such

implementations is to perform a comparative study of the performances of FAST with those of the previous schemes in software. To this end, we have carried out detailed software implementations of the IEEE standards XCB [83] and EME2 [62] as well as AEZ (instantiated with the encryption function of the AES block cipher) along with similar implementations of variants of FAST. For a fair comparison, we have incorporated similar efficiency measures in all the implementations.

As mentioned earlier, this implementation is targeted towards modern Intel processors and is in Intel intrinsics using the specialised AES-NI instructions and the `pclmulqdq` instruction. The code for the software implementation of FAST is publicly available from `https://github.com/sebatighosh/FAST`. The implementation of FAST covers both the fixed length and the general settings. We provide timing results for the Skylake and the Kabylake processors of Intel.

Results arising from the implementations show that the new proposal compares favourably to the most important previous constructions in software. For the fixed length setting, the best speed achieved by FAST on the Intel Skylake platform is 1.24 cycles per byte (cpb). In comparison, XCB, EME2 and AEZ achieve speeds of 1.92 cpb, 2.07 cpb and 1.74 cpb respectively. The corresponding figures on Kabylake for FAST, XCB, EME2 and AEZ are 1.19, 1.85, 1.99 and 1.70 cpb respectively. Further timing details are provided later.

Dispensing with invertibility: There are several concrete TES proposals in the literature. Most of these proposals including the ones that have been standardised are modes of operations of a block cipher and use both the encryption and the decryption functions of the underlying block cipher. FAST, on the other hand, uses a PRF and does not require the invertibility property of a block cipher. The PRF itself may be instantiated using the encryption function of a block cipher such as AES. This provides two distinct advantages.

1. From a practical point of view, the advantage is that the decryption function of the block cipher does not require to be implemented. This is an advantage in hardware implementation since it results in a smaller hardware. A software implementation also benefits by requiring a smaller size code.

2. From a theoretical point of view, a block cipher is modelled as a strong pseudo-random permutation (SPRP). A PRF assumption on the encryption function of a block cipher is a weaker assumption than an SPRP assumption on the block cipher. So security of FAST can be based on a weaker assumption on the underlying block cipher.

As mentioned earlier, a previous work [100] had pointed out the possibility of using only the encryption function of a block cipher to build a TES. The work was more at a conceptual level using generic components and some unnecessary operations. It did not provide any specific instantiation or implementation. Subsequent to [100], the constructions AEZ [67] and FMix [19] proposed single key TESs using only the encryption function of the block cipher. FMix is a sequential scheme while AEZ is parallelisable. Later we discuss in more details several issues regarding the comparison of FAST to previous schemes.

Parallelisable: At a top level, the construction applies a Feistel layer of encryption on the first two message blocks and sandwiches a counter type mode of operation in-between two layers of hashing for the rest of the message. The counter mode is fully parallelisable. This leads to efficient implementations in both software and hardware.

Design of hash functions: We provide instantiations using two kinds of hash functions both of which are based on arithmetic over the finite field $\mathbb{F}_{2^n}$. The first kind of hash function is based on the usual polynomial based hashing using Horner's rule. The second kind is based on a class of polynomials [15] which was later called BRW polynomials [98]. For tackling variable length inputs, a combination of BRW and Horner based hashing called Hash2L [30] turns out to be advantageous. For the fixed length setting, we show instantiations using Horner and BRW while for the general setting, we use the vector version vecHorner of Horner and the vector version vecHash2L of Hash2L.

Provable security treatment: The security of the proposed scheme is analysed following the standard provable security methodology. The theoretical notion of security of a TES is shown to hold under the assumption that the encryption function of the underlying block cipher is a PRF. The proof requires the hash functions to satisfy certain properties. We show that the hash functions obtained from Horner, vecHorner, BRW and vecHash2L satisfy the required properties. Concrete security bounds are derived for the different instantiations. These bounds show that the security of FAST is adequate for practical purposes and is comparable to those achieved in previous designs.

Promising candidate for standardisation: As mentioned earlier, both the IEEE [3] standardisation of TESs, namely EME2 and XCB, are patented algorithms. To the best of our knowledge, till date there is no unpatented algorithm which has been standardised. Apart from offering superior performance guarantees with respect to previous schemes XCB, EME2 and AEZ, it is our hope that FAST will also fill the gap of providing an attractive solution which is unencumbered by intellectual property claims.

This chapter is based on the work [29].

7.1 Preliminaries

Throughout this chapter, we fix a positive integer $\eta \geq 3$.

Notation: Let α be a binary string. We define the following additional terminology for this chapter.

- $\mathsf{pad}_n(\alpha)$: For $n > 0$, if α is the empty string, then $\mathsf{pad}_n(\alpha)$ will denote the string 0^n; while if α is non-empty, then $\mathsf{pad}_n(\alpha)$ will denote $\alpha||0^i$, where $i \geq 0$ is the minimum integer such that n divides $\mathsf{len}(\alpha||0^i)$.

- $\mathsf{parse}_n(\alpha)$: Let α be such that $\mathsf{len}(\alpha) \geq 2n$; $\mathsf{parse}_n(\alpha)$ denotes $(\alpha_1, \alpha_2, \alpha_3)$ where $\mathsf{len}(\alpha_1) = \mathsf{len}(\alpha_2) = n$ and $\alpha = \alpha_1||\alpha_2||\alpha_3$. In other words, $\mathsf{parse}_n(\alpha)$ divides the string α into three parts with the first two parts having length n bits each with the remaining bits of α (if any) forming the third part.

- *Number of n-bit blocks:* Let the expression $\mathsf{format}_n(\mathsf{pad}_n(\alpha))$ return $\alpha_1||\cdots||\alpha_{\mathfrak{m}}$. We will say that the number of n-bit blocks in α is $\mathfrak{m}$. Note that if α is the empty string, then $\mathsf{pad}(\alpha)$ is 0^n and so $\mathsf{format}_n(\mathsf{pad}_n(\alpha))$ is also 0^n whence $\mathfrak{m} = 1$, i.e., as per our formalism, the empty string has one n-bit block. The number of n-bit blocks in α will be denoted by $\mathfrak{l}(\alpha)$.
  For a vector of binary strings $\beta = (\beta_1, \ldots, \beta_k)$, by the number of n-bit blocks in β we will mean the sum of the numbers of n-bit blocks in the strings $\beta_1, \ldots, \beta_k$. The number of n-bit blocks in β will be denoted by $\mathfrak{t}(\beta)$.

- $\mathsf{superBlks}_{n,\eta}(\alpha)$: $\mathsf{superBlks}_{n,\eta}(\alpha)$ denotes the vector of strings $(\alpha_1, \ldots, \alpha_\ell)$ obtained as $(\alpha_1, \ldots, \alpha_\ell) \leftarrow \mathsf{format}_{n\eta}(\mathsf{pad}_n(\alpha))$. For $1 \leq i \leq \ell - 1$, α_i is an $n\eta$-bit string while α_ℓ is a string whose length is at most $n\eta$ and is divisible by n. The strings $\alpha_1, \ldots, \alpha_\ell$ are called super-blocks. The first $\ell - 1$ of these super-blocks consist of exactly η n-bit blocks while the last super-block consists of at most η n-bit blocks. *We will say that the number of super-blocks in α is ℓ.*

7.2 Construction

Formally, $\mathsf{FAST} = (\mathsf{FAST.Encrypt}, \mathsf{FAST.Decrypt})$ where

$$\mathsf{FAST.Encrypt}, \mathsf{FAST.Decrypt} : \mathcal{K} \times \mathcal{T} \times \mathcal{P} \to \mathcal{P}, \tag{7.1}$$

- $\mathcal{K}$ is a finite non-empty set called the key space,
- $\mathcal{T}$ is a finite non-empty set called the tweak space and
- $\mathcal{P}$ denotes both the message and the ciphertext spaces such that for any string $P \in \mathcal{P}$, $\mathsf{len}(P) > 2n$. So, for any $P \in \mathcal{P}$, the number of n-bit blocks in $\mathsf{pad}_n(P)$ is at least three. This requirement will be called the *length condition* on $\mathcal{P}$.

We emphasise that $\mathcal{P}$ does not necessarily contain all strings of lengths greater than $2n$. We provide the precise definitions of $\mathcal{P}$ for specific instantiations later.

Let $K \in \mathcal{K}$ and $T \in \mathcal{T}$. For $P \in \mathcal{P}$, we write $\mathsf{FAST.Encrypt}_K(T, P)$ in order to denote $\mathsf{FAST.Encrypt}(K, T, P)$; and for $C \in \mathcal{P}$, we write $\mathsf{FAST.Decrypt}_K(T, C)$ in order to denote $\mathsf{FAST.Decrypt}(K, T, C)$.

The definitions of $\mathsf{FAST.Encrypt}_K(T, P)$ and $\mathsf{FAST.Decrypt}_K(T, C)$ are given in Table 7.1. These definitions use the functions $\mathbf{H}_\tau$, $\mathbf{G}_\tau$, $\mathbf{H}'_\tau$ and $\mathbf{G}'_\tau$ which themselves are defined using two hash functions h and h' in the following manner.

$$\begin{array}{rcl}
\mathbf{H}_\tau(P_1, P_2, P_3, T) & = & (P_1 \oplus h_\tau(T, P_3), P_2 \oplus \tau(P_1 \oplus h_\tau(T, P_3))); \\
\mathbf{G}_\tau(X_1, X_2, X_3, T) & = & (X_1 \oplus h_\tau(T, X_3), X_2 \oplus \tau X_1); \\
\mathbf{H}'_\tau(C_1, C_2, C_3, T) & = & (C_1 \oplus \tau(C_2 \oplus h'_\tau(T, C_3)), C_2 \oplus h'_\tau(T, C_3)); \\
\mathbf{G}'_\tau(Y_1, Y_2, Y_3, T) & = & (Y_1 \oplus \tau Y_2, Y_2 \oplus h'_\tau(T, Y_3)).
\end{array} \tag{7.2}$$

The schematic diagrams of $\mathbf{H}$ and $\mathbf{G}$ are given in Figure 7.1. The diagrams for $\mathbf{H}'$ and $\mathbf{G}'$ will be similar.

From the definitions of $\mathbf{H}_\tau, \mathbf{G}_\tau$ and $\mathbf{H}'_\tau, \mathbf{G}'_\tau$ it is easy to verify the following properties.

$$\begin{array}{lll} \mathbf{H}_\tau(P_1, P_2, P_3, T) = (A_1, A_2) & \text{implies} & \mathbf{G}_\tau(A_1, A_2, P_3, T) = (P_1, P_2); \\ \mathbf{H}'_\tau(C_1, C_2, C_3, T) = (B_1, B_2) & \text{implies} & \mathbf{G}'_\tau(B_1, B_2, C_3, T) = (C_1, C_2). \end{array} \tag{7.3}$$

Note that for fixed τ, P_3 and T, $\mathbf{H}_\tau(\cdot, \cdot, P_3, T)$ and $\mathbf{G}_\tau(\cdot, \cdot, P_3, T)$ are inverses of one another and similarly, $\mathbf{H}'_\tau(\cdot, \cdot, P_3, T)$ and $\mathbf{G}'_\tau(\cdot, \cdot, P_3, T)$ are inverses of one another.

The hash functions h and h' required in the definitions of $\mathbf{H}, \mathbf{G}$ and $\mathbf{H}', \mathbf{G}'$ respectively and the other components used in Table 7.1 are given below.

1. The two hash functions h and h' are defined in the following manner.

$$h, h' : \mathbb{F}_{2^n} \times \mathcal{T} \times \mathcal{M} \to \mathbb{F}_{2^n}, \tag{7.4}$$

   where

$$\mathcal{M} \;=\; \{\beta : \alpha || \beta \in \mathcal{P} \text{ for some } \alpha \in \{0,1\}^{2n}\}; \tag{7.5}$$

   $\mathbb{F}_{2^n}$ is the key space and also the digest space, $\mathcal{T}$ is the tweak space and $\mathcal{M}$ is the message space for the hash functions. For $\tau \in \mathbb{F}_{2^n}$, $T \in \mathcal{T}$ and $M \in \mathcal{M}$, we will write $h_\tau(T, M)$ (resp. $h'_\tau(T, M)$) to denote $h(\tau, T, M)$ (resp. $h'(\tau, T, M)$). Note that in FAST, both h and h' share the same key τ. Later we discuss the properties required of the pair of hash functions (h, h') and how to construct such pairs using standard hash functions.

2. A PRF $\{F_K\}_{K \in \mathcal{K}}$ where for $K \in \mathcal{K}$, $F_K : \{0,1\}^n \to \{0,1\}^n$. The PRF is used in the Ctr mode as given in (3.3). Since strings in $\mathcal{P}$ are of length greater than $2n$, the Ctr mode is applied to non-empty strings.

3. A fixed n-bit string fStr.

4. Sub-routines Feistel and Feistel^{-1} which are shown in Table 7.2.

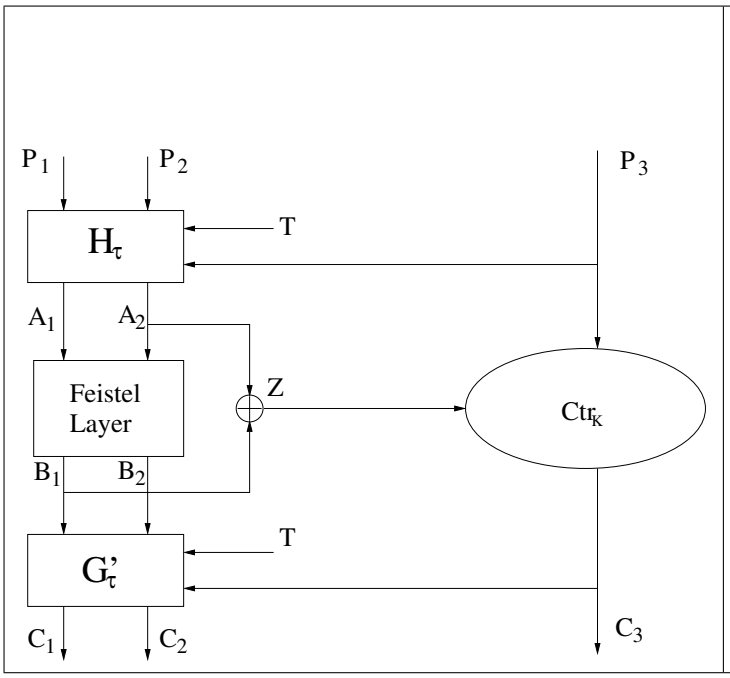
From the descriptions of $\mathsf{FAST.Encrypt}_K(T, P)$ and $\mathsf{FAST.Decrypt}_K(T, C)$ in Table 7.1, the following two facts are easy to verify. For $K \in \mathcal{K}$, $T \in \mathcal{T}$ and $P \in \mathcal{P}$,

$$\mathsf{FAST.Decrypt}_K(T, \mathsf{FAST.Encrypt}_K(T, P)) \;=\; P; \tag{7.6}$$

$$\mathsf{len}\,(\mathsf{FAST.Encrypt}_K(T, P)) \;=\; \mathsf{len}(P). \tag{7.7}$$

From (7.6), it follows that the decryption function of FAST is the inverse of the encryption function, while (7.7) shows that the length of the ciphertext produced by the encryption function is equal to the length of the plaintext.

Table 7.1: Encryption and decryption algorithms for FAST.



Algorithm $\mathsf{FAST.Encrypt}_K(T, P)$
1. $\tau \leftarrow F_K(\mathsf{fStr})$;
2. $(P_1, P_2, P_3) \leftarrow \mathsf{parse}_n(P)$;
3. $(A_1, A_2) \leftarrow \mathbf{H}_\tau(P_1, P_2, P_3, T)$;
4. $(B_1, B_2) \leftarrow \mathsf{Feistel}_K(A_1, A_2)$;
5. $Z \leftarrow A_2 \oplus B_1$;
6. $C_3 \leftarrow \mathsf{Ctr}_{K,Z}(P_3)$;
7. $(C_1, C_2) \leftarrow \mathbf{G}'_\tau(B_1, B_2, C_3, T)$;
8. return $(C_1||C_2||C_3)$.

Algorithm $\mathsf{FAST.Decrypt}_K(T, C)$
1. $\tau \leftarrow F_K(\mathsf{fStr})$;
2. $(C_1, C_2, C_3) \leftarrow \mathsf{parse}_n(C)$;
3. $(B_1, B_2) \leftarrow \mathbf{H}'_\tau(C_1, C_2, C_3, T)$;
4. $(A_1, A_2) \leftarrow \mathsf{Feistel}_K^{-1}(B_1, B_2)$;
5. $Z \leftarrow A_2 \oplus B_1$;
6. $P_3 \leftarrow \mathsf{Ctr}_{K,Z}(C_3)$;
7. $(P_1, P_2) \leftarrow \mathbf{G}_\tau(A_1, A_2, P_3, T)$;
8. return $(P_1||P_2||P_3)$.

Remarks:

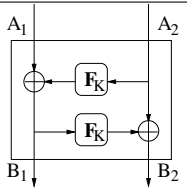
1. From the schematic diagram of the encryption algorithm in Table 7.1, $\mathsf{FAST.Encrypt}_K(T, P)$ can be seen as consisting of three distinct layers – a hashing layer using the hash function H_τ; an encryption layer consisting of the two-round Feistel and the counter mode; and the hashing layer using the hash function G'_τ. The quantity τ is the key for the hashing layers and is not used in the encryption layer while K is used only in the encryption layer and not in the hashing layers.

2. The quantity Z in $\mathsf{FAST.Encrypt}_K(T, P)$ and $\mathsf{FAST.Decrypt}_K(T, C)$, given in Table 7.1 is defined to be equal to $A_2 \oplus B_1$. In an earlier version, we had defined Z to be equal to $P_2 \oplus C_1$. The suggestion to define Z as $A_2 \oplus B_1$ is due to Mridul Nandi. This saves a few cycles in a pipelined hardware implementation when F is instantiated with AES; it has no effect on the efficiency of software implementation.

7.3 Instantiations of FAST

Certain properties are required from the pair of hash functions (h, h'). These properties will be used in the security argument to show that in an information theoretic setting, the adversary's probability of breaking the security of FAST is low. The specific properties that will be required are formalised below.

Table 7.2: A two-round Feistel construction required in Table 7.1.



$\mathsf{Feistel}_K(A_1, A_2)$	$\mathsf{Feistel}_K^{-1}(B_1, B_2)$
1. $B_1 \leftarrow A_1 \oplus F_K(A_2)$;	1. $A_2 \leftarrow B_2 \oplus F_K(B_1)$;
2. $B_2 \leftarrow A_2 \oplus F_K(B_1)$;	2. $A_1 \leftarrow B_1 \oplus F_K(A_2)$;
return (B_1, B_2).	return (A_1, A_2).

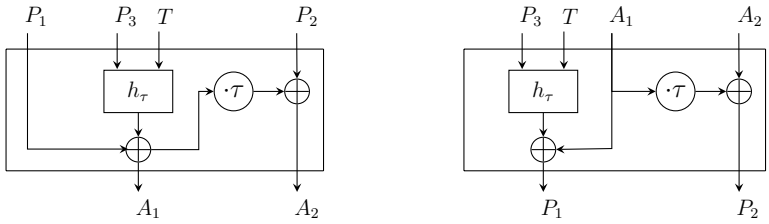


Figure 7.1: The hash functions **H** and **G**.

Definition 9. *Let (h, h') be a pair of hash functions where $h, h' : \mathbb{F}_{2^n} \times \mathcal{T} \times \mathcal{M} \to \mathbb{F}_{2^n}$ satisfy the following properties. For any $(T, M), (T', M') \in \mathcal{T} \times \mathcal{M}$, with $(T, M) \neq (T', M')$; any $\alpha, \beta \in \mathbb{F}_{2^n}$; and τ chosen uniformly at random from $\mathbb{F}_{2^n}$:*

$$\begin{aligned}
\Pr[\tau(h_\tau(T, M) \oplus \alpha) = \beta] &\leq \epsilon_1(\mathfrak{t}, \mathfrak{l}); && (7.8)\\
\Pr[\tau(h'_\tau(T, M) \oplus \alpha) = \beta] &\leq \epsilon_1(\mathfrak{t}, \mathfrak{l}); && (7.9)\\
\Pr[\tau(h_\tau(T, M) \oplus h_\tau(T', M') \oplus \alpha) = \beta] &\leq \epsilon_2(\mathfrak{t}, \mathfrak{l}, \mathfrak{t}', \mathfrak{l}'); && (7.10)\\
\Pr[\tau(h'_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) = \beta] &\leq \epsilon_2(\mathfrak{t}, \mathfrak{l}, \mathfrak{t}', \mathfrak{l}'). && (7.11)
\end{aligned}$$

For any $(T, M), (T', M') \in \mathcal{T} \times \mathcal{M}$; any $\alpha, \beta \in \mathbb{F}_{2^n}$; and τ chosen uniformly at random from $\mathbb{F}_{2^n}$:

$$\Pr[\tau(h_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) = \beta] \leq \epsilon_2(\mathfrak{t}, \mathfrak{l}, \mathfrak{t}', \mathfrak{l}'). \quad (7.12)$$

Here $\mathfrak{t} \equiv \mathfrak{t}(T)$, $\mathfrak{t}' \equiv \mathfrak{t}(T')$, $\mathfrak{l} \equiv \mathfrak{l}(M)$, $\mathfrak{l}' \equiv \mathfrak{l}(M')$; and ϵ_1 and ϵ_2 are functions of $\mathfrak{t}, \mathfrak{l}, \mathfrak{t}'$ and $\mathfrak{l}'$. Then (h, h') is said to be an (ϵ_1, ϵ_2)-eligible pair of hash functions.

Before considering the specific instantiations of FAST, we briefly discuss the hash functions used for the purpose.

7.3.1 Hash Functions

For the following two standard hash functions, the details are available in Chapter 3. Here we repeat only the definitions.

- **Polynomials:** For $\ell \geq 0$, let $\mathsf{Horner} : \mathbb{F}_{2^n} \times (\mathbb{F}_{2^n})^\ell \to \mathbb{F}_{2^n}$ be defined as follows.

$$\begin{aligned}&\mathsf{Horner}(\tau, m_1, m_2, \cdots, m_\ell)\\ &= \begin{cases} \mathbf{0}, & \text{if } \ell = 0;\\ m_1\tau^{\ell-1} \oplus m_2\tau^{\ell-2} \oplus \cdots \oplus m_{\ell-1}\tau \oplus m_\ell, & \text{if } \ell > 0. \end{cases}\end{aligned}$$

  We denote $\mathsf{Horner}(\tau, m_1, \ldots, m_\ell)$ by $\mathsf{Horner}_\tau(m_1, \ldots, m_\ell)$. $\mathsf{Horner}_\tau(m_1, \ldots, m_\ell)$, as a polynomial in τ, has degree at most $\ell - 1$.

- **BRW polynomials:** For $\ell \geq 0$, let $\mathsf{BRW} : \mathbb{F}_{2^n} \times (\mathbb{F}_{2^n})^\ell \to \mathbb{F}_{2^n}$ be defined as follows. We write $\mathsf{BRW}_\tau(\cdots)$ to denote $\mathsf{BRW}(\tau, \cdots)$.

  - $\mathsf{BRW}_\tau() = \mathbf{0}$;
  - $\mathsf{BRW}_\tau(m_1) = m_1$;
  - $\mathsf{BRW}_\tau(m_1, m_2) = m_1\tau \oplus m_2$;
  - $\mathsf{BRW}_\tau(m_1, m_2, m_3) = (\tau \oplus m_1)(\tau^2 \oplus m_2) \oplus m_3$;
  - $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\ell)$
    $= (\tau^k \oplus m_k) \times \mathsf{BRW}_\tau(m_1, \cdots, m_{k-1}) \oplus \mathsf{BRW}_\tau(m_{k+1}, \cdots, m_\ell)$;
    if $k \in \{4, 8, 16, 32, \cdots\}$ and $k \leq \ell < 2k$.

  From the definition it follows that for $\ell \geq 3$, $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\ell)$ is a monic polynomial and for $\ell = 0, 1, 2$, $\mathsf{BRW}_\tau(m_1, \ldots, m_\ell) = \mathsf{Horner}_\tau(m_1, \ldots, m_\ell)$.

7.3.1.1 Hash function vecHorner

Let

$$\mathcal{VD} = \bigcup_{k=0}^{255} \{(M_1, \ldots, M_k) : M_i \in \{0,1\}^*,\ 0 \leq \mathsf{len}(M_i) \leq 2^{n-16} - 1\}. \tag{7.13}$$

The upper bound of 255 on k ensures that the value of k fits in a byte and the upper bound of $2^{n-16}-1$ on the lengths of strings ensures that the lengths of such strings fit into an $(n-16)$-bit binary string. The definition of $\mathsf{vecHorner} : \mathbb{F}_{2^n} \times \mathcal{VD} \to \mathbb{F}_{2^n}$ is shown in Table 7.3 where we write $\mathsf{vecHorner}_\tau(\cdot)$ to denote $\mathsf{vecHorner}(\tau, \cdot)$. From a top level abstraction, vecHorner can be seen as a two-step process: the first step is to perform a one-one encoding of the input vector of strings into a single string; and the second layer consists of applying the Horner's rule to the encoded string. The degree of $\mathsf{vecHorner}_\tau(M_1, \ldots, M_k)$ is at most $k + \sum_{i=1}^{k} \mathfrak{m}_i$ and its constant term is $\mathbf{0}$. Here $\mathfrak{m}_i = \mathfrak{l}(M_i)$, $i = 1, \ldots, k$.

The following result shows that vecHorner is an AXU family.

Proposition 7. *Let $k \geq k' \geq 0$; $\mathbf{M} = (M_1, \ldots, M_k)$ and $\mathbf{M}' = (M'_1, \ldots, M'_{k'})$ be two distinct vectors in $\mathcal{VD}$ and $\alpha \in \mathbb{F}_{2^n}$. For a uniform random $\tau \in \mathbb{F}_{2^n}$,*

$$\Pr_\tau\left[\mathsf{vecHorner}_\tau(\mathbf{M}) \oplus \mathsf{vecHorner}_\tau(\mathbf{M}') = \alpha\right] \leq \frac{\max\left(k + \sum_{i=1}^{k} \mathfrak{m}_i, k' + \sum_{j=1}^{k'} \mathfrak{m}'_j\right)}{2^n} \tag{7.14}$$

where $\mathfrak{m}_i$ (resp. $\mathfrak{m}'_j$) is the number of n-bit blocks in $\mathsf{pad}_n(M_i)$ (resp. $\mathsf{pad}_n(M'_j)$).

Table 7.3: Computations of vecHorner and vecHash2L. The string 1^n denotes the element of $\mathbb{F}_{2^n}$ whose binary representation consists of the all-one string. Here η is a positive integer ≥ 3 and $d(\eta)$ denote the degree of $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$, where $m_1, \ldots, m_\eta \in \mathbb{F}_{2^n}$.

$\mathsf{vecHorner}_\tau(M_1, \ldots, M_k)$	$\mathsf{vecHash2L}_\tau(M_1, \ldots, M_k)$
if $k = 0$ return $1^n\tau$;	if $k = 0$ return $1^n\tau$;
$\mathsf{digest} \leftarrow \mathbf{0}$;	$\mathsf{digest} \leftarrow \mathbf{0}$;
for $i \leftarrow 1, \ldots, k-1$ do	for $i \leftarrow 1, \ldots, k-1$ do
$(M_{i,1}, \ldots, M_{i,\mathfrak{m}_i}) \leftarrow \mathsf{format}_n(\mathsf{pad}_n(M_i))$;	$(M_{i,1}, \ldots, M_{i,\ell_i}) \leftarrow \mathsf{superBlks}_{n,\eta}(M_i)$;
$L_i \leftarrow \mathsf{bin}_n(\mathsf{len}(M_i))$;	$L_i \leftarrow \mathsf{bin}_n(\mathsf{len}(M_i))$;
for $j \leftarrow 1, \ldots, \mathfrak{m}_i$ do	for $j \leftarrow 1, \ldots, \ell_i$ do
$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus M_{i,j}$;	$\mathsf{digest} \leftarrow \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{i,j})$;
end for;	end for;
$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus L_i$;	$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus L_i$;
end for;	end for;
$(M_{k,1}, \ldots, M_{k,\mathfrak{m}_k}) \leftarrow \mathsf{format}_n(\mathsf{pad}_n(M_k))$;	$(M_{k,1}, \ldots, M_{k,\ell_k}) \leftarrow \mathsf{superBlks}_{n,\eta}(M_k)$;
$L_k \leftarrow \mathsf{bin}_8(k)\|\|0^8\|\|\mathsf{bin}_{n-16}(\mathsf{len}(M_k))$;	$L_k \leftarrow \mathsf{bin}_8(k)\|\|0^8\|\|\mathsf{bin}_{n-16}(\mathsf{len}(M_k))$;
for $j \leftarrow 1, \ldots, \mathfrak{m}_k$ do	for $j \leftarrow 1, \ldots, \ell_k$ do
$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus M_{k,j}$;	$\mathsf{digest} \leftarrow \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{k,j})$;
end for;	end for;
$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus L_k$;	$\mathsf{digest} \leftarrow \tau\mathsf{digest} \oplus L_k$;
$\mathsf{digest} \leftarrow \tau\mathsf{digest}$;	$\mathsf{digest} \leftarrow \tau\mathsf{digest}$;
return digest.	return digest.

Proof. Let $p(\tau) = \mathsf{vecHorner}_\tau(\mathbf{M}) \oplus \mathsf{vecHorner}_\tau(\mathbf{M}') \oplus \alpha$. If $p(\tau)$ is a non-zero polynomial, then the degree of $p(\tau)$ is at most $\max\left(k + \sum_{i=1}^{k} \mathfrak{m}_i, k' + \sum_{j=1}^{k'} \mathfrak{m}'_j\right)$. The probability that a uniform random τ is a root of $p(\tau)$ is at most the stated bound. So, it is sufficient to argue that $p(\tau)$ is non-zero.

If $k' = 0$, then, as $\mathbf{M} \neq \mathbf{M}'$, $k > 0$. In this case, $\mathsf{vecHorner}_\tau(\mathbf{M}') = 1^n\tau$ and the coefficient of τ in $\mathsf{vecHorner}_\tau(\mathbf{M})$ is $L_k \neq 1^n$. Hence, in this case $p(\tau)$ is a non-zero polynomial.

Let M_{i_1,i_2} (resp. M'_{j_1,j_2}) be the n-bit blocks obtained from $\mathbf{M}$ (resp. $\mathbf{M}'$) using format. If $k > k' > 0$, then the coefficient of τ in $p(\tau)$ is $L_k \oplus L'_{k'} \neq \mathbf{0}$ and so $p(\tau)$ is a non-zero polynomial. So, suppose $k = k'$. If there is an i such that $L_i \neq L'_i$, let i be the maximum such index. Using the maximality of i it is possible to argue that $L_i \oplus L'_i$ occurs as a coefficient of some power of τ in $p(\tau)$ and again it follows that $p(\tau)$ is a non-zero polynomial. So, now suppose that $L_i = L'_i$ for all $1 \leq i \leq k = k'$. Since $\mathbf{M} \neq \mathbf{M}'$, there must be an i and j such that $M_{i,j} \neq M'_{i,j}$ again showing that $p(\tau)$ is a non-zero polynomial. □

7.3.1.2 Hash function vecHash2L [30]

Two universal hash functions, namely Hash2L and vecHash2L, have been defined earlier [30]. Here we only recall the definition of vecHash2L since we will not be using the hash function Hash2L in this chapter. The definition of $\mathsf{vecHash2L} : \mathbb{F}_{2^n} \times \mathcal{VD} \to \mathbb{F}_{2^n}$ is given in Table 7.3 where we write $\mathsf{vecHash2L}_\tau(\cdot)$ to denote $\mathsf{vecHash2L}(\tau, \cdot)$. For better understanding of the function one may refer to [30]. The degree of $\mathsf{vecHash2L}_\tau(M_1, \ldots, M_k)$ is at most $(d(\eta) + 1)(\ell_1 + \cdots + \ell_k) + k$, and its constant term is 0. The values of $\ell_1, \ldots, \ell_k$ are defined by the algorithm given in Table 7.3. Theorem 2 of [30] shows that vecHash2L is an AXU family. More precisely, the following is proved. Let $k \geq k' \geq 0$; $\mathbf{M} = (M_1, \ldots, M_k)$ and $\mathbf{M}' = (M'_1, \ldots, M'_{k'})$ be two distinct vectors in $\mathcal{VD}$. For a uniform random $\tau \in \mathbb{F}_{2^n}$ and for any $\alpha \in \mathbb{F}_{2^n}$,

$$\Pr_\tau\left[\mathsf{vecHash2L}_\tau(\mathbf{M}) \oplus \mathsf{vecHash2L}_\tau(\mathbf{M}') = \alpha\right] \leq \frac{\max\left(k + (d(\eta)+1)\Lambda, k' + (d(\eta)+1)\Lambda'\right)}{2^n} \tag{7.15}$$

where $\Lambda = \sum_{i=1}^{k} \ell_i$ and $\Lambda' = \sum_{j=1}^{k'} \ell'_j$; ℓ_i (resp. ℓ'_j) is the number of super-blocks in M_i (resp. M'_j).

Note that the hash function vecHash2L is parameterised by the value of η. *In the rest of the chapter, we will assume that $\eta + 1$ is a power of two so that the degree $d(\eta)$ of $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ is η.*

7.3.2 Specific Instantiations

For specific instantiations of FAST, we consider the following two scenarios.

Fixed length setting $\mathsf{Fx}_\mathfrak{m}$ for some positive integer $\mathfrak{m} > 2$: For this setting, in (7.1), we define

$$\mathcal{T} = \{0,1\}^n, \quad \mathcal{P} = \{0,1\}^{\mathfrak{m}n} \text{ and so } \mathcal{M} = \{0,1\}^{n(\mathfrak{m}-2)}. \tag{7.16}$$

In other words, a tweak T is an n-bit string while plaintexts and ciphertexts consist of $\mathfrak{m}$ n-bit strings. This particular setting is suited for disk encryption application, where for a fixed n, the number of blocks $\mathfrak{m}$ in a message is determined by the size of a disk sector. In this case, for $M \in \mathcal{M}$, $\mathfrak{l}(M) = \mathfrak{m} - 2$ and for $T \in \mathcal{T}$, $\mathfrak{t}(T) = 1$. By the length condition on $\mathcal{P}$, we must have $\mathfrak{m} \geq 3$.

Consider the encryption and decryption algorithms of FAST. The number of n-bit blocks in P (resp. C) is $\mathfrak{m}$ and so the number of n-bit blocks in P_3 (resp. C_3) is $\mathfrak{m} - 2$. The hash functions h and h' are invoked in FAST as $h(T, P_3)$ and $h'(T, C_3)$. So, in Definition 9, $\mathcal{T} = \mathbb{F}_{2^n}$ and $\mathcal{M} = (\mathbb{F}_{2^n})^{\mathfrak{m}-2}$ and we have

$$h, h' : \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \times (\mathbb{F}_{2^n})^{\mathfrak{m}-2} \to \mathbb{F}_{2^n}. \tag{7.17}$$

For the setting of $\mathsf{Fx}_{\mathfrak{m}}$, we describe two instantiations of h and h', one with Horner and the other with BRW. The corresponding instantiations of FAST will be denoted as FAST[$\mathsf{Fx}_{\mathfrak{m}}$, Horner] and FAST[$\mathsf{Fx}_{\mathfrak{m}}$, BRW].

General setting Gn: Let $\mathfrak{k}$ be a fixed integer in the range $\{0, \ldots, 254\}$. For this setting, in (7.1), we define

$$\mathcal{T} = \bigcup_{k=0}^{\mathfrak{k}} \{(T_1, \ldots, T_k) : 0 \leq \mathsf{len}(T_i) \leq 2^{n-16} - 1\}; \tag{7.18}$$

$$\mathcal{P} = \bigcup_{i>2n}^{2^{n-16}-1} \{0,1\}^i; \text{ and so} \tag{7.19}$$

$$\mathcal{M} = \bigcup_{i>0}^{2^{n-16}-2n-1} \{0,1\}^i. \tag{7.20}$$

A tweak T is a vector $T = (T_1, \ldots, T_k)$ where $0 \leq k \leq \mathfrak{k}$ and each T_i is a binary string. Since $\mathfrak{k} \leq 254$, $\mathfrak{k} + 1 \leq 255$ and so the binary representation of $\mathfrak{k} + 1$ will fit in a byte.

For $P \in \mathcal{P}$, suppose $M \in \mathcal{M}$ is such that $P = X||M$ for some binary string X of length $2n$. Then $\mathfrak{l}(M) = \mathfrak{m} - 2$, where $\mathfrak{m}$ is the number of blocks in $\mathsf{pad}_n(P)$. For a tweak $T = (T_1, \ldots, T_k)$, $\mathfrak{t}(T) = \sum_{i=1}^{k} \mathfrak{m}_i$, where $\mathfrak{m}_i$ is the number of blocks in $\mathsf{pad}_n(T_i)$.

The parameter $\mathfrak{k}$ controls the maximum number of components that can appear in a tweak. This does not imply that the number of components in all the tweaks is equal to $\mathfrak{k}$. Rather, the number of components in a tweak is between 0 and $\mathfrak{k}$. So, the above definition of the tweak space models tweaks as vectors having variable number of components. Since we put an upper bound of 254 on $\mathfrak{k}$, one possibility is to do away with the parameter $\mathfrak{k}$ and replace it with the value 254. The reason we do not do this is the following. The parameter $\mathfrak{k}$ enters the security bound. If we replace $\mathfrak{k}$ by 254, then this value would enter the security bound. If in practice, the actual value of $\mathfrak{k}$ is much less than 254 (as it is likely to be), then using 254 instead of $\mathfrak{k}$ will lead to a looser security bound than what it should actually be. It is to avoid this unnecessary looseness in the security bound that we introduce and work with the parameter $\mathfrak{k}$.

For the setting of Gn, we describe two instantiations of h and h'. One of these is based on vecHorner while the other is based on vecHash2L. The parameter $\mathfrak{k}$ is required in both cases while the parameter η is required only in the case of vecHash2L. The instantiations of FAST in the general setting with vecHorner and vecHash2L will be denoted as $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ and $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \eta, \mathsf{vecHash2L}]$ respectively.

In the general setting, the lengths of the plaintexts can vary. Also, the tweak space has a rich structure which provides considerable flexibility in applications. Examples of such applications have been mentioned in the introduction. On the downside, the specific instantiations of the general setting are somewhat slower than the corresponding instantiations for the fixed length setting. So, for targeted applications such as disk encryption, it would be preferable to use the fixed length setting leaving out some of the extra overheads incurred in the general setting.

7.3.2.1 Hash functions h and h' for $\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}}, \mathsf{Horner}]$

Fix a positive integer $\mathfrak{m} \geq 3$ so that the length condition on $\mathcal{P}$ is satisfied. The hash functions h, h' are defined using Horner as follows:

$$h_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2}) = \tau\mathsf{Horner}_\tau(\mathbf{1}, X_1, \ldots, X_{\mathfrak{m}-2}, T); \quad (7.21)$$

$$h'_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2}) = \tau^2\mathsf{Horner}_\tau(\mathbf{1}, X_1, \ldots, X_{\mathfrak{m}-2}, T). \quad (7.22)$$

Note that $\mathsf{Horner}_\tau(\mathbf{1}, X_1, \ldots, X_{\mathfrak{m}-2}, T)$ is a monic polynomial in τ of degree $\mathfrak{m}-1$. Consequently, h and h' are monic polynomials in τ of degrees $\mathfrak{m}$ and $\mathfrak{m}+1$ respectively whose constant terms are zero.

Proposition 8. *Let $\mathfrak{m} \geq 3$ be an integer. The pair (h, h') of hash functions defined in (7.21) and (7.22) for the construction* $\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}}, \mathsf{Horner}]$ *is an (ϵ_1, ϵ_2)-eligible pair, where $\epsilon_1 = \epsilon_2 = (\mathfrak{m}+2)/2^n$.*

Proof. In this case, for $M \in \mathcal{M}$, $\mathfrak{l}(M) = \mathfrak{m}-2$ and for $T \in \mathcal{T}$, $\mathfrak{t}(T) = 1$. We write $\mathfrak{l}$ and $\mathfrak{t}$ instead of $\mathfrak{l}(M)$ and $\mathfrak{t}(T)$.

The polynomials $\tau(h_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2})\oplus\alpha)\oplus\beta$ and $\tau(h'_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2})\oplus\alpha)\oplus\beta$ are monic polynomials of degrees $\mathfrak{l}+\mathfrak{t}+2 = \mathfrak{m}+1$ and $\mathfrak{l}+\mathfrak{t}+3 = \mathfrak{m}+2$ in τ respectively. So, the probability that a uniform random τ in $\mathbb{F}_{2^n}$ is a root of $\tau(h_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2})\oplus\alpha)\oplus\beta$ (resp. $\tau(h'_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2})\oplus\alpha)\oplus\beta$) is $(\mathfrak{l}+\mathfrak{t}+2)/2^n = (\mathfrak{m}+1)/2^n$ (resp. $(\mathfrak{l}+\mathfrak{t}+3)/2^n = (\mathfrak{m}+2)/2^n$). This shows the value of ϵ_1.

Let $X = X_1||\cdots||X_{\mathfrak{m}-2}$ and $X' = X'_1||\cdots||X'_{\mathfrak{m}-2}$ and T, T' be such that $(T, X) \neq (T', X')$. Then $h_\tau(T, X)\oplus h_\tau(T', X')$ is a non-zero polynomial of degree at most $\mathfrak{l}+\mathfrak{t} = \mathfrak{m}-1$ whose constant term is zero. This is because the leading terms of $h_\tau(T, X)$ and $h_\tau(T', X')$ will cancel out in the sum $h_\tau(T, X)\oplus h_\tau(T', X')$ so that its degree will be at most $\mathfrak{m}-1$; $(T, X) \neq (T', X')$ ensures that $h_\tau(T, X)\oplus h_\tau(T', X')$ is a non-zero polynomial; and the constant terms of both $h_\tau(T, X)$ and $h_\tau(T', X')$ are zero. As a result, $\tau(h_\tau(T, X)\oplus h_\tau(T', X')\oplus\alpha)\oplus\beta$ is a non-zero polynomial in τ of degree at most $\mathfrak{m}$. So, the probability that a uniform random τ is a root of this polynomial is at most $(\mathfrak{l}+\mathfrak{t}+1)/2^n = \mathfrak{m}/2^n$. A similar reasoning shows that the probability that a uniform random τ is a root of $\tau(h'_\tau(T, X)\oplus h'_\tau(T', X')\oplus\alpha)\oplus\beta$ is at most $(\mathfrak{l}+\mathfrak{t}+2)/2^n = (\mathfrak{m}+1)/2^n$.

For any (T, X) and (T', X'), the polynomial $h_\tau(T, X) \oplus h'_\tau(T', X')$ is a monic polynomial of degree $\mathfrak{l} + \mathfrak{t} + 2 = \mathfrak{m} + 1$ whose constant term is zero. Consequently, the polynomial $\tau(h_\tau(T, X) \oplus h'_\tau(T', X') \oplus \alpha) \oplus \beta$ is a monic polynomial of degree $\mathfrak{m}+2$ and so the probability that a uniform random τ is a root of this polynomial is $(\mathfrak{l} + \mathfrak{t} + 3)/2^n = (\mathfrak{m} + 2)/2^n$. This shows the value of ϵ_2. □

7.3.2.2 Hash functions h and h' for FAST[$\mathsf{Fx}_\mathfrak{m}$, BRW]

Fix an integer $\mathfrak{m} \geq 4$. From the length condition on $\mathcal{P}$, we only need $\mathfrak{m} \geq 3$ and the condition $\mathfrak{m} \geq 4$ is a special requirement for FAST[$\mathsf{Fx}_\mathfrak{m}$, BRW] as we explain below. In this case, the hash functions h, h' are defined using BRW as follows:

$$h_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2}) = \tau\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T); \quad (7.23)$$
$$h'_\tau(T, X_1||\cdots||X_{\mathfrak{m}-2}) = \tau^2\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T). \quad (7.24)$$

Note that from the definition of BRW polynomials, for $\mathfrak{m} = 3$, the polynomial $\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T)$ is not necessarily monic, while for $\mathfrak{m} \geq 4$, the polynomial $\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T)$ is necessarily monic. It is to ensure the monic property that we enforce the condition $\mathfrak{m} \geq 4$ for FAST[$\mathsf{Fx}_\mathfrak{m}$, BRW]. An alternative would have been to prepend $\mathbf{1}$ as in the case of FAST[$\mathsf{Fx}_\mathfrak{m}$, Horner]. This though would create complications which do not seem to be necessary for the fixed length setting. Instead, we use this technique later in the context of the general setting.

Recall that the degree of $\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T)$ is denoted as $d(\mathfrak{m} - 1)$. So, h and h' are also monic polynomials of degrees $1 + d(\mathfrak{m} - 1)$ and $2 + d(\mathfrak{m} - 1)$ respectively whose constant terms are zero.

Proposition 9. *Let $\mathfrak{m} \geq 4$ be an integer. The pair (h, h') of hash functions defined in (7.23) and (7.24) for the construction* FAST[$\mathsf{Fx}_\mathfrak{m}$, BRW] *is an (ϵ_1, ϵ_2)-eligible pair, where $\epsilon_1 = \epsilon_2 = (3+d(\mathfrak{m}-1))/2^n$. Further, if $\mathfrak{m}$ is a power of two, then (h, h') is an $((\mathfrak{m}+2)/2^n, (\mathfrak{m}+2)/2^n)$-eligible pair.*

Proof. The proof is analogous to the proof of Proposition 8. It is required to use the expression $d(\mathfrak{m}-1)$ for the degree of $\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T)$ and further the injectivity of the map $(X_1, \ldots, X_{\mathfrak{m}-2}, T) \longmapsto \mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T)$ ensures that for $(T, X) \neq (T', X')$, the polynomial $\mathsf{BRW}_\tau(X_1, \ldots, X_{\mathfrak{m}-2}, T) \oplus \mathsf{BRW}_\tau(X'_1, \ldots, X'_{\mathfrak{m}-2}, T')$ is not zero.

The last statement follows from the previously mentioned fact that $d(\mathfrak{m} - 1) = \mathfrak{m} - 1$ if and only if $\mathfrak{m} \geq 4$ is a power of two. □

Remark: In the case where $\mathfrak{m}$ is a power of two, (h, h') is an $((\mathfrak{m}+2)/2^n, (\mathfrak{m}+2)/2^n)$-eligible pair for both FAST[$\mathsf{Fx}_\mathfrak{m}$, Horner] and FAST[$\mathsf{Fx}_\mathfrak{m}$, BRW].

7.3.2.3 Hash functions h and h' for FAST[Gn, $\mathfrak{k}$, vecHorner]

In this setting, we define $h, h' : \mathbb{F}_{2^n} \times \mathcal{T} \times \mathcal{M} \to \mathbb{F}_{2^n}$ where $\mathcal{T}$ and $\mathcal{M}$ are given by (7.18) and (7.20) respectively. For $T = (T_1, \ldots, T_k) \in \mathcal{T}$ and $M \in \mathcal{M}$, let $\mathfrak{d} = \mathfrak{t}(T) + \mathfrak{l}(M) + k + 2$.

We define

$$h_\tau(T, M) = \tau^{\mathfrak{d}} \oplus \mathsf{vecHorner}_\tau(T_1, \ldots, T_k, M); \quad (7.25)$$
$$h'_\tau(T, M) = \tau(\tau^{\mathfrak{d}} \oplus \mathsf{vecHorner}_\tau(T_1, \ldots, T_k, M)). \quad (7.26)$$

It is easy to see that h and h' are monic polynomials of degrees $\mathfrak{d}$ and $\mathfrak{d}+1$ respectively whose constant terms are zero.

The computation of $\tau^{\mathfrak{d}} \oplus \mathsf{vecHorner}_\tau(T_1, \ldots, T_k, M)$ can be done by the following simple modification of the algorithm for computing vecHorner shown in Table 7.3. *The initialisation of* digest *using* $\mathsf{digest} = \mathbf{0}$ *is to be replaced with* $\mathsf{digest} = \mathbf{1}$.

Proposition 10. *The hash functions h and h' defined in (7.25) and (7.26) respectively for the construction* $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ *form an (ϵ_1, ϵ_2)-eligible pair, where*

$$\epsilon_1 = \frac{\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 4}{2^n}; \qquad \epsilon_2 = \frac{\max(\mathfrak{t} + \mathfrak{l}, \mathfrak{t}' + \mathfrak{l}') + \mathfrak{k} + 4}{2^n}.$$

Note: In this case, both ϵ_1 and ϵ_2 depend linearly on the total number of n-bit blocks in the tweak ($\mathfrak{t}$), the number of n-bit blocks in the message ($\mathfrak{l}$) and the upper bound on the number of components in the tweak ($\mathfrak{k}$).

Proof. For $T = (T_1, \ldots, T_k)$, recall that $\mathfrak{t} = \mathfrak{t}(T) = \sum_{i=1}^{k} \mathfrak{m}_i$ where $\mathfrak{m}_i$ is the number of n-bit blocks in $\mathsf{pad}_n(T_i)$. Also, $\mathfrak{l} = \mathfrak{l}(M)$ is the number of n-bit blocks in $\mathsf{pad}_n(M)$.

The degree of $h_\tau(T, M)$ is $\mathfrak{d} = \mathfrak{t} + \mathfrak{l} + k + 2 \leq \mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 2$ and the degree of $h'_\tau(T, M)$ is $\mathfrak{d} + 1 = \mathfrak{t} + \mathfrak{l} + k + 3 \leq \mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 3$. So, the polynomial $\tau(h_\tau(T, M) \oplus \alpha) \oplus \beta$ is a monic polynomial of degree at most $\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 3$ and the polynomial $\tau(h'_\tau(T, M) \oplus \alpha) \oplus \beta$ is a monic polynomial of degree at most $\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 4$. This shows the value of ϵ_1.

Consider $(T', M') \neq (T, M)$ where $T' = (T'_1, \ldots, T'_{k'})$, $\mathfrak{t}' = \mathfrak{t}(T')$ and $\mathfrak{l}' = \mathfrak{l}(M')$. Without loss of generality assume that $k \geq k'$. Let $p(\tau) = \tau(h_\tau(T, M) \oplus h_\tau(T', M') \oplus \alpha) \oplus \beta$. If the degrees of $h_\tau(T, M)$ and $h_\tau(T', M')$ are not equal, then $p(\tau)$ is a polynomial of degree $\max(\mathfrak{t} + \mathfrak{l} + k + 3, \mathfrak{t}' + \mathfrak{l}' + k' + 3) \leq \max(\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 3, \mathfrak{t}' + \mathfrak{l}' + \mathfrak{k} + 3)$. So, suppose that the degrees of $h_\tau(T, M)$ and $h_\tau(T', M')$ are equal. The leading monic terms of the two polynomials cancel out. If $p(\tau)$ is a non-zero polynomial, then it has maximum degree $\max(\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 2, \mathfrak{t}' + \mathfrak{l}' + \mathfrak{k} + 2)$. So, it is sufficient to show that $p(\tau)$ is a non-zero polynomial. This argument is similar to that of Proposition 7. Further, a similar argument applies for h' where the degree of $\tau(h'_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) \oplus \beta$ is at most $\max(\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 4, \mathfrak{t}' + \mathfrak{l}' + \mathfrak{k} + 4)$.

Now consider (T, M) and (T', M') which are not necessarily distinct and let $p(\tau) = \tau(h_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) \oplus \beta$. The coefficient of τ in $h_\tau(T, M)$ is $L = \mathsf{bin}_8(k + 1) || 0^8 || \mathsf{bin}_{n-16}(\mathsf{len}(M)) \neq \mathbf{0}$ which is the coefficient of τ^2 in τh_τ. The coefficient of τ^2 in $\tau h'_\tau(T', M')$ is $\mathbf{0}$ and so the coefficient of τ^2 in $p(\tau)$ is $L \neq \mathbf{0}$. So, $p(\tau)$ is a non-zero polynomial. The degree of $p(\tau)$ is at most $\max(\mathfrak{t} + \mathfrak{l} + \mathfrak{k} + 3, \mathfrak{t}' + \mathfrak{l}' + \mathfrak{k} + 4)$.

This completes the proof. □

7.3.2.4 Hash functions h and h' for FAST[Gn, $\mathfrak{k}, \eta$, vecHash2L]

In this setting, we define $h, h' : \mathbb{F}_{2^n} \times \mathcal{T} \times \mathcal{M} \to \mathbb{F}_{2^n}$ where $\mathcal{T}$ and $\mathcal{M}$ are given by (7.18) and (7.20) respectively. For $T = (T_1, \ldots, T_k) \in \mathcal{T}$ and $M \in \mathcal{M}$, let the number of super-blocks in $\mathsf{pad}_n(T_i)$ be ℓ_i and the number of super-blocks in $\mathsf{pad}_n(M)$ be ℓ. Let $\mathfrak{d} = (d(\eta) + 1)(\ell_1 + \cdots + \ell_k + \ell) + k + 2$. We define

$$\begin{aligned} h_\tau(T, M) &= \tau^{\mathfrak{d}} \oplus \mathsf{vecHash2L}_\tau(T_1, \ldots, T_k, M); & (7.27)\\ h'_\tau(T, M) &= \tau(\tau^{\mathfrak{d}} \oplus \mathsf{vecHash2L}_\tau(T_1, \ldots, T_k, M)). & (7.28) \end{aligned}$$

The definition of vecHash2L requires choosing the value η. As mentioned earlier, we will assume that η is chosen so that $\eta + 1$ is a power of two and so $d(\eta) = \eta$. The computation of $\tau^{\mathfrak{d}} \oplus \mathsf{vecHash2L}_\tau(T_1, \ldots, T_k, M)$ can be done by the following simple modification of the algorithm for computing vecHash2L shown in Table 7.3. *The initialisation of* digest *using* $\mathsf{digest} = \mathbf{0}$ *is to be replaced with* $\mathsf{digest} = \mathbf{1}$.

Proposition 11. *Let the parameter $\eta \geq 3$ required in the definition of* vecHash2L *be such that $\eta+1$ is a power of two. The hash functions h and h' defined in (7.27) and (7.28) respectively for the construction* FAST[Gn, $\mathfrak{k}, \eta$, vecHash2L] *form an (ϵ_1, ϵ_2)-eligible pair, where*

$$\begin{aligned} \epsilon_1 &= \frac{((\eta+1)/\eta)(\mathfrak{t} + \mathfrak{l}) + (\mathfrak{k}+1)(\eta+2) + 3}{2^n};\\ \epsilon_2 &= \frac{((\eta+1)/\eta)\max(\mathfrak{t} + \mathfrak{l}, \mathfrak{t}' + \mathfrak{l}') + (\mathfrak{k}+1)(\eta+2) + 3}{2^n}. \end{aligned}$$

Note: In this case, once the parameter η is fixed, both ϵ_1 and ϵ_2 depend linearly on the total number of n-bit blocks in the tweak ($\mathfrak{t}$), the number of n-bit blocks in the message ($\mathfrak{l}$) and the upper bound on the number of components in the tweak ($\mathfrak{k}$).

Proof. Since $\eta + 1 \geq 4$ is a power of two, $d(\eta) = \eta$.

For $i = 1, \ldots, k$, let the number of super-blocks in $\mathsf{pad}_n(T_i)$ be ℓ_i and the number of super-blocks in $\mathsf{pad}_n(M)$ be ℓ. For $i = 1, \ldots, k$, let the number of n-bit blocks in $\mathsf{pad}_n(T_i)$ be $\mathfrak{m}_i$, so that $\mathfrak{t} = \mathfrak{t}(T) = \sum_{i=1}^{k} \mathfrak{m}_i$ and the number of n-bit blocks in $\mathsf{pad}_n(M)$ be $\mathfrak{l}$. In $\mathsf{pad}_n(T_i)$, each of the first $\ell_i - 1$ super-blocks contains exactly η n-bit blocks and the last super-block contains at most η blocks. Since the total number of n-bit blocks in $\mathsf{pad}_n(T_i)$ is $\mathfrak{m}_i$, we have $\mathfrak{m}_i > \eta(\ell_i - 1)$ from which we obtain $(\eta+1)\ell_i < \mathfrak{m}_i((\eta+1)/\eta) + \eta + 1$. Similarly, $(\eta+1)\ell < \mathfrak{l}((\eta+1)/\eta) + \eta + 1$. We have

$$\begin{aligned} \mathfrak{d} &= (d(\eta)+1)(\ell_1 + \cdots + \ell_k + \ell) + k + 2\\ &\leq (d(\eta)+1)(\ell_1 + \cdots + \ell_k + \ell) + \mathfrak{k} + 2\\ &= (\eta+1)(\ell_1 + \cdots + \ell_k + \ell) + \mathfrak{k} + 2 \quad (\text{since } d(\eta) = \eta)\\ &< ((\eta+1)/\eta)(\mathfrak{t} + \mathfrak{l}) + (k+1)(\eta+1) + \mathfrak{k} + 2\\ &\leq ((\eta+1)/\eta)(\mathfrak{t} + \mathfrak{l}) + (\mathfrak{k}+1)(\eta+2) + 1. \end{aligned} \tag{7.29}$$

So, the degree of $\tau h_\tau(T, M)$ is $\mathfrak{d} + 1$ which is at most $((\eta+1)/\eta)(\mathfrak{t} + \mathfrak{l}) + (\mathfrak{k}+1)(\eta+2) + 2$ and the degree of $\tau h'_\tau(T, M)$ is $\mathfrak{d} + 2$ which is at most $((\eta+1)/\eta)(\mathfrak{t} + \mathfrak{l}) + (\mathfrak{k}+1)(\eta+2) + 3$. This shows the value of ϵ_1.

Consider $(T', M') \neq (T, M)$ where $T' = (T'_1, \ldots, T'_{k'})$, $\mathfrak{t}' = \mathfrak{t}(T')$ and $\mathfrak{l}' = \mathfrak{l}(M')$. Let $\mathfrak{d}'$ be the degree of $h_\tau(T', M')$. For any $\alpha, \beta \in \mathbb{F}_{2^n}$, we wish to bound the probability (over uniform random choice of τ in $\mathbb{F}_{2^n}$) that the polynomial $p_1(\tau) = \tau(h_\tau(T, M) \oplus h_\tau(T', M') \oplus \alpha) \oplus \beta$ is zero. If $\mathfrak{d} \neq \mathfrak{d}'$, then $p_1(\tau)$ is a monic polynomial of degree $\max(\mathfrak{d}+1, \mathfrak{d}'+1)$ and so the probability that it is zero is at most $\max(\mathfrak{d}+1, \mathfrak{d}'+1)/2^n$. If $\mathfrak{d} = \mathfrak{d}'$, then $p_1(\tau)$ is zero if and only if the polynomial

$$p_2(\tau) = \tau(\mathsf{vecHash2L}_\tau(T_1, \ldots, T_k, M) \oplus \mathsf{vecHash2L}_\tau(T'_1, \ldots, T'_{k'}, M') \oplus \alpha) \oplus \beta$$

is zero. Using Theorem 2 of [30] (see (7.15)), we have this probability to be at most $\max(\mathfrak{d}, \mathfrak{d}')/2^n$. So, the probability that $p_1(\tau)$ is zero is at most $\max(\mathfrak{d}+1, \mathfrak{d}'+1)/2^n$. Similarly, the probability that $\tau(h'_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) \oplus \beta$ is zero is at most $\max(\mathfrak{d}+2, \mathfrak{d}'+2)/2^n$.

Consider (T, M) and (T', M') which are not necessarily distinct. Fix $\alpha, \beta \in \mathbb{F}_{2^n}$ and consider $p(\tau) = \tau(h_\tau(T, M) \oplus h'_\tau(T', M') \oplus \alpha) \oplus \beta$. The coefficient of τ in

$$h_\tau(T, M) = \tau^{\mathfrak{d}} \oplus \mathsf{vecHash2L}_\tau(T_1, \ldots, T_k, M)$$

is

$$L = \mathsf{bin}_8(k+1)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M)) \neq \mathbf{0},$$

which is the coefficient of τ^2 in $\tau h_\tau(T, M)$. The coefficient of τ^2 in $\tau h'_\tau(T', M')$ is $\mathbf{0}$ and so the coefficient of τ^2 in $p(\tau)$ is $L \neq \mathbf{0}$. So, $p(\tau)$ is a non-zero polynomial. The degree of $p(\tau)$ is at most $\max(\mathfrak{d}+1, \mathfrak{d}'+2) \leq \max(\mathfrak{d}+2, \mathfrak{d}'+2)$. This shows the value of ϵ_2. □

7.4 Security

In this section, we provide the formal definitions and the formal security statement for FAST.

7.4.1 Pseudo-Random Function

The pseudo-random function (PRF) and its security notion have already been defined in Chapter 3. Here we define only the query complexity of the adversary when it queries an oracle corresponding to the PRF. Suppose the adversary $\mathcal{A}$ makes a total of q queries which are $X^{(1)}, \ldots, X^{(q)}$. The query complexity of $\mathcal{A}$ is the sum total of the number of n-bit blocks in $\mathsf{pad}_n(X^{(s)})$, $s = 1, \ldots, q$.

7.4.2 Tweakable Enciphering Scheme

The tweakable enciphering scheme has already been defined in Chapter 3. The notion of security for a TES that we consider is that of indistinguishability from uniform random strings. This is the standard security model used for a TES and implies other notions of security. For details, we refer to [65].

Let $\mathfrak{F}$ be the set of all functions f from $\mathcal{T} \times \mathcal{P}$ to $\mathcal{P}$ such that for any $T \in \mathcal{T}$ and $P \in \mathcal{P}$, $\mathsf{len}(f(T, P)) = \mathsf{len}(P)$. Let ρ_1 and ρ_2 be two functions chosen independently and uniformly at random from $\mathfrak{F}$.

An adversary $\mathcal{A}$ attacking a TES has access to two oracles which we will call the left and the right oracles. Both the oracles are functions from $\mathfrak{F}$. An input to the left oracle is of the form (T, P) and the response is C, while an input to the right oracle is of the form (T, C) and the response is P. The adversary $\mathcal{A}$ adaptively queries its oracles possibly interweaving its queries to its left and right oracles. At the end, $\mathcal{A}$ outputs a bit.

We assume that the adversary does not make any pointless query. This means that $\mathcal{A}$ does not repeat a query to any of its oracles; does not query the right oracle with (T, C) if it received C in response to a query (T, P) made to its left oracle; and does not query the left oracle with (T, P) if it received P in response to a query (T, C) made to its right oracle. The advantage of $\mathcal{A}$ in breaking TES is defined as follows:

$$\begin{aligned}&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathcal{A})\\ &= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{TES.Encrypt}_K(\cdot,\cdot),\mathsf{TES.Decrypt}_K(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1\right].\end{aligned} \tag{7.30}$$

Suppose the adversary $\mathcal{A}$ makes q queries $(T^{(1)}, X^{(1)}), \ldots, (T^{(q)}, X^{(q)})$ where $T^{(s)} = (T^{(s)}_1, \ldots, T^{(s)}_{k^{(s)}})$ and $X^{(s)}$ is either $P^{(s)}$ or $C^{(s)}$. For $j = 1, \ldots, k^{(s)}$, let $\mathfrak{m}^{(s)}_j$ be the number of n-bit blocks in $\mathsf{pad}_n(T^{(s)}_j)$ and let $\mathfrak{m}^{(t)}$ be the number of n-bit blocks in $\mathsf{pad}_n(X^{(t)})$. We will write $\mathfrak{t}^{(s)}$ to denote $\mathfrak{t}(T^{(s)})$. Let $X^{(s)} = X^{(s)}_1 || X^{(s)}_2 || X^{(s)}_3$ with $\mathsf{len}(X^{(s)}_1) = \mathsf{len}(X^{(s)}_2) = n$ and $\mathsf{len}(X^{(s)}_3) \geq 1$. Then $X^{(s)}_3 \in \mathcal{M}$ and $\mathfrak{l}(X^{(s)}_3) = \mathfrak{m}^{(s)} - 2$. We will write $\mathfrak{n}^{(s)}$ to denote $\mathfrak{l}(X^{(s)}_3)$.

The tweak query complexity θ, the message query complexity ω and the total query complexity σ are defined as follows.

$$\theta = \sum_{s=1}^{q} \mathfrak{t}(T^{(s)}) = \sum_{s=1}^{q} \mathfrak{t}^{(s)}; \tag{7.31}$$

$$\omega = \sum_{t=1}^{q} \mathfrak{m}^{(t)}; \tag{7.32}$$

$$\sigma = \theta + \omega. \tag{7.33}$$

By $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathfrak{T}, q, \theta, \omega)$ we denote the maximum of $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathcal{A})$ over all adversaries $\mathcal{A}$ which run in time $\mathfrak{T}$, make q queries and have tweak query complexity θ and message query complexity ω. TES is said to be $(\mathfrak{T}, q, \theta, \omega, \varepsilon)$-secure if $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathcal{A}) \leq \varepsilon$ for all $\mathcal{A}$ running in time $\mathfrak{T}$, making a total of q queries with tweak query complexity θ and message query complexity ω.

7.4.3 Security of FAST

The security proof for FAST is the following.

Theorem 10. *Let n be a positive integer; $\mathbb{F}_{2^n}$ is represented using some fixed irreducible polynomial of degree n over $GF(2)$; $\{F_K\}_{K \in \mathcal{K}}$ where for $K \in \mathcal{K}$, $F_K : \{0,1\}^n \to \{0,1\}^n$;*

(h, h') is an (ϵ_1, ϵ_2)-eligible pair of hash functions, where $h, h' : \mathbb{F}_{2^n} \times \mathcal{T} \times \mathcal{M} \to \mathbb{F}_{2^n}$; and fStr *is a fixed n-bit string used to build the TES*

$$\mathsf{FAST} = (\mathsf{FAST.Encrypt}, \mathsf{FAST.Decrypt})$$

given in Table 7.1. Fix q, $\omega \geq q$ to be positive integers and θ to be a non-negative integer. For all $\mathfrak{T} > 0$,

$$\begin{aligned} \mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}}(\mathfrak{T}, q, \theta, \omega) &\leq \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T} + \mathfrak{T}', \omega + 1, \omega + 1) + \Delta(\mathsf{FAST}), \quad \textit{where} \\ \Delta(\mathsf{FAST}) &= 2\omega \left(\sum_{s=1}^{q} \epsilon_1^{(s)} \right) + 3 \sum_{1 \leq s < t \leq q} \epsilon_2^{(s,t)} + \sum_{s=1}^{q} \epsilon_2^{(s,s)} + \frac{3\omega^2}{2^n}; \end{aligned} \tag{7.34}$$

$\mathfrak{T}'$ is the time required to answer q queries with tweak query complexity θ and message query complexity ω plus some bookkeeping time; and for $1 \leq s, t \leq q$, $\epsilon_1^{(s)} = \epsilon_1(\mathfrak{t}^{(s)}, \mathfrak{n}^{(s)})$, $\epsilon_2^{(s,t)} = \epsilon_2(\mathfrak{t}^{(s)}, \mathfrak{n}^{(s)}, \mathfrak{t}^{(t)}, \mathfrak{n}^{(t)})$.

Proof. Let $\mathcal{A}$ be an adversary attacking FAST. We use $\mathcal{A}$ to build an adversary $\mathcal{B}$ attacking the PRF-property of F. $\mathcal{B}$ has access to an oracle which is either $F_K(\cdot)$ for a uniform random K in $\mathcal{K}$, or, the oracle is ρ which is a uniform random function from $\{0,1\}^n$ to $\{0,1\}^n$. Adversary $\mathcal{B}$ uses the (ϵ_1, ϵ_2)-eligible pair of hash functions (h, h') to set up an instance of FAST and invokes $\mathcal{A}$ to attack this instance. $\mathcal{A}$ makes a number of oracle queries to the encryption and decryption oracles of FAST. $\mathcal{B}$ uses its own oracle and the hash functions h and h' to compute the responses which it provides to $\mathcal{A}$. At the end, $\mathcal{A}$ outputs a bit and $\mathcal{B}$ outputs the same bit. Note that both encryption and decryption queries by $\mathcal{A}$ can be answered using the oracle of $\mathcal{B}$ and the hash functions h, h'.

The running time of $\mathcal{B}$ is the running time of $\mathcal{A}$ along with the time required to compute the responses to the queries made by $\mathcal{A}$ using $\mathcal{B}$'s oracle plus some bookkeeping time which includes the time for set-up. So, the total running time of $\mathcal{B}$ is $\mathfrak{T} + \mathfrak{T}'$ as desired. Further, to answer $\mathcal{A}$'s queries, $\mathcal{B}$ needs to make a query to its oracle on fStr and to answer the s-th query, it needs to make $\mathfrak{m}^{(s)}$ queries to its oracle. So, the total number of times $\mathcal{B}$ queries its oracle is $1 + \sum_{s=1}^{q} \mathfrak{m}^{(s)} = \omega + 1$. Since each query of $\mathcal{B}$ consists of a single n-bit block, the query complexity is also $\omega + 1$.

If the oracle to $\mathcal{B}$ is the real oracle, i.e., the oracle is F_K, then $\mathcal{A}$ gets to interact with the real encryption and decryption oracles of FAST. So,

$$\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{B}^{F_K(\cdot)} \Rightarrow 1\right] = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{FAST.Encrypt}_K(\cdot,\cdot),\mathsf{FAST.Decrypt}_K(\cdot,\cdot)} \Rightarrow 1\right]. \tag{7.35}$$

Denote by FAST_ρ the instance of FAST where F_K is replaced by ρ. If the oracle to $\mathcal{B}$ is the random oracle, i.e., the oracle is ρ, then

$$\Pr\left[\mathcal{B}^{\rho(\cdot)} \Rightarrow 1\right] = \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right]. \tag{7.36}$$

So,

$$\begin{aligned} \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) &= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{B}^{F_K(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{B}^{\rho(\cdot)} \Rightarrow 1\right] \\ &= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{FAST.Encrypt}_K(\cdot,\cdot),\mathsf{FAST.Decrypt}_K(\cdot,\cdot)} \Rightarrow 1\right] \\ &\quad - \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right]. \end{aligned} \tag{7.37}$$

The advantages of $\mathcal{A}$ and $\mathcal{B}$ are related as follows. Recall that $\mathfrak{F}$ is the set of all functions f from $\mathcal{T} \times \mathcal{P}$ to $\mathcal{P}$ such that for any $T \in \mathcal{T}$ and $P \in \mathcal{P}$, $\mathsf{len}(f(T,P)) = \mathsf{len}(P)$. Let $\rho_1(\cdot,\cdot)$ and $\rho_2(\cdot,\cdot)$ be two independent and uniform random functions from $\mathfrak{F}$.

$$\begin{aligned}
&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}}(\mathcal{A}) \\
&= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{FAST.Encrypt}_K(\cdot,\cdot),\mathsf{FAST.Decrypt}_K(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1\right] \\
&= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{FAST.Encrypt}_K(\cdot,\cdot),\mathsf{FAST.Decrypt}_K(\cdot,\cdot)} \Rightarrow 1\right] \\
&\quad - \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right] \\
&\quad + \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1\right] \\
&= \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right] \\
&\quad - \Pr\left[\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1\right]. \qquad (7.38)
\end{aligned}$$

There are two events to consider, namely,

$$\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1 \text{ and } \mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1.$$

Consider the event $\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1$. Suppose $\mathcal{A}$ makes a total of q queries with tweak query complexity θ and message query complexity ω. For $1 \le s \le q$, let $\mathsf{ty}^{(s)} = \mathsf{enc}$ if the s-th query is an encryption query and $\mathsf{ty}^{(s)} = \mathsf{dec}$ if the s-th query is a decryption query. Denote the tweak, the plaintext and the ciphertext associated with the s-th query by $T^{(s)}$, $P^{(s)} = P_1^{(s)}||P_2^{(s)}||P_3^{(s)}$ and $C^{(s)} = C_1^{(s)}||C_2^{(s)}||C_3^{(s)}$ respectively. We have $\mathfrak{t}^{(s)} = \mathfrak{t}(T^{(s)})$ and $\mathfrak{m}^{(s)}$ is the number of n-bit blocks in $\mathsf{pad}_n(P^{(s)})$ and $\mathsf{pad}_n(C^{(s)})$. Also, $\mathfrak{n}^{(s)} = \mathfrak{l}(P_3^{(s)}) = \mathfrak{l}(C_3^{(s)}) = \mathfrak{m}^{(s)} - 2$.

The interaction of $\mathcal{A}$ with the oracle in this setting is given by the game G_{real} which is shown in Table 7.4. In this game, the random function ρ is built incrementally. Whenever a "new" input to ρ is received, the output is chosen independently and uniformly at random. The variable bad is set to true if it turns out that two inputs to ρ collide. Let $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$ be the event that bad is set to true in the game G_{real}. Also, by $\mathcal{A}^{G_{\mathsf{real}}} \Rightarrow 1$ we denote the event that $\mathcal{A}$ outputs 1 in the game G_{real}. Note that $\mathcal{A}^{G_{\mathsf{real}}} \Rightarrow 1$ is exactly the event $\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1$.

If $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$ does not occur, then the boxed instruction in game G_{real} is not executed. The absence of the boxed instruction does not affect the probability of $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$. We consider the distributions of the plaintexts and ciphertexts when $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$ does not occur. Let $Y_1^{(s)}$ denote the output of $\mathsf{Ch}\text{-}\rho(A_2^{(s)})$ and $Y_2^{(s)}$ denote the output of $\mathsf{Ch}\text{-}\rho(B_1^{(s)})$. Suppose $\mathsf{ty}^{(s)} = \mathsf{enc}$, i.e., the s-th query is an encryption query. Then from game G_{real}, we can write

$$\begin{aligned}
C_1^{(s)} &= Y_1^{(s)} \oplus P_1^{(s)} \oplus \tau C_2^{(s)} \oplus \tau h'_\tau(T^{(s)}, C_3^{(s)}) \oplus h_\tau(T^{(s)}, P_3^{(s)}); \\
C_2^{(s)} &= Y_2^{(s)} \oplus P_2^{(s)} \oplus \tau P_1^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)}) \oplus \tau h_\tau(T^{(s)}, P_3^{(s)}); \\
C_{3,i}^{(s)} &= S_i^{(s)} \oplus P_{3,i}^{(s)} \text{ for } i = 1, \ldots, \mathfrak{m}^{(s)} - 3; \\
C_{3,\mathfrak{m}^{(s)}-2}^{(s)} &= \mathsf{first}_{r^{(s)}}(D^{(s)}) \oplus P_{3,\mathfrak{m}^{(s)}-2}^{(s)};
\end{aligned}$$

Table 7.4: Game G_{real}.

Subroutine $\mathsf{Ch}\text{-}\rho(X)$
$Y \xleftarrow{\$} \{0,1\}^n$;
if $X \in \mathcal{D}$ then $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{Y \leftarrow \rho(X);}$ endif;
$\rho(X) \leftarrow Y$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$; **return**$(Y)$;
Initialization:
$\tau \xleftarrow{\$} \{0,1\}^n$; $\mathcal{D} \leftarrow \{\mathsf{fStr}\}$; $\mathsf{bad} \leftarrow \mathsf{false}$.

$\mathsf{ty}^{(s)} = \mathsf{enc}$: input $(T^{(s)}, P^{(s)})$	$\mathsf{ty}^{(s)} = \mathsf{dec}$: input $(T^{(s)}, C^{(s)})$
$(P_1^{(s)}, P_2^{(s)}, P_3^{(s)}) \leftarrow \mathsf{parse}_n(P^{(s)})$;	$(C_1^{(s)}, C_2^{(s)}, C_3^{(s)}) \leftarrow \mathsf{parse}_n(C^{(s)})$;
$A_1^{(s)} \leftarrow P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})$;	$B_2^{(s)} \leftarrow C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$;
$A_2^{(s)} \leftarrow \tau A_1^{(s)} \oplus P_2^{(s)}$;	$B_1^{(s)} \leftarrow C_1^{(s)} \oplus \tau B_2^{(s)}$;
$B_1^{(s)} \leftarrow A_1^{(s)} \oplus \mathsf{Ch}\text{-}\rho(A_2^{(s)})$;	$A_2^{(s)} \leftarrow B_2^{(s)} \oplus \mathsf{Ch}\text{-}\rho(B_1^{(s)})$;
$B_2^{(s)} \leftarrow A_2^{(s)} \oplus \mathsf{Ch}\text{-}\rho(B_1^{(s)})$;	$A_1^{(s)} \leftarrow B_1^{(s)} \oplus \mathsf{Ch}\text{-}\rho(A_2^{(s)})$;
$C_1^{(s)} \leftarrow \tau B_2^{(s)} \oplus B_1^{(s)}$;	$P_2^{(s)} \leftarrow \tau A_1^{(s)} \oplus A_2^{(s)}$;
$Z^{(s)} \leftarrow A_2^{(s)} \oplus B_1^{(s)}$;	$Z^{(s)} \leftarrow A_2^{(s)} \oplus B_1^{(s)}$;
for $i = 1$ to $\mathfrak{m}^{(s)} - 3$ do	for $i = 1$ to $\mathfrak{m}^{(s)} - 3$ do
$J_i^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(i)$;	$J_i^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(i)$;
$S_i^{(s)} \leftarrow \mathsf{Ch}\text{-}\rho(J_i^{(s)})$;	$S_i^{(s)} \leftarrow \mathsf{Ch}\text{-}\rho(J_i^{(s)})$;
$C_{3,i}^{(s)} \leftarrow P_{3,i}^{(s)} \oplus S_i^{(s)}$;	$P_{3,i}^{(s)} \leftarrow C_{3,i}^{(s)} \oplus S_i^{(s)}$;
end for;	end for;
$J_{\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(\mathfrak{m}^{(s)} - 2)$;	$J_{\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(\mathfrak{m}^{(s)} - 2)$;
$D^{(s)} \leftarrow \mathsf{Ch}\text{-}\rho(J_{\mathfrak{m}^{(s)}-2}^{(s)})$;	$E^{(s)} \leftarrow \mathsf{Ch}\text{-}\rho(J_{\mathfrak{m}^{(s)}-2}^{(s)})$;
$C_{3,\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow P_{3,\mathfrak{m}^{(s)}-2}^{(s)} \oplus \mathsf{first}_{r^{(s)}}(D^{(s)})$;	$P_{3,\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow C_{3,\mathfrak{m}^{(s)}-2}^{(s)} \oplus \mathsf{first}_{r^{(s)}}(E^{(s)})$;
$C_2^{(s)} \leftarrow B_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$;	$P_1^{(s)} \leftarrow A_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})$;
return $(C_1^{(s)} \| C_2^{(s)} \| C_3^{(s)})$.	return $(P_1^{(s)} \| P_2^{(s)} \| P_3^{(s)})$.

When $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$ does not occur, $Y_1^{(s)}, Y_2^{(s)}, S_i^{(s)}, (i = 1, \ldots, \mathfrak{m}^{(s)}-3), D^{(s)}$ are independent and uniform random strings. From the above relations, it is easy to argue that the ciphertext $C^{(s)}$ is also independent and uniform random. A similar argument shows that when $\mathsf{ty}^{(s)} = \mathsf{dec}$, i.e., the query is a decryption query, then $P^{(s)}$ is an independent and uniform random string. So, if $\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})$ does not occur, then the adversary obtains independent and uniform random strings as responses to all its queries.

In the next step, the game G_{real} is modified to the game G_{int}. This game is shown in Table 7.5. In this game, the outputs of ρ are not chosen directly. Instead, these are defined from the plaintexts and the ciphertexts. For a enciphering query, the ciphertext is chosen independently and uniformly at random while for a deciphering query, the plaintext is chosen independently and uniformly at random. The outputs of ρ are defined from these

Table 7.5: Game G_{int}.

Subroutine $\mathsf{ChkDom}(X)$
if $X \in \mathcal{D}$ then $\mathsf{bad} \leftarrow \mathsf{true}$; endif;
$\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$;
Initialization:
$\tau \xleftarrow{\$} \{0,1\}^n$; $\mathcal{D} \leftarrow \{\mathsf{fStr}\}$; $\mathsf{bad} \leftarrow \mathsf{false}$.

$\mathsf{ty}^{(s)} = \mathsf{enc}$: input $(T^{(s)}, P^{(s)})$	$\mathsf{ty}^{(s)} = \mathsf{dec}$: input $(T^{(s)}, C^{(s)})$
$(P_1^{(s)}, P_2^{(s)}, P_3^{(s)}) \leftarrow \mathsf{parse}_n(P^{(s)})$;	$(C_1^{(s)}, C_2^{(s)}, C_3^{(s)}) \leftarrow \mathsf{parse}_n(C^{(s)})$;
$C_1^{(s)} \xleftarrow{\$} \{0,1\}^n$; $C_2^{(s)} \xleftarrow{\$} \{0,1\}^n$;	$P_1^{(s)} \xleftarrow{\$} \{0,1\}^n$; $P_2^{(s)} \xleftarrow{\$} \{0,1\}^n$;
for $i = 1, \ldots, \mathfrak{m}^{(s)} - 3$ do $C_{3,i}^{(s)} \xleftarrow{\$} \{0,1\}^n$;	for $i = 1, \ldots, \mathfrak{m}^{(s)} - 3$ do $P_{3,i}^{(s)} \xleftarrow{\$} \{0,1\}^n$;
$W^{(s)} \xleftarrow{\$} \{0,1\}^n$; $C_{3,\mathfrak{m}^{(s)}-2} \leftarrow \mathsf{first}_{r^{(s)}}(W^{(s)})$;	$V^{(s)} \xleftarrow{\$} \{0,1\}^n$; $P_{3,\mathfrak{m}^{(s)}-2} \leftarrow \mathsf{first}_{r^{(s)}}(V^{(s)})$;
$A_1^{(s)} \leftarrow P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})$;	$B_2^{(s)} \leftarrow C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$;
$A_2^{(s)} \leftarrow \tau A_1^{(s)} \oplus P_2^{(s)}$; $\mathsf{ChkDom}(A_2^{(s)})$;	$B_1^{(s)} \leftarrow C_1^{(s)} \oplus \tau B_2^{(s)}$; $\mathsf{ChkDom}(B_1^{(s)})$;
$Y_1^{(s)} \leftarrow C_1^{(s)} \oplus P_1^{(s)} \oplus \tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)}))$ $\oplus h_\tau(T^{(s)}, P_3^{(s)})$;	$Y_2^{(s)} \leftarrow C_2^{(s)} \oplus P_2^{(s)} \oplus \tau P_1^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$ $\oplus \tau h_\tau(T^{(s)}, P_3^{(s)})$;
$\rho(A_2^{(s)}) \leftarrow Y_1^{(s)}$;	$\rho(B_1^{(s)}) \leftarrow Y_2^{(s)}$;
$B_1^{(s)} \leftarrow A_1^{(s)} \oplus Y_1^{(s)}$; $\mathsf{ChkDom}(B_1^{(s)})$;	$A_2^{(s)} \leftarrow B_2^{(s)} \oplus Y_2^{(s)}$; $\mathsf{ChkDom}(A_2^{(s)})$;
$Y_2^{(s)} \leftarrow C_2^{(s)} \oplus P_2^{(s)} \oplus \tau P_1^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$ $\oplus \tau h_\tau(T^{(s)}, P_3^{(s)})$;	$Y_1^{(s)} \leftarrow C_1^{(s)} \oplus P_1^{(s)} \oplus \tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)}))$ $\oplus h_\tau(T^{(s)}, P_3^{(s)})$;
$\rho(B_1^{(s)}) \leftarrow Y_2^{(s)}$;	$\rho(A_2^{(s)}) \leftarrow Y_1^{(s)}$;
$B_2^{(s)} \leftarrow A_2^{(s)} \oplus Y_2^{(s)}$;	$A_1^{(s)} \leftarrow B_1^{(s)} \oplus Y_1^{(s)}$;
$C_1^{(s)} \leftarrow \tau B_2^{(s)} \oplus B_1^{(s)}$;	$P_2^{(s)} \leftarrow \tau A_1^{(s)} \oplus A_2^{(s)}$;
$Z^{(s)} \leftarrow A_2^{(s)} \oplus B_1^{(s)}$;	$Z^{(s)} \leftarrow A_2^{(s)} \oplus B_1^{(s)}$;
for $i = 1$ to $\mathfrak{m}^{(s)} - 3$ do	for $i = 1$ to $\mathfrak{m}^{(s)} - 3$ do
$J_i^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(i)$; $\mathsf{ChkDom}(J_i^{(s)})$;	$J_i^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(i)$; $\mathsf{ChkDom}(J_i^{(s)})$;
$\rho(J_i^{(s)}) \leftarrow C_{3,i}^{(s)} \oplus P_{3,i}^{(s)}$;	$\rho(J_i^{(s)}) \leftarrow C_{3,i}^{(s)} \oplus P_{3,i}^{(s)}$;
end for;	end for;
$J_{\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(\mathfrak{m}^{(s)} - 2)$;	$J_{\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(\mathfrak{m}^{(s)} - 2)$;
$\mathsf{ChkDom}(J_{\mathfrak{m}^{(s)}-2}^{(s)})$;	$\mathsf{ChkDom}(J_{\mathfrak{m}^{(s)}-2}^{(s)})$;
$D^{(s)} \leftarrow W^{(s)} \oplus (P_{3,\mathfrak{m}^{(s)}-2}^{(s)} \| 0^{n-r^{(s)}})$;	$E^{(s)} \leftarrow V^{(s)} \oplus (C_{3,\mathfrak{m}^{(s)}-2}^{(s)} \| 0^{n-r^{(s)}})$;
$\rho(J_{\mathfrak{m}^{(s)}-2}^{(s)}) \leftarrow D^{(s)}$;	$\rho(J_{\mathfrak{m}^{(s)}-2}^{(s)}) \leftarrow E^{(s)}$;
return $(C_1^{(s)} \| C_2^{(s)} \| C_3^{(s)})$.	return $(P_1^{(s)} \| P_2^{(s)} \| P_3^{(s)})$.

in the following manner.

$$\begin{aligned}
\rho(A_2^{(s)}) &= Y_1^{(s)} \leftarrow C_1^{(s)} \oplus P_1^{(s)} \oplus \tau C_2^{(s)} \oplus \tau h'_\tau(T^{(s)}, C_3^{(s)}) \oplus h_\tau(T^{(s)}, P_3^{(s)}); \\
\rho(B_1^{(s)}) &= Y_2^{(s)} \leftarrow C_2^{(s)} \oplus P_2^{(s)} \oplus \tau P_1^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)}) \oplus \tau h_\tau(T^{(s)}, P_3^{(s)}); \\
\rho(J_i^{(s)}) &= C_{3,i}^{(s)} \oplus P_{3,i}^{(s)} \text{ for } i = 1, \ldots, \mathfrak{m}^{(s)} - 3; \\
\rho(J_{\mathfrak{m}^{(s)}-2}^{(s)}) &= \begin{cases} D^{(s)} \oplus (P_{3,\mathfrak{m}^{(s)}-2}^{(s)} \| 0^{n-r^{(s)}}) & \text{if } \mathsf{ty}^{(s)} = \mathsf{enc}; \\ E^{(s)} \oplus (C_{3,\mathfrak{m}^{(s)}-2}^{(s)} \| 0^{n-r^{(s)}}) & \text{if } \mathsf{ty}^{(s)} = \mathsf{dec}; \end{cases}
\end{aligned} \tag{7.39}$$

As for an encryption query, $C_1^{(s)}, C_2^{(s)}, C_{3,1}^{(s)}, \ldots, C_{3,\mathfrak{m}^{(s)}-3}^{(s)}, D^{(s)}$ are chosen independently and uniformly at random, from (7.39) it follows that the outputs of ρ are also independent and uniformly distributed. Similarly, for a decryption query, $P_1^{(s)}, P_2^{(s)}, P_{3,1}^{(s)}, \ldots, P_{3,\mathfrak{m}^{(s)}-3}^{(s)}, E^{(s)}$ are chosen independently and uniformly at random. Again, from (7.39) it follows that the outputs of ρ are also independent and uniformly distributed. So, as in game G_{real}, in game G_{int} also the outputs of ρ are independent and uniformly distributed.

Let $\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})$ be the event that the variable bad is set to true in the game G_{int}. Let $\mathcal{A}^{G_{\mathsf{int}}} \Rightarrow 1$ denote the event that $\mathcal{A}$ outputs 1 in the game G_{int}. From the description of the games, it follows that if bad does not occur, then $\mathcal{A}$'s views in both G_{real} and G_{int} are the same. Also, the probabilities that bad occurs in the two games are equal. This gives the following.

Claim 1.

$$\begin{aligned} \Pr\left[\left(\mathcal{A}^{G_{\mathsf{real}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})}\right] &= \Pr\left[\left(\mathcal{A}^{G_{\mathsf{int}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})}\right]; \\ \Pr\left[\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})\right] &= \Pr\left[\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})\right]. \end{aligned}$$

Next, the game G_{int} is changed to the game G_{rnd} which is shown in Table 7.6. In this game, there is no ρ. For an enciphering query, the ciphertext is chosen independently and uniformly at random and for a deciphering query, the plaintext is chosen independently and uniformly at random. These are returned to $\mathcal{A}$. After the interaction is over, in the finalisation step, the internal random variables are included in $\mathcal{D}$ and bad is set to true if there is a collision in $\mathcal{D}$. Let $\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})$ be the event that the variable bad is set to true in the game G_{rnd}. Let $\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1$ denote the event that $\mathcal{A}$ outputs 1 in the game G_{rnd}. If bad does not occur, then in both the games G_{int} and G_{rnd}, $\mathcal{A}$ obtains independent and uniform random strings as responses to all its queries. Also, the probabilities that bad occurs in the two games are equal. So, we have the following.

Claim 2.

$$\begin{aligned} \Pr\left[\left(\mathcal{A}^{G_{\mathsf{int}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})}\right] &= \Pr\left[\left(\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})}\right]; \\ \Pr\left[\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})\right] &= \Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})\right]. \end{aligned}$$

Note that the event $\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1$ is exactly the event $\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1$.

Using (7.38) along with Claims 1 and 2, we have the following.

$$\begin{aligned} &\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}}(\mathcal{A}) \\ &= \Pr\left[\mathcal{A}^{\mathsf{FAST}_\rho.\mathsf{Encrypt}(\cdot,\cdot),\mathsf{FAST}_\rho.\mathsf{Decrypt}(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\rho_1(\cdot,\cdot),\rho_2(\cdot,\cdot)} \Rightarrow 1\right] \\ &\qquad\qquad + \mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) \\ &= \mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) + \Pr\left[\mathcal{A}^{G_{\mathsf{real}}} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1\right] \\ &\leq \mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) + \Pr\left[\left(\mathcal{A}^{G_{\mathsf{real}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})}\right] + \Pr\left[\mathsf{Bad}_{\mathsf{real}}(\mathcal{A})\right] \\ &\qquad\qquad - \Pr\left[\left(\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})}\right] \end{aligned}$$

Table 7.6: Game G_{rnd}

Respond to the s^{th} adversary query as follows:

if $\mathsf{ty}^{(s)} = \mathsf{enc}$; $C_1^{(s)}||C_2^{(s)}||C_{3,1}^{(s)}||\ldots||C_{3,\mathfrak{m}^{(s)}-3}^{(s)}||D^{(s)} \xleftarrow{\$} \{0,1\}^{n\mathfrak{m}^{(s)}}$;
$C_{3,\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow \mathsf{first}_{r^{(s)}}(D^{(s)})$ **return** $C^{(s)} = C_1^{(s)}||C_2^{(s)}||C_{3,1}^{(s)}||\ldots||C_{3,\mathfrak{m}^{(s)}-2}^{(s)}$;
if $\mathsf{ty}^{(s)} = \mathsf{dec}$; $P_1^{(s)}||P_2^{(s)}||P_{3,1}^{(s)}||\ldots||P_{3,\mathfrak{m}^{(s)}-3}^{(s)}||E^{(s)} \xleftarrow{\$} \{0,1\}^{n\mathfrak{m}^{(s)}}$;
$P_{3,\mathfrak{m}^{(s)}-2}^{(s)} \leftarrow \mathsf{first}_{r^{(s)}}(E^{(s)})$ **return** $P^{(s)} = P_1^{(s)}||P_2^{(s)}||P_{3,1}^{(s)}||\ldots||P_{3,\mathfrak{m}^{(s)}-2}^{(s)}$;

Finalisation:

$\mathcal{D} \leftarrow \{\mathsf{fStr}\}$; $\mathsf{bad} \leftarrow \mathsf{false}$; $\tau \xleftarrow{\$} \{0,1\}^n$;
for $s = 1$ to q do
  $A_1^{(s)} \leftarrow P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})$;
  $B_2^{(s)} \leftarrow C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})$;
  $A_2^{(s)} \leftarrow \tau A_1^{(s)} \oplus P_2^{(s)} = \tau(P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})) \oplus P_2^{(s)}$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{A_2^{(s)}\}$;
  $B_1^{(s)} \leftarrow C_1^{(s)} \oplus \tau B_2^{(s)} = C_1^{(s)} \oplus \tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)}))$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{B_1^{(s)}\}$;
  $Z^{(s)} \leftarrow A_2^{(s)} \oplus B_1^{(s)}$;
  for $i = 1$ to $\mathfrak{m}^{(s)} - 2$;
    $J_i^{(s)} \leftarrow Z^{(s)} \oplus \mathsf{bin}_n(i) = A_2^{(s)} \oplus B_1^{(s)} \oplus \mathsf{bin}_n(i)$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{J_i^{(s)}\}$;
  end for;
end for;
if (some value occurs more than once in $\mathcal{D}$) then $\mathsf{bad} \leftarrow \mathsf{true}$ endif;

$$\begin{aligned} &= \mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}) + \Pr\left[\left(\mathcal{A}^{G_{\mathsf{int}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})}\right] + \Pr\left[\mathsf{Bad}_{\mathsf{int}}(\mathcal{A})\right] \\ &\qquad\qquad - \Pr\left[\left(\mathcal{A}^{G_{\mathsf{rnd}}} \Rightarrow 1\right) \wedge \overline{\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})}\right] \\ &= \mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}) + \Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})\right]. \end{aligned} \tag{7.40}$$

Adversary $\mathcal{A}$ runs in time $\mathfrak{T}$. We instead consider an adversary $\mathcal{C}$ which is allowed unbounded runtime and also unbounded memory. Consider the interaction of $\mathcal{C}$ with the oracle in the game G_{rnd} and define the event $\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})\right]$ in a manner analogous to $\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})\right]$. Clearly, we have

$$\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{A})\right] \;\leq\; \Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})\right]. \tag{7.41}$$

So, it is sufficient to upper bound $\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})\right]$. Since $\mathcal{C}$ has unbounded computational power, without loss of generality, we may assume that $\mathcal{C}$ is deterministic.

Upper bound on $\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})\right]$: An upper bound on $\Pr\left[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})\right]$ is obtained by showing that in the game G_{rnd} the event that two random variables in $\mathcal{D}$ are equal occurs with low probability. The main crux of the whole proof is to argue this in the various cases that arise in considering the different pairs of random variables from $\mathcal{D}$. The claims below tackle all the different cases that can arise.

Claim 3. For $1 \leq s \leq q$, $\Pr[A_2^{(s)} = \mathsf{fStr}] \leq \epsilon_1^{(s)}$.

Proof.

$$\begin{aligned} \Pr[A_2^{(s)} = \mathsf{fStr}] &= \Pr[\tau P_1^{(s)} \oplus \tau h_\tau(T^{(s)}, P_3^{(s)}) \oplus P_2^{(s)} = \mathsf{fStr}] \\ &= \Pr[\tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus P_1^{(s)}) = P_2^{(s)} \oplus \mathsf{fStr}] \\ &\leq \epsilon_1^{(s)}. \end{aligned}$$

The last inequality follows from (7.8). □

Claim 4. For $1 \leq s \leq q$, $\Pr[B_1^{(s)} = \mathsf{fStr}] \leq \epsilon_1^{(s)}$.

Proof.

$$\begin{aligned} \Pr[B_1^{(s)} = \mathsf{fStr}] &= \Pr[\tau C_2^{(s)} \oplus \tau h'_\tau(T^{(s)}, C_3^{(s)}) \oplus C_1^{(s)} = \mathsf{fStr}] \\ &= \Pr[\tau(h'_\tau(T^{(s)}, C_3^{(s)}) \oplus C_2^{(s)}) = C_1^{(s)} \oplus \mathsf{fStr}] \\ &\leq \epsilon_1^{(s)}. \end{aligned}$$

The last inequality follows from (7.9). □

Claim 5. For $1 \leq s \leq q$, $1 \leq i \leq \mathfrak{m}^{(s)} - 2$, $\Pr[J_i^{(s)} = \mathsf{fStr}] = 1/2^n$.

Proof.

$$\begin{aligned} J_i^{(s)} \oplus \mathsf{fStr} &= Z^{(s)} \oplus \mathsf{bin}_n(i) \oplus \mathsf{fStr} \\ &= A_2^{(s)} \oplus B_1^{(s)} \oplus \mathsf{bin}_n(i) \oplus \mathsf{fStr} \\ &= \tau(P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})) \oplus P_2^{(s)} \oplus C_1^{(s)} \oplus \tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})) \\ &\quad \oplus \mathsf{bin}_n(i) \oplus \mathsf{fStr}. \end{aligned}$$

When $\mathsf{ty}^{(s)} = \mathsf{enc}$, then $C_1^{(s)}$ is an n-bit uniform random string which is independent of the other quantities and when $\mathsf{ty}^{(s)} = \mathsf{dec}$, then $P_2^{(s)}$ is an n-bit uniform random string which is independent of the other quantities. So in both cases we have the required probability. □

Claim 6. For $s \neq t$, $\Pr[A_2^{(s)} = A_2^{(t)}] \leq \max\{\epsilon_2^{(s,t)}, 1/2^n\}$.

Proof.

$$A_2^{(s)} \oplus A_2^{(t)} = \tau(P_1^{(s)} \oplus P_1^{(t)}) \oplus \tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus h_\tau(T^{(t)}, P_3^{(t)})) \oplus P_2^{(s)} \oplus P_2^{(t)}.$$

There are four cases to consider.

Case 1: $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{enc}$. There are two sub-cases.

(a) **Case 1a:** $(T^{(s)}, P_1^{(s)}, P_3^{(s)}) = (T^{(t)}, P_1^{(t)}, P_3^{(t)})$.
As the adversary is not allowed to repeat a query, hence $(T^{(s)}, P_1^{(s)}, P_3^{(s)}) = (T^{(t)}, P_1^{(t)}, P_3^{(t)})$ implies $P_2^{(s)} \neq P_2^{(t)}$ and so $\Pr[A_2^{(s)} = A_2^{(t)}] = 0$.

(b) **Case 1b:** $(T^{(s)}, P_1^{(s)}, P_3^{(s)}) \neq (T^{(t)}, P_1^{(t)}, P_3^{(t)})$.
If $(T^{(s)}, P_3^{(s)}) = (T^{(t)}, P_3^{(t)})$, then $P_1^{(s)} \neq P_1^{(t)}$ and so $A_2^{(s)} \oplus A_2^{(t)} = \tau(P_1^{(s)} \oplus P_1^{(t)}) \oplus P_2^{(s)} \oplus P_2^{(t)}$ is a non-zero polynomial in τ of degree 1. Thus, $\Pr[A_2^{(s)} = A_2^{(t)}] = 1/2^n$.
If $(T^{(s)}, P_3^{(s)}) \neq (T^{(t)}, P_3^{(t)})$, then

$$\begin{aligned} &\Pr[A_2^{(s)} = A_2^{(t)}] \\ &= \Pr[\tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus h_\tau(T^{(t)}, P_3^{(t)}) \oplus P_1^{(s)} \oplus P_1^{(t)}) = P_2^{(s)} \oplus P_2^{(t)}] \\ &\leq \epsilon_2^{(s,t)}. \end{aligned}$$

The last inequality follows from (7.10).

Case 2: $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{dec}$. In this case all of $P_1^{(s)}, P_1^{(t)}, P_2^{(s)}, P_2^{(t)}$ are independent and uniformly distributed n-bit strings and so $\Pr[A_2^{(s)} = A_2^{(t)}] = 1/2^n$.

Case 3: $\mathsf{ty}^{(s)} = \mathsf{enc}$ and $\mathsf{ty}^{(t)} = \mathsf{dec}$. In this case $P_1^{(t)}$ and $P_2^{(t)}$ are independent and uniformly distributed n-bit strings and so $\Pr[A_2^{(s)} = A_2^{(t)}] = 1/2^n$.

Case 4: $\mathsf{ty}^{(s)} = \mathsf{dec}$ and $\mathsf{ty}^{(t)} = \mathsf{enc}$. In this case $P_1^{(s)}$ and $P_2^{(s)}$ are independent and uniformly distributed n-bit strings and so $\Pr[A_2^{(s)} = A_2^{(t)}] = 1/2^n$.

□

Claim 7. For $s \neq t$, $\Pr[B_1^{(s)} = B_1^{(t)}] \leq \max\{\epsilon_2^{(s,t)}, 1/2^n\}$.

The proof is almost the same as the proof of Claim 6.

Claim 8. For $1 \leq s, t \leq q$, $\Pr[A_2^{(s)} = B_1^{(t)}] \leq \max\{\epsilon_2^{(s,t)}, 1/2^n\}$.

Proof.

$$\begin{aligned} & A_2^{(s)} \oplus B_1^{(t)} \\ &= \tau(P_1^{(s)} \oplus C_2^{(t)}) \oplus \tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus h'_\tau(T^{(t)}, C_3^{(t)})) \oplus (P_2^{(s)} \oplus C_1^{(t)}). \end{aligned}$$

There are four cases.

Case 1: $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{enc}$. In this case, $C_1^{(t)}$ is an independent and uniform random n-bit string and so $\Pr[A_2^{(s)} = B_1^{(t)}] = 1/2^n$.

Case 2: $\mathsf{ty}^{(s)} = \mathsf{enc}$ and $\mathsf{ty}^{(t)} = \mathsf{dec}$. We have

$$\begin{aligned} & \Pr[A_2^{(s)} = B_1^{(t)}] \\ &= \Pr[\tau(P_1^{(s)} \oplus C_2^{(t)}) \oplus \tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus h'_\tau(T^{(t)}, C_3^{(t)})) = P_2^{(s)} \oplus C_1^{(t)}] \\ &= \Pr[\tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus h'_\tau(T^{(t)}, C_3^{(t)}) \oplus P_1^{(s)} \oplus C_2^{(t)}) = P_2^{(s)} \oplus C_1^{(t)}] \\ &\leq \epsilon_2^{(s,t)}. \end{aligned}$$

The last inequality follows from (7.12).

Case 3: $\mathsf{ty}^{(s)} = \mathsf{dec}$ and $\mathsf{ty}^{(t)} = \mathsf{enc}$. In this case, $P_2^{(s)}$ is an independent and uniform random n-bit string and so $\Pr[A_2^{(s)} = B_1^{(t)}] = 1/2^n$.

Case 4: $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{dec}$. In this case also, $P_2^{(s)}$ is an independent and uniform random n-bit string and so $\Pr[A_2^{(s)} = B_1^{(t)}] = 1/2^n$.

□

Claim 9. For $1 \leq s, t \leq q$ and $1 \leq i \leq \mathfrak{m}^{(t)} - 2$, $\Pr[A_2^{(s)} = J_i^{(t)}] \leq \epsilon_1^{(s)}$.

Proof.

$$\begin{aligned} \Pr[A_2^{(s)} = J_i^{(t)}] &= \Pr[A_2^{(s)} = A_2^{(t)} \oplus B_1^{(t)} \oplus \mathsf{bin}_n(i)] \\ &= \Pr[\tau(h_\tau(T^{(s)}, P_3^{(s)}) \oplus P_1^{(s)}) \oplus P_2^{(s)} \\ &\qquad = \tau(P_1^{(t)} \oplus h_\tau(T^{(t)}, P_3^{(t)})) \oplus P_2^{(t)} \\ &\qquad \oplus C_1^{(t)} \oplus \tau(C_2^{(t)} \oplus h'_\tau(T^{(t)}, C_3^{(t)})) \oplus \mathsf{bin}_n(i)]. \end{aligned}$$

First suppose that $s \neq t$. If $\mathsf{ty}^{(t)} = \mathsf{enc}$, then $C_1^{(t)}$ is a uniform n-bit string which is independent of the other quantities and if $\mathsf{ty}^{(t)} = \mathsf{dec}$, then $P_2^{(t)}$ is a uniform n-bit string which is independent of the other quantities. In both cases, the above probability is $1/2^n$.

So, suppose that $s = t$. Then the required probability reduces to

$$\begin{aligned}\Pr[A_2^{(s)} = J_i^{(s)}] &= \Pr[B_1^{(s)} = \mathsf{bin}_n(i)] \\ &= \Pr[\tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})) = C_1^{(s)} \oplus \mathsf{bin}_n(i)] \\ &\leq \epsilon_1^{(s)}.\end{aligned}$$

If $\mathsf{ty}^{(s)} = \mathsf{enc}$, then $C_1^{(s)}$ is a uniform n-bit string which is independent of the other quantities and so the probability is equal to $1/2^n$; on the other hand, if $\mathsf{ty}^{(s)} = \mathsf{dec}$, then the last inequality follows from (7.9). □

Claim 10. For $1 \leq s, t \leq q$ and $1 \leq i \leq \mathfrak{m}^{(t)} - 2$, $\Pr[B_1^{(s)} = J_i^{(t)}] \leq \epsilon_1^{(s)}$.

The proof is almost the same as the proof of Claim 9.

Claim 11. For $1 \leq s, t \leq q$, $1 \leq i \leq \mathfrak{m}^{(s)} - 2$, $1 \leq j \leq \mathfrak{m}^{(t)} - 2$ and $(s, i) \neq (t, j)$, $\Pr[J_i^{(s)} = J_j^{(t)}] \leq 1/2^n$.

Proof.

$$\begin{aligned}J_i^{(s)} \oplus J_j^{(t)} &= A_2^{(s)} \oplus B_1^{(s)} \oplus A_2^{(t)} \oplus B_1^{(t)} \oplus \mathsf{bin}_n(i) \oplus \mathsf{bin}_n(j) \\ &= \tau(P_1^{(s)} \oplus h_\tau(T^{(s)}, P_3^{(s)})) \oplus P_2^{(s)} \oplus C_1^{(s)} \oplus \tau(C_2^{(s)} \oplus h'_\tau(T^{(s)}, C_3^{(s)})) \\ &\quad \oplus \tau(P_1^{(t)} \oplus h_\tau(T^{(t)}, P_3^{(t)})) \oplus P_2^{(t)} \oplus C_1^{(t)} \oplus \tau(C_2^{(t)} \oplus h'_\tau(T^{(t)}, C_3^{(t)})) \\ &\quad \oplus \mathsf{bin}_n(i) \oplus \mathsf{bin}_n(j).\end{aligned}$$

If $s = t$, then $i \neq j$ and so $\Pr[J_i^{(s)} = J_j^{(t)}] = \Pr[\mathsf{bin}_n(i) = \mathsf{bin}_n(j)] = 0$. Suppose that $s \neq t$. There are four cases to consider.

- If $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{enc}$, then both $C_1^{(s)}$ and $C_1^{(t)}$ are independent and uniform random strings which are independent of the other quantities.
- If $\mathsf{ty}^{(s)} = \mathsf{ty}^{(t)} = \mathsf{dec}$, then both $P_2^{(s)}$ and $P_2^{(t)}$ are independent and uniform random strings which are independent of the other quantities.
- If $\mathsf{ty}^{(s)} = \mathsf{enc}$ and $\mathsf{ty}^{(t)} = \mathsf{dec}$, then both $C_1^{(s)}$ and $P_2^{(t)}$ are independent and uniform random strings which are independent of the other quantities.
- If $\mathsf{ty}^{(s)} = \mathsf{dec}$ and $\mathsf{ty}^{(t)} = \mathsf{enc}$, then both $P_2^{(s)}$ and $C_1^{(t)}$ are independent and uniform random strings which are independent of the other quantities.

From the above it follows that if $s \neq t$, then in all cases the probability is equal to $1/2^n$. Thus, the claim follows. □

By Claims 3 to 11 and the union bound we have

$$\begin{aligned}&\Pr[\mathsf{Bad}_{\mathsf{rnd}}(\mathcal{C})] \\ &\leq 2\sum_{s=1}^{q} \epsilon_1^{(s)} + \sum_{s=1}^{q} \left(\frac{\mathfrak{m}^{(s)} - 2}{2^n}\right) + \sum_{1 \leq s < t \leq q} 2\left(\epsilon_2^{(s,t)} + \frac{1}{2^n}\right)\end{aligned}$$

$$
\begin{aligned}
&+\sum_{1\le s\le t\le q}\left(\epsilon_2^{(s,t)}+\frac{1}{2^n}\right)+2\left(\sum_{s=1}^{q}\epsilon_1^{(s)}\right)\left(\sum_{t=1}^{q}(\mathfrak{m}^{(t)}-2)\right)\\
&+\frac{1}{2^n}\binom{\sum_{s=1}^{q}(\mathfrak{m}^{(s)}-2)}{2}\\
=\;& 2\left(\sum_{s=1}^{q}\epsilon_1^{(s)}\right)\left(1+\sum_{t=1}^{q}(\mathfrak{m}^{(t)}-2)\right)+\frac{3}{2^n}\frac{q(q-1)}{2}+\frac{q}{2^n}+3\sum_{1\le s<t\le q}\epsilon_2^{(s,t)}\\
&\qquad+\sum_{s=1}^{q}\epsilon_2^{(s,s)}+\sum_{s=1}^{q}\left(\frac{\mathfrak{m}^{(s)}-2}{2^n}\right)+\frac{1}{2^n}\binom{\sum_{s=1}^{q}(\mathfrak{m}^{(s)}-2)}{2}\\
\le\;& 2\left(\sum_{s=1}^{q}\epsilon_1^{(s)}\right)(1+\omega-2q)+\frac{2q^2}{2^n}+3\sum_{1\le s<t\le q}\epsilon_2^{(s,t)}+\sum_{s=1}^{q}\epsilon_2^{(s,s)}\\
&\qquad+\frac{\omega-2q}{2^n}+\frac{1}{2^n}\frac{\omega(\omega-1)}{2}\\
\le\;& 2\omega\left(\sum_{s=1}^{q}\epsilon_1^{(s)}\right)+3\sum_{1\le s<t\le q}\epsilon_2^{(s,t)}+\sum_{s=1}^{q}\epsilon_2^{(s,s)}+\frac{3\omega^2}{2^n}.
\end{aligned}
\tag{7.42}
$$

Putting together (7.40), (7.41) and (7.42) we obtain

$$
\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}}(\mathcal{A}) \;\le\; \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B})+2\omega\left(\sum_{s=1}^{q}\epsilon_1^{(s)}\right)+3\sum_{1\le s<t\le q}\epsilon_2^{(s,t)}+\sum_{s=1}^{q}\epsilon_2^{(s,s)}+\frac{3\omega^2}{2^n}.
$$

The relations between the resources of $\mathcal{A}$ and $\mathcal{B}$ have been stated earlier. Maximising the left hand side on the resources shows the required result and completes the proof of Theorem 10. □

We have the following consequences of Theorem 10 for the specific instantiations of FAST.

Corollary 4. *Let $\mathfrak{m}\ge 3$ be a fixed positive integer. Let q and $\sigma\ge q$ be positive integers and $\omega=\sigma-q$. Consider the instantiations* $\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}},\mathsf{Horner}]$ *and* $\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}},\mathsf{BRW}]$ *of* FAST*. Then for all $\mathfrak{T}>0$,*

$$
\begin{aligned}
&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}},\mathsf{Horner}]}(\mathfrak{T},q,q,\omega)\\
&\quad\le\; \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}',\omega+1,\omega+1)+\frac{1}{2^n}\left(5\omega^2+\omega+\frac{11\omega q}{2}+3q^2+2q\right).
\end{aligned}
\tag{7.43}
$$

If $\mathfrak{m}\ge 4$, then for all $\mathfrak{T}>0$,

$$
\begin{aligned}
&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}[\mathsf{Fx}_{\mathfrak{m}},\mathsf{BRW}]}(\mathfrak{T},q,q,\omega)\\
&\quad\le\; \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}',\omega+1,\omega+1)\\
&\qquad+\frac{1}{2^n}\left(6\omega q+\frac{9q^2}{2}+3q+3\omega^2+d(\mathfrak{m}-1)\left(2\omega q+\frac{3q^2}{2}+q\right)\right)\\
&\quad\le\; \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}',\omega+1,\omega+1)+\frac{1}{2^n}\left(7\omega^2+3\omega q+2\omega\right).
\end{aligned}
\tag{7.44}
$$

Further, if $\mathfrak{m} \geq 4$ *is a power of two, then for all* $\mathfrak{T} > 0$,

$$\begin{aligned}&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m},\mathsf{BRW}]}(\mathfrak{T}, q, q, \omega)\\ &\quad\leq\ \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}', \omega+1, \omega+1) + \frac{1}{2^n}\left(5\omega^2+\omega+\frac{11\omega q}{2}+3q^2+2q\right). \qquad (7.45)\end{aligned}$$

Proof. Let $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{Horner}])$ and $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}])$ denote the expressions for Δ in (7.34) when $\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{Horner}]$ and $\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}]$ are respectively used.

In the setting of $\mathsf{Fx}_\mathfrak{m}$, each query consists of a single n-bit block for the tweak and $\mathfrak{m}$ n-bit blocks for the plaintext or the ciphertext, i.e., $\mathfrak{m}^{(s)} = \mathfrak{m}$ for all $1 \leq s \leq q$. So, $\theta = q$, $\omega = q\mathfrak{m}$ and $\sigma = \theta + \omega = q(\mathfrak{m}+1)$. Also, $\mathfrak{m} \geq 3$ and $q \geq 1$ imply that $q < \omega$ and $\mathfrak{m} \leq \omega$.

From Proposition 8, we have $\epsilon_1^{(s)} = \epsilon_2^{(s,t)} = (\mathfrak{m}+2)/2^n$ for $1 \leq s, t \leq q$. Using these, $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{Horner}])$ can be upper bounded as given in (7.43).

From Proposition 9, we have $\epsilon_1^{(s)} = \epsilon_2^{(s,t)} = (3 + d(\mathfrak{m}-1))/2^n$ for $1 \leq s, t \leq q$. Using these, $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}])$ achieves the first bound and using $d(\mathfrak{m}-1) \leq 2(\mathfrak{m}-1)-1$ yields the second bound.

In the case where $\mathfrak{m} \geq 4$ is a power of two, then $d(\mathfrak{m}-1) = \mathfrak{m}-1$. Also, from Proposition 9, we have $\epsilon_1^{(s)} = \epsilon_2^{(s,t)} = (\mathfrak{m}+2)/2^n$ and so $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}]) = \Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{Horner}])$ which shows the required statement. □

Remarks:

1. As mentioned above, the query complexity σ is equal to the sum of the tweak query complexity θ and the message query complexity ω, i.e., $\sigma = \theta + \omega$. For the setting of Fx, $\theta = q$ so that $\sigma = q + \omega$. Using this, the bounds in (7.43), (7.44) and (7.45) are all of the form $\mathfrak{c}\sigma^2$ for some small constant $\mathfrak{c}$. This is the typical form of the bound that is obtained for other constructions in the literature. So, FAST does not suffer any security loss compared to previous constructions.

2. Consider the application of FAST to disk encryption of 4096-byte sectors. Then $\mathfrak{m} = 2^8$. Using $\omega = \mathfrak{m}q$, the bounds for $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{Horner}])$ and $\Delta(\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}])$ given by (7.43) and (7.45) respectively both reduce to the following form.

$$\frac{1}{2^n}\left(q^2(5\cdot 2^{16}+1441)+258q\right) < \frac{1}{2^n}\left(5\sigma^2+2\sigma\right).$$

Let us consider a numerical example to illustrate the above bound. Suppose that an adversary provides a total of 2^{40} bytes as part of plaintext/ciphertext in all its queries. Assuming the size of a disk sector to be 4096 bytes, the 2^{40} bytes is provided in a total of 2^{28} queries, i.e., $q = 2^{28}$. With this value of q, the above security bound is approximately 2^{-54}.

Corollary 5. *Let* q, $\omega \geq q$ *be positive integers and* θ *be a non-negative integer. Consider the instantiations* $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ *and* $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \eta, \mathsf{vecHash2L}]$ *of* FAST. *Then for all*

$\mathfrak{T} > 0$,

$$\begin{aligned}\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST[Gn},\mathfrak{k},\mathsf{vecHorner}]}(\mathfrak{T},q,\theta,\omega) &\le \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}',\omega+1,\omega+1) \\ &\quad + \frac{5\omega^2+2\omega\theta+\omega+\theta}{2^n} + \frac{3q(\theta+\omega)+(\mathfrak{k}+2)((2\omega+1)q+3q^2)+6q^2}{2^n};\end{aligned} \tag{7.46}$$

$$\begin{aligned}&\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{FAST[Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}]}(\mathfrak{T},q,\theta,\omega) \\ &\le \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathfrak{T}+\mathfrak{T}',\omega+1,\omega+1) \\ &\quad + \frac{(3+2(\eta+1)/\eta)\omega^2+(2(\eta+1)/\eta)\omega\theta+((\eta+1)/\eta)(\omega+\theta)}{2^n} \\ &\quad + \frac{3q((\eta+1)/\eta)(\theta+\omega)+((2\omega+1)q+3q^2)(\mathfrak{k}+1)(\eta+2)+3q(2\omega+1)+9q^2}{2^n}.\end{aligned} \tag{7.47}$$

Proof. Let $\Delta(\mathsf{FAST[Gn},\mathfrak{k},\mathsf{vecHorner}])$ and $\Delta(\mathsf{FAST[Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}])$ denote the expressions for Δ in (7.34) when $\mathsf{FAST[Gn},\mathfrak{k},\mathsf{vecHorner}]$ and $\mathsf{FAST[Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}]$ are respectively used.

First consider $\mathsf{FAST[Gn},\mathfrak{k},\mathsf{vecHorner}]$. From Proposition 10, $\epsilon_1^{(s)} = (\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)}+\mathfrak{k}+4)/2^n$ and $\epsilon_2^{(s,t)} = (\max(\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)},\mathfrak{t}^{(t)}+\mathfrak{n}^{(t)})+\mathfrak{k}+4)/2^n$. So, $\epsilon_2^{(s,s)} = \epsilon_1^{(s)}$. We have

$$\begin{aligned}\sum_{s=1}^{q}\epsilon_1^{(s)} &= \sum_{s=1}^{q}\frac{\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)}+\mathfrak{k}+4}{2^n} = \frac{\theta+\omega}{2^n}+\frac{q(\mathfrak{k}+2)}{2^n}; \\ \sum_{1\le s<t\le q}\epsilon_2^{(s,t)} &\le \sum_{1\le s<t\le q}\frac{\max(\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)},\mathfrak{t}^{(t)}+\mathfrak{n}^{(t)})+\mathfrak{k}+4}{2^n} \le \frac{q(\theta+\omega)}{2^n}+\frac{q^2(\mathfrak{k}+4)}{2^n}.\end{aligned}$$

Using these in Theorem 10, we obtain

$$\begin{aligned}\Delta(\mathsf{FAST[Gn},\mathfrak{k},\mathsf{vecHorner}]) &\le \frac{5\omega^2+2\omega\theta+\omega+\theta}{2^n} \\ &\quad + \frac{3q(\theta+\omega)+(\mathfrak{k}+2)((2\omega+1)q+3q^2)+6q^2}{2^n}.\end{aligned}$$

Now consider $\mathsf{FAST[Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}]$. From Proposition 11,

$$\begin{aligned}\epsilon_1^{(s)} &= \frac{((\eta+1)/\eta)(\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)})+(\mathfrak{k}+1)(\eta+2)+3}{2^n}; \\ \epsilon_2^{(s,t)} &= \frac{((\eta+1)/\eta)\max(\mathfrak{t}^{(s)}+\mathfrak{n}^{(s)},\mathfrak{t}^{(t)}+\mathfrak{n}^{(t)})+(\mathfrak{k}+1)(\eta+2)+3}{2^n}.\end{aligned}$$

So, $\epsilon_2^{(s,s)} = \epsilon_1^{(s)}$. We have

$$\begin{aligned}\sum_{s=1}^{q}\epsilon_1^{(s)} &\le \frac{((\eta+1)/\eta)(\theta+\omega)+q(\mathfrak{k}+1)(\eta+2)+3q}{2^n}; \\ \sum_{1\le s<t\le q}\epsilon_2^{(s,t)} &\le \frac{q((\eta+1)/\eta)(\theta+\omega)+q^2(\mathfrak{k}+1)(\eta+2)+3q^2}{2^n};\end{aligned}$$

Using these in Theorem 10, we obtain

$$\begin{aligned}&\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}])\\ &\leq \frac{(3+2(\eta+1)/\eta)\omega^2+(2(\eta+1)/\eta)\omega\theta+((\eta+1)/\eta)(\omega+\theta)}{2^n}\\ &\quad+\frac{3q((\eta+1)/\eta)(\theta+\omega)+((2\omega+1)q+3q^2)(\mathfrak{k}+1)(\eta+2)+3q(2\omega+1)+9q^2}{2^n}.\end{aligned}$$

□

Remarks:

1. Recall that $\mathfrak{k}$ is the number of components in the tweak while η is the number of blocks in a BRW super-block. So, $\mathfrak{k}$ and η are constants, i.e., they are independent of n.

2. The query complexity σ is equal to the sum of the tweak query complexity θ and the message query complexity ω, i.e., $\sigma = \theta+\omega$. Using this, it can be seen that the bounds on

$$\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\mathsf{vecHorner}]) \text{ and } \Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}])$$

given by (7.46) and (7.47) respectively are of the form $\mathfrak{c}\sigma^2$ for some constant $\mathfrak{c}$. We provide illustrations below. As mentioned earlier, this is the typical form of the security bound for earlier constructions and so FAST does not have any security loss compared to the known constructions.

3. In our implementations, we take $\eta = 31$ and we use the bounds $(\eta+1)/\eta < 2$, $2(\eta+1)/\eta < 3$ and $3(\eta+1)/\eta < 4$. The value of $\mathfrak{k}$ and the lengths of the tweaks depend on the application. We provide two illustrations.
Case $\mathfrak{k} = 1$ and each tweak is an n-bit string: In this scenario, the bounds for $\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\mathsf{vecHorner}])$ and $\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}])$ can be shown to be respectively less than

$$\frac{5\sigma^2+\sigma q+12q^2+\sigma+3q}{2^n} \quad \text{and} \quad \frac{6\sigma^2+135\sigma q+72q^2+2\sigma+69q}{2^n}.$$

Case $\mathfrak{k} = 8$ where the components of a tweak can have variable lengths:
The bounds for $\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\mathsf{vecHorner}])$ and $\Delta(\mathsf{FAST}[\mathsf{Gn},\mathfrak{k},\eta,\mathsf{vecHash2L}])$ can be shown to be respectively less than

$$\frac{1}{2^n}\left(5\omega^2+2\omega\theta+\omega+\theta\right)+\frac{1}{2^n}\left(3q(\omega+\theta)+10((2\omega+1)q+3q^2)+6q^2\right) \tag{7.48}$$

$$\text{and } \frac{1}{2^n}\left(6\omega^2+3\omega\theta+2(\omega+\theta)+6q(\theta+\omega)+300q(2\omega+1)+900q^2\right). \tag{7.49}$$

The expressions in (7.48) and (7.49) are determined by the values of q, ω and θ. These quantities are in turn determined by the manner in which the adversary makes its queries. Since the adversary can make queries of varying lengths, it is not possible to obtain further simplifications of the expressions given in (7.48) and (7.49).

7.5 Comparison

This section provides a comparison of the design features of FAST with previously proposed TESs. As mentioned earlier, an important application of a TES is disk encryption. The literature, on the other hand, contains a number of proposals for disk encryption which do not provide provable security as TESs; our comparison does not include such schemes.

Note that FAST and all the previous TESs present in the literature provide security upto birthday bound. In that sense FAST is as good as its predecessors. Hence, we do not consider the security aspect in the comparison.

Table 7.7: Comparison of different tweakable enciphering schemes according to computational efficiency. [BC] denotes the number of block cipher calls; [M] denotes the number of field multiplications; [D] denotes the number of doubling ('multiplication by x') operations;

type	scheme	[BC]	[M]	[D]
enc-mix-enc	CMC [65]	$2\mathfrak{m}+1$	–	–
	EME2* [62]	$2\mathfrak{m}+1+\mathfrak{m}/n$	–	2
	AEZ [67]	$(5\mathfrak{m}+4)/2$	–	$\frac{\mathfrak{m}-2}{4}$
	FMix [19]	$2\mathfrak{m}+1$	–	–
hash-enc-hash	XCB [81]	$\mathfrak{m}+1$	$2(\mathfrak{m}+3)$	–
	HCTR [109]	$\mathfrak{m}$	$2(\mathfrak{m}+1)$	–
	HCHfp [37]	$\mathfrak{m}+2$	$2(\mathfrak{m}-1)$	–
	TET [63]	$\mathfrak{m}+1$	$2\mathfrak{m}$	$2(\mathfrak{m}-1)$
	HEH-BRW[98]	$\mathfrak{m}+1$	$2+2\lfloor(\mathfrak{m}-1)/2\rfloor$	$2(\mathfrak{m}-1)$
	TESX with BRW [100]	$\mathfrak{m}+1$	$4+2\lfloor(\mathfrak{m}-1)/2\rfloor$	$2(\mathfrak{m}-1)$
	FAST[$\mathsf{Fx}_{\mathfrak{m}}$, Horner]	$\mathfrak{m}+1$	$2\mathfrak{m}+1$	–
	FAST[$\mathsf{Fx}_{\mathfrak{m}}$, BRW]	$\mathfrak{m}+1$	$2+2\lfloor(\mathfrak{m}-1)/2\rfloor$	–

Several block cipher based TES constructions essentially use a layer of encryption using a mode of operation of the block cipher sandwiched between two layers of hashing. Differences arise in the choice of the mode of operation, the choice of the hash functions and other details.

1. For the mode of operation, the electronic codebook mode (ECB) has been suggested in TET [63] and HEH [98] while some form of the counter mode of operation has been used in XCB [81, 83], HCTR [109] and HCH [37]. In this chapter, we use the counter mode of operation as described in the scheme HCTR [109].

2. For the hash functions, XCB, HCTR and HCH essentially use polynomial hashing based on Horner's rule. The cost of hashing in TET is higher. BRW based hashing has been suggested for HEH and implemented in hardware for fixed length messages [33].

All of the above mentioned TESs require both the encryption and the decryption functions of the block cipher. The possibility of using only the encryption function of a block cipher

to build a TES has been reported [100] and for the convenience of description let us denote this scheme by TESX. The present scheme is based upon the idea behind TESX. In terms of similarity, both TESX and the scheme in the present chapter use a Feistel layer on the first two blocks and a counter mode of operation on the rest of the blocks. There are several differences in the two constructions.

1. For TESX, inside the Feistel layer, the hash function h is used to process A_1 and B_2. The key for this hash function is τ' which is independent of the key τ which is used for the hash function outside the Feistel layer. In contrast, FAST is organised such that the Feistel layer consists of only two encryption rounds and the entire hashing using a single key τ takes place outside the Feistel layer. In summary, TESX uses two hash keys while FAST uses a single hash key. This is a significant practical difference between the two schemes.

2. For TESX, the hash of P_3 is masked with β_1 and XORed to both P_1 and P_2 and the hash of C_3 is masked with β_2 and XORed to both C_1 and C_2. FAST does away with the maskings with β_1 and β_2; the hash of P_3 is XORed to only P_1; and the hash of C_3 is XORed to only C_2.

3. For TESX, the seed to the counter mode is generated as $A_1 \oplus P_2 \oplus C_1 \oplus B_2$. FAST generates the seed to the counter mode as $A_2 \oplus B_1$.

4. The counter mode suggested in TESX requires a doubling operation for each block. The counter mode used in FAST is given by (3.3) and is based on HCTR [109]; this counter mode does not require the doubling operation.

Some of the above differences such as reducing the hash key size and avoiding doubling operations are important from a practical point of view while the others are simplifications obtained by removing unnecessary operations. Keeping the similarities and the differences in mind, it would be proper to view the present scheme as a fine-tuned version of TESX. On the other hand, one may note that the two designs are sufficiently different which is the reason that FAST requires a separate complete proof. The fine tuning is required from an engineering point of view where the goal is to obtain an efficient and clean design. More importantly, we present detailed software implementations of different instantiations of FAST and thereby actually demonstrate the advantages of the new proposal in comparison to the previous works.

Another class of block cipher based TESs such as CMC [65], EME [66, 62] (the scheme EME2 is essentially EME* [62]), AEZ [67] and FMix [19] essentially uses several layers of encryption using a mode of operation of a block cipher. CMC, EME and FMix use two layers of encryption whereas AEZ uses three layers of encryption. These TESs do not use any hash function. Out of these CMC and FMix are sequential while EME and AEZ are parallelisable. AEZ and FMix require only the encryption function of the underlying block cipher whereas CMC and EME require both the encryption and the decryption functions of the block cipher. The costs of encryption for CMC, EME and FMix are roughly two block cipher calls per block of the message and for AEZ the cost is roughly two-and-half block

Table 7.8: Comparison of different tweakable enciphering schemes according to practical and implementation simplicity. [BCK] denotes the number of block cipher keys; and [HK] denotes the number of blocks in the hash key.

type	scheme	[BCK]	[HK]	dec module	parallel
enc-mix-enc	CMC [65]	1	–	reqd	no
	EME2* [62]	1	2	reqd	yes
	AEZ [67]	1	2	not reqd	yes
	FMix [19]	1	–	not reqd	no
hash-enc-hash	XCB [81]	3	2	reqd	yes
	HCTR [109]	1	1	reqd	yes
	HCHfp [37]	1	1	reqd	yes
	TET [63]	2	3	reqd	yes
	HEH-BRW[98]	1	1	reqd	yes
	TESX with BRW [100]	1	2	not reqd	yes
	FAST[$\mathsf{Fx}_{\mathfrak{m}}$, Horner]	1	–	not reqd	yes
	FAST[$\mathsf{Fx}_{\mathfrak{m}}$, BRW]	1	–	not reqd	yes

Note: for both Tables 7.7 and 7.8, the block size is n bits, the tweak is a single n-bit block and the number of blocks $\mathfrak{m} \geq 3$ in the message is fixed.

cipher calls per block of the message. CMC and FMix do not use any doubling operation while both EME and AEZ use doubling operations. Since AEZ and EME are parallelisable while CMC and FMix are not, any reasonable implementations of AEZ and EME will be faster than both CMC and FMix. Later we provide software implementation results which show the superiority of FAST in comparison to both AEZ and EME and hence also imply the superiority of FAST over CMC and FMix.

Table 7.7 and 7.8 compares FAST to previously proposed schemes. From the viewpoint of efficiency, it is preferable to have schemes which are parallelisable. This would eliminate CMC and FMix from the comparison. Further, again from an efficiency point of view it would be preferable to have schemes which use only the encryption module of a block cipher. This restricts the comparison to AEZ and TESX. As explained above, the current construction is a fine-tuned version of TESX and Table 7.7 and 7.8 shows the comparative advantage in terms of operation counts and key size. The inherent simplification of the design of FAST over that of TESX is not captured by these parameters. Since the design approaches of FAST and AEZ are different, the comparison between FAST and AEZ cannot be determined only from the operation counts.

Among the various schemes that have been proposed, only XCB and EME2 (which is essentially EME* [62]) have been standardised. Further, AEZ is a more recent proposal and has received a fair amount of attention as part of the CAESAR[3] competition. So, it is important to provide more detailed comparison to XCB, EME2 and AEZ. Below we provide details of the software implementations of FAST and the performance results of these

[3] `https://competitions.cr.yp.to/caesar.html`

implementations in comparison to those of XCB, EME2 and AEZ. For the purpose of such comparison, we have made efficient software implementations of XCB, EME2 and AEZ. In Section 7.7 we provide a brief overview of the software implementation of AEZ. This is of some independent interest.

There have been proposals for constructing TESs from stream ciphers. The proposal [34] provides a scheme called STES which uses hardware oriented stream ciphers from the Estream portfolio to develop TES with small hardware footprint and low power consumption. A recent proposal [39] provides two TESs called Adiantum and HPolyC aimed at entry level processors. Adiantum uses four primitives, namely, AES, XChaCha12, NH and Poly1305 while HPolyC uses AES, XChaCHa12 and Poly1305. These schemes are targeted towards processors which do not provide processor support for AES and binary field multiplication. The target applications of STES and Adiantum/HPolyC are different from that of FAST. None of these schemes are expected to be as efficient as FAST on modern Intel processors, or more generally on processors which provide support for AES and binary field multiplication.

7.6 Software Implementation

In this section, we describe implementations of the various instantiations of FAST in software. For the implementation, we set $n = 128$ and so the underlying field is $\mathbb{F}_{2^{128}}$.

The implementation of the PRF F is done using the encryption function of AES. This may seem like a mismatch since the encryption function of AES is invertible while a PRF does not require invertibility. On the other hand, the PRF assumption on the encryption function of AES is widely believed to hold up to the birthday bound. Since we do not claim security beyond the birthday bound, the use of the encryption function of AES as a PRF is justified. The main reason for choosing AES is its universal acceptance.

Our target platforms for software implementation were modern Intel processors which support the AES-NI instructions which includes the 64-bit binary polynomial multiplication. The relevant Intel instructions for the two main tasks are the following.

- **Computation of AES:** The instructions relevant here are `aeskeygenassist` (for round key generation); `aesenc` (for one round of AES encryption) and `aesenclast` (for the last round of AES encryption). There are additional instructions for AES decryption. We do not mention these, since we do not require the AES decryption module.

- **Computation of polynomial multiplication:** The relevant instruction is `pclmulqdq`. This instruction takes as input two 64-bit unsigned integers representing two polynomials each of maximum degree 63 over $GF(2)$ and returns a 128-bit quantity which represents the product of these two polynomials over $GF(2)$.

For software implementation, the relevant parameters are latency and throughput. These are defined[4] as follows: "Latency is the number of processor clocks it takes for an instruction to have its data available for use by another instruction. Throughput is the number of

[4] `https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput`

processor clocks it takes for an instruction to execute or perform its calculations." On Skylake the latency/throughput figures of `aesenc`, `aesenclast` and `pclmulqdq` are 4/1.

7.6.1 Implementation of the Hash Functions

The several variants of the hash functions used in this chapter are all based on finite field computations over $\mathbb{F}_{2^n}$. As for the implementation, we choose $n = 128$, addition over this field is a simple XOR operation of 128-bit data types. Multiplication, on the other hand, is more involved.

As mentioned in Chapter 3, we consider the field $\mathbb{F}_{2^n}$ to be represented using the irreducible polynomial $\psi(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$ over $GF(2)$. The elements of $\mathbb{F}_{2^n}$ are represented using polynomials over $GF(2)$ of degrees at most 127. Let $\alpha(x)$ and $\beta(x)$ be two such polynomials. The multiplication of $\alpha(x)$ and $\beta(x)$ in $\mathbb{F}_{2^n}$ consists of the following two operations. Compute the polynomial multiplication of $\alpha(x)$ and $\beta(x)$ over $GF(2)$ and let $c(x)$ be the result. Then $c(x)$ is a polynomial over $GF(2)$ of degree at most 254. The product of $\alpha(x)$ and $\beta(x)$ over $\mathbb{F}_{2^n}$ is $c(x)$ mod $\psi(x)$. The above computation consists of two distinct steps, namely polynomial multiplication followed by reduction.

Polynomial multiplication: The instruction `pclmulqdq` multiplies two polynomials over $GF(2)$ of degrees at most 63 each and returns a polynomial of degree at most 126. This is a 64-bit polynomial multiplication over $GF(2)$. Our requirement is a 128-bit polynomial multiplication over $GF(2)$. Using the direct schoolbook method, a 128-bit polynomial multiplication can be computed using four 64-bit polynomial multiplications and hence using four `pclmulqdq` calls. Karatsuba's algorithm, on the other hand, allows the computation of a 128-bit polynomial multiplication using three `pclmulqdq` calls at the cost of a few extra XOR operations. Due to the low latency of `pclmulqdq` on Skylake processors, it turns out that schoolbook is faster than Karatsuba. This has been reported by Gueron[5] and we also observed this in our experiments. So, we opted to implement 128-bit polynomial multiplication using the schoolbook method.

Reduction: Efficient computation of $c(x)$ mod $\psi(x)$ has been discussed earlier [58]. It was shown that this operation can be efficiently computed using two `pclmulqdq` instructions along with a few other shifts and xors. A more detailed description of this procedure can also be found in Chapter 5. We implemented reduction using this method.

Horner: As mentioned earlier, $\mathsf{Horner}_\tau(X_1, \ldots, X_\mathfrak{m})$ can be computed using $\mathfrak{m} - 1$ multiplications in $\mathbb{F}_{2^n}$. Each multiplication consists of a polynomial multiplication followed by a reduction. Doing this directly, would lead to a count of $\mathfrak{m} - 1$ polynomial multiplications and $\mathfrak{m} - 1$ reductions.

The efficiency can be improved by using a delayed (or lazy) reduction strategy. Let $i > 1$ be a positive integer and suppose the powers $1, \tau, \tau^2, \ldots, \tau^{i-1}, \tau^i$ are available (i.e., the powers $\tau^2, \ldots, \tau^{i-1}, \tau^i$ have been pre-computed and stored). The expression $X_1\tau^{i-1} + \cdots + X_{i-1}\tau +$

[5] `https://github.com/Shay-Gueron/AES-GCM-SIV/`

X_i over $\mathbb{F}_{2^n}$ can be computed using $i-1$ polynomial multiplications followed by a single reduction. Extension to handle arbitrary number of blocks is easy. For simplicity, assume that $i|\mathfrak{m}$ and $\lambda = \mathfrak{m}/i$. The $\mathfrak{m}$ blocks are divided into λ groups of i blocks each. Each group of i blocks is processed and suppose the outputs are $Y_1, Y_2, \ldots, Y_\lambda$. Then $\mathsf{Horner}_\tau(X_1, \ldots, X_\mathfrak{m}) = \tau^i(\cdots \tau^i(\tau^i Y_1 \oplus Y_2) \oplus \cdots \oplus Y_{\lambda-1}) \oplus Y_\lambda$. Processing of a single such group of i blocks requires $i-1$ polynomial multiplications and a single reduction plus a multiplication by τ^i. Note that the computation of $\tau^i Y_1 \oplus Y_2$ is done by performing the polynomial multiplication of τ^i and Y_1, computing Y_2 without the final reduction, adding the two results and then performing a reduction. Further, this strategy is also carried out for the intermediate computations. So, processing a group of i blocks requires i polynomial multiplications and a single reduction except for the last group. In the case where i does not divide $\mathfrak{m}$, it is easy to modify this strategy to handle this case. We have implemented this strategy for $i = 8$ (for use in Horner and vecHorner) and $i = 9$ (for use in vecHash2L). This strategy of delayed reduction has been earlier used [60] in the context of evaluation of POLYVAL which is a variant of Horner.

There is another technique which can result in efficiency improvement. The sequence $X_1, \ldots, X_\mathfrak{m}$ is decimated into j subsequences $X_1, X_{j+1}, \ldots;\ X_2, X_{j+2}, \ldots;\ \ldots;\ X_j, X_{2j}, \ldots;$ each subsequence is computed as a polynomial in τ^j and then the results are combined together to obtain the final result. This is a well known technique and has been discussed in more detail in Chapter 5. The advantage of this technique is that the j sub-sequences can be computed in parallel. In software the ability to batch j independent multiplications allows the processor to efficiently pre-fetch and pipeline the corresponding operations. We have experimented with $j = 1$, 2 and 3 and later we report timing results for $j = 3$. There are cases, however, where $j = 1$ provides slightly better performance than $j = 3$.

vecHorner: The computation of vecHorner essentially boils down to Horner computation on several different blocks. The implementation of Horner is extended to implement the computation of vecHorner.

BRW: Implementation of BRW arises as part of implementing $\mathsf{FAST}[\mathsf{Fx}_\mathfrak{m}, \mathsf{BRW}]$. For this implementation, we chose $\mathfrak{m} = 256$ and $n = 128$ (corresponding to a 4096-byte disk sector to be encrypted using AES). With $\mathfrak{m} = 256$, BRW is invoked on $\mathfrak{m} - 2 + 1 = 255$ (the first two message blocks are not hashed while the single tweak block is hashed) blocks. The implementation of BRW on 255 blocks has been done in the following manner. Write

$$\mathsf{BRW}_\tau(X_1, \ldots, X_{255}) = (X_{128} \oplus \tau^{128}) \times \mathsf{BRW}_\tau(X_1, \ldots, X_{127}) \oplus \mathsf{BRW}_\tau(X_{129}, \ldots, X_{255}).$$

This shows that the 255-block BRW computation can be broken down into 2 127-block BRW computations. Continuing, we break up the 255-block BRW computation into 8 31-block BRW computations. A completely loop unrolled 31-block BRW computation can be implemented using 15 polynomial multiplications and 8 reductions [30] and has been discussed in Chapter 5. We use this implementation of 31-block BRW computation to build the 255-block BRW computation. This strategy requires 127 polynomial multiplications and 71 reductions. Following the delayed reduction strategy for BRW computation [30] discussed in Chapter 5, it is possible to have a completely loop unrolled 255-block BRW computation requiring 127

polynomial multiplications and 64 reductions. The code for such a loop unrolled implementation would be quite complex and could lead to a substantial performance penalty. This is the reason why we have chosen to build the 255-block BRW computation from the (loop unrolled) implementation of the 31-block BRW computation.

vecHash2L: The function vecHash2L is parameterised by two quantities, namely the block size n and the super-block size η. The use of AES fixes n to be 128. We have taken $\eta = 31$. This requires the implementations of 31-block BRW and also i-block BRW for $i = 1, \ldots, 30$ to tackle the last super-block which can possibly have less than 31 blocks. As mentioned earlier, an implementation of Hash2L for $n = 128$ and $\eta = 31$ was reported in [30], but, the implementation of vecHash2L was not considered in that work. The computation of vecHash2L can be conceptually seen as 31-block BRW computations whose outputs are combined using Horner. Additionally, after each component, the length block is processed. As discussed above, the computation of Horner can be improved by using the delayed reduction strategy and the decimation technique. We have experimented with various combinations and later we report the results for 3-decimated implementations with and without the delayed reduction strategy.

7.6.2 Implementation of FAST

We have described several variants of FAST. Software implementations of these variants are built from the implementation of AES and the implementations of the various hash functions. The AES based parts consist of the Ctr mode and the Feistel layer while the hash functions are built from either Horner or BRW in case of the fixed length setting and are built from either vecHorner or vecHash2L in the general setting. We describe these aspects below.

Key schedule generation: All versions of FAST use a single key K which is the key to the underlying PRF F_K. Instantiating F_K with the encryption function of AES requires generating the round keys. This is a one-time activity and is done using the instruction `aeskeygenassist`. The generated round keys are stored and used in both the Ctr mode and the Feistel layer.

Ctr: The Ctr mode defined in (3.3) requires a PRF F which is implemented using the encryption function of AES. Each invocation of the encryption function can be implemented using `aesenc` followed by `aesenclast`. The invocations of `aesenc` can be speeded up using an interleaving of multiple AES invocations. The AES encryption calls in the Ctr mode are fully parallelisable. Let $i \geq 1$ be a positive integer. The computations of the AES calls in the Ctr mode are done in batches of i calls each. The inputs to one batch of i encryptions are prepared; then the first rounds of AES encryptions of this batch of i encryptions are computed using i calls to `aesenc`; this is followed by the second rounds of this batch of i encryptions again using `aesenc` and so on. This ensures that the second round of any AES encryption does not have to wait to obtain the output of the first round. This interleaved strategy leads to substantial speed-up over computing the complete AES encryptions one

after another. In our implementation, we have used $i = 8$ which follows the earlier work by Gueron[6].

Feistel: The Feistel layer has two calls to AES encryptions. These calls are not parallelisable. So, these calls are implemented using a sequence of `aesenc` followed by a single call to `aesenclast` to perform the computation of a single AES encryption.

Hash key generation: The hash key τ is obtained by applying F_K to fStr. This is a one-time operation and the value of τ does not change during the life-time of K. So, it is possible to generate τ once and store it securely along with K. More generally, it is also possible to use a uniform random τ as the hash key instead of generating it by applying F_K to fStr. This will not affect the security analysis, but, will increase the key storage requirement. Alternatively, it is possible to generate τ once per session. The cost of generating τ is amortised over all the encryptions/decryptions per session and hence is negligible. Timing results provided later include the time for generating τ.

FAST[$\mathsf{Fx}_{\mathfrak{m}}$, Horner]: In the setting of $\mathsf{Fx}_{\mathfrak{m}}$, tweaks consist of a single n-bit block while plaintexts and ciphertexts consist of $\mathfrak{m}$ n-bit blocks. In our implementation, we have taken $\mathfrak{m} = 256$ so that the total number of bytes in a plaintext/ciphertext is 4096. As mentioned earlier, this corresponds to the size of a modern disk sector. In this case, $P_3 = (P_{3,1}, \ldots, P_{3,\mathfrak{m}-2})$ consists of $(\mathfrak{m} - 2)$ n-bit blocks and the hash function $\mathsf{Horner}_\tau(1, P_{3,1}, \ldots, P_{3,\mathfrak{m}-2}, T)$ needs to be computed. An implementation of Horner as mentioned above is used. This requires a total of 255 polynomial multiplications and a total of 32 reductions. Counting a single polynomial multiplication as 4 `pclmulqdq` and a reduction as 2 `pclumulqdq`, the total number of `pclmulqdq` calls required is 1084.

FAST[$\mathsf{Fx}_{\mathfrak{m}}$, BRW]: As above, we take $\mathfrak{m} = 256$. The requirement is to compute

$$\mathsf{BRW}_\tau(P_{3,1}, \ldots, P_{3,\mathfrak{m}-2}, T).$$

This is done as described above which requires 127 polynomial multiplications and 71 reductions. The total number of `pclmulqdq` calls required is 650. This is 434 calls lesser than that required for computing $\mathsf{Horner}_\tau(1, P_{3,1}, \ldots, P_{3,\mathfrak{m}-2}, T)$. So, one would expect FAST[$\mathsf{Fx}_{\mathfrak{m}}$, BRW] to be faster than FAST[$\mathsf{Fx}_{\mathfrak{m}}$, Horner]. Our implementation shows a speed-up, but, not as much as one might expect from the counts of the `pclmulqdq` calls. Indeed instruction cache and pipelining are rather complicated issues and precise information about these issues for Intel processors are not easily available[7]. So, it is possible that the code for BRW that we have developed can be tuned further to obtain speed improvements.

FAST[Gn, $\mathfrak{k}$, vecHorner]: This requires the implementation of the hash function vecHorner which is an easy extension of the implementation of Horner.

[6] Interleaving of 8 AES encryptions has been called a sweat point in `https://crypto.stanford.edu/RealWorldCrypto/slides/gueron.pdf`

[7] `https://blog.cr.yp.to/20140517-insns.html`

$\mathsf{FAST}[\mathsf{Gn}, \mathbb{t}, \eta, \mathsf{vecHash2L}]$**:** This requires the implementation of the hash function vecHash2L. The implementation of this hash function has been discussed above.

7.6.3 Timing Results

In this section, we provide timing results for the software implementations of all the four variants of FAST, i.e. for both the settings of Fx and Gn. As mentioned earlier the corresponding code is publicly available from

`https://github.com/sebatighosh/FAST`.

In the setting of Fx, messages are 4096 bytes long, i.e., each message consists of 256 128-bit blocks and the tweak is a single 128-bit block. The timing results are shown in Table 7.9 along with the timing results for XCB, EME2 and AEZ.

In the setting of Gn, timing measurements are separately reported for messages of lengths 512, 1024, 4096 and 8192 bytes. For tweaks, the number of components has been considered to be 2, 3 and 4 and the sum of the lengths of the components of the tweaks has been taken to be 1024 bytes: For tweaks with 2 components, each component has length 512 bytes; for tweaks with 3 components, the 3 components have lengths 336, 336 and 352 bytes; whereas for tweaks with 4 components, each component has length 256 bytes. Two columns of measurements are shown for $\mathsf{FAST}[\mathsf{Gn}, \mathbb{t}, \eta, \mathsf{vecHash2L}]$. The column with the heading 'delayed' reports measurements for the case where the Horner layer in vecHash2L has been implemented using the delayed reduction strategy while the column with the heading 'normal' reports measurements for the case where the Horner layer in vecHash2L has been implemented without using the delayed reduction strategy. The timing results are shown in Table 7.10.

The timing measurements were taken on two platforms.

- **Skylake:** The processor was Intel Core i7-6500U @ 2.5GHz. The operating system was 64-bit Ubuntu 14.04 LTS and the C codes were complied using GCC version 5.5.0.
- **Kabylake:** The processor was Intel Core i7-7700 @ 3.6GHz. The operating system was 64-bit Ubuntu 18.04 LTS and the C codes were complied using GCC version 7.3.0.

For the setting of Fx, we have carried out efficient implementations of XCB, EME2 and AEZ. In the implementation of AEZ, for the underlying block cipher, the full AES has been used unlike the reduced round versions considered earlier [67]. XCB uses hashing based on Horner's rule and we have used the same delayed reduction strategy in the implementation of this hashing as we did in the implementation of the hash function for $\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{Horner}]$. The efficient software implementations of XCB, EME2 and AEZ are of independent interest. We provide a brief description of the software implementation of AEZ in Section 7.7. Timing results from the setting of Fx show that all three of XCB, EME2 and AEZ are slower than FAST. Consequently, we do not compare FAST to XCB, EME2 and AEZ in the setting of Gn.

Based on Tables 7.9 and 7.10, we make the following observations.

Table 7.9: Comparison of the cycles per byte measure of FAST with those of XCB, EME2 and AEZ in the setting of Fx_{256}.

scheme	Skylake	Kabylake
XCB	1.92	1.85
EME2	2.07	1.99
AEZ	1.74	1.70
$\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{Horner}]$	1.63	1.56
$\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{BRW}]$	1.24	1.19

1. In the setting of Fx, $\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{BRW}]$ is faster than $\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{Horner}]$. Moreover, $\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{Horner}]$ and $\mathsf{FAST}[\mathsf{Fx}_{256}, \mathsf{BRW}]$ are both faster than all three of XCB, EME2 and AEZ.

2. In the setting of Gn for vecHorner, the speed decreases with increase in message length while for vecHash2L the speed increases with increase in message length. In both cases, for the same message length, the speed mostly does not vary much with increase in the number of components in the tweak. In the case of vecHash2L, using the delayed reduction strategy for implementing the Horner layer results in improved speed than an implementation without using delayed reduction. Overall, on Kabylake $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, 31, \mathsf{vecHash2L}]$ is faster than $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ while on Skylake $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, 31, \mathsf{vecHash2L}]$ is faster than $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ for longer messages.

Remark: From Table 7.9, it may be noted that AEZ is faster than EME2. From Table 7.7, it can be seen that the number of block cipher calls made by AEZ is more than that made by EME2. So, the fact that in practice AEZ turns out to be faster may be surprising. The explanation lies in the difference in the number of doubling operations made by EME2 and AEZ. From Table 7.7, EME2 requires roughly 2[BC]+2[D] operations per block whereas AEZ requires roughly 2.5[BC]+0.25[D] operations per block. Executing AES instructions in groups using pipelining results in very fast AES timings. Our experiment on the Skylake processor shows that AES requires about 0.65 cycles per byte. In contrast, while the doubling operation should in theory be much faster, there is no support for 128-bit shift and consequently doubling takes about 0.29 cycles per byte. (We refer to [38] for an elaborate discussion on various strategies for constant time doubling operation.) Using these figures, the operations count of 2[BC]+2[D] for EME2 translates to about 1.88 cycles per byte while the operations count of 2.5[BC]+0.25[D] for AEZ translates to about 1.70 cycles per byte. This provides an explanation of why AEZ is faster than EME2. Note that EME2 requires additional block cipher calls and so the actual observed time of 2.07 cycles per byte for EME2 is a bit higher than the estimated 1.88 cycles per byte whereas the observed and the estimated timings for AEZ are quite close.

Table 7.10: Report of cycles per byte measure for the setting of Gn for $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, \mathsf{vecHorner}]$ and $\mathsf{FAST}[\mathsf{Gn}, \mathfrak{k}, 31, \mathsf{vecHash2L}]$.

		Skylake			Kabylake		
msg len (bytes)	$\mathfrak{k}$	vecHorner	vecHash2L (delayed)	vecHash2L (normal)	vecHorner	vecHash2L (delayed)	vecHash2L (normal)
512	2	1.51	1.38	1.59	1.42	1.32	1.56
	3	1.40	1.38	1.39	1.32	1.31	1.35
	4	1.34	1.37	1.36	1.26	1.31	1.33
1024	2	1.53	1.34	1.48	1.42	1.27	1.42
	3	1.45	1.34	1.34	1.35	1.27	1.30
	4	1.40	1.33	1.32	1.29	1.27	1.30
4096	2	1.57	1.30	1.35	1.45	1.24	1.30
	3	1.54	1.29	1.31	1.43	1.24	1.27
	4	1.51	1.29	1.30	1.40	1.24	1.26
8192	2	1.57	1.27	1.32	1.45	1.22	1.27
	3	1.56	1.27	1.30	1.44	1.22	1.25
	4	1.54	1.27	1.30	1.43	1.22	1.25

7.7 Additional Material on Implementation of AEZ

For $\alpha \in \{0,1\}^{128}$ and $i \in \mathbb{N}$, the following operation has been defined [67] in the context of AEZ:

$$i \cdot \alpha \;=\; \begin{cases} \mathbf{0} & \text{if} \quad i = 0; \\ \alpha & \text{if} \quad i = 1; \\ (\alpha \ll 1) \oplus (\mathsf{msb}(\alpha) \cdot 135) & \text{if} \quad i = 2; \\ 2 \cdot (j \cdot \alpha) & \text{if} \quad i = 2j > 2; \\ (2j \cdot \alpha) \oplus \alpha & \text{if} \quad i = 2j+1 > 2. \end{cases} \tag{7.50}$$

The operation in (7.50) corresponding to $i = 2$ is the doubling operation. (See Section 8.1).

There are different versions of AEZ [67] built from different variants of AES. The version which is relevant here is the one where the proper AES algorithm is used. Messages of lengths at least $2n$ bits are handled differently from messages of lengths less than $2n$ bits. For our purpose, we will be considering the portion which can handle messages of lengths at least $2n$ bits. This portion has been called AEZ-Core [67]. By AEZ, we will denote AEZ-Core[AES].

The length of the message is written as $2nk + \mu$ bits with $0 \leq \mu < 2n$ and $k \geq 1$. Below we provide an overview of the encryption algorithm of AEZ where $\mu = 0$. Let $\mathfrak{m} = 2\mathfrak{s} + 2$ and consider a message having $\mathfrak{m}$ blocks with total length $n(2\mathfrak{s} + 2)$ bits. The message is partitioned into two parts: The first part consists of $2\mathfrak{s}n$ bits organised as $2\mathfrak{s}$ n-bit blocks $M_1, M'_1, \ldots, M_{\mathfrak{s}}, M'_{\mathfrak{s}}$ and the second part consists of $2n$ bits organised as 2 n-bit blocks M_{x} and M_{y}. The ciphertext blocks are C_i, C'_i, $i = 1, \ldots, \mathfrak{s}$ and $C_{\mathsf{x}}, C_{\mathsf{y}}$.

At a conceptual level, this encryption consists of three layers. The first and the third layers consist of a sequence of 2-round Feistel networks where each Feistel network requires 2 block cipher calls. The second layer is a mixing layer and requires one block cipher call for each i. Let E denote the encryption function of AES and for $\alpha \in \{0,1\}^n$, define $\widetilde{E}_K^{i,j}(\alpha) = E_K(\alpha \oplus (i+1) \cdot I \oplus j \cdot J)$ where $I = E_K(\mathbf{0})$ and $J = E_K(\mathbf{1})$ [67]. The encryption proceeds as follows:

- *First layer:* for $i = 1, \ldots, \mathfrak{s}$, $W_i = M_i \oplus \widetilde{E}_K^{1,i}(M'_i)$; $X_i = M'_i \oplus \widetilde{E}_K^{0,0}(W_i)$; $S_{\mathsf{x}} = \widetilde{E}_K^{0,1}(M_{\mathsf{y}}) \oplus M_{\mathsf{x}} \oplus X \oplus \Delta$; $S_{\mathsf{y}} = \widetilde{E}_K^{-1,1}(S_{\mathsf{x}}) \oplus M_{\mathsf{y}}$;
- *Second layer:* for $i = 1, \ldots, \mathfrak{s}$, $S'_i = \widetilde{E}_K^{2,i}(S)$; $Y_i = S'_i \oplus W_i$; $Z_i = S'_i \oplus X_i$;
- *Third layer:* for $i = 1, \ldots, \mathfrak{s}$, $C'_i = Y_i \oplus \widetilde{E}_K^{0,0}(Z_i)$; $C_i = Z_i \oplus \widetilde{E}_K^{1,i}(C'_i)$; $C_{\mathsf{y}} = S_{\mathsf{x}} \oplus \widetilde{E}_K^{-1,2}(S_{\mathsf{y}})$; $C_{\mathsf{x}} = S_{\mathsf{y}} \oplus \widetilde{E}_K^{0,2}(C_{\mathsf{y}}) \oplus \Delta \oplus Y$;

Here $X = X_1 \oplus \cdots \oplus X_{\mathfrak{s}}$, $Y = Y_1 \oplus \cdots \oplus Y_{\mathfrak{s}}$, $S = S_{\mathsf{x}} \oplus S_{\mathsf{y}}$ and Δ is obtained by processing the tweak.

Remarks:

1. For implementation, we have considered $n = 128$ and $\mathfrak{m} = 256$, i.e., messages of lengths equal to 4096 bytes. So, writing $\mathfrak{m} = 2\mathfrak{s} + 2$ we have $\mathfrak{s} = 127$.
2. In our implementations, we have taken $\Delta = \mathbf{0}$, i.e., we have ignored the processing of the tweak. Since the resulting implementations of AEZ turn out to be less efficient than FAST, considering the processing of the tweak for AEZ will result in further slowdown compared to FAST.

7.7.1 Software Implementation

The design of AEZ is based on OTR [86] and the parallelism in AEZ is the same as that in OTR. The encryptions in the first layer can be divided into two classes – one class consisting of the encryptions of $W_1, W_2, \ldots$ and the other class consisting of the encryptions of $M'_1, M'_2, \ldots$. Following the pipelining strategy for AES described in Section 7.6.2, the encryptions in each class have to be bunched into groups of eight. The encryptions proceed as follows. One bunch of M'_i's is encrypted followed by the corresponding bunch of W_i's; then the next bunch of M'_i's is encrypted followed by the corresponding bunch of W_i's and so on. After all the encryptions in the first layer are over, the encryptions in the second layer are to be computed. These encryptions are independent and can be executed in groups of eight. The strategy for executing the encryptions in the third layer is similar to that of the first layer. Though somewhat complicated, the above mentioned strategy can be used to obtain the benefits of pipelined AES execution.

One important efficiency issue is that of computing the values $j \cdot J$. As briefly mentioned in the work introducing AEZ [67], by storing some of the previously generated values, it is possible to efficiently generate the required values. We provide some details. A queue of

values $2 \cdot J, 3 \cdot J, \ldots$ is maintained. When $j \cdot J$ is computed, it is added to the tail of the queue. If the head of the queue contains $\jmath \cdot J$, then this value is used to generate $2\jmath \cdot J$ and $(2\jmath+1) \cdot J$ and then the entry $\jmath \cdot J$ is deleted from the queue. Using this strategy, the computation of $2\jmath \cdot J$ from $\jmath \cdot J$ requires one doubling and the computation of $(2\jmath + 1) \cdot J$ from $\jmath \cdot J$ requires one XOR. Overall, the computations of $2\jmath \cdot J$ and $(2\jmath + 1) \cdot J$ require one doubling and one XOR. For the $2\mathfrak{s}$ blocks in the first part, the operation (7.50) will be required to be applied $\mathfrak{s}$ times. This will require $\lfloor \mathfrak{s}/2 \rfloor$ doubling operations and $\lfloor (\mathfrak{s} - 1)/2 \rfloor$ XOR operations. Since $\mathfrak{m} = 2\mathfrak{s} + 2$, the computations of all the masks require $\lfloor (\mathfrak{m} - 2)/4 \rfloor$ doubling operations.

7.8 Summary

In this chapter, we have proposed a new family of TESs called FAST. Several instantiations of FAST have been presented both for the popular application of TESs in disk encryption and for a wide variety of more general applications. Such applications of TESs are natural, but new in the literature. FAST has the attractive features of using a single field element as the key, being parallelisable and not requiring the inverse function of the underlying block cipher. One major contribution here is detailed and careful software implementation of all the instantiations of FAST for modern Intel processors. Comparative timing measurements show that FAST compares very favourably to the IEEE disk encryption standards XCB and EME2 as well as the more recent proposal AEZ. Depending on this, FAST indeed becomes a deserving candidate for standardisation and deployment.

Chapter 8

Breaking Tweakable Enciphering Schemes using Simon's Algorithm

The threat of the possible advent of quantum computers in large scale has motivated the entire cryptography community to search for quantum safe primitives. Adequate cryptanalysis of the current primitives is essential to guess the impact of quantum algorithms on the classical cryptography. Several works have already been carried out in this direction in the area of symmetric key ciphers.

Among them, a series of works [75, 76, 68, 23, 42, 24] have already identified the impact of Simon's period finding quantum algorithm [104] on certain modes of operation. In this chapter, we continue this line of work on using Simon's algorithm to the cryptanalysis of several tweakable enciphering schemes (TESs) [65]. TESs provide several important cryptographic functionalities including that of full disk encryption. We refer to [65] for a description of how a TES can be used for disk encryption and to [29] for more general functionalities. Some TESs have also been standardised [3].

Here, we consider five TESs, namely, CMC [65], EME [66, 62], TET [63], XCB [82, 83] and FAST [29]. CMC was the first TES to be proposed; IEEE has standardised [3] XCB and EME; TET uses invertible universal hash; and presently FAST provides the most recent development.

Following Kaplan et al. [68], the attacks that we describe are essentially based on an algorithm to solve the following problem.

Simon's problem: Given a function $f : \{0,1\}^{\mathtt{m}} \rightarrow \{0,1\}^n$ and the promise that there exists $s \in \{0,1\}^{\mathtt{m}} \setminus 0^{\mathtt{m}}$ such that for all $x \neq y$, $f(x) = f(y)$ if and only if $x \oplus y = s$, find s.

The quantity s is called the period of the function. Simon [104] described a quantum algorithm which, with high probability, finds the period of f with $O(\mathtt{m})$ quantum queries to the function f and additional polynomial time classical computation. The description in [104] required f to be a 2-to-1 function which was later modified to a looser condition in [68].

For each of the TES that we consider, we construct a function f based on the encryption algorithm of the TES. The function f has a period which is based on variables that are used during the computation, but is not revealed as part of the ciphertext. Applying Simon's algorithm to f uncovers the period and reveals the internal secret variable. In the cases of TET, XCB and FAST, obtaining the period of f reveals a portion of the secret key of the TES resulting in a partial key recovery attack. For all the five TESs, we show that using the period, it is possible to construct two distinct plaintexts such that designated portions of the corresponding ciphertexts are equal. This results in distinguishing attacks on all the TESs under consideration.

This chapter is based on the work [50].

8.1 Preliminaries

Throughout this chapter, n is fixed a positive integer. A block cipher is a function $E : \mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$, where $\mathcal{K}$ is a finite non-empty set and for each $K \in \mathcal{K}$, $E_K(\cdot) \stackrel{\Delta}{=} E(K, \cdot)$ is a permutation of $\{0,1\}^n$. The integer n denotes the block size and K is the key of the block cipher. The corresponding decryption function is $D : \mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$, where for each $K \in \mathcal{K}$, $D_K(\cdot) \stackrel{\Delta}{=} D(K, \cdot)$ is the inverse of $E_K(\cdot)$, i.e., for any $x \in \{0,1\}^n$, $D_K(E_K(x)) = x$.

8.1.1 Tweakable Enciphering Scheme

The notion of security of tweakable enciphering scheme that we consider is that of indistinguishability from a random oracle which returns independent and uniform random strings of appropriate lengths. This implies other notions of security (see [65]). We do not repeat the formal definition of security of TES since this will not be required here.

Four of the five TESs that we consider, namely, XCB, TET, CMC and EME are built using n-bit block ciphers. The security proofs of the TESs assume the underlying block cipher to be strong pseudo-random permutation. The other TES that we consider, namely FAST, is built using a n-bit to n-bit pseudo-random function. Assuming the underlying primitive (block cipher or pseudo-random function) to be secure, the security proofs of all the five TESs provide an upper bound on the advantage of an adversary in distinguishing the TES from a random oracle. The upper bound is essentially of the form $c\sigma_n^2/2^n$, where c is a small constant and σ_n is the number of n-bit blocks in all the queries made by the adversary. Ignoring the constant c, at a broad level the proofs show that the TESs are secure up to about $2^{n/2}$ adversarially chosen n-bit blocks.

8.1.2 Simon's Algorithm with Spurious Collisions

Simon's problem is a promise problem, i.e., the function f has to satisfy the stated condition for Simon's algorithm to work. There may be functions for which there is an $s \in \{0,1\}^{\mathtt{m}} \setminus 0^{\mathtt{m}}$, such that for all $(x, y) \in \{0,1\}^{\mathtt{m}} \times \{0,1\}^{\mathtt{m}}, x \oplus y \in \{0^{\mathtt{m}}, s\} \Rightarrow f(x) = f(y)$, but $f(x) = f(y)$ does not necessarily imply $x \oplus y \in \{s, 0^{\mathtt{m}}\}$, i.e., there could be a $\mathtt{t}$ different from s and $0^{\mathtt{m}}$, such that for some x, $f(x) = f(x \oplus \mathtt{t})$. Such a collision is called a spurious collision. This issue was considered in [68], which defined the notion of approximate promise problem. For $f : \{0,1\}^{\mathtt{m}} \rightarrow \{0,1\}^n$ such that $f(x \oplus s) = f(x)$ for all x, the following quantity was defined in [68].

$$\varepsilon(f, s) \quad = \quad \mathsf{max}_{\mathtt{t} \in \{0,1\}^{\mathtt{m}} \setminus \{0,s\}} \Pr_x[f(x) = f(x \oplus \mathtt{t})]. \tag{8.1}$$

Note that here the probability is taken over the random choice of x. If f satisfies the promise in Simon's problem and has period s, then $\varepsilon(f, s) = 0$. We say that a function $f : \{0,1\}^{\mathtt{m}} \rightarrow \{0,1\}^n$ satisfies the promise in Simon's problem approximately, if there is an s

such that $f(x) = f(x \oplus s)$ for all x and $0 < \varepsilon(f, s) < 1$. A modification of Simon's algorithm to solve the approximate promise problem has been considered in [68] where the following result was proved.

Theorem 11 (Kaplan et al. [68]). *If $\varepsilon(f, s) \leq p_0 < 1$, then for any constant c, Simon's algorithm returns s with $c\mathfrak{m}$ quantum queries, with probability at least $1 - (2(\frac{1+p_0}{2})^c)^{\mathfrak{m}}$.*

If f satisfies the approximate promise problem, then Theorem 11 shows that s can be recovered with high probability.

Remarks:

1. A function satisfying the promise in Simon's problem is a 2-to-1 function. Simon had considered a slightly different problem. Given a function $f : \{0,1\}^{\mathfrak{m}} \rightarrow \{0,1\}^n$ which is known to be either injective or 2-to-1, determine the correct condition and if f is 2-to-1, then determine its period.

2. In his formulation, Simon required $n \geq \mathfrak{m}$. The analysis of Simon's algorithm, on the other hand, goes through without the condition $n \geq \mathfrak{m}$ and later works [68, 23, 43] have indeed also considered $n < \mathfrak{m}$.

8.2 Outline of the Attacks

The attacks that we describe are based on Simon's algorithm and are distinguishing attacks. For three of the TESs, namely XCB, TET and FAST, the attacks also reveal part of the secret key.

Suppose the adversary is provided black-box access to Π, where Π is either the encryption algorithm of a TES (which we write as Π is real) or Π is a random oracle (which we write as Π is random). The goal of the adversary is to determine whether Π is real or random. Using Π we define a function f. If Π is random, then f is a random function. On the other hand, if Π is real, then either f satisfies the promise in Simon's problem, or the approximate promise mentioned in Section 8.1.2. Simon's algorithm is applied to f which requires making quantum queries to the given black-box. This is a strong attack model and has been adopted in previous works [68, 23, 43].

If Π is random, then the output of Simon's algorithm on f will be a random string. On the other hand, if Π is real, then with high probability Simon's algorithm will return the period of f. So, given the output of Simon's algorithm, some further work is required to determine whether Π is real or random. This work consists of making two classical queries to the black box. These queries are built from the output of Simon's algorithm. If Π is real, then we show that the outputs of the two classical queries satisfy a pre-defined relation, while if Π is random, then the outputs of the two classical queries satisfy the same relation with very low probability. So, looking at the outputs of the two classical queries, it becomes possible to determine whether Π is real or random.

The above provides the broad outline of the attacks on the TESs. In the subsequent sections, we do not repeat the above strategy. Instead, we provide the definition of f when Π

is real, the two classical queries and the pre-defined relation that their outputs satisfy when Π is real. Plugging these two tools into the above attack strategy provides the complete attacks on the individual TESs.

As mentioned above, for some of the TESs, we show that f satisfies an approximate promise. The proof of approximate promise requires upper bounding the probability of spurious collisions. Since the definition of f is based on the block cipher, to bound the probability of spurious collisions of f, we need to make an assumption on the underlying block cipher. The assumption that we make is to consider the block cipher to behave like a uniform random function. Since a block cipher is an injective map, it would be appropriate to assume the block cipher to behave like a uniform random permutation. If the number of inputs on which the block cipher is invoked is below the (quantum) birthday bound, then it is reasonable to consider the block cipher to behave like a uniform random function. In our applications, we will consider the application of the block cipher to only a few (at most six) inputs.

The analyses of the probabilities of spurious collisions for the various TESs have a common structure. Suppose s is the period of f. We start by considering a non-zero $\mathsf{t} \neq s$ which maximises the probability of spurious collisions. The requirement is to bound the probability $f(x) = f(x \oplus \mathsf{t})$. Then for any event $\mathcal{E}$, we have

$$\Pr[f(x) = f(x \oplus \mathsf{t})] \leq \Pr[f(x) = f(x \oplus \mathsf{t}) | \mathcal{E}] + \Pr[\overline{\mathcal{E}}]. \tag{8.2}$$

In the analyses of the individual TESs, we identify a suitable event $\mathcal{E}$ and obtain upper bounds on the two terms in the right hand side of (8.2).

In Section 8.3, we describe the attacks on XCB, TET and FAST. The attacks on XCB, TET and FAST require the key of the underlying universal hash function to be non-zero. Since the hash key is a random n-bit quantity, it is zero with probability $1/2^n$ which is negligible for $n = 128$ or larger. These attacks also recover the hash key. In Section 8.4, we describe the distinguishing attacks on CMC and EME. For EME, we require the internal variable L to be non-zero. Since it is the output of a block cipher instantiated with a random key, it is zero with probability $1/2^n$ which is negligible for $n = 128$ or larger.

Offline Simon's Algorithm

The attacks that we describe require quantum access to the encryption algorithms of the respective TESs. A recent work [24] has shown that for some symmetric key algorithms, it is possible to do away with the requirement of quantum access to the encryption algorithms. The quantum computations are done in an offline manner while all the queries to the encryption algorithms are classical. In particular, Simon's algorithm is applied in an offline mode. Such attacks are more practical than attacks which require quantum access to the encryption algorithms.

The core observation in [24] is that it is possible to determine whether a function of the form $f_1 \oplus f_2$ has a period without any quantum query to f_2 if there is a suitable quantum state corresponding to f_2. Various examples of the idea are provided in [24]. For the TESs that we have considered, we tried to apply the idea from [24] to obtain attacks which do not

require quantum access to the encryption algorithms. Our efforts were not successful. It was not clear how to modify the functions with period that we constructed for the various TESs to the form $f_1 \oplus f_2$ which seems to be required to apply the technique of [24]. Our inability does not mean that offline Simon's algorithm is not applicable to these TESs. There could be other ways of constructing the functions in the desired form. This though seems to require more work.

8.3 Partial Key Recovery Attacks

8.3.1 XCB

XCB was proposed by McGrew and Fluhrer [82]. A later variant [83] was standardised by IEEE [3]. We describe the quantum attack on the standardised version [83] of XCB. A similar attack also works on the previous version.

XCB is built using a block cipher and a polynomial hash function. The key K of XCB is the same as the key of the underlying block cipher. XCB defines a tweak space. In our attack, we will fix the tweak to be the empty string $\mathbf{e}$.

XCB can be used with an n-bit block cipher. For the sake of convenience, we fix $n = 128$. Let E_K denote the encryption function of the underlying block cipher instantiated with the key K. Using E_K, XCB derives the keys K_e, K_d, K_c and τ. Here τ is used as the key to a polynomial hash function called GHASH, K_c is the key to the counter mode of encryption and K_e and K_d are used as shown in Figure 8.1. The counter mode Ctr uses the function incr to obtain successive values to be encrypted.

Our attack considers 4-block messages. So, we briefly describe the encryption of 4-block messages with reference to Figure 8.1. The message is partitioned into a single block and a 3-block message. As per the specification of XCB, the quantity A is equal to the last block of the message and the quantity V is the last block of the ciphertext. In more details, if $y_1||y_2||y_3||y_4$ is the 4-block ciphertext corresponding to a 4-block message, then $V = y_4$ and $R = y_1||y_2||y_3$. The functions h_1 and h_2 in Figure 8.1 are polynomial hash functions using the key τ. The counter mode Ctr uses Q as the initialisation vector. The rest of the encryption algorithm can be understood from Figure 8.1. We provide more details as part of the attack.

Fix $m, \chi_0, \chi_1 \in \{0,1\}^n$, such that $\chi_0 \neq \chi_1$; let b denote a bit. For the standardised version [83], we define the following function.

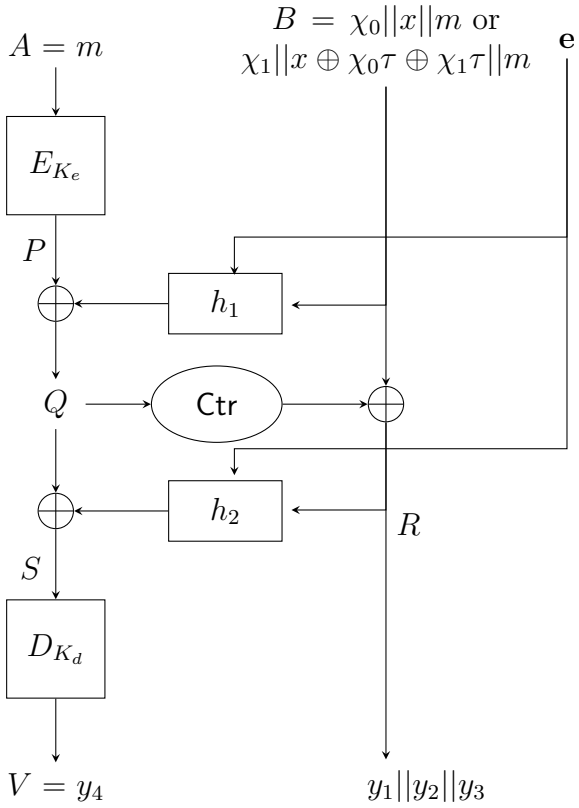
$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\rightarrow \{0,1\}^n \\ (b, x) &\overset{f}{\mapsto} y_3, \text{ where } y_1||y_2||y_3||y_4 \longleftarrow \mathsf{XCB.Encrypt}_K(\mathbf{e}, \chi_b||x||m||m). \end{aligned} \tag{8.3}$$

The function f defined in (8.3) satisfies the following property.

Proposition 12. *Let $b, b' \in \{0,1\}, x, x' \in \{0,1\}^n$. Suppose that the hash key τ is non-zero. Then, $f(b,x) = f(b',x') \Leftrightarrow x \oplus x' = \chi_b\tau \oplus \chi_{b'}\tau$, where χ_0 and χ_1 are as fixed before.*

Proof. Let γ be a 128-bit string which is formed by concatenating the 64-bit binary repre-

Figure 8.1: Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus \chi_0\tau \oplus \chi_1\tau||m||m$ with tweak $\mathbf{e}$ under XCB.



sentation of 128 and the 64-bit binary representation of 512. For the input $\chi_b||x||m||m$,

$$\begin{aligned} A &= m; \\ B &= \chi_b||x||m; \\ P &= E_{K_e}(m); \\ Q &= E_{K_e}(m) \oplus \chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus \gamma\tau; \\ R &= \chi_b \oplus E_{K_c}(Q)||x \oplus E_{K_c}(\mathsf{incr}(Q))||m \oplus E_{K_c}(\mathsf{incr}(\mathsf{incr}(Q))); \end{aligned}$$

For the input $\chi_{b'}||x \oplus \chi_b\tau \oplus \chi_{b'}\tau||m||m$,

$$\begin{aligned} A' &= m; \\ B' &= \chi_{b'}||x \oplus \chi_b\tau \oplus \chi_{b'}\tau||m; \\ P' &= E_{K_e}(m); \\ Q' &= E_{K_e}(m) \oplus \chi_{b'}\tau^4 \oplus x\tau^3 \oplus \chi_b\tau^4 \oplus \chi_{b'}\tau^4 \oplus m\tau^2 \oplus \gamma\tau \\ &= E_{K_e}(m) \oplus x\tau^3 \oplus \chi_b\tau^4 \oplus m\tau^2 \oplus \gamma\tau; \\ R' &= \chi_{b'} \oplus E_{K_c}(Q')||x \oplus \chi_b\tau \oplus \chi_{b'}\tau \oplus E_{K_c}(\mathsf{incr}(Q'))||m \oplus E_{K_c}(\mathsf{incr}(\mathsf{incr}(Q'))); \end{aligned}$$

We observe, $Q = Q'$ results in equality of last blocks of R and R'. So, the third blocks of the outputs are same, establishing one direction of the result.

For the other direction, we have

$$\begin{aligned} y_3 = y_3' &\Rightarrow m \oplus E_{K_c}(\mathsf{incr}(\mathsf{incr}(Q))) = m \oplus E_{K_c}(\mathsf{incr}(\mathsf{incr}(Q'))) \\ &\Rightarrow Q = Q' \\ &\Rightarrow E_{K_e}(m) \oplus \chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus \gamma\tau = E_{K_e}(m) \oplus \chi_{b'}\tau^4 \oplus x'\tau^3 \oplus m\tau^2 \oplus \gamma\tau \\ &\Rightarrow \chi_b\tau^4 \oplus x\tau^3 = \chi_{b'}\tau^4 \oplus x'\tau^3 \\ &\Rightarrow x \oplus x' = \chi_b\tau \oplus \chi_{b'}\tau. \end{aligned}$$

□

Classical queries: Given the period $1||s = 1||\tau(\chi_0 \oplus \chi_1)$, the two classical queries required in Section 8.2 are the following. The first query is $\chi_0||x||m||m$ with output $y_1||y_2||y_3||y_4$ and the second query is $\chi_1||x \oplus s||m||m$ with output $y_1'||y_2'||y_3'||y_4'$. From the proof of Proposition 12 we have that $y_3 = y_3'$ which defines the relation between the outputs of the two classical queries.

Partial key recovery: Once $s = \tau(\chi_0 \oplus \chi_1)$ has been obtained, since χ_0 and χ_1 are distinct, from s, one obtains the hash key τ as $\tau = s(\chi_0 \oplus \chi_1)^{-1}$.

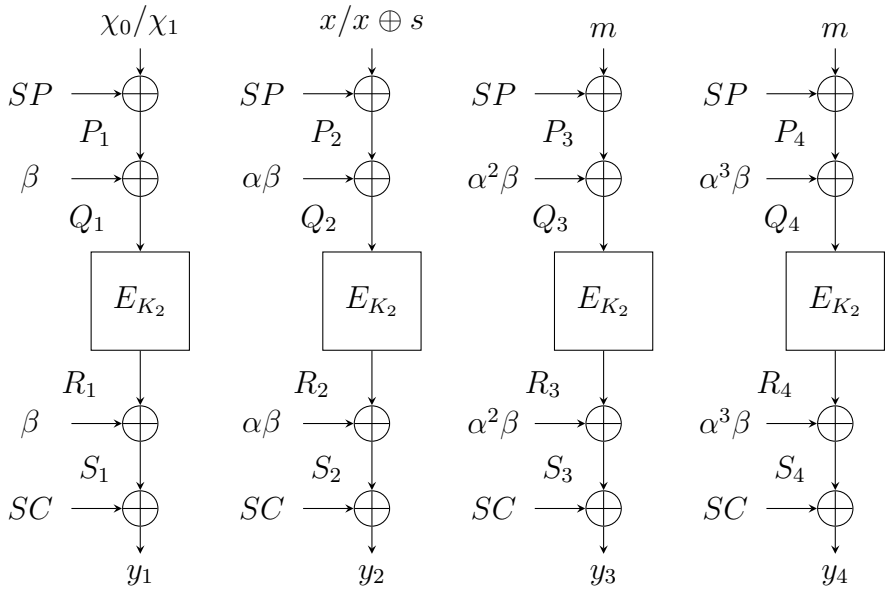
8.3.2 TET

TET [63] has key space $\mathcal{K} \times \mathcal{K}$, where $\mathcal{K}$ is the key space for the underlying block cipher having block size n. The tweak space is $\mathcal{T} = \{0,1\}^*$. The message space is $\mathcal{P} = \{0,1\}^{\mathtt{m}}$

where $\mathfrak{m} \in [n, 2^n - 1]$. Fix arbitrary (K_1, K_2) from the key-space and arbitrary T from the tweak-space.

The attack against TET also considers 4-block messages. Hence, we briefly explain the encryption of 4-block messages with reference to Figure 8.2. E_{K_2} is the encryption function of the underlying block cipher instantiated with the key K_2. The encryption consists of five layers; the first, second, fourth and fifth being masking layers and the third layer being application of E_{K_2}. For a 4-block message $x_1||x_2||x_3||x_4$ and hash key τ, $SP = \kappa^{-1}\left(x_1\tau^4 \oplus x_2\tau^3 \oplus x_3\tau^2 \oplus x_4\tau\right)$, where $\kappa = 1 \oplus \tau \oplus \tau^2 \oplus \tau^3 \oplus \tau^4$ which is assumed to be non-zero. The exact definitions of α, β and SC are not required for our purpose, so, we skip these details and refer to [63] for their definitions. We only note that the tweak T is used in determining β.

Figure 8.2: Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus \chi_0\tau \oplus \chi_1\tau||m||m$ under TET.



Fix $m, \chi_0, \chi_1 \in \{0,1\}^n$, such that $\chi_0 \neq \chi_1$; let b denote a bit and we define the following function.

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\rightarrow \{0,1\}^n \\ (b, x) &\overset{f}{\mapsto} y_3 \oplus y_4, \text{ where } y_1||y_2||y_3||y_4 \longleftarrow \mathsf{TET.Encrypt}_{K_1,K_2}(T, \chi_b||x||m||m). \end{aligned} \tag{8.4}$$

The function f defined in (8.4) satisfies the following property.

Proposition 13. *Let $b, b' \in \{0,1\}, x \in \{0,1\}^n$. Suppose that the hash key τ is non-zero. Then, $f(b, x) = f(b', x \oplus \chi_b\tau \oplus \chi_{b'}\tau)$, where χ_0, χ_1 and τ are as described earlier.*

Proof. For the input $\chi_b||x||m||m$, we have,

$$\begin{aligned} SP &= \kappa^{-1}(\chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus m\tau); \\ Q_3 &= m \oplus SP \oplus \alpha^2\beta; \\ Q_4 &= m \oplus SP \oplus \alpha^3\beta; \\ S_3 &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus \alpha^2\beta; \\ S_4 &= E_{K_2}(m \oplus SP \oplus \alpha^3\beta) \oplus \alpha^3\beta; \\ y_3 \oplus y_4 &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP \oplus \alpha^3\beta) \oplus \alpha^3\beta; \end{aligned}$$

For the input $\chi_{b'}||x \oplus \chi_b\tau \oplus \chi_{b'}\tau||m||m$, we have,

$$\begin{aligned} SP' &= \kappa^{-1}(\chi_{b'}\tau^4 \oplus x\tau^3 \oplus \chi_b\tau^4 \oplus \chi_{b'}\tau^4 \oplus m\tau^2 \oplus m\tau) = \kappa^{-1}(\chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus m\tau); \\ Q'_3 &= m \oplus SP' \oplus \alpha^2\beta; \\ Q'_4 &= m \oplus SP' \oplus \alpha^3\beta; \\ S'_3 &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus \alpha^2\beta; \\ S'_4 &= E_{K_2}(m \oplus SP' \oplus \alpha^3\beta) \oplus \alpha^3\beta; \\ y'_3 \oplus y'_4 &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP' \oplus \alpha^3\beta) \oplus \alpha^3\beta. \end{aligned}$$

Now, $SP = SP'$ implies $y_3 \oplus y_4 = y'_3 \oplus y'_4$.
Hence, the proposition is proved. □

The above proposition establishes that $1||\chi_0\tau \oplus \chi_1\tau$ is a period of f. Proposition 13 falls short of showing that f satisfies the promise of Simon's problem. We show below, that f satisfies an approximate promise.

Proposition 14. *Assume that the block cipher E instantiated with a uniform random key, behaves like a uniform random function. Suppose that the hash key τ is non-zero. Then, for f defined in (8.4), $\varepsilon(f, 1||\chi_0\tau \oplus \chi_1\tau) \leq 5/2^n$.*

Proof. Let $\upsilon||\mathtt{t} \notin \{0||0^n, 1||\chi_0\tau \oplus \chi_1\tau\}$ be such that the probability of $f(b, x) = f(b \oplus \upsilon, x \oplus \mathtt{t})$ is maximised.

Case 1: Suppose $\upsilon = 0$. Then $\mathtt{t}$ is necessarily non-zero. Let,

$$SP = \kappa^{-1}(\chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus m\tau); \qquad SP' = \kappa^{-1}(\chi_b\tau^4 \oplus x\tau^3 \oplus \mathtt{t}\tau^3 \oplus m\tau^2 \oplus m\tau). \quad (8.5)$$

As $\mathtt{t}$ is necessarily non-zero, $SP \neq SP'$.
We have,

$$\begin{aligned} f(b, x) &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP \oplus \alpha^3\beta) \oplus \alpha^3\beta; \\ f(b, x \oplus \mathtt{t}) &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP' \oplus \alpha^3\beta) \oplus \alpha^3\beta. \end{aligned}$$

Then $f(b, x) = f(b, x \oplus \mathtt{t})$ if and only if $X_1 = X_2$, where

$$\begin{aligned} X_1 &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus E_{K_2}(m \oplus SP \oplus \alpha^3\beta); \\ X_2 &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus E_{K_2}(m \oplus SP' \oplus \alpha^3\beta). \end{aligned}$$

Let $\mathcal{E}$ be the event that $(m \oplus SP \oplus \alpha^2\beta)$, $(m \oplus SP \oplus \alpha^3\beta)$, $(m \oplus SP' \oplus \alpha^2\beta)$ and $(m \oplus SP' \oplus \alpha^3\beta)$ are distinct. As $SP \neq SP'$, clearly $(m \oplus SP \oplus \alpha^2\beta)$ and $(m \oplus SP' \oplus \alpha^2\beta)$ are distinct and so are $(m \oplus SP \oplus \alpha^3\beta)$ and $(m \oplus SP' \oplus \alpha^3\beta)$. As β is generated through the application of a PRF, the probability that $(\alpha^2\beta = \alpha^3\beta)$ is $1/2^n$. With similar reasoning probability that $SP \oplus SP' \oplus \alpha^2\beta \oplus \alpha^3\beta = 0$ is $1/2^n$. So, we have $\Pr[\overline{\mathcal{E}}] = 4/2^n$. Conditioned on $\mathcal{E}$, and under the assumption that E behaves like a uniform random function, the probability that $X_1 = X_2$ is at most $1/2^n$. Using (8.2) we have $\Pr_{b,x}[f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})] = \Pr[X_1 = X_2] \leq 5/2^n$.

Case 2: Suppose $\upsilon = 1$. Then $\mathsf{t} \neq \chi_0\tau \oplus \chi_1\tau$. Let $b' = b \oplus 1$. Let,

$$SP = \kappa^{-1}(\chi_b\tau^4 \oplus x\tau^3 \oplus m\tau^2 \oplus m\tau); \qquad SP' = \kappa^{-1}(\chi_{b'}\tau^4 \oplus x\tau^3 \oplus \mathsf{t}\tau^3 \oplus m\tau^2 \oplus m\tau) \quad (8.6)$$

As $\mathsf{t} \neq \chi_0\tau \oplus \chi_1\tau$, $SP \neq SP'$.
We have,

$$\begin{aligned} f(b,x) &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP \oplus \alpha^3\beta) \oplus \alpha^3\beta; \\ f(b \oplus \upsilon, x \oplus \mathsf{t}) &= f(b', x \oplus \mathsf{t}) \\ &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus \alpha^2\beta \oplus E_{K_2}(m \oplus SP' \oplus \alpha^3\beta) \oplus \alpha^3\beta. \end{aligned}$$

So, again $f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})$ if and only if $X_1 = X_2$, where

$$\begin{aligned} X_1 &= E_{K_2}(m \oplus SP \oplus \alpha^2\beta) \oplus E_{K_2}(m \oplus SP \oplus \alpha^3\beta); \\ X_2 &= E_{K_2}(m \oplus SP' \oplus \alpha^2\beta) \oplus E_{K_2}(m \oplus SP' \oplus \alpha^3\beta). \end{aligned}$$

A reasoning similar to Case 1 shows that $\Pr_{b,x}[f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})]$ is at most $5/2^n$. □

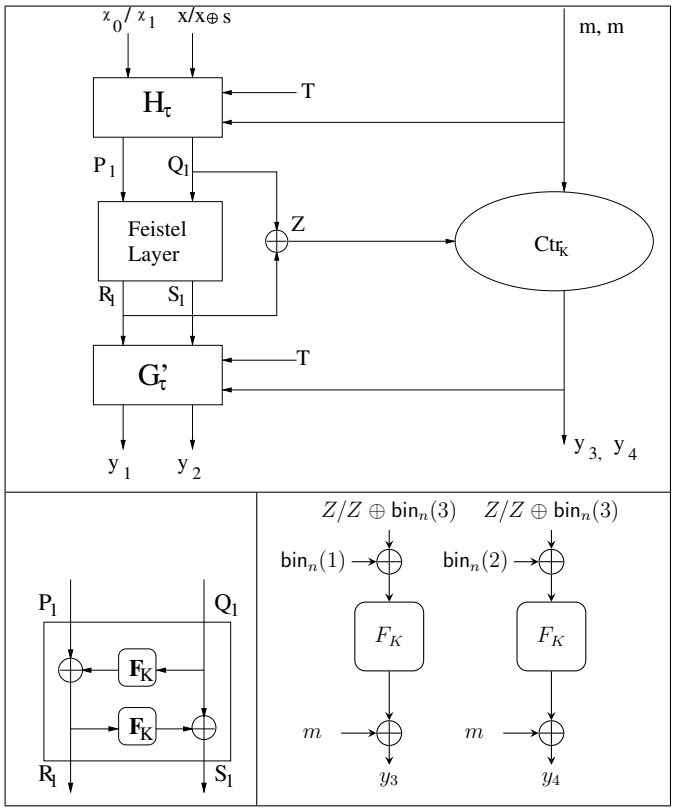
Classical queries: Given the period $1||s = 1||\tau(\chi_0 \oplus \chi_1)$, the two classical queries required in Section 8.2 are the following. The first query is $\chi_0||x||m||m$ with output $y_1||y_2||y_3||y_4$ and the second query is $\chi_1||x \oplus s||m||m$ with output $y_1'||y_2'||y_3'||y_4'$. From the proof of Proposition 13 we have that $y_3 \oplus y_4 = y_3' \oplus y_4'$ which defines the relation between the outputs of the two classical queries.

Partial key recovery: Once $s = \tau(\chi_0 \oplus \chi_1)$ has been obtained, since χ_0 and χ_1 are distinct, from s, one obtains the hash key τ as $\tau = s(\chi_0 \oplus \chi_1)^{-1}$.

8.3.3 FAST

FAST was proposed by Chakraborty, Ghosh, López and Sarkar [29]. It is built using a fixed input length pseudo-random function and an appropriate hash function. The key K of FAST is the same as the key of the underlying pseudo-random function. The pseudo-random function maps n-bit strings to n-bit strings. For the sake of concreteness, we fix $n = 128$. Let F_K denote the pseudo-random function instantiated with the key K. FAST is targeted towards two application scenarios. We describe the quantum attack on the instantiation targeted towards the specific task of disk encryption. In this case, the tweak

Figure 8.3: Enciphering a 4-block message $\chi_0||x||m||m$ or $\chi_1||x \oplus s||m||m$ under FAST.



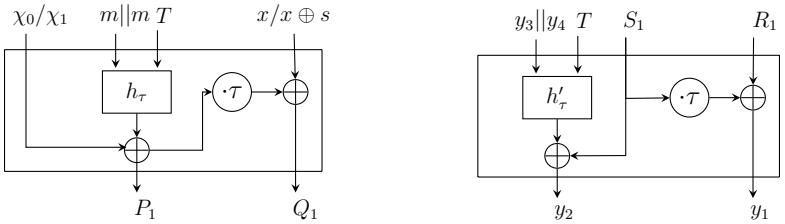
space is $\mathcal{T} = \{0,1\}^n$ and the message space is $\mathcal{P} = \{0,1\}^{\mathfrak{m}n}$, where $\mathfrak{m} > 2$ is determined by the size of a disk sector. In our attack we will fix the tweak to be an arbitrary $T \in \mathcal{T}$.

Our attack considers 4-block messages. So, we briefly describe the encryption of 4-block messages with reference to Figures 8.3 and 8.4. From a top level view, FAST consists of three distinct layers - hash-encrypt-hash. The hashing layers $\mathbf{H}$ and $\mathbf{G}'$ are based on two universal hash functions h and h', both having the key τ and $h' = \tau h$. For more details of these functions we refer to [29]. The encryption layer consists of a two-round Feistel network and a counter mode Ctr. The two-round Feistel is built using the PRF F_K and processes the first two blocks of the plaintext. The third and fourth blocks are encrypted in a counter mode built using F_K. The offset for the counter mode is derived from the input and output of the Feistel layer. The input of the Feistel layer is obtained by processing the plaintext and the tweak through the first hash layer. The second hash layer generates the first two blocks of the ciphertext by processing the output of the Feistel layer and the third and fourth blocks of the ciphertext. Some more details are provided as part of the attack.

Fix $m, \chi_0, \chi_1 \in \{0,1\}^n$, such that $\chi_0 \oplus \chi_1 = 0^{126}11$; let b denote a bit. We define the following function.

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\rightarrow \{0,1\}^n \\ (b, x) &\overset{f}{\mapsto} y_3 \oplus y_4, \text{ where } y_1||y_2||y_3||y_4 \longleftarrow \mathsf{FAST.Encrypt}_K(T, \chi_b||x||m||m). \end{aligned} \tag{8.7}$$

Figure 8.4: The hash functions **H** (left) and **G′** (right).



The function f defined in (8.7) satisfies the following property.

Proposition 15. *Let $b, b' \in \{0,1\}, x \in \{0,1\}^n$. Suppose that the hash key τ is non-zero. Then, $f(b,x) = f(b', x \oplus \chi_b\tau \oplus \chi_{b'}\tau)$. χ_0, χ_1 and τ are as described before.*

Proof. For the input $(T, \chi_b||x||m||m)$,

$$\begin{aligned}
P_1 &= \chi_b \oplus h_\tau(T, m||m);\\
Q_1 &= x \oplus \tau(\chi_b \oplus h_\tau(T, m||m));\\
R_1 &= \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1);\\
Z &= Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1);\\
y_3 &= m \oplus F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1));\\
y_4 &= m \oplus F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(2));\\
y_3 \oplus y_4 &= F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1))\\
&\qquad \oplus F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(2));
\end{aligned}$$

For the input $(T, \chi_{b'}||x \oplus \chi_b\tau \oplus \chi_{b'}\tau||m||m)$,

$$\begin{aligned}
P'_1 &= \chi_{b'} \oplus h_\tau(T, m||m);\\
Q'_1 &= x \oplus \chi_b\tau \oplus \chi_{b'}\tau \oplus \tau(\chi_{b'} \oplus h_\tau(T, m||m))\\
&= x \oplus \chi_b\tau \oplus \tau h_\tau(T, m||m);\\
R'_1 &= \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1);\\
Z' &= Q'_1 \oplus \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1);\\
y'_3 &= m \oplus F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1));\\
y'_4 &= m \oplus F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(2));\\
y'_3 \oplus y'_4 &= F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1))\\
&\qquad \oplus F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(2))\\
&= F_K(Q'_1 \oplus \chi_b \oplus \mathsf{bin}_n(3) \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1))\\
&\quad \oplus F_K(Q'_1 \oplus \chi_b \oplus \mathsf{bin}_n(3) \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(2)),\\
&\qquad\qquad (\text{as } \chi_b \oplus \chi_{b'} = \mathsf{bin}_n(3))\\
&= F_K(Q'_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(2))\\
&\qquad \oplus F_K(Q'_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1)).
\end{aligned}$$

We observe that $Q_1 = Q'_1$, which implies $y_3 \oplus y_4 = y'_3 \oplus y'_4$. This proves the proposition. □

The above discussion establishes that $1||\chi_0\tau \oplus \chi_1\tau$ is a period of f. Proposition 15 falls short of showing that f satisfies the promise of Simon's problem. We show below, that f satisfies an approximate promise.

Proposition 16. *Assume that the PRF F instantiated with a uniform random key, behaves like a uniform random function. Suppose that the hash key τ is non-zero. Then, for f defined in (8.7), $\varepsilon(f, 1||\chi_0\tau \oplus \chi_1\tau) \leq \frac{3}{2^n}$.*

Proof. Let $\upsilon||\mathsf{t} \notin \{0||0^n, 1||\chi_0\tau \oplus \chi_1\tau\}$ be such that the probability of $f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})$ is maximised.

- Case 1: Suppose $\upsilon = 0$. Then t is necessarily non-zero. We have

$$\begin{aligned} f(b,x) &= F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1)) \\ &\quad \oplus F_K(Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(2)); \\ f(b, x \oplus \mathsf{t}) &= F_K(Q_1 \oplus \mathsf{t} \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1 \oplus \mathsf{t}) \oplus \mathsf{bin}_n(1)) \\ &\quad \oplus F_K(Q_1 \oplus \mathsf{t} \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1 \oplus \mathsf{t}) \oplus \mathsf{bin}_n(2)). \end{aligned}$$

  Let

$$\begin{aligned} c_1 &= Q_1 \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1), \\ c_2 &= Q_1 \oplus \mathsf{t} \oplus \chi_b \oplus h_\tau(T, m||m) \oplus F_K(Q_1 \oplus \mathsf{t}) \oplus \mathsf{bin}_n(1), \\ v &= \mathsf{bin}_n(1) \oplus \mathsf{bin}_n(2); \end{aligned}$$

  Hence

$$\begin{aligned} f(b,x) &= F_K(c_1) \oplus F_K(c_1 \oplus v); \\ f(b, x \oplus \mathsf{t}) &= F_K(c_2) \oplus F_K(c_2 \oplus v). \end{aligned}$$

  Let $\mathcal{E}$ be the event that c_1, $c_1 \oplus v$, c_2 and $c_2 \oplus v$ are distinct. Clearly c_1 and $c_1 \oplus v$ are distinct and so are c_2 and $c_2 \oplus v$. Now we consider the following events.

  - $\mathcal{E}_1 := c_1 = c_2$ or, equivalently $F_K(Q_1) \oplus F_K(Q_1 \oplus \mathsf{t}) = \mathsf{t}$. As $\mathsf{t} \neq 0$ and F_K is assumed to be a uniform random function, hence $\Pr[\mathcal{E}_1] = \frac{1}{2^n}$.
  - $\mathcal{E}_2 := c_1 = c_2 \oplus v$ or, equivalently $F_K(Q_1) \oplus F_K(Q_1 \oplus \mathsf{t}) = \mathsf{t} \oplus v$. As in the previous case, $\Pr[\mathcal{E}_2] = \frac{1}{2^n}$.

  Hence, $\overline{\mathcal{E}} := \mathcal{E}_1 \cup \mathcal{E}_2$. Note that since v is a non-zero string $\mathcal{E}_1$ and $\mathcal{E}_2$ are disjoint. Hence, $\Pr[\overline{\mathcal{E}}] = \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2] = \frac{2}{2^n}$. Conditioned on $\mathcal{E}$, and under the assumption that F_K behaves like a uniform random function, the probability that $f(b,x) = f(b, x \oplus \mathsf{t})$ is at most $1/2^n$. Using (8.2) we have $\Pr_{b,x}[f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})] \leq 3/2^n$.

- Case 2: Suppose $\upsilon = 1$. Then $\mathsf{t} \neq \chi_0\tau \oplus \chi_1\tau$. Let $b' = b \oplus 1$, $Q'_1 = x \oplus \mathsf{t} \oplus \tau(\chi_{b'} \oplus h_\tau(T, m||m))$. We have

$$\begin{aligned} f(b,x) &= F_K(Q_1 \oplus \chi_b \oplus h_\tau(T,m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1)) \\ &\quad \oplus F_K(Q_1 \oplus \chi_b \oplus h_\tau(T,m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(2)); \\ f(b \oplus \upsilon, x \oplus \mathsf{t}) &= f(b', x \oplus \mathsf{t}) \\ &= F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T,m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1)) \\ &\quad \oplus F_K(Q'_1 \oplus \chi_{b'} \oplus h_\tau(T,m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(2)); \end{aligned}$$

  Let

$$\begin{aligned} c_1 &= Q_1 \oplus \chi_b \oplus h_\tau(T,m||m) \oplus F_K(Q_1) \oplus \mathsf{bin}_n(1) \\ c_2 &= Q'_1 \oplus \chi_{b'} \oplus h_\tau(T,m||m) \oplus F_K(Q'_1) \oplus \mathsf{bin}_n(1), \\ v &= \mathsf{bin}_n(1) \oplus \mathsf{bin}_n(2); \end{aligned}$$

  Hence

$$\begin{aligned} f(b,x) &= F_K(c_1) \oplus F_K(c_1 \oplus v); \\ f(b \oplus \upsilon, x \oplus \mathsf{t}) &= F_K(c_2) \oplus F_K(c_2 \oplus v). \end{aligned}$$

  A reasoning similar to Case 1 shows that $\Pr_{b,x}[f(b,x) = f(b \oplus \upsilon, x \oplus \mathsf{t})]$ is at most $3/2^n$.

□

Classical queries: Given the period $1||s = 1||\tau(\chi_0 \oplus \chi_1)$, the two classical queries required in Section 8.2 are the following. The first query is $\chi_0||x||m||m$ with output $y_1||y_2||y_3||y_4$ and the second query is $\chi_1||x \oplus \tau(\chi_0 \oplus \chi_1)||m||m$ with output $y'_1||y'_2||y'_3||y'_4$. From the proof of Proposition 15 we have that $y_3 \oplus y_4 = y'_3 \oplus y'_4$ which defines the relation between the outputs of the two classical queries.

Partial key recovery: Once $s = \tau(\chi_0 \oplus \chi_1)$ has been obtained, since χ_0 and χ_1 are distinct, from s, one obtains the hash key τ as $\tau = s(\chi_0 \oplus \chi_1)^{-1}$.

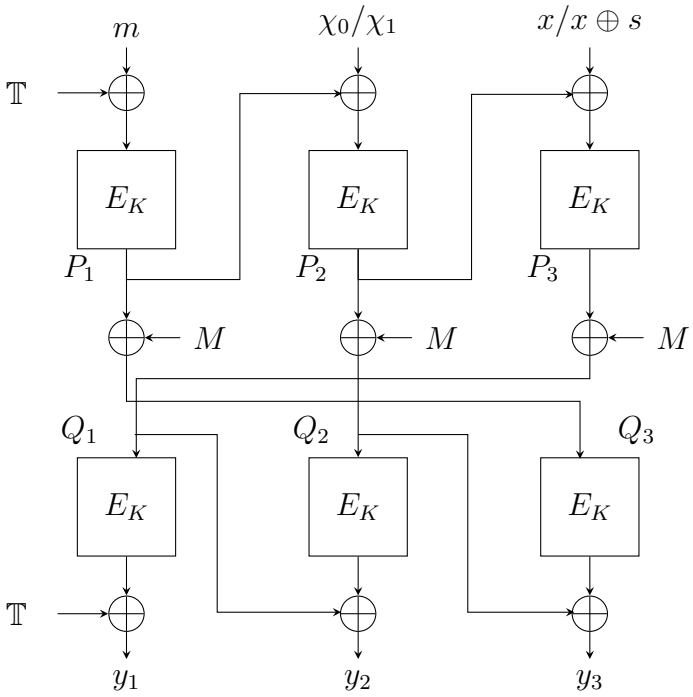
8.4 Distinguishing Attacks

8.4.1 CMC

CMC was proposed by Halevi and Rogaway [65], in 2003. It is based on the CBC mode of operation of a block cipher. The block length of the block cipher can be assumed to be n-bit. CMC has the key space $\mathcal{K} \times \mathcal{K}$, where $\mathcal{K}$ is the key space for the underlying block cipher and the tweak space $\mathcal{T} = \{0,1\}^n$. The message space of CMC is $\mathcal{P} = \bigcup_{i \in I} \{0,1\}^i$ for some non-empty index set $I \subseteq \mathbb{N}$. Let E_K denote the encryption function of the underlying block cipher instantiated with the key K.

Our attack considers 3-block messages. Hence, we briefly describe the encryption of 3-block messages with reference to Figure 8.5. Let CMC be instantiated with the key $(K, \tilde{K}) \in \mathcal{K} \times \mathcal{K}$. In our attack we will fix the tweak to be an arbitrary $T \in \mathcal{T}$. $\tilde{K}$ is used as the key to the block cipher E only to produce $\mathbb{T}$ from T. At a conceptual level, the CMC encryption function consists of three layers. The first layer is essentially CBC encryption on the message blocks, followed by a layer of masking and the third layer is CBC decryption. The rest of the encryption algorithm can be understood from Figure 8.5. We provide more details as part of the attack.

Figure 8.5: Enciphering a 3-block message $m||\chi_0||x$ or $m||\chi_1||x \oplus s$ under CMC. Correspondingly, $M = 2(P_1 \oplus P_3)$ and $M' = 2(P'_1 \oplus P'_3)$.



Fix $m, \chi_0, \chi_1 \in \{0,1\}^n$, such that $\chi_0 \neq \chi_1$; let b denote a bit and we define the following function.

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\rightarrow \{0,1\}^n \\ (b, x) &\overset{f}{\mapsto} y_1, \text{ where } y_1||y_2||y_3 \longleftarrow \mathsf{CMC.Encrypt}_{K,\tilde{K}}(T, m||\chi_b||x). \end{aligned} \quad (8.8)$$

The function f defined in (8.8) satisfies the following property.

Proposition 17. *Let $b, b' \in \{0,1\}, x, x' \in \{0,1\}^n$. Then, $f(b,x) = f(b',x') \Leftrightarrow x \oplus x' = E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_b) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_{b'})$, where the constants χ_0 and χ_1 are as fixed before.*

Proof. Let $s = E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_1)$. For the input $m||\chi_0||x$,

$$\begin{aligned} P_1 &= E_K(m \oplus \mathbb{T}); \\ P_2 &= E_K(\chi_0 \oplus E_K(m \oplus \mathbb{T})); \\ P_3 &= E_K(x \oplus E_K(\chi_0 \oplus E_K(m \oplus \mathbb{T}))); \\ M &= 2(P_1 \oplus P_3); \\ Q_1 &= P_3 \oplus M; \\ y_1 &= E_K(Q_1) \oplus \mathbb{T}; \end{aligned} \tag{8.9}$$

For the input $m||\chi_1||x \oplus s$,

$$\begin{aligned} P'_1 &= E_K(m \oplus \mathbb{T}); \\ P'_2 &= E_K(\chi_1 \oplus E_K(m \oplus \mathbb{T})); \\ P'_3 &= E_K(x \oplus s \oplus E_K(\chi_1 \oplus E_K(m \oplus \mathbb{T}))) \\ &= E_K(x \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_1) \oplus E_K(\chi_1 \oplus E_K(m \oplus \mathbb{T}))) \\ &= E_K(x \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0)); \\ M' &= 2(P'_1 \oplus P'_3); \\ Q'_1 &= P'_3 \oplus M'; \\ y'_1 &= E_K(Q'_1) \oplus \mathbb{T}. \end{aligned} \tag{8.10}$$

We see $P_1 = P'_1$ and $P_3 = P'_3$ implying $M = M'$; $P_3 = P'_3$ and $M = M'$ together imply $Q_1 = Q'_1$, which finally establishes $y_1 = y'_1$. This proves one direction of the proposition. Now we see the other direction.

$$\begin{aligned} f(b,x) = f(b',x') &\Rightarrow y_1 = y'_1 \\ &\Rightarrow E_K(Q_1) \oplus \mathbb{T} = E_K(Q'_1) \oplus \mathbb{T} \\ &\Rightarrow Q_1 = Q'_1 \\ &\Rightarrow P_3 \oplus M = P'_3 \oplus M' \\ &\Rightarrow P_3 \oplus 2(P_1 \oplus P_3) = P'_3 \oplus 2(P'_1 \oplus P'_3) \\ &\Rightarrow P_3 \oplus 2(P_1 \oplus P_3) = P'_3 \oplus 2(P_1 \oplus P'_3) \text{ (as } P'_1 = P_1) \\ &\Rightarrow P_3 = P'_3 \\ &\Rightarrow E_K(x \oplus E_K(\chi_b \oplus E_K(m \oplus \mathbb{T}))) = E_K(x' \oplus E_K(\chi_{b'} \oplus E_K(m \oplus \mathbb{T}))) \\ &\Rightarrow x \oplus E_K(\chi_b \oplus E_K(m \oplus \mathbb{T})) = x' \oplus E_K(\chi_{b'} \oplus E_K(m \oplus \mathbb{T})) \\ &\Rightarrow x \oplus x' = E_K(\chi_b \oplus E_K(m \oplus \mathbb{T})) \oplus E_K(\chi_{b'} \oplus E_K(m \oplus \mathbb{T})). \end{aligned}$$

□

The above proposition proves that $1||E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_1)$ is a period for the function f and f is a 2-to-1 function. So, Simon's algorithm applied to f

uncovers this period with high probability.

Obtaining the period $1||s$, where $s = E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_1)$ provides a distinguishing attack against CMC.

Classical queries: Given the period $1||s = 1||E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_0) \oplus E_K(E_K(m \oplus \mathbb{T}) \oplus \chi_1)$, the two classical queries required in Section 8.2 are the following. The first query is $m||\chi_0||x$ with output $y_1||y_2||y_3$ and the second query is $m||\chi_1||x \oplus s$ with output $y_1'||y_2'||y_3'$. From the proof of Proposition 17 we have that $y_1 = y_1'$ which defines the relation between the outputs of the two classical queries.

8.4.2 EME

EME also was proposed by Halevi and Rogaway [66]. It was later extended to handle arbitrary length messages by Halevi [62] and the resulting scheme was called EME*. EME* has been standardised as a TES by IEEE [3] in the name EME2.

Our attack considers 3-block messages. For this message length the constructions EME and EME2 are identical, with only the minor replacement of the tweak by a function of the tweak in the latter. Hence, we will describe the attack in the context of EME only.

EME has the key space $\mathcal{K}$, the same as the underlying block cipher having block size n; the tweak space is $\mathcal{T} = \{0,1\}^n$. The message space of EME is $\mathcal{P} = \{0,1\}^n \cup \{0,1\}^{2n} \cup \cdots \cup \{0,1\}^{n^2}$. The key K of EME is the same as the key of the underlying block cipher. Let E_K denote the encryption function of the underlying block cipher instantiated with the key K. In our attack we fix the tweak to be an arbitrary $T \in \mathcal{T}$.

As our attack considers 3-block messages, we will briefly describe the encryption of 3-block messages with reference to Figure 8.6. The encryption of each of the message blocks consists of five layers: initial masking followed by an application of E_K, then another masking followed by another application of E_K and the final masking. One of the masking elements is $L = 2E_K(0^n)$. The middle layer of masking for the first block is of different form from that for second and third blocks. The rest of the encryption algorithm can be understood from Figure 8.6. We provide more details as part of the attack.

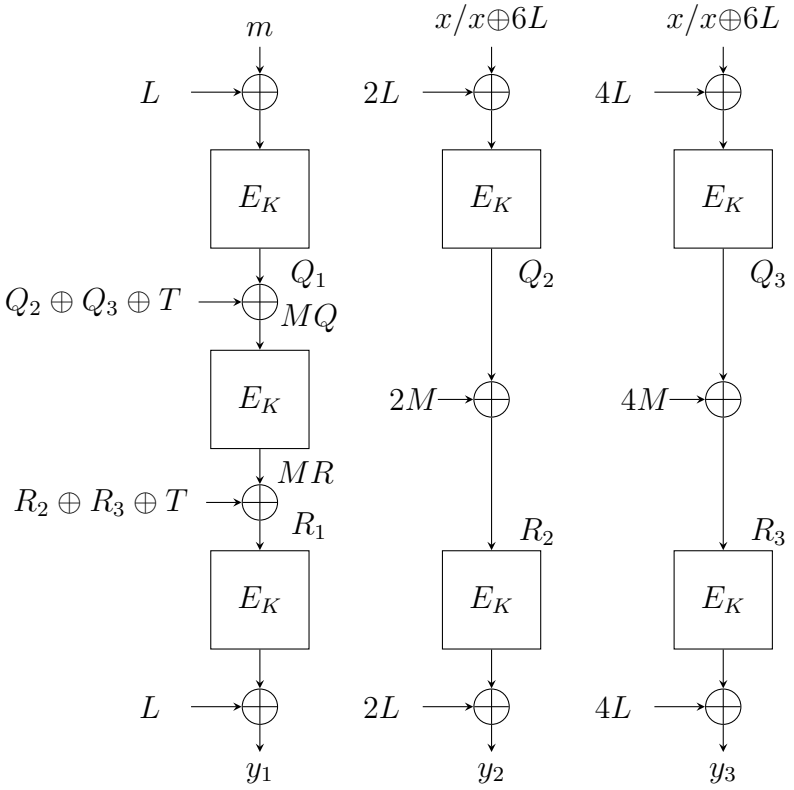
Notation: Let x denote a root of the primitive polynomial used to represent $GF(2^n)$. Note that, $2L$ represents L times the field element denoted by x. Similarly, $4L$ represents L times the field element denoted by x^2; $6L$ represents L times the field element denoted by $x^2 \oplus x$. Hence, $6L = (x^2 \oplus x)L = 4L \oplus 2L$. This notation has been used in [66] and so we follow this notation.

Fix $m \in \{0,1\}^n$ and we define the following function.

$$\begin{aligned} f : \{0,1\}^n &\rightarrow \{0,1\}^n \\ x &\overset{f}{\mapsto} y_1, \text{ where } y_1||y_2||y_3 \longleftarrow \mathsf{EME.Encrypt}_K(T, m||x||x). \end{aligned} \tag{8.11}$$

The function f defined in (8.11) satisfies the following property.

Figure 8.6: Enciphering a 3-block message $m||x||x$ or $m||x \oplus 6L||x \oplus 6L$ under EME. Correspondingly, $M = MQ \oplus E_K(MQ)$, where $MQ = E_K(m \oplus L) \oplus E_K(x \oplus 2L) \oplus E_K(x \oplus 4L) \oplus T$ and $M' = MQ' \oplus E_K(MQ')$ where $MQ' = E_K(m \oplus L) \oplus E_K(x \oplus 4L) \oplus E_K(x \oplus 2L) \oplus T$.



Proposition 18. *Let $x \in \{0,1\}^n$. Then, $f(x) = f(x \oplus 6L)$, where L is as defined before and suppose it is non-zero.*

Proof. Consider the two inputs $m||x||x$ and $m||(x \oplus 6L)||(x \oplus 6L)$.

For the input $m||x||x$,

$$\begin{aligned}
Q_1 &= E_K(m \oplus L);\\
Q_2 &= E_K(x \oplus 2L);\\
Q_3 &= E_K(x \oplus 4L);\\
MQ &= E_K(m \oplus L) \oplus E_K(x \oplus 2L) \oplus E_K(x \oplus 4L) \oplus T;\\
MR &= E_K(MQ);\\
M &= MQ \oplus MR;\\
R_2 &= E_K(x \oplus 2L) \oplus 2M;\\
R_3 &= E_K(x \oplus 4L) \oplus 4M;\\
SR &= E_K(x \oplus 2L) \oplus 2M \oplus E_K(x \oplus 4L) \oplus 4M;\\
R_1 &= MR \oplus SR \oplus T;\\
y_1 &= L \oplus E_K(R_1) = E_K(MR \oplus SR \oplus T) \oplus L; \qquad (8.12)
\end{aligned}$$

For the input $m||(x \oplus 6L)||(x \oplus 6L)$,

$$\begin{aligned}
Q'_1 &= E_K(m \oplus L);\\
Q'_2 &= E_K(x \oplus 4L);\\
Q'_3 &= E_K(x \oplus 2L);\\
MQ' &= E_K(m \oplus L) \oplus E_K(x \oplus 4L) \oplus E_K(x \oplus 2L) \oplus T;\\
MR' &= E_K(MQ');\\
M' &= MQ' \oplus MR';\\
R'_2 &= E_K(x \oplus 4L) \oplus 2M';\\
R'_3 &= E_K(x \oplus 2L) \oplus 4M';\\
SR' &= E_K(x \oplus 4L) \oplus 2M' \oplus E_K(x \oplus 2L) \oplus 4M';\\
R'_1 &= MR' \oplus SR' \oplus T;\\
y'_1 &= L \oplus E_K(R'_1) = E_K(MR' \oplus SR' \oplus T) \oplus L. \qquad (8.13)
\end{aligned}$$

From above we see that $MQ = MQ'$; hence, $MR = MR'$ and $M = M'$; $M = M'$ implies $SR = SR'$. Hence, we see, $y_1 = y'_1$, proving the proposition. □

The above discussion establishes that $6L$ is a period of f. Proposition 18 falls short of showing that f satisfies the promise of Simon's problem. We show below, that f satisfies an approximate promise.

Proposition 19. *Assume that the block cipher E instantiated with a uniform random key, behaves like a uniform random function. Suppose L is non-zero. Then, for f defined in (8.11), $\varepsilon(f, 6L) \leq 1/2^{n-1}$.*

Proof. Let $\mathtt{t} \notin \{0^n, 6L\}$ be such that the probability of $f(x) = f(x \oplus \mathtt{t})$ is maximised.

We have

$$\begin{aligned}
f(x) &= L \oplus E_K(MR \oplus SR \oplus T);\\
f(x \oplus \mathtt{t}) &= L \oplus E_K(MR' \oplus SR' \oplus T);\ \text{where } MR = E_K(MQ), MR' = E_K(MQ'),\\
MQ &= E_K(m \oplus L) \oplus E_K(x \oplus 2L) \oplus E_K(x \oplus 4L) \oplus T,\\
MQ' &= E_K(m \oplus L) \oplus E_K(x \oplus \mathtt{t} \oplus 2L) \oplus E_K(x \oplus \mathtt{t} \oplus 4L) \oplus T,\\
SR &= E_K(x \oplus 2L) \oplus 2M \oplus E_K(x \oplus 4L) \oplus 4M,\\
SR' &= E_K(x \oplus \mathtt{t} \oplus 2L) \oplus 2M' \oplus E_K(x \oplus \mathtt{t} \oplus 4L) \oplus 4M',\\
M &= MQ \oplus MR, M' = MQ' \oplus MR'.
\end{aligned}$$

Then $f(x) = f(x \oplus \mathtt{t}) \Leftrightarrow E_K(MR \oplus SR \oplus T) = E_K(MR' \oplus SR' \oplus T) \Leftrightarrow MR \oplus SR = MR' \oplus SR'$. So,

$$\begin{aligned}
&\Pr[f(x) = f(x \oplus \mathtt{t})]\\
&= \Pr[MR \oplus MR' = SR \oplus SR']\\
&= \Pr[MR \oplus MR' = MQ \oplus MQ' \oplus 6M \oplus 6M'] \text{ (as } SR \oplus SR' = MQ \oplus MQ' \oplus 6M \oplus 6M')\\
&= \Pr[MQ \oplus MR \oplus MQ' \oplus MR' = 6M \oplus 6M']\\
&= \Pr[M \oplus M' = 6M \oplus 6M'] \qquad (8.14)\\
&= \Pr[M = M'] \qquad (8.15)\\
&= \Pr[E_K(MQ) \oplus E_K(MQ') = MQ \oplus MQ'] \qquad (8.16)
\end{aligned}$$

The explanation for obtaining (8.15) from (8.14) is the following. From (8.14), we have $7M = 7M'$. Recall that here 7 is a shorthand for the polynomial $x^2 \oplus x \oplus 1$, where x is the root of the degree n primitive polynomial used to represent the field. So, $x^2 \oplus x \oplus 1$ is an invertible element in the field and hence the equality $M = M'$ follows.

Let $\mathcal{E}$ be the event $MQ \neq MQ'$. Under the assumption that E_K behaves like a uniform random function, $\Pr[E_K(MQ) \oplus E_K(MQ') = MQ \oplus MQ' | \mathcal{E}] \leq 1/2^n$. Since $\mathtt{t} \notin \{0^n, 6L\}$ and L is non-zero, $x \oplus 2L$, $x \oplus 4L$, $x \oplus \mathtt{t} \oplus 2L$ and $x \oplus \mathtt{t} \oplus 4L$ are distinct. As a result, under the assumption that E_K behaves like a uniform random function, we have, $\Pr[\overline{\mathcal{E}}] \leq 1/2^n$. From (8.2) and (8.16) we have $\Pr[f(x) = f(x \oplus \mathtt{t})] \leq 2/2^n$. □

Classical queries: Given the period $6L$, the two classical queries required in Section 8.2 are the following. The first query is $m||x||x$ with output $y_1||y_2||y_3$ and the second query is $m||x \oplus 6L||x \oplus 6L$ with output $y_1'||y_2'||y_3'$. From the proof of Proposition 18 we have that $y_1 = y_1'$ which defines the relation between the outputs of the two classical queries.

8.5 Summary

In this chapter, we have shown the applicability of Simon's period finding quantum algorithm to the cryptanalysis of five TESs, namely, CMC, EME, XCB, TET and FAST. Although each of these TESs has standard proof of security in the classical world, this chapter shows

that none of them is secure against a quantum adversary. For XCB, TET and FAST, partial key recovery attacks are possible. For all of the five TESs, we have shown distinguishing attacks. This shows that for the quantum world, none of these TESs will work and we need to find one. This piece of information adds another stepping stone in the journey towards the quantum era.

Chapter 9

Future Research Possibilities

Modes of operations is a pretty old branch of research in the area of Cryptology. It is a widely spread area and a huge volume of interesting works have been done here. As a result it is already a immensely developed area, while at the same time being open enough to attract further attention. We believe each of the five works on which this thesis is based, has significantly contributed in some way or the other towards further development of the area. The Chapters 4, 5 and 7 give algorithms or schemes which are more efficient in some important platform than other candidates of its category present in the literature. The Chapter 6 adds to the primitive MAC, a new feature which has important practical motivation, but has not gained much attention before. The Chapter 8 shows the need of having a TES which will be able to resist a quantum adversary.

- In Chapter 4, we have described an efficient non-recursive algorithm to evaluate BRW polynomials which works for any number of blocks. This algorithm has been used to define two concrete hash functions. Implementations of the hash functions using instructions available on modern Intel processors show promising timing results making the hash functions worthy candidates for actual deployment. The speed-up that we have achieved is mainly due to the algorithmic improvement. On the other hand, there is a possibility of further speed-up by trying to aggresively optimise the code and by considering the details of instruction level pipelining issues. Implementation in assembly is also possibly another promising choice. Future implementation efforts may attempt such works.

- In Chapter 5, we have shown how to combine the BRW family of polynomials with the Horner based polynomial evaluation to design a new hash function. The number of multiplications required for computing the digest is a little more than that for BRW polynomials. The advantage is that the implementation difficulties of BRW polynomials for variable length messages are eliminated. The combination is a two-level hash with BRW at the lower level and Horner at the higher level. The hash key is a single field element and has been appropriately used to work for both the levels. Concrete instantiations of the hash function over binary fields have been reported. The idea, on the other hand, is quite general and applies to other fields as well. A possible future work is to explore this idea to build concrete hash functions over other finite fields.

- In Chapter 6, we have considered the problem of constructing variable tag length MAC schemes. Several variants obtained from the Wegman-Carter MAC scheme have been shown to be insecure. One of these variants is proved to be secure. This scheme is extended to obtain constructions of single-key nonce-based variable tag length MAC schemes using either a stream cipher or a short-output PRF. To the best of our knowledge, this is the first time that in the literature such an extensive formal treatment of

this notion has been considered. An immediate future work can be efficient implementation of some of the proposed schemes.

- In Chapter 7, we have presented a tweakable enciphering scheme called FAST. Instantiations of the scheme for both fixed length messages with single block tweaks and variable length messages with very general tweaks have been described. A detailed security analysis in the style of reductionist security proof has been provided. Software implementations of both kinds of instantiations have been made. The instantiation for fixed length messages with single block tweaks is appropriate for low-level disk encryption. The implementation results show that the new scheme outperforms previous schemes which makes the new scheme an attractive option for designers and standardisation bodies. Again an implementation in assembly is worth trying as future endeavour for further speed-up.

- Chapter 8 shows that some of the well known TESs which are secure in the classical world are broken in the quantum world. This leaves us with the following question. Can some simple modifications of these schemes make them quantum secure? Perhaps future research will answer this question.

Bibliography

[1] Public comments on the XTS-AES mode. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/XTS/collected_XTS_comments.pdf`.

[2] IEEE Std 1619-2007: Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. Available at: `http://standards.ieee.org/findstds/standard/1619-2007.html`, 2008.

[3] IEEE Std 1619.2-2010: IEEE standard for wide-block encryption for shared storage media. `http://standards.ieee.org/findstds/standard/1619.2-2010.html`, March 2011.

[4] Post-Quantum Cryptography standardisation: A process initiated by NIST for standardising quantum-safe public-key cryptosystems. Available at: `https://csrc.nist.gov/projects/post-quantum-cryptography`, 2017-ongoing.

[5] IETF RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. Available at: `https://tools.ietf.org/html/rfc8446`, August 2018.

[6] Gorjan Alagic, Anne Broadbent, Bill Fefferman, Tommaso Gagliardoni, Christian Schaffner, and Michael St. Jules. Computational security of quantum encryption. In Anderson C. A. Nascimento and Paulo S. L. M. Barreto, editors, *Information Theoretic Security - 9th International Conference, ICITS 2016, Tacoma, WA, USA, August 9-12, 2016, Revised Selected Papers*, volume 10015 of *Lecture Notes in Computer Science*, pages 47–71, 2016.

[7] Mayuresh Vivekanand Anand, Ehsan Ebrahimi Targhi, Gelo Noel Tabia, and Dominique Unruh. Post-quantum security of the cbc, cfb, ofb, ctr, and XTS modes of operation. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 44–63. Springer, 2016.

[8] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012.

[9] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Koblitz [70], pages 1–15.

[10] Mihir Bellare, David Cash, and Sriram Keelveedhi. Ciphers that securely encipher their own keys. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 423–432. ACM, 2011.

[11] Côme Berbain and Henri Gilbert. On the security of IV dependent stream ciphers. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2007.

[12] Daniel J. Bernstein. The Salsa20 family of stream ciphers. `http://cr.yp.to/papers.html#salsafamily`. Document ID: 31364286077dcdff8e4509f9ff3139ad. Date: 2007.12.25.

[13] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[14] Daniel J. Bernstein. Stronger security bounds for Wegman-Carter-Shoup authenticators. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 2005.

[15] Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. `http://cr.yp.to/papers.html#pema`.

[16] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field macs. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[17] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. KECCAK sponge function family main document. NIST SHA-3 Submission (updated), `http://keccak.noekeon.org/`, January 2009.

[18] Sreyosi Bhattacharyya and Palash Sarkar. Improved SIMD implementation of poly1305. *IET Inf. Secur.*, 14(5):521–530, 2020.

[19] Ritam Bhaumik and Mridul Nandi. An inverse-free single-keyed tweakable enciphering scheme. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 159–180. Springer, 2015.

[20] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1999.

[21] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun

Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69. Springer, 2011.

[22] Dan Boneh and Mark Zhandry. Quantum-secure message authentication codes. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 592–608. Springer, 2013.

[23] Xavier Bonnetain. Quantum key-recovery on full AEZ. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, volume 10719 of *Lecture Notes in Computer Science*, pages 394–406. Springer, 2017.

[24] Xavier Bonnetain, Akinori Hosoyamada, María Naya-Plasencia, Yu Sasaki, and André Schrottenloher. Quantum attacks without superposition queries: The offline Simon's algorithm. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 552–583. Springer, 2019.

[25] Anne Broadbent and Stacey Jeffery. Quantum homomorphic encryption for circuits of low t-gate complexity. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 609–629. Springer, 2015.

[26] CAESAR. Competition for Authenticated Encryption: Security, Applicability, and Robustness. `http://competitions.cr.yp.to/caesar.html`.

[27] Anne Canteaut and Kapalee Viswanathan, editors. *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*. Springer, 2004.

[28] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[29] Debrup Chakraborty, Sebati Ghosh, Cuauhtemoc Mancillas López, and Palash Sarkar. FAST: disk encryption and beyond. *Advances in Mathematics of Communications*, 2020. `https://www.aimsciences.org/article/doi/10.3934/amc.2020108`.

[30] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. *IACR Trans. Symmetric Cryptol.*, 2017(1):106–128, 2017.

[31] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. Hash2L implementations (128 and 256-bit). 2017. `https://github.com/sebatighosh/HASH2L.git`.

[32] Debrup Chakraborty, Vicente Hernandez-Jimenez, and Palash Sarkar. Another look at XCB. *Cryptography and Communications*, 7(4):439–468, 2015.

[33] Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez, and Palash Sarkar. Efficient hardware implementations of BRW polynomials and tweakable enciphering schemes. *IEEE Trans. Computers*, 62(2):279–294, 2013.

[34] Debrup Chakraborty, Cuauhtemoc Mancillas-López, and Palash Sarkar. STES: A stream cipher based low cost scheme for securing stored data. *IEEE Trans. Computers*, 64(9):2691–2707, 2015.

[35] Debrup Chakraborty and Mridul Nandi. An improved security bound for HCTR. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 289–302. Springer, 2008.

[36] Debrup Chakraborty and Palash Sarkar. A new mode of encryption providing a tweakable strong pseudo-random permutation. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2006.

[37] Debrup Chakraborty and Palash Sarkar. HCH: A new tweakable enciphering scheme using the hash-counter-hash approach. *IEEE Transactions on Information Theory*, 54(4):1683–1699, 2008.

[38] Debrup Chakraborty and Palash Sarkar. On modes of operations of a block cipher for authentication and authenticated encryption. *Cryptography and Communications*, 8(4):455–511, 2016.

[39] Paul Crowley and Eric Biggers. Adiantum: length-preserving encryption for entry-level processors. *IACR Trans. Symmetric Cryptol.*, 2018(4):39–61, 2018.

[40] Ivan Damgård, Jakob Funder, Jesper Buus Nielsen, and Louis Salvail. Superposition attacks on cryptographic protocols. In Carles Padró, editor, *Information Theoretic Security - 7th International Conference, ICITS 2013, Singapore, November 28-30, 2013, Proceedings*, volume 8317 of *Lecture Notes in Computer Science*, pages 142–161. Springer, 2013.

[41] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Remark on variable tag lengths and OMD. `https://groups.google.com/forum/#!searchin/crypto-competitions/Remark$20on$20variable$20tag$20lengths$20and$20OMD%7Csort:date/crypto-competitions/sekKDsIJvwU/5_V_TzZQaWYJ`, accessed on 15 November, 2019, 2014.

[42] Xiaoyang Dong, Bingyou Dong, and Xiaoyun Wang. Quantum attacks on some feistel block ciphers. *Des. Codes Cryptogr.*, 88(6):1179–1203, 2020.

[43] Xiaoyang Dong and Xiaoyun Wang. Quantum key-recovery attack on feistel structures. *Sci. China Inf. Sci.*, 61(10):102501:1–102501:7, 2018.

[44] Morris Dworkin. Recommendation for block cipher modes of operations: the CMAC mode for authentication, May 2005. National Institute of Standards and Technology, U.S. Department of Commerce. NIST Special Publication 800-38B.

[45] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, `csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`, November 2007.

[46] Morris J. Dworkin. SP 800-38E. Recommendation for Block Cipher Modes of Operation: the XTS-AES Mode for Confidentiality on Storage Devices. Technical report, Gaithersburg, MD, United States, 2010.

[47] Hal Finney. CFRG discussion on UMAC. `https://marc.info/?l=cfrg&m=143336318427069&w=2`, accessed on 15 November, 2019, 2005.

[48] Hal Finney. CFRG discussion on UMAC. `https://marc.info/?l=cfrg&m=143336318527072&w=2`, accessed on 15 November, 2019, 2005.

[49] Tommaso Gagliardoni, Andreas Hülsing, and Christian Schaffner. Semantic security and indistinguishability in the quantum world. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, volume 9816 of *Lecture Notes in Computer Science*, pages 60–89. Springer, 2016.

[50] Sebati Ghosh and Palash Sarkar. Breaking tweakable enciphering schemes using simon's algorithm. *IACR Cryptol. ePrint Arch.*, 2019:724, 2019.

[51] Sebati Ghosh and Palash Sarkar. BRW efficient non-recursive implementations (128 and 256-bit). 2019. `https://github.com/sebatighosh/BRW`.

[52] Sebati Ghosh and Palash Sarkar. Evaluating Bernstein-Rabin-Winograd polynomials. *Des. Codes Cryptogr.*, 87(2-3):527–546, 2019.

[53] Sebati Ghosh and Palash Sarkar. Variants of Wegman-Carter message authentication code supporting variable tag lengths. *Des. Codes Cryptogr.*, 2021. `https://doi.org/10.1007/s10623-020-00840-w`.

[54] Edgar N. Gilbert, F. Jessie MacWilliams, and Neil J. A. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53:405–424, 1974.

[55] Martin Goll and Shay Gueron. Vectorization of poly1305 message authentication code. In *2015 12th International Conference on Information Technology-New Generations*, pages 145–150, 2015.

[56] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996.

[57] Shay Gueron. AES-GCM-SIV implementations (128 and 256-bit). 2016. `https://github.com/Shay-Gueron/AES-GCM-SIV`.

[58] Shay Gueron and Michael E. Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010.

[59] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the gcm mode. 2010. Intel white paper.

[60] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: specification and analysis. *IACR Cryptol. ePrint Arch.*, 2017:168, 2017.

[61] Shay Gueron and Yehuda Lindell. Gcm-siv: full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 109–119, 2015.

[62] Shai Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. In Canteaut and Viswanathan [27], pages 315–327.

[63] Shai Halevi. Invertible universal hashing and the TET encryption mode. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 412–429. Springer, 2007.

[64] Shai Halevi and Hugo Krawczyk. Security under key-dependent inputs. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 466–475. ACM, 2007.

[65] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer, 2003.

[66] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.

[67] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2015.

[68] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 207–237. Springer, 2016.

[69] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, Tuplehash and Parallelhash. NIST Special Publication 800–185, `https://doi.org/10.6028/NIST.SP.800-185`, December 2016.

[70] Neal Koblitz, editor. *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[71] Ted Krovetz. UMAC: Message authentication code using universal hashing. `https://tools.ietf.org/html/draft-krovetz-umac-05.html`, accessed on 15 November, 2019., 2005.

[72] Ted Krovetz and Wei Dai. VMAC: Message authentication code using universal hashing. 2007.

[73] Ted Krovetz and Phillip Rogaway. Fast universal hashing with small keys and no preprocessing: The polyr construction. In Dongho Won, editor, *ICISC*, volume 2015 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2000.

[74] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[75] Hidenori Kuwakado and Masakatu Morii. Quantum distinguisher between the 3-round feistel cipher and the random permutation. In *IEEE International Symposium on Information Theory, ISIT 2010, June 13-18, 2010, Austin, Texas, USA, Proceedings*, pages 2682–2685. IEEE, 2010.

[76] Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type even-mansour cipher. In *Proceedings of the International Symposium on Information Theory and its Applications, ISITA 2012, Honolulu, HI, USA, October 28-31, 2012*, pages 312–316. IEEE, 2012.

[77] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptogr. Eng.*, 6(3):171–185, 2016.

[78] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

[79] Atul Luykx and Bart Preneel. Optimal forgeries against polynomial-based macs and GCM. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 445–467. Springer, 2018.

[80] James H. Manger. Attacker changing tag length in OCB. `https://mailarchive.ietf.org/arch/msg/cfrg/gJtV9FCw92MguqqhxrSNUyIDZIw`, accessed on 15 November, 2019., 2013.

[81] David A. McGrew and Scott R. Fluhrer. The extended codebook (XCB) mode of operation. Cryptology ePrint Archive, Report 2004/278, 2004. `http://eprint.iacr.org/`.

[82] David A. McGrew and Scott R. Fluhrer. The extended codebook (XCB) mode of operation. *IACR Cryptol. ePrint Arch.*, 2004:278, 2004.

[83] David A. McGrew and Scott R. Fluhrer. The security of the extended codebook (xcb) mode of operation. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2007.

[84] David A. McGrew and John Viega. Arbitrary block length mode, 2004. `http://grouper.ieee.org/groups/1619/email/pdf00005.pdf`.

[85] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In Canteaut and Viswanathan [27], pages 343–355.

[86] Kazuhiko Minematsu. Parallelizable rate-1 authenticated encryption from pseudorandom functions. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2014.

[87] Mridul Nandi. On the minimum number of multiplications necessary for universal hash functions. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2014.

[88] Mridul Nandi. Bernstein bound on WCS is tight - repairing luykx-preneel optimal forgeries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara,*

CA, USA, August 19-23, 2018, Proceedings, Part II, volume 10992 of *Lecture Notes in Computer Science*, pages 213–238. Springer, 2018.

[89] Wim Nevelsteen and Bart Preneel. Software performance of universal hash functions. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 24–41. Springer, 1999.

[90] Mike Ounsworth. Footguns as an axis of security analysis. `https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/l2iYk-8sGnI`, accessed on 15 November, 2019, 2019.

[91] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.

[92] Reza Reyhanitabar, Serge Vaudenay, and Damian Vizár. Authenticated encryption with variable stretch. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 396–425, 2016.

[93] P. Rogaway and D. Wagner. A critique of ccm. Cryptology ePrint Archive, Report 2003/070, 2003. `https://eprint.iacr.org/2003/070`.

[94] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[95] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[96] Reihaneh Safavi-Naini, Viliam Lisý, and Yvo Desmedt. Economically optimal variable tag length message authentication. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, volume 10322 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 2017.

[97] Palash Sarkar. A general mixing strategy for the ECB-Mix-ECB mode of operation. *Inf. Process. Lett.*, 109(2):121–123, 2008.

[98] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4759, 2009.

[99] Palash Sarkar. A trade-off between collision probability and key size in universal hashing using polynomials. *Des. Codes Cryptography*, 58(3):271–278, 2011.

[100] Palash Sarkar. Tweakable enciphering schemes using only the encryption function of a block cipher. *Inf. Process. Lett.*, 111(19):945–955, 2011.

[101] Palash Sarkar. A new multi-linear universal hash family. *Des. Codes Cryptogr.*, 69(3):351–367, 2013.

[102] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[103] Victor Shoup. On fast and provably secure message authentication based on universal hashing. In Koblitz [70], pages 313–328.

[104] Daniel R. Simon. On the power of quantum computation. *SIAM J. Comput.*, 26(5):1474–1483, 1997.

[105] Vladimir Soukharev, David Jao, and Srinath Seshadri. Post-quantum security models for authenticated encryption. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2016.

[106] Douglas R. Stinson. Universal hashing and authentication codes. *Des. Codes Cryptography*, 4(4):369–380, 1994.

[107] UMAC. CFRG discussion on UMAC. `http://marc.info/?l=cfrg&m=143336318427068&w=2`, accessed on 15 November, 2019, 2005.

[108] David Wagner. CFRG discussion on UMAC. `https://marc.info/?l=cfrg&m=143336318527073&w=2`, accessed on 15 November, 2019, 2005.

[109] Peng Wang, Dengguo Feng, and Wenling Wu. HCTR: A variable-input-length enciphering mode. In Dengguo Feng, Dongdai Lin, and Moti Yung, editors, *CISC*, volume 3822 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2005.

[110] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[111] Shmuel Winograd. A new algorithm for inner product. *IEEE Transactions on Computers*, 17:693–694, 1968.