

**Cryptographic Primitives
Based on
Reverse Decryption**

*Thesis submitted
for the award of the degree
of
Doctor of Philosophy
by
Suyash Kandele*

under the guidance of

Dr. Souradyuti Paul



Department of Electrical Engineering and Computer Science
Indian Institute of Technology Bhilai

January 2020

To

Nehal Kandele

My Sister, My Motivator, My Strength.

My Partner in Crime...

© 2020 Suyash Kandele. All rights reserved.

Thesis Certificate

This is to certify that the thesis entitled "**Cryptographic Primitives Based on Reverse Decryption**", submitted by **Suyash Kandele (ID No. 11710050)** to Indian Institute of Technology Bhilai, for the award of the degree of **Doctor of Philosophy** is a bonafide record of research work carried out by the student under my supervision. The contents of this thesis, in full or in parts, have not been separately submitted to any Institute or University for the award of a degree or diploma.



Date: 22nd January 2020

Place: IIT Bhilai, Raipur

Souradyuti Paul

Associate Professor

Department of EECS

Indian Institute of Technology Bhilai

Acknowledgements

First and foremost, I would like to thank the *Almighty* for bestowing his blessings on me to make it possible for me to pursue Ph.D.

Any achievement is impossible without the unconditional support from the Family. I would like to express my gratitude to my Parents and my Sister, for the immense love, trust, care and motivation that they have always showered on me.

The first name that comes to my mind when I think of my Ph.D. is my thesis supervisor Dr. Souradyuti Paul, without whom I would have never been able to begin this exciting and thrilling journey of my doctoral study. For me, he has been a mentor, a motivator, a friend and an example himself. Allowing me freedom of thoughts, expression and timeline, he has always supported me to explore even the tiniest idea I have thought of. His support through the tough times, and his cheer in the successes has always boosted my self-confidence. The four and a half years of academic collaboration with him have been an experience unforgettable for the lifetime.

In the past years, I have met many cryptographers working in diverse areas. They have given me insights of various aspects of cryptology, that has helped me to acquire significant knowledge. I would like to thank Prof. Bimal Kumar Roy (Indian Statistical Institute Kolkata), Prof. Bart Preneel (Katholieke Universiteit Leuven), Dr. Manoj Prabhakaran (Indian Institute of Technology Bombay), Dr. Elena Andreeva (Technical University of Denmark), Dr. Bart Mennink (Radboud University) and Dr. Mridul Nandi (Indian Statistical Institute Kolkata).

I express my thankfulness to Dr. Nithin V. George, Dr. Anirban Dasgupta, Dr. Bireswar Das, Dr. Neeldhara Misra, Dr. Manoj Gupta, Dr. Madhumita Sengupta, Dr. Arnapurna Rath and Dr. Malavika Subramanyam, the faculties of Indian Institute of Technology Gandhinagar, for being huge source of knowledge and motivation for me.

I am grateful to Prof. Kamalakar Karlapalem (International Institute of Information Technology, Hyderabad), Dr. Manu Awasthi (Ashoka University), Dr. Dinesh Garg (India Research Laboratory, Bangalore), Dr.

Subrahmanyam Kalyanasundaram (Indian Institute of Technology Hyderabad) and Dr. Veena Bansal (Indian Institute of Technology Kanpur) for their untiring support and innovative ideas of teaching and guiding.

I would like to thank Prof. Rajat Moona (Director, IIT Bhilai), Prof. R.K. Ghosh, Dr. Santosh Biswas (HOD, EECS), Dr. Jose Immanuel R (Dean of Academic Affairs), Dr. Dhiman Saha, Dr. Rishi Ranjan Singh, Dr. Sk Subidh Ali, Dr. Barun Gorain, Dr. Amit Kumar Dhar, Dr. Vinod Reddy and Dr. Subhajit Sidhanta, the faculties of Indian Institute of Technology Bhilai, for their constant support and encouragement.

I extend my thanks to the staff members at Indian Institute of Technology Gandhinagar, especially, Mr. Ram Babu Bhagat (Deputy Registrar, Estt. & Admin, and PIO), Ms. Meena Joshi (Assistant Registrar & APIO), Mr. B. V. Puvar, Mr. Shreejit B Menon, Mr. Yogesh Dattatraya Jade, Mr. Krupesh Chauhan, Mr. Hemant K. Gupta, Mr. Piyush Vankar, Ms. Sunita Menon, Ms. Hani Khamar, Mr. Viral Y. Shah, Mr. Harish Singh and Mr. Vishwajeet Mishra.

My heartfelt thanks to the staff members at Indian Institute of Technology Bhilai, especially, Mr. Ashok K. Gupta (Deputy Registrar, Administration), Mr. Gautam Ramani (Deputy Registrar, Finance & Accounts), Mr. Nihar Ranjan Barick (Assistant Registrar, Academic & Students' Affairs), Mr. Hemant Verma, Mrs. R. Neelima Gowthami, Mr. Sunil Kumar, Mr. Bhumesh Thakre, Mr. Kunal Kumar Waldekar, Mr. Pawan Kumar Mishra, Ms. Shubhangshu Sharma, Ms. Khushboo Bais, Mr. K. Aditya, Mr. Naveen Rai and Mr. Yogesh Sinha.

My heartfelt thanks to Mrs. Rajni Moona, for her dedication in building this new Institute and for constantly motivating the students to prosper in life.



Suyash Kandele

IIT Bhilai

Abstract

Keywords: *Reverse Decryption, Message-Locked Encryption (MLE), Key Assignment Scheme (KAS), All-Or-Nothing Transform (AONT), and Authenticated Encryption (AE).*

With the exponential rise in *Internet*-based applications (*e.g.* secure computation, e-commerce, and blockchains) and with the emergence of new technologies (*e.g.* *Artificial Intelligence (AI)*, *Machine Learning (ML)*, and *Internet-of-Things (IoT)*), data is being generated at an astonishing rate. Most of this data is outsourced to public databases, such as the Cloud, for better availability and reduced storage costs. However, in this new environment, preservation of data in a secure manner, has become a major challenge.

In this thesis, we have designed and analysed various types of cryptographic primitives with potential applications in the *Cloud* and *IoT* devices. These primitives include *file-updatable message-locked encryption (FMLE)*, *key assignment scheme with authenticated encryption (KAS-AE)*, *all-or-nothing transform (AONT)*, *all-or-nothing encryption (AONE)* and *one-time AONE*. Note that, of all these, the *FMLE*, *KAS-AE* and *one-time AONE* are introduced for the first time. These cryptographic algorithms have been shown to exhibit various security properties, such as data confidentiality, data integrity, and authentication.

On the construction side, in total, we have designed *eleven* highly-efficient schemes (2 *FMLEs*, 3 *KAS-AEs*, 2 *AONTs*, 2 *AONES*, and 2 *one-time AONES*). Except one *KAS-AE* construction, all remaining ten constructions exploit the very rare *reverse decryption* property inherent in the *APE* authenticated encryption and *FP* hash mode of operation. Very interestingly, this *reverse decryption* property allows us to design cryptographic schemes, where seemingly diverse sets of security properties coexist. For example, the encryption key of our *one-time AONE* construction is a public value – like the *public key* of a *public key cryptosystem (PKC)* – where the *decryption key* is message-dependent – unlike the case in a *PKC*, but similar to that in a *message-locked encryption (MLE)*. Leveraging the power of the *reverse decryption* property to the maximum advantage, we are also able to design new constructions of all the aforementioned primitives whose performance and security bounds are several orders of magnitude better than that of the existing ones.

For all constructions, we have provided rigorous security proofs using some of the most sophisticated tools available in modern cryptology, such as *code-based game playing technique*, *PRP/PRF switching lemma*, *triangle inequality*, *principle of inclusion-exclusion*, and various graph-theoretic results. We have also provided several interesting open problems.

List of Publications

- Suyash Kandele and Souradyuti Paul. “*Message-locked encryption with file update*”. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 678–695. Springer, 2018. [The 43-page full version is on *IACR e-print archive* at <https://eprint.iacr.org/2018/422.pdf>]
- Suyash Kandele and Souradyuti Paul. “*Key assignment scheme with authenticated encryption*”. *IACR Transactions on Symmetric Cryptology*, 2018(4):150–196, 2018. (This work was presented at *FSE 2019* held in Paris.) [The 47-page full version is also available on *IACR e-print archive* at <https://eprint.iacr.org/2018/1233.pdf>]

Under Review

- Suyash Kandele and Souradyuti Paul. “*Efficient AONTs and AONES Based on Permutations*”. The 39-page manuscript has been submitted to *Design, Codes and Cryptography* on 3rd October 2019.

Contents

Thesis Certificate	v
Acknowledgements	vii
Abstract	ix
List of Publications	xi
Contents	xiii
List of Tables	xvii
List of Figures	xix
List of Abbreviations	xxv
List of Symbols	xxvii
1 Introduction	1
1.1 What is Reverse Decryption?	2
1.2 Motivation	3
1.3 Summary of the Thesis Work	3
1.4 Main Technical Results	4
1.4.1 File-Updatable Message-Locked Encryption (FMLE)	4
1.4.2 Key Assignment Scheme with Authenticated Encryption (KAS-AE)	5
1.4.3 All-Or-Nothing Transform (AONT)	6
1.4.4 All-Or-Nothing Encryption (AONE)	6
1.4.5 One-Time AONE	7
1.5 Organization of the Thesis	8
2 Cryptologic Background	9
	xiii

2.1	Introduction to Cryptology	9
2.1.1	Cryptography	10
2.1.2	Cipher	10
2.1.3	Cryptanalysis: Types of Attacks	15
2.1.4	A Note on the Security Parameter λ	17
2.2	Basic Cryptographic Primitives	17
2.2.1	Ideal Permutation	17
2.2.2	Random Function	18
2.2.3	Hash Function	18
2.2.4	Symmetric Encryption (SE)	23
2.2.5	Authenticated Encryption (AE)	25
2.3	Useful Techniques	32
3	Advanced Tools and Techniques	35
3.1	Introduction	35
3.2	Graphs	36
3.2.1	Dijkstra's Algorithm	37
3.2.2	Posets, Chains and Access Graphs	38
3.2.3	Graph Algorithms	40
3.3	Deduplication	46
3.4	Message-Locked Encryption (MLE)	48
3.4.1	Message-source \mathcal{S}	49
3.4.2	Definition	49
3.4.3	Existing MLE schemes	55
3.4.4	Proof-of-Ownership Function	58
3.4.5	Updatable block-level message-locked encryption (UMLE)	59
3.4.6	Dictionary Attack	67
3.5	Hierarchical Access Control (HAC)	67
3.6	Key Assignment Scheme (KAS)	70
3.6.1	Definition	70
3.6.2	Existing KAS schemes	74
4	Message-Locked Encryption with File Update	81
4.1	Introduction	81
4.2	Contribution of this Chapter	82
4.3	<i>FMLE</i> : A New Cryptographic Primitive	82
4.3.1	Syntax	82
4.3.2	Correctness	84
4.3.3	Security definitions	85
4.4	Deduplication: An Application of <i>FMLE</i>	90
4.5	Practical <i>FMLE</i> Constructions from existing MLE schemes	93

4.5.1	Construction F-CE	94
4.5.2	Construction F-HCE2	97
4.5.3	Construction F-RCE	100
4.6	Practical <i>FMLE</i> Construction from existing <i>UMLE</i> scheme	104
4.6.1	Construction F-UMLE	104
4.7	New Efficient <i>FMLE</i> Schemes	107
4.7.1	Construction $\hat{\Phi}$	107
4.7.2	Construction $\tilde{\Phi}$	119
4.7.3	Resistance of $\hat{\Phi}$ and $\tilde{\Phi}$ against Dictionary Attack	128
4.8	Comparison of <i>FMLE</i> schemes	128
4.9	Conclusion and Future Work	131
5	Key Assignment Scheme with Authenticated Encryption	133
5.1	Introduction	133
5.2	Contribution of this Chapter	134
5.3	<i>KAS-AE</i> : A New Cryptographic Primitive	135
5.3.1	Syntax	135
5.3.2	Correctness	136
5.3.3	Security definitions	137
5.4	<i>HAC</i> : An Application of <i>KAS-AE</i>	140
5.5	<i>KAS-AE</i> from <i>KAS</i> and <i>AE</i>	142
5.5.1	A natural construction Ψ and an attack	142
5.5.2	Construction $\dot{\Psi}_\Omega$: A secure (yet inefficient) <i>KAS-AE</i>	144
5.6	<i>Modified Chain Partition</i> Construction $\vec{\Psi}$: <i>KAS-AE</i> from <i>KAS-AE-chain</i>	148
5.6.1	<i>KAS-AE-chain</i> constructions	148
5.6.2	The <i>Modified Chain Partition</i>	186
5.6.3	Security of <i>Modified Chain Partition</i>	189
5.7	Construction $\breve{\Psi}$: <i>KAS-AE</i> based on <i>MLE</i>	190
5.7.1	Description of $\breve{\Psi}$	190
5.7.2	Security of $\breve{\Psi}$	195
5.8	Building <i>KAS-AE</i> by Tweaking <i>APE</i> and <i>FP</i>	197
5.8.1	Construction $\hat{\Psi}$: <i>KAS-AE</i> based on <i>APE</i>	197
5.8.2	Construction $\tilde{\Psi}$: <i>KAS-AE</i> based on <i>FP</i>	205
5.9	Comparison of <i>KAS-AE</i> schemes	212
5.10	Conclusion and Future Work	216
6	All-Or-Nothing Transform (AONT)	217
6.1	Introduction	217
6.2	Contribution of this Chapter	218
6.3	Related Work	219

6.4	<i>All-Or-Nothing Transform (AONT)</i>	220
6.4.1	Syntax	220
6.4.2	Correctness	220
6.4.3	Security definitions	221
6.5	New <i>AONT</i> Constructions	222
6.5.1	Construction $\hat{\Pi}$	222
6.5.2	Construction $\tilde{\Pi}$	230
6.6	Comparison of <i>AONT</i> schemes	233
6.7	Conclusion and Future Work	235
7	One-Time AONE: A variant of AONE	237
7.1	Introduction	237
7.2	Contribution of this Chapter	238
7.3	Related work	239
7.4	<i>All-Or-Nothing Encryption (AONE)</i>	240
7.4.1	Syntax	240
7.4.2	Correctness	240
7.4.3	Security Definitions	241
7.5	New <i>AONE</i> Constructions	243
7.5.1	Construction $\hat{\chi}$	244
7.5.2	Construction $\tilde{\chi}$	256
7.6	Comparison	262
7.7	<i>One-Time AONE: A Novel AONE variant</i>	264
7.7.1	Syntax	265
7.7.2	Correctness	265
7.7.3	Security Definitions	265
7.8	New <i>One-Time AONE</i> Constructions	268
7.8.1	Construction $\hat{\Upsilon}$	268
7.8.2	Construction $\tilde{\Upsilon}$	272
7.9	Conclusion and Future Work	277
8	Conclusion and Future Scope	279
Bibliography		283
About the Author		299

List of Tables

4.1	Comparison of running time complexities of <i>FMLE</i> schemes.	129
4.2	Comparison of storage requirement and security properties of <i>FMLE</i> schemes.	130
5.1	Comparison table for <i>KAS-AE</i> schemes described in Section 5.5.2.2	213
5.2	Comparison table for <i>KAS-AE</i> schemes described in Section 5.6.	214
5.3	Comparison table for <i>KAS-AE</i> schemes described in Sections Sec- tions 5.7 and 5.8.	215
6.1	Comparison table for various <i>AONT</i> schemes.	234
7.1	Comparison table for various <i>AONE</i> schemes.	263

List of Figures

2.1	Pictorial description of the <i>blockcipher</i> \mathcal{BC} .	11
2.2	Pictorial description of ECB Mode.	13
2.3	Pictorial description of CBC Mode.	13
2.4	Pictorial description of CFB Mode.	14
2.5	Pictorial description of OFB Mode.	15
2.6	Pictorial description of CTR Mode.	16
2.7	Security game HF-CR for the <i>hash function</i> ϑ .	19
2.8	Pictorial description of the <i>hash function</i> FP. \mathcal{H} .	20
2.9	Algorithmic description of the <i>hash function</i> FP. \mathcal{H} .	20
2.10	Pictorial description of the <i>hash function</i> Sponge. \mathcal{H} .	22
2.11	Algorithmic description of the <i>hash function</i> Sponge. \mathcal{H} .	22
2.12	Security game SE-KR for the <i>SE scheme</i> ρ .	24
2.13	Security game SE-IND for the <i>SE scheme</i> ρ .	25
2.14	Security game AE-IND for the <i>AE scheme</i> ϱ .	27
2.15	Security game AE-INT for the <i>AE scheme</i> ϱ .	28
2.16	Algorithmic description of the <i>key generation function</i> APE. $\mathcal{G}\mathcal{E}\mathcal{N}$.	29
2.17	Pictorial description of the <i>encryption function</i> APE. \mathcal{E} .	30
2.18	Algorithmic description of the <i>encryption function</i> APE. \mathcal{E} .	30
2.19	Pictorial description of the <i>decryption function</i> APE. \mathcal{D} .	31
2.20	Algorithmic description of the <i>decryption function</i> APE. \mathcal{D} .	32
2.21	Pictorial representation of the <i>triangle inequality</i> .	33
3.1	Algorithmic description of the <i>Dijkstra's algorithm</i> .	38
3.2	Algorithmic description of the <i>all_succ</i> function.	40
3.3	Algorithmic description of the <i>ch_seq</i> function.	41
3.4	Algorithmic description of the <i>height</i> function.	42
3.5	Algorithmic description of the <i>max_isect</i> function.	43
3.6	Algorithmic description of the <i>max_isect_chs</i> function.	43
3.7	Algorithmic description of the <i>nodes_at_level</i> function.	44
3.8	Algorithmic description of the <i>path</i> function.	45
3.9	Algorithmic description of the <i>vertex_in_order</i> function.	45
3.10	UPLOAD protocol for <i>MLE-based deduplication</i> .	47

3.11 DOWNLOAD protocol for <i>MLE</i> -based <i>deduplication</i>	47
3.12 Security game MLE-PRV for the <i>MLE</i> scheme ϕ	51
3.13 Security game MLE-PRV\$ for the <i>MLE</i> scheme ϕ	52
3.14 Security games MLE-STC and MLE-TC for the <i>MLE</i> scheme ϕ . .	53
3.15 Security game MLE-KR for the <i>MLE</i> scheme ϕ	54
3.16 Algorithmic description of the <i>MLE</i> scheme CE	56
3.17 Algorithmic description of the <i>MLE</i> scheme HCE2	57
3.18 Algorithmic description of the <i>MLE</i> scheme RCE	57
3.19 Security game UMLE-PRV for the <i>UMLE</i> scheme ζ	63
3.20 Security games UMLE-STC and UMLE-TC for the <i>UMLE</i> scheme ζ	64
3.21 Security game UMLE-CXH for the <i>UMLE</i> scheme ζ	65
3.22 Security game UMLE-UNC for the <i>UMLE</i> scheme ζ	66
3.23 ESTABLISH protocol for <i>KAS</i> -based <i>HAC</i>	68
3.24 KEY DERIVE protocol for <i>KAS</i> -based <i>HAC</i>	69
3.25 DECRYPTION protocol for <i>KAS</i> -based <i>HAC</i>	69
3.26 Security game KAS-KI for the <i>KAS</i> Ω	72
3.27 Security game KAS-SKI for the <i>KAS</i> Ω	72
3.28 Security game KAS-KR for the <i>KAS</i> Ω	73
3.29 Algorithmic description of the <i>key generation</i> function $\vec{\Omega}.\mathcal{G}\mathcal{E}\mathcal{N}$. .	78
3.30 Algorithmic description of the <i>key derivation</i> function $\vec{\Omega}.\mathcal{D}\mathcal{E}\mathcal{R}$. .	78
 4.1 Security game FMLE-PRV for the <i>FMLE</i> scheme Φ	85
4.2 Security game FMLE-PRV\$ for the <i>FMLE</i> scheme Φ	86
4.3 Security games FMLE-STC and FMLE-TC for the <i>FMLE</i> scheme Φ	88
4.4 Security game FMLE-UNC for the <i>FMLE</i> scheme Φ	89
4.5 Security game FMLE-CXH for the <i>FMLE</i> scheme Φ	90
4.6 UPLOAD protocol for <i>FMLE</i> -based <i>deduplication</i>	91
4.7 DOWNLOAD protocol for <i>FMLE</i> -based <i>deduplication</i>	92
4.8 UPDATE protocol for <i>FMLE</i> -based <i>deduplication</i>	93
4.9 Algorithmic description of the <i>encryption</i> function F-CE. \mathcal{E}	94
4.10 Algorithmic description of the <i>decryption</i> function F-CE. \mathcal{D}	95
4.11 Algorithmic description of the <i>update</i> function F-CE. \mathcal{U}	95
4.12 Algorithmic description of the <i>PoW algorithm</i> for prover F-CE. \mathcal{P} . .	96
4.13 Algorithmic description of the <i>PoW algorithm</i> for verifier F-CE. \mathcal{V} . .	96
4.14 Algorithmic description of the <i>encryption</i> function F-HCE2. \mathcal{E} . .	97
4.15 Algorithmic description of the <i>decryption</i> function F-HCE2. \mathcal{D} . .	98
4.16 Algorithmic description of the <i>update</i> function F-HCE2. \mathcal{U}	99
4.17 Algorithmic description of the <i>PoW algorithm</i> for prover F-HCE2. \mathcal{P} . .	99
4.18 Algorithmic description of the <i>PoW algorithm</i> for verifier F-HCE2. \mathcal{V} . .	100

4.19	Algorithmic description of the <i>encryption</i> function $\mathbf{F}\text{-RCE}.\mathcal{E}$.	101
4.20	Algorithmic description of the <i>decryption</i> function $\mathbf{F}\text{-RCE}.\mathcal{D}$.	101
4.21	Algorithmic description of the <i>update</i> function $\mathbf{F}\text{-RCE}.\mathcal{U}$.	102
4.22	Algorithmic description of the <i>PoW algorithm for prover</i> $\mathbf{F}\text{-RCE}.\mathcal{P}$.	103
4.23	Algorithmic description of the <i>PoW algorithm for verifier</i> $\mathbf{F}\text{-RCE}.\mathcal{V}$.	103
4.24	Algorithmic description of the <i>encryption</i> function $\mathbf{F}\text{-UMLE}.\mathcal{E}$.	105
4.25	Algorithmic description of the <i>decryption</i> function $\mathbf{F}\text{-UMLE}.\mathcal{D}$.	105
4.26	Algorithmic description of the <i>update</i> function $\mathbf{F}\text{-UMLE}.\mathcal{U}$.	106
4.27	Algorithmic description of the <i>PoW algorithm for prover</i> $\mathbf{F}\text{-UMLE}.\mathcal{P}$.	106
4.28	Algorithmic description of the <i>PoW algorithm for verifier</i> $\mathbf{F}\text{-UMLE}.\mathcal{V}$.	107
4.29	Pictorial description of the <i>encryption</i> function $\hat{\Phi}.\mathcal{E}$.	108
4.30	Algorithmic description of the <i>encryption</i> function $\hat{\Phi}.\mathcal{E}$.	109
4.31	Pictorial description of the <i>decryption</i> function $\hat{\Phi}.\mathcal{D}$.	109
4.32	Algorithmic description of the <i>decryption</i> function $\hat{\Phi}.\mathcal{D}$.	110
4.33	Pictorial description of the <i>update</i> function $\hat{\Phi}.\mathcal{U}$.	111
4.34	Algorithmic description of the <i>update</i> function $\hat{\Phi}.\mathcal{U}$.	112
4.35	Pictorial description of the <i>PoW algorithm for prover</i> $\hat{\Phi}.\mathcal{P}$.	113
4.36	Algorithmic description of the <i>PoW algorithm for prover</i> $\hat{\Phi}.\mathcal{P}$.	113
4.37	Algorithmic description of the <i>PoW algorithm for verifier</i> $\hat{\Phi}.\mathcal{V}$.	114
4.38	Games used in the proof of Theorem 1.	118
4.39	Algorithmic description of $\hat{\Phi}^{(1)}$ and $\hat{\Phi}^{(2)}$.	119
4.40	Reduction used in Theorem 2.	120
4.41	Pictorial description of the <i>encryption</i> function $\tilde{\Phi}.\mathcal{E}$.	121
4.42	Algorithmic description of the <i>encryption</i> function $\tilde{\Phi}.\mathcal{E}$.	121
4.43	Pictorial description of the <i>decryption</i> function $\tilde{\Phi}.\mathcal{D}$.	122
4.44	Algorithmic description of the <i>decryption</i> function $\tilde{\Phi}.\mathcal{D}$.	122
4.45	Pictorial description of the <i>update</i> function $\tilde{\Phi}.\mathcal{U}$.	123
4.46	Algorithmic description of the <i>update</i> function $\tilde{\Phi}.\mathcal{U}$.	124
4.47	Pictorial description of the <i>PoW algorithm for prover</i> $\tilde{\Phi}.\mathcal{P}$.	126
4.48	Algorithmic description of the <i>PoW algorithm for prover</i> $\tilde{\Phi}.\mathcal{P}$.	126
4.49	Algorithmic description of the <i>PoW algorithm for verifier</i> $\tilde{\Phi}.\mathcal{V}$.	127
5.1	Security game KASAE-KR for the <i>KAS-AE</i> scheme Ψ .	137
5.2	Security game KASAE-IND for the <i>KAS-AE</i> scheme Ψ .	138
5.3	Security game KASAE-INT for the <i>KAS-AE</i> scheme Ψ .	139
5.4	ESTABLISH protocol for <i>KAS-AE</i> -based <i>HAC</i> .	140
5.5	KEY DERIVE protocol for <i>KAS-AE</i> -based <i>HAC</i> .	141
5.6	DECRYPTION protocol for <i>KAS-AE</i> -based <i>HAC</i> .	141
5.7	Algorithmic description of the <i>encryption</i> function $\acute{\Psi}.\mathcal{E}$.	143
5.8	Algorithmic description of the <i>key derivation</i> function $\acute{\Psi}.\mathcal{DER}$.	143
5.9	Algorithmic description of the <i>decryption</i> function $\acute{\Psi}.\mathcal{D}$.	143

5.10	Algorithmic description of the <i>encryption</i> function $\hat{\Psi}_\Omega.\mathcal{E}$	145
5.11	Algorithmic description of the <i>key derivation</i> function $\hat{\Psi}_\Omega.\mathcal{DER}$	146
5.12	Algorithmic description of the <i>decryption</i> function $\hat{\Psi}_\Omega.\mathcal{D}$	146
5.13	Pictorial description of the <i>encryption</i> function $\bar{\psi}.\mathcal{E}$	149
5.14	Algorithmic description of the <i>encryption</i> function $\bar{\psi}.\mathcal{E}$	150
5.15	Pictorial description of the <i>key derivation</i> function $\bar{\psi}.\mathcal{DER}$	150
5.16	Algorithmic description of the <i>key derivation</i> function $\bar{\psi}.\mathcal{DER}$	151
5.17	Pictorial description of the <i>decryption</i> function $\bar{\psi}.\mathcal{D}$	151
5.18	Algorithmic description of the <i>decryption</i> function $\bar{\psi}.\mathcal{D}$	152
5.19	Pictorial description of the <i>encryption</i> function $\check{\psi}.\mathcal{E}$	154
5.20	Algorithmic description of the <i>encryption</i> function $\check{\psi}.\mathcal{E}$	154
5.21	Pictorial description of the <i>key derivation</i> function $\check{\psi}.\mathcal{DER}$	155
5.22	Algorithmic description of the <i>key derivation</i> function $\check{\psi}.\mathcal{DER}$	155
5.23	Pictorial description of the <i>decryption</i> function $\check{\psi}.\mathcal{D}$	156
5.24	Algorithmic description of the <i>decryption</i> function $\check{\psi}.\mathcal{D}$	157
5.25	Reduction used in Theorem 15	158
5.26	Algorithmic description of message-source $\mathcal{S}_{u_{i_1},M}$ used in Theorem 15	159
5.27	Reduction used in Theorem 16	160
5.28	Algorithmic description of message-source \mathcal{S}_{M_0,M_1} used in Theorem 16	161
5.29	Reduction used in Theorem 17	162
5.30	Pictorial description of the function \mathcal{F}_1^π	163
5.31	Algorithmic description of the function \mathcal{F}_1^π	163
5.32	Pictorial description of the function \mathcal{F}_2^π	164
5.33	Algorithmic description of the function \mathcal{F}_2^π	164
5.34	Pictorial description of the <i>encryption</i> function $\dot{\psi}.\mathcal{E}$	165
5.35	Algorithmic description of the <i>encryption</i> function $\dot{\psi}.\mathcal{E}$	165
5.36	Pictorial description of the <i>key derivation</i> function $\dot{\psi}.\mathcal{DER}$	166
5.37	Algorithmic description of the <i>key derivation</i> function $\dot{\psi}.\mathcal{DER}$	167
5.38	Pictorial description of the <i>decryption</i> function $\dot{\psi}.\mathcal{D}$	167
5.39	Algorithmic description of the <i>decryption</i> function $\dot{\psi}.\mathcal{D}$	168
5.40	Games used in the proof of Theorem 18	171
5.41	Algorithmic description of $\dot{\psi}^{(2)}$ and $\dot{\psi}^{(3)}$	172
5.42	Games used in the proof of Theorem 19	175
5.43	Games used in the proof of Theorem 20	178
5.44	Pictorial description of the function \mathcal{G}_1^π	179
5.45	Algorithmic description of the function \mathcal{G}_1^π	179
5.46	Pictorial description of the function \mathcal{G}_2^π	180
5.47	Algorithmic description of the function \mathcal{G}_2^π	181
5.48	Pictorial description of the <i>encryption</i> function $\ddot{\psi}.\mathcal{E}$	182

5.49	Algorithmic description of the <i>encryption</i> function $\ddot{\psi}.\mathcal{E}$	182
5.50	Pictorial description of the <i>key derivation</i> function $\ddot{\psi}.\mathcal{DER}$	183
5.51	Algorithmic description of the <i>key derivation</i> function $\ddot{\psi}.\mathcal{DER}$	183
5.52	Pictorial description of the <i>decryption</i> function $\ddot{\psi}.\mathcal{D}$	184
5.53	Algorithmic description of the <i>decryption</i> function $\ddot{\psi}.\mathcal{D}$	184
5.54	Algorithmic description of the <i>encryption</i> function $\vec{\Psi}.\mathcal{E}$	187
5.55	Algorithmic description of the <i>key derivation</i> function $\vec{\Psi}.\mathcal{DER}$	188
5.56	Algorithmic description of the <i>decryption</i> function $\vec{\Psi}.\mathcal{D}$	188
5.57	Pictorial description of the <i>encryption</i> function $\check{\Psi}.\mathcal{E}$	191
5.58	Algorithmic description of the <i>encryption</i> function $\check{\Psi}.\mathcal{E}$	192
5.59	Pictorial description of the <i>key derivation</i> function $\check{\Psi}.\mathcal{DER}$	193
5.60	Algorithmic description of the <i>key derivation</i> function $\check{\Psi}.\mathcal{DER}$	194
5.61	Pictorial description of the <i>decryption</i> function $\check{\Psi}.\mathcal{D}$	195
5.62	Algorithmic description of the <i>decryption</i> function $\check{\Psi}.\mathcal{D}$	196
5.63	Pictorial description of the <i>encryption</i> function $\hat{\Psi}.\mathcal{E}$	198
5.64	Algorithmic description of the <i>encryption</i> function $\hat{\Psi}.\mathcal{E}$	199
5.65	Pictorial description of the <i>key derivation</i> function $\hat{\Psi}.\mathcal{DER}$	200
5.66	Algorithmic description of the <i>key derivation</i> function $\hat{\Psi}.\mathcal{DER}$	201
5.67	Pictorial description of the <i>decryption</i> function $\hat{\Psi}.\mathcal{D}$	202
5.68	Algorithmic description of the <i>decryption</i> function $\hat{\Psi}.\mathcal{D}$	203
5.69	Pictorial description of the <i>encryption</i> function $\tilde{\Psi}.\mathcal{E}$	206
5.70	Algorithmic description of the <i>encryption</i> function $\tilde{\Psi}.\mathcal{E}$	207
5.71	Pictorial description of the <i>key derivation</i> function $\tilde{\Psi}.\mathcal{DER}$	208
5.72	Algorithmic description of the <i>key derivation</i> function $\tilde{\Psi}.\mathcal{DER}$	209
5.73	Pictorial description of the <i>decryption</i> function $\tilde{\Psi}.\mathcal{D}$	210
5.74	Algorithmic description of the <i>decryption</i> function $\tilde{\Psi}.\mathcal{D}$	211
6.1	Security game AONT-AON for <i>AONT</i> scheme Π	221
6.2	Pictorial description of the <i>transformation</i> function $\hat{\Pi}.\mathcal{T}$	223
6.3	Algorithmic description of the <i>transformation</i> function $\hat{\Pi}.\mathcal{T}$	223
6.4	Pictorial description of the <i>inverse transformation</i> function $\hat{\Pi}.\mathcal{I}$	224
6.5	Algorithmic description of the <i>inverse transformation</i> function $\hat{\Pi}.\mathcal{I}$	224
6.6	Games used in the proof of Theorem 36	228
6.7	Algorithmic description of $\hat{\Pi}^{(2)}$ and $\hat{\Pi}^{(3)}$	229
6.8	Algorithmic description of $\hat{\Pi}^{(3)}$ and $\hat{\Pi}^{(4)}$	229
6.9	Pictorial description of the <i>transformation</i> function $\tilde{\Pi}.\mathcal{T}$	230
6.10	Algorithmic description of the <i>transformation</i> function $\tilde{\Pi}.\mathcal{T}$	231
6.11	Pictorial description of the <i>inverse transformation</i> function $\tilde{\Pi}.\mathcal{I}$	231
6.12	Algorithmic description of the <i>inverse transformation</i> function $\tilde{\Pi}.\mathcal{I}$	232

7.1	Security game AONE-KR for <i>AONE</i> scheme χ	241
7.2	Security game AONE-IND for <i>AONE</i> scheme χ	242
7.3	Security game AONE-AON for <i>AONE</i> scheme χ	243
7.4	Algorithmic description of the <i>key generation</i> function $\hat{\chi}.\mathcal{GEN}$. .	244
7.5	Pictorial description of the <i>encryption</i> function $\hat{\chi}.\mathcal{E}$	245
7.6	Algorithmic description of the <i>encryption</i> function $\hat{\chi}.\mathcal{E}$	245
7.7	Pictorial description of the <i>decryption</i> function $\hat{\chi}.\mathcal{D}$	246
7.8	Algorithmic description of the <i>decryption</i> function $\hat{\chi}.\mathcal{D}$	247
7.9	Games used in the proof of AONE-KR security in Theorem 38. .	250
7.10	Algorithmic description of $\hat{\chi}^{(2)}$ and $\hat{\chi}^{(3)}$	251
7.11	Games used in the proof of AONE-IND security in Theorem 39. .	253
7.12	Games used in the proof of AONE-AON security in Theorem 40. .	257
7.13	Algorithmic description of $\hat{\chi}^{(3)}$ and $\hat{\chi}^{(4)}$	258
7.14	Algorithmic description of the <i>key generation</i> function $\tilde{\chi}.\mathcal{GEN}$. .	258
7.15	Pictorial description of the <i>encryption</i> function $\tilde{\chi}.\mathcal{E}$	259
7.16	Algorithmic description of the <i>encryption</i> function $\tilde{\chi}.\mathcal{E}$	259
7.17	Pictorial description of the <i>decryption</i> function $\tilde{\chi}.\mathcal{D}$	260
7.18	Algorithmic description of the <i>decryption</i> function $\tilde{\chi}.\mathcal{D}$	260
7.19	Security game OTAONE-KR for <i>one-time AONE</i> scheme Υ	266
7.20	Security game OTAONE-IND for <i>one-time AONE</i> scheme Υ . . .	267
7.21	Security game OTAONE-AON for <i>one-time AONE</i> scheme Υ . .	267
7.22	Pictorial description of the <i>encryption</i> function $\hat{\Upsilon}.\mathcal{E}$	269
7.23	Algorithmic description of the <i>encryption</i> function $\hat{\Upsilon}.\mathcal{E}$	269
7.24	Pictorial description of the <i>decryption</i> function $\hat{\Upsilon}.\mathcal{D}$	270
7.25	Algorithmic description of the <i>decryption</i> function $\hat{\Upsilon}.\mathcal{D}$	271
7.26	Pictorial description of the <i>encryption</i> function $\tilde{\Upsilon}.\mathcal{E}$	273
7.27	Algorithmic description of the <i>encryption</i> function $\tilde{\Upsilon}.\mathcal{E}$	274
7.28	Pictorial description of the <i>decryption</i> function $\tilde{\Upsilon}.\mathcal{D}$	275
7.29	Algorithmic description of the <i>decryption</i> function $\tilde{\Upsilon}.\mathcal{D}$	275

List of Abbreviations

AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AONE	All-Or-Nothing Encryption
AONT	All-Or-Nothing Transform
APE	Authenticated Permutation-based Encryption
BFS	Breadth First Search
BL-MLE	Block-Level Message-Locked Encryption
CBC	Cipher Block Chaining mode of operation
CE	Convergent Encryption
CFB	Cipher Feedback mode of operation
CTR	Counter mode of operation
DAG	Directed Acyclic Graph
DFS	Depth First Search
ECB	Electronic Code Book mode of operation
FMLE	File-Updatable Message-Locked Encryption
HAC	Hierarchical Access Control
HCE2	Hash and Convergent Encryption 2
IoT	Internet-of-Things
KAS	Key Assignment Scheme
KAS-AE	Key Assignment Scheme with Authenticated Encryption
MLE	Message-Locked Encryption
OFB	Output Feedback mode of operation
PKC	Public Key Cryptosystem
PoW	Proof-of-ownership
poset	partially ordered set
RCE	Randomized Convergent Encryption

SE	Symmetric Encryption
UMLE	Updatable Block-Level Message-Locked Encryption

List of Symbols

Symbol	Operation
$:$	assignment
$=$	equality comparison
\oplus , XOR	<i>exclusive-or</i>
\parallel	<i>concatenation</i>
$ S $	<i>length</i> of string S or the <i>cardinality</i> of set S
$\{0, 1\}^\ell$	set of all binary strings of length ℓ
$\{0, 1\}^*$	set of all binary strings $\bigcup^{i \geq 0} \{0, 1\}^i$
$M \xleftarrow{\$} \{0, 1\}^k$	choosing an element from $\{0, 1\}^k$ according to the uniform distribution, and assigning it to M
\emptyset	empty set
\perp	invalid string
ϵ	empty string
$[s]$	the set $\{1, 2, \dots, s\}$
\mathbb{N}	the set of all <i>natural numbers</i>
$M[i]$	i -th block of message M
$M[i][j]$	j -th bit of i -th block of message M
$\lceil temp \rceil$	ceiling function on $temp$
\boldsymbol{M}	sequence of strings or vector of strings
$\boldsymbol{M}^{(i)}$	i -th string in \boldsymbol{M}
$\ \boldsymbol{M}\ $	number of strings in \boldsymbol{M}
$\boldsymbol{M}^{(0)} \circ \boldsymbol{M}^{(1)}$	sequence of two strings $\boldsymbol{M}^{(0)}$ and $\boldsymbol{M}^{(1)}$
$\{0, 1\}^{**}$	infinite set of all binary strings of any length
$(M, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda)$	assignment of outputs given randomly and uniformly by message-source \mathcal{S} to M and Z .
$G = (V, E)$	graph G with <i>vertex set</i> V and <i>edge set</i> E

$(\mathbf{M}^{(u)})_{u \in V}$	sequence of strings $\mathbf{M}^{(u)}$, where $u \in V$
$level[u]$	length of path from <i>root</i> node to node u in graph G
h	maximum-depth of the tree
u_j^i	j -th node from root in the <i>chain</i> C_i .

CHAPTER 1

Introduction

The *Internet*, defined as the network of interconnected autonomous communicating devices, has seen explosive growth in the last few decades. It is attributed to the fact that this *technology* has been able to cater to the ever-increasing needs of the human civilization, all the way. These needs have been addressed through innovative applications in the domains of e-commerce, e-governance, centralized/remote storage and access, secure computation, etc.

With the emergence of the *Internet-of-Things (IoT)*, lightweight devices have become of paramount importance, because of their potential use in it. Also, for increased availability and reduced storage costs pertaining to the *Internet* data, it is, nowadays, increasingly getting stored in the public storage, known as the *Cloud*. Being publicly accessible, the data in the *Cloud* needs to be administered with adequate caution. *IoT* is also intricately connected to the *Cloud*, since the former dumps data into the latter.

In this thesis, our prime focus is to design and analyse various cryptographic protocols and primitives, that have potential applications in the (lightweight) *IoT* devices and the *Cloud*. To that end, finally, we were able to design five primitives, namely, *file-updatable message-locked encryption (FMLE)*, *key assignment scheme with authenticated encryption (KAS-AE)*, *all-or-nothing transform (AONT)*, *all-or-nothing encryption (AONE)* and *one-time AONE*, with usefulness in the *IoT* devices and *Cloud*.

Interestingly, the design of our schemes substantially relies on a unique property, known as the *reverse decryption*, to be found in a handful of cryptographic schemes. We discuss this in the subsequent sections.

1.1 What is Reverse Decryption?

In the *reverse decryption* property, the direction of *decryption* is opposite to that of the *encryption*. It was first discovered by Andreeva *et al.* in the *APE* authenticated encryption (DIAC 2012, FSE 2014), and independently by Paul *et al.* in a very different *FP* hash mode of operation (Indocrypt 2012) [ABB⁺14, PHG12]. By virtue of *reverse decryption* property, unless the last ciphertext-block is available, it is impossible to recover even a single bit of the plaintext (except for a random guess). There is one more interesting feature: if a single bit is flipped in an intermediate ciphertext-block, then none of the preceding message-blocks can be decrypted successfully. Interestingly, except these two constructions, to the best of our knowledge, no other scheme exists with this property.

We would like to emphasize that the *reverse decryption* is necessarily an *offline decryption*, but not the other way round. In the *offline decryption* property of an *authenticated encryption* scheme, until the last ciphertext-block arrives at the receiver’s end, the verification of the entire plaintext is not completed, and for this reason, not even a single bit of plaintext is released [HRRV15]. For example, *Artemia*, *ASCON*, *STRIBOB*, *sp-AELM* etc. have the *offline decryption* property, but they do not possess the *reverse decryption* property [AAB15, ACS15, DEMS14, Saa14].

As mentioned earlier, the *reverse decryption* property is only found in *APE* and *FP*, both of which use a permutation as the basic building block. Usage of permutation-based constructions has gained importance and attention over the past decade, mainly because of the following reasons: a blockcipher often comes with a weak *key schedule* algorithm [DSST17, KSW96, Knu95]; when implemented in hardware, it usually requires a larger number of gates (this often becomes prohibitive in *lightweight* devices) [ABB⁺14, MP12]; also, from the theoretical standpoint, an ideal (block)cipher is a stronger assumption (than that of an ideal permutation) [Nat12]. Additionally, KECCAK, a permutation-based hash-function, has been selected as the winner of the SHA-3 standardization competition; this construction has exhibited several orders of magnitude more hardware throughput than all the competing blockcipher-based candidates. They have attributed such an advantage of KECCAK’s to its permutation-based structure.

However, we would like to point out that we do not claim that all the permutation-based primitives are better than the blockcipher-based ones. Importance of widely-used blockciphers, such as, *AES*, is undeniable. We, however, seek a fundamentally different permutation-based approach, where

certain blockciphers are not suitable to achieve specific goals. We want to emphasize that permutations have several advantages such as absence of *key schedule*, that may be exploited in various applications where blockciphers fall short.

1.2 Motivation

We derive our motivation from the following question:

Can we exploit the *reverse decryption* property to design new and efficient cryptographic primitives to solve various real-life problems?

Since, intuitively, the *reverse decryption* property can only be supported by permutation-based constructions, any primitive supporting the *reverse decryption* would automatically inherit the advantages of a permutation as well. Therefore, there is an added motivation to search for applications that can benefit from the *reverse decryption*.

1.3 Summary of the Thesis Work

Our core technical contributions can be divided into three parts: (1) definition and formalization of new (as well as existing) cryptographic primitives; (2) design of new constructions; and (3) determination of security bounds of the proposed constructions.

In the direction of (1), we did the following:

- We proposed three new cryptographic primitives, namely, *file-updatable message-locked encryption (FMLE)*, *key assignment scheme with authenticated encryption (KAS-AE)*, and *one-time all-or-nothing encryption (one-time AONE)*.
- We refined the definition of the cryptographic primitive *all-or-nothing transform (AONT)* by concretely specifying the parameters and correctness conditions (details in the subsequent sections).
- We gave a flexible definition of an existing cryptographic primitive, namely, *all-or-nothing encryption (AONE)*, by liberating it from being *always* constructed from the *AONT* and *SE* (details in the subsequent sections). Now, *AONE* can be constructed using primitives other than *AONT* and *SE*.

In the direction of (2), we have designed the following highly efficient schemes: 2 *FMLEs*, 3 *KAS-AEs*, 2 *AONTs*, 2 *AONEs* and 2 *one-time*

AONEs. With the exception of one *KAS-AE* construction based on *MLE*, all others are based on the *reverse decryption* property designed using easy-to-invert permutations.¹ In the subsequent sections, we have discussed them elaborately.

As regards (3), to compute the security bounds of our aforementioned constructions, we relied on the following ingredients: the *PRP/PRF switching lemma*, the *code-based game playing technique*, and the *triangle inequality*. The initial segment of each proof follows a common framework, where: at the first step, the permutation π (in the construction) is replaced by a random function rf ; and then the permutation π^{-1} is replaced by a random function rf' . The adversarial advantage for this step is computed using the *PRP/PRF switching lemma*, depending on the number of queries to the permutation and the random function. After this step, we design a sequence of pairs of games, where each pair behaves *identically* until the *bad* flag is set; this is exactly what is required in a proof based on *code-based game playing technique*. Now, we compute the probability of the event when the *bad* flag is set for all pairs in the sequence one by one, and then sum them all up to get the final security bound.

1.4 Main Technical Results

In this section, we give a brief overview of the main technical results. Their full descriptions can be found in Chapters 4, 5, 6 and 7.

1.4.1 File-Updatable Message-Locked Encryption (FMLE)

We have conceptualized and formalized a new cryptographic primitive called *FMLE*; it is a special variant of an already existing cryptographic primitive *message-locked encryption (MLE)*; *FMLE* is *MLE* with two additional functions, namely, *update* and *proof-of-ownership (PoW)*. On the other hand, *FMLE* can be viewed as the generalization of another cryptographic

¹Our permutation is not implemented using a table lookup. Instead, the output is computed by processing an input using an algorithm, say π , and its inverse using another algorithm, say π^{-1} . We claim that both π and π^{-1} are easy-to-compute. A good example of such (π, π^{-1}) is KECCAK/SHA-3 permutation. An easy-to-invert permutation should be contrasted with a hard-to-invert permutation, such as, $\alpha \rightarrow \alpha^g$, where it is hard to determine α from α^g . Note that we only claim that (π, π^{-1}) are easy-to-compute, but this does not mean that the permutation has structural simplicity leading to security vulnerabilities. Designing an easy-to-invert permutation with various cryptographic properties is a non-trivial task.

primitive, namely, *efficiently updatable block-level message-locked encryption (UMLE)*, which itself is a variant of *MLE*. However, the main difference is that the latter is necessarily based on *block-level message-locked encryption (BL-MLE)*, but the former may or may not be [ZC17]. Effectively, we remove the constraint of *always* using *BL-MLE* to design *UMLE*. This allows for *UMLE* constructions that do not necessarily follow the existing rigid framework, and, eventually, results in efficient *FMLE* schemes.

We have designed two novel permutation-based *FMLE* constructions, exploiting the *reverse decryption* property of *APE* authenticated encryption and *FP* hash mode of operation. Our schemes have been proven to be secure under the following security notions: *privacy*, *tag consistency*, *context hiding*, and *uncheatability*. Since the existing *UMLE* schemes are intrinsically parallel algorithms, where all blocks can be processed independently using the underlying *BL-MLE* scheme, the resistance to the *dictionary attack* is limited to the entropy of a single block. In contrast, our *FMLE* constructions are randomized and inherently sequential (by the virtue of the *reverse decryption* property); therefore, they are secure against *dictionary attacks*.

On comparison, our proposed schemes outperform all the existing schemes – both with respect to memory requirements and execution costs.

1.4.2 Key Assignment Scheme with Authenticated Encryption (KAS-AE)

We have proposed a brand new cryptographic primitive, namely, *KAS-AE*, which can be regarded as the existing cryptographic primitive *key assignment scheme (KAS)* endowed with an additional function of *authenticated encryption (AE)*. The *KAS* primitive has been used to design a protocol to implement *hierarchical access control (HAC)*; *HAC* is a mechanism that allows the classes of people in an organisation with varying levels of privileges to access data based on their positions. We observe that *HAC* protocol has, so far, been designed using the *KAS* primitive which lacks the authentication mechanism. We first combine *KAS* and *AE* in a novel way to design a new *KAS-AE* primitive, which is faster than the trivial combination of *KAS* and *AE*, and demonstrate how it will be effectively used in the *HAC* protocol.

To motivate this new idea, we have proposed a total of eight secure *KAS-AE* constructions with varying degrees of efficiencies and construction subtleties: (1) in the first construction, we show a secure way of combining *KAS* and *AE* independently to build a *KAS-AE* scheme; (2-5) our next

four *KAS-AE* constructions are based on first building *KAS-AE* schemes for linear graphs (or totally ordered sets), and then combining them to support arbitrary access graphs (i.e. partially ordered sets); and (6-8) the last three constructions are the most efficient *KAS-AE* constructions that are based on a novel use of *MLE*, *APE* and *FP*, respectively.

Our schemes are secure under all the proposed security notions of *KAS-AE*, namely, *key recovery*, *privacy*, and *tag consistency*. On comparison, our schemes outperform all the existing schemes, both with respect to memory requirements and execution costs.

1.4.3 All-Or-Nothing Transform (AONT)

We have refined the definition of an already existing cryptographic primitive *AONT*. The notion of *AONT* was first conceived by Rivest more than two decades ago (1997), to be used as a preprocessing step to enhance the security of conventional *blockcipher-based symmetric encryption (SE)* against the *key recovery* attack [Riv97]. In such an attack, the adversary exploits the property that, on the decryption side, a plaintext block can be viewed as a function of just *one* (or at most two) ciphertext block(s), rather than a function of *all* of them. The security property, called *all-or-nothing (AON)*, where the recovery of each plaintext bit requires the knowledge of *all* the ciphertext bits, is responsible for enhancing the security of various real-life applications.

We enrich the existing definition in the following way: we concretely define the scope and the usage of the *setup* algorithm; we clearly parameterize the *message-space* and the *pseudo-message-space*; we quantify the pseudo-message expansion; and, most importantly, we establish the various correctness properties.

Interestingly, we observed that all the practical *AONT* schemes have, so far, been constructed using blockciphers [Des00, Riv97]. This thesis explores how to design efficient *AONT* constructions based on a fixed permutation, and, for the first time, describes two concrete permutation-based *AONTs*. We gave security bounds for all the schemes against various security properties. We observe that our schemes outperform all the existing ones, both with respect to memory requirements and running-times.

1.4.4 All-Or-Nothing Encryption (AONE)

AONE is an existing cryptographic primitive, proposed more than three decades ago. Since its inception, it has always been defined in terms of *AONT* and *symmetric encryption (SE)*. We, for the first time, define *AONE*

as an independent cryptographic primitive, where it is not necessary to construct it from $AONT$ and SE ; thereby, our $AONE$ definition is much more flexible. From a high level, $AONE$ has three algorithms: a *key generation* algorithm, that generates a key for the encryption and decryption; an *encryption* algorithm, that takes the pair of key and message as input, and outputs a ciphertext, which is computed in a way that enforces the *all-or-nothing* property; and a *decryption* algorithm, which decrypts the ciphertext into a message in such a way that if a few bits in the ciphertext are missing or incorrect, then the original message will not be recovered (with a high probability). The security properties of $AONE$ are amalgamation of that of $AONT$ and SE .

Next, we proposed concrete constructions of permutation-based $AONEs$. We compute the security bounds for each construction for all the security notions. We have given a detailed comparison of all the existing and proposed $AONE$ schemes, where we observe that our schemes outperform the existing ones, both with respect to memory requirements and running-times.

1.4.5 One-Time AONE

We have conceptualized and designed a novel cryptographic primitive that does not require any *encryption key*, nevertheless, generates the decryption key that is message-dependent. We call this primitive *one-time AONE* because the decryption key changes every encryption. This primitive is very unique as it supports various conflicting properties: like a *public key cryptosystem (PKC)*, it is inherently an asymmetric cipher, since the encryption/decryption keys are always different; on the other hand, unlike *PKC*, its decryption key is dependent on the message; although its security properties are *identical* to that of $AONE$, very oddly, unlike $AONE$, it does not require any (encryption) *key generation* algorithm, making this primitive a suitable candidate to be used in applications, where the secure storage of secret encryption key is an issue. Since in *one-time AONE*, a fresh decryption key is generated every encryption, thus, the differential attack is significantly mitigated.

Our *one-time AONE* constructions are based on a fixed permutation, instead of a blockcipher. These constructions are unique in a way that they do not require any *encryption key*. We achieve this property by using the *reverse decryption* mechanism. Based on our extensive analysis, designing such *one-time AONE* constructions without the *reverse decryption* property seems hard.

1.5 Organization of the Thesis

In Chapter 2, we discuss the fundamental concepts in cryptology along with some basic cryptographic primitives. Then we discuss some advanced cryptographic protocols, primitives and tools in Chapter 3. In Chapter 4, we elaborately describe a new cryptographic primitive *FMLE* and its two highly efficient constructions. Then, we propose a new cryptographic primitive *KAS-AE* and give nine constructions of it in Chapter 5. In Chapter 6, we enhance the definition of an existing cryptographic primitive *AONT* and give two permutation-based constructions for *AONT*. In Chapter 7, we describe *AONE* and its variant *one-time AONE*, and design two highly efficient constructions for each of the two primitives. Finally, we conclude in Chapter 8 outlining various ways to extend the work of this thesis.

CHAPTER 2

Cryptologic Background

Nowadays, *Internet* data is stored in the public storage, and is, therefore, vulnerable to unauthorized modifications and access by the attackers. *Cryptologic primitives* and *protocols* address these security issues.

In this chapter, we give a brief introduction to *Cryptology*, discuss the most basic algorithms of it, and finally provide some mathematical foundation on how to quantify cryptographic security bounds.

2.1 Introduction to Cryptology

Cryptology is the science of designing and protecting systems which aim to hide information from unintended users and making it available to legitimate ones [Til99]. The word *cryptology* is derived from two Greek words *kryptos* and *logos*, meaning “hidden” and “study”, respectively [Sim06]. The history of *cryptology* dates back to about 1900 BC, when Egyptians used the *hieroglyphs* to hide information. Since then, for over thousands of years, cryptologic techniques have been regarded as an art. However, the modern cryptology has gradually emerged as a rigorous scientific and engineering discipline, involving mathematics, computer science, electrical engineering, communication science, and physics, among others, for protecting data and identity in all major applications on the *Internet*, as well as in the digital world [Wik19a]. Based on the nature of activities, *cryptology* has been classified into *cryptography* and *cryptanalysis*; the former deals with the design of cryptosystems, while the latter devotes to attacking them.

2.1.1 Cryptography

Cryptography is the branch of *cryptology* that deals with the designing of cryptosystems. It aims to establish mechanisms for the secure communication between two (or more) parties over an insecure channel (such as a computer network or a telephone line) preventing the unintended users, called *adversaries*, to gain access to this information. Besides this, with the evolution of the Internet technology and the diverse applications running on it, these cryptosystems have encompassed numerous other mechanisms such as assuring data integrity, key exchange, key agreement, user authentication, and secret sharing, among others [KL07].

Goals of Cryptography

There are three basic goals of *cryptography* [Til99]. They are explained below.

1. **Confidentiality.** The prime focus of *confidentiality* is to prevent any *adversary* from deducing any information from the encoded message in the insecure channel or the storage media. It is usually achieved using *encryption schemes*, and is integrated into the cryptosystem by the *privacy* security notion.
2. **Integrity.** The *integrity* property targets to safeguard the data (whether stored or being transmitted) from any unintentional modifications during transmission or intentional tampering by an adversary. Use of *hash function* ensures integrity of data.
3. **Authentication.** The purpose of *authentication* is to prove the identity of the sender to the receiver. In other words, by the virtue of *authentication* property, the sender, at a later stage, cannot deny the fact that the data received was sent by him/her. Employment of *message authentication codes* or *digital signatures* ensure the *authentication* property. It is integrated into the cryptosystem by the *tag consistency* security notion.

2.1.2 Cipher

A *cipher* is a collection of cryptographic algorithms that realize a specific security service [Wik19b]. Generally, a *cipher* targets the *privacy* security notion. It consists of three algorithms – *key generation*, *encryption* and *decryption* – and the associated sets of *key-space*, *message-space* and

ciphertext-space. The decryption key of a *cipher* always remains secret to everybody except the decryptor, therefore, it is called the *secret key* of the cipher.

Symmetric and Asymmetric Cipher

A *cipher*, where the encryption and decryption keys are identical, is called a *symmetric cipher*. Some of the most popular examples of *symmetric ciphers* are *Data Encryption Standard (DES)*, *triple DES (3DES)*, and *Advanced Encryption Standard (AES)* [BDC05, DC05, MVM09].

In an *asymmetric cipher*, the encryption and decryption keys are different; the encryption key is made public, and is therefore, known as the *public key*; the decryption key remains secret, and is called *private key*. In this case, the *key generation* algorithm always generates a pair of (*public, private*) keys. Some of the most popular examples of *asymmetric ciphers* are *RSA encryption*, *Schnorr signature*, and *El-Gamal encryption*.

Blockcipher

The pictorial description of the *blockcipher* is given in Figure 2.1. Below we elaborately discuss the syntax and correctness of a *blockcipher*.

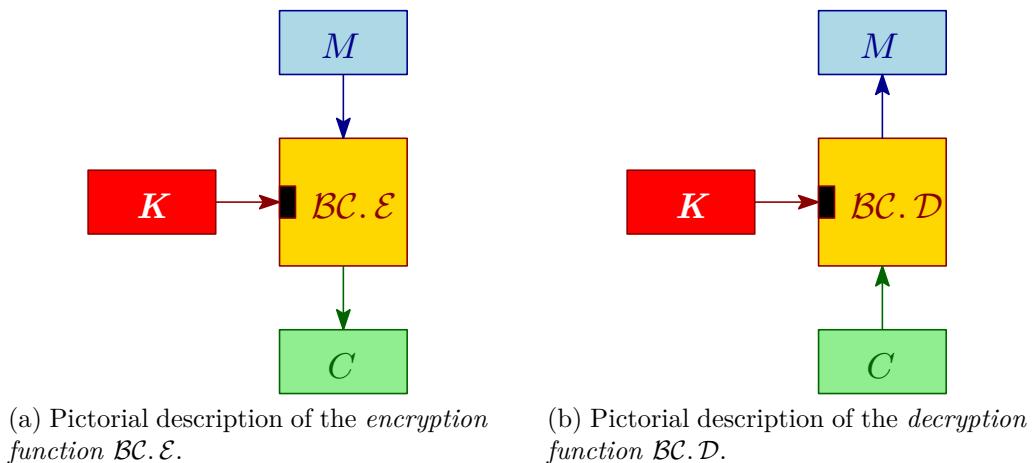


Figure 2.1: Pictorial description of the *blockcipher* \mathcal{BC} .

SYNTAX. Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *blockcipher* $\mathcal{BC} = (\mathcal{BC}.\mathcal{E}, \mathcal{BC}.\mathcal{D})$ is a pair of algorithms over the *setup* algorithm $\mathcal{BC}.\text{Setup}$, satisfying the following conditions. The blockcipher \mathcal{BC} with the security parameter λ is associated with the *key-space* $\mathcal{K}^{(\mathcal{BC})} = \{0, 1\}^\lambda$, *message-space* $\mathcal{M}^{(\mathcal{BC})} = \{0, 1\}^\lambda$ and *ciphertext-space* $\mathcal{C}^{(\mathcal{BC})} = \{0, 1\}^\lambda$.

1. The PPT *setup* algorithm $\mathcal{BC}.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\mathcal{BC})}$ and a key $K \in \mathcal{K}^{(\mathcal{BC})}$.
2. The PPT *encryption* algorithm $\mathcal{BC}.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\mathcal{BC})}$, key $K \in \mathcal{K}^{(\mathcal{BC})}$ and message $M \in \mathcal{M}^{(\mathcal{BC})}$, and returns ciphertext $C := \mathcal{BC}.\mathcal{E}(\text{params}^{(\mathcal{BC})}, K, M)$, where $C \in \mathcal{C}^{(\mathcal{BC})}$. The pictorial description is given in Figure 2.1(a).
3. The *decryption* algorithm $\mathcal{BC}.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\mathcal{BC})}$, key $K \in \mathcal{K}^{(\mathcal{BC})}$ and ciphertext $C \in \mathcal{C}^{(\mathcal{BC})}$, and returns message $M := \mathcal{BC}.\mathcal{D}(\text{params}^{(\mathcal{BC})}, K, C)$, where $M \in \mathcal{M}^{(\mathcal{BC})}$. The pictorial description is given in Figure 2.1(b).

CORRECTNESS. This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the correctness of a *blockcipher* can be defined as follows.

Suppose $(\text{params}^{(\mathcal{BC})}, K) := \mathcal{BC}.\text{Setup}(1^\lambda)$. Then, the *correctness* of \mathcal{BC} requires that $\mathcal{BC}.\mathcal{D}(\text{params}^{(\mathcal{BC})}, K, \mathcal{BC}.\mathcal{E}(\text{params}^{(\mathcal{BC})}, K, M)) = M$, for all $(\lambda, K, M) \in \mathbb{N} \times \mathcal{K}^{(\mathcal{BC})} \times \mathcal{M}^{(\mathcal{BC})}$.

Modes of operation: ciphers using a blockcipher

To allow a *blockcipher* to operate on a message of arbitrary length, a *mode of operation* is used [Sti95].

- **Electronic Code Book (ECB) Mode.** In this mode of operation, after parsing the message into fixed-length blocks, each message-block is encrypted using a *blockcipher* with the identical key. This mode is vulnerable to *dictionary attack*, because of the lack of *diffusion*. The pictorial description is given in Figure 2.2. Mathematically, *ECB* encryption of message $M[i]$ under key K using the blockcipher \mathcal{BC} to obtain ciphertext $C[i]$, can be expressed as $C[i] := \mathcal{BC}.\mathcal{E}(K, M[i])$.
- **Cipher Block Chaining (CBC) Mode.** In this mode of operation, after parsing the message into fixed-length blocks, for each block in the sequence, the following operations are performed: first the message-block is XORed with the previous ciphertext-block; and then it is encrypted using a *blockcipher* under the identical key. For the encryption of the first message-block, initialization value *IV* is considered as the preceding ciphertext-block. The pictorial description is given in Figure 2.3. Mathematically, *CBC* encryption of message $M[i]$ under key K using the blockcipher \mathcal{BC} to obtain ciphertext $C[i]$, can be

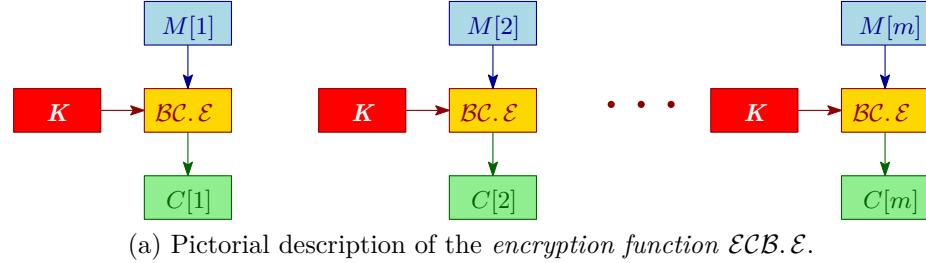
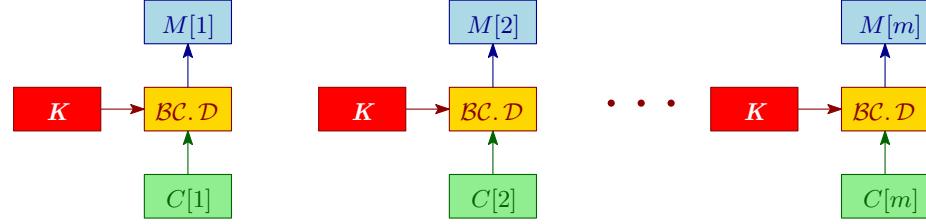
(a) Pictorial description of the *encryption function* $\mathcal{ECB} \cdot \mathcal{E}$.(b) Pictorial description of the *decryption function* $\mathcal{ECB} \cdot \mathcal{D}$.

Figure 2.2: Pictorial description of ECB Mode.

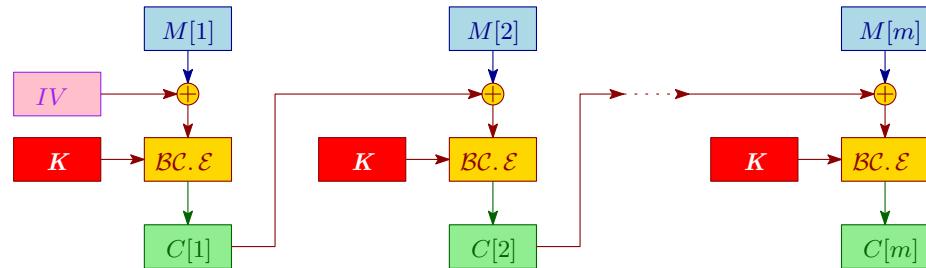
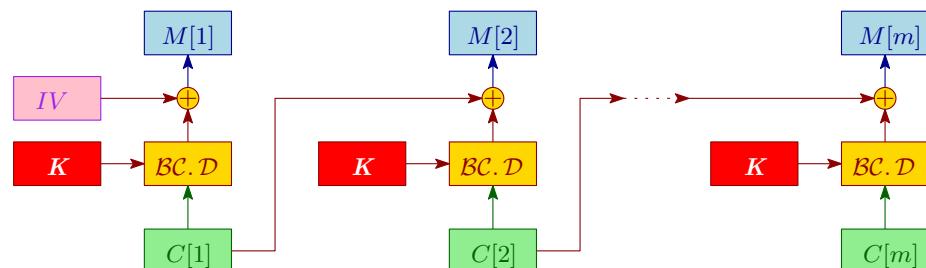
(a) Pictorial description of the *encryption function* $\mathcal{CBC} \cdot \mathcal{E}$.(b) Pictorial description of the *decryption function* $\mathcal{CBC} \cdot \mathcal{D}$.

Figure 2.3: Pictorial description of CBC Mode.

expressed as $C[i] := \mathcal{BC} \cdot \mathcal{E}(K, (C[i-1] \oplus M[i]))$, where $C[i-1]$ is the preceding ciphertext-block.

- **Cipher Feedback (CFB) Mode.** In this mode of operation, after parsing

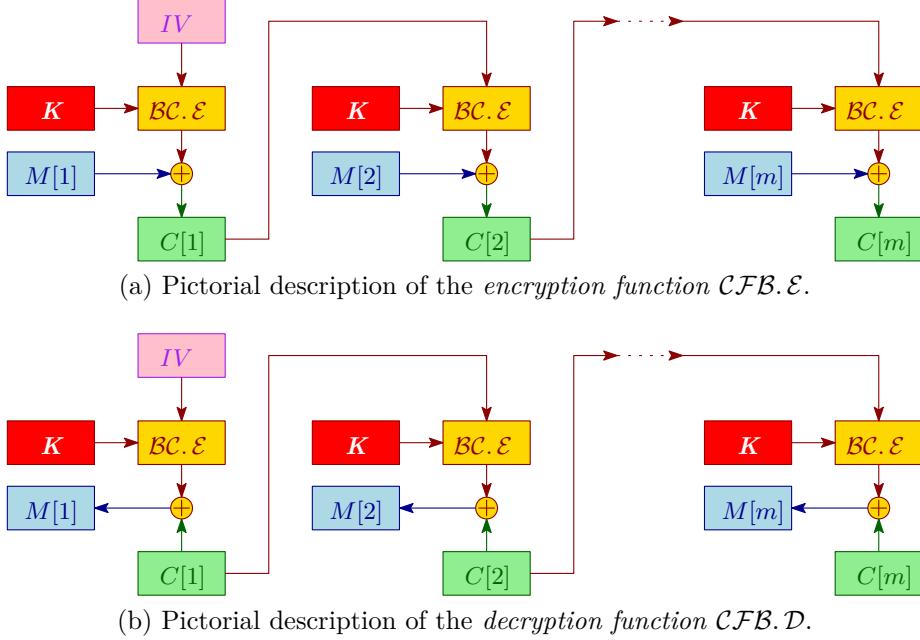


Figure 2.4: Pictorial description of CFB Mode.

the message into fixed-length blocks, for each message-block in the sequence, the following operations are performed: first the preceding ciphertext-block is encrypted using a *blockcipher* under the identical key; and then its output is XORed with the message-block to obtain the ciphertext-block. For the encryption of the first block of message, the preceding ciphertext-block is the initialization value IV . The pictorial description is given in Figure 2.4. Mathematically, *CFB* encryption of message $M[i]$ under key K using the blockcipher \mathcal{BC} to obtain ciphertext $C[i]$, can be expressed as $C[i] := M[i] \oplus \mathcal{BC.E}(K, C[i-1])$, where $C[i-1]$ is the preceding ciphertext-block.

- **Output Feedback (OFB) Mode.** In this mode of operation, after parsing the message into fixed-length blocks, for each message-block in the sequence, the following operations are performed: first the output of previous *blockcipher* is encrypted using a *blockcipher* under the identical key; and then this output is XORed with message-block to obtain the ciphertext-block. For the first message-block, initialization value IV is considered as output of previous *blockcipher*. The pictorial description is given in Figure 2.5. Mathematically, *OFB* encryption of message $M[i]$ under key K using the blockcipher \mathcal{BC} to obtain ciphertext $C[i]$, can be expressed as $C[i] := M[i] \oplus \mathcal{BC.E}(K, (M[i] -$

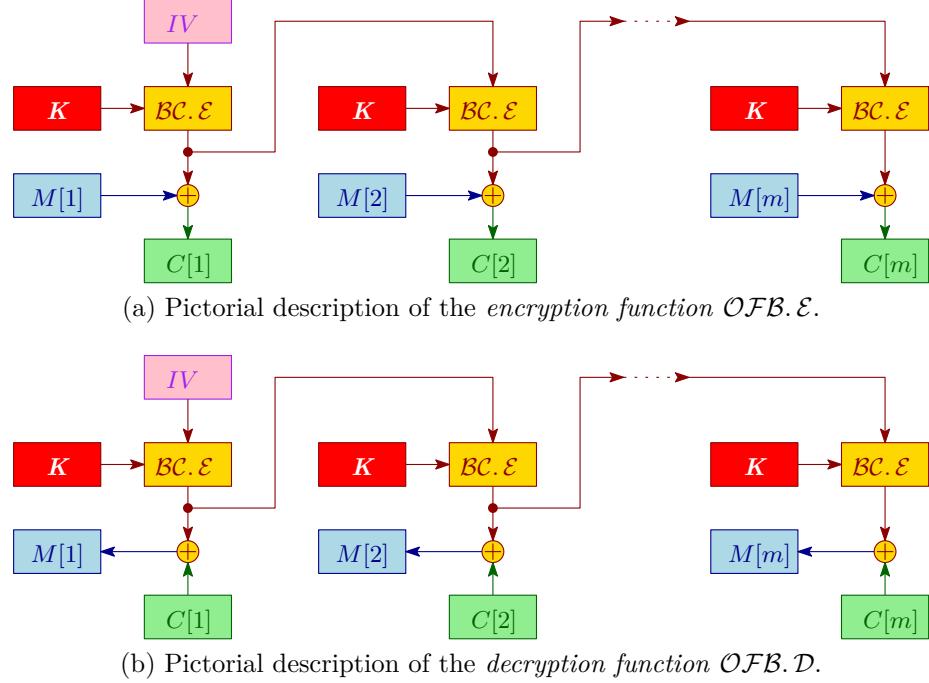


Figure 2.5: Pictorial description of OFB Mode.

$1] \oplus C[i-1])$, where $M[i-1]$ and $C[i-1]$ are the preceding message-block and ciphertext-block respectively.

- **Counter (CTR) Mode.** In this mode of operation, after parsing the message into fixed-length blocks, for each message-block in the sequence, the following operations are performed: first the counter value is incremented; then the counter is encrypted under the identical key; and finally this output is XORed with message-block to obtain the ciphertext-block. The pictorial description is given in Figure 2.6. Mathematically, CTR encryption of message $M[i]$ under key K using the blockcipher \mathcal{BC} to obtain ciphertext $C[i]$, can be expressed as $C[i] := M[i] \oplus \mathcal{BC}.\mathcal{E}(K, (CTR + i - 1))$, where CTR is the counter variable.

2.1.3 Cryptanalysis: Types of Attacks

Based on the amount of information available to the adversary, we classify cryptanalytic attacks on the *cipher* into the following four categories [Sti95]. They are as follows:

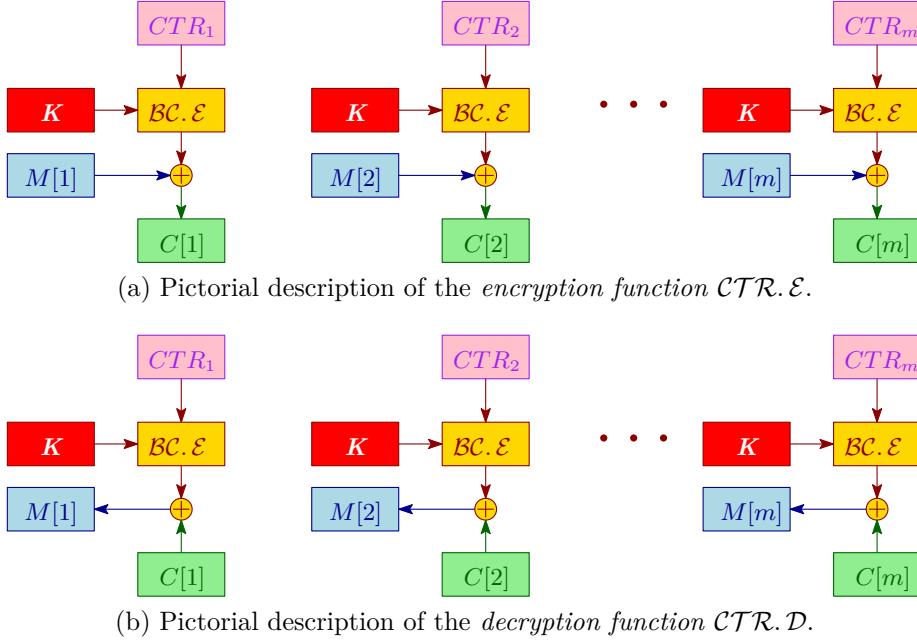


Figure 2.6: Pictorial description of CTR Mode.

1. **Ciphertext-only attack (COA).** This is the most fundamental attack, where the adversary is given the least information possible. In this attack, the adversary is supplied with the ciphertext, and then he/she tries to deduce the corresponding plaintext.
2. **Known-plaintext attack (KPA).** This attack endows adversary with more power than COA. The adversary is given access to pairs of plaintext and ciphertext. The adversary's challenge is to determine information about the underlying plaintext or some other ciphertext.
3. **Chosen-plaintext attack (CPA).** This attack endows adversary with more power than KPA. The adversary is now given access to encryption *oracle* with the limited number of queries.
4. **Chosen-ciphertext attack (CCA).** This attack endows adversary with the maximum power. The adversary is given access to both encryption and decryption *oracles*, with the limited number of queries for each of them.

2.1.4 A Note on the Security Parameter λ

The security parameter λ , is a tunable parameter that demarcates the level or amount of security (in bits) in a cryptographic primitive. Usually, for the conventional applications, the value of λ is 128, 192 or 256 [Nat01]. For the applications requiring higher security levels, λ can be set to 512, 1024 or even higher values [Nat15]. For lightweight cryptography, the desired value of λ is 80, 96 or 112 [Nat17].

The λ is one of the independent parameters in a primitive, and has to be supplied, right at the first step, while the primitive is being initialized using the *setup* algorithm. The *setup* algorithm, using λ , computes all other parameters (e.g. block-size, tag-size, key-size etc.) needed to achieve the λ -bit security, which are, in turn, then given to all other algorithms for correct computation. For example, one parameter given to the *encryption* algorithm \mathcal{E} (or to *decryption* algorithm \mathcal{D}) is the plaintext (or ciphertext) which is concatenation of certain blocks; here, the block-size is computed by the *setup* algorithm using λ .

2.2 Basic Cryptographic Primitives

In this section, we discuss a few basic cryptographic primitives (both theoretical and practical) used for building applications, as well as for security analysis.

2.2.1 Ideal Permutation

Let $\pi/\pi^{-1}: \{0, 1\}^n \mapsto \{0, 1\}^n$ be a pair of oracles. The pair π/π^{-1} is called an *ideal permutation* if the following three properties are satisfied.

1. $\pi^{-1}(\pi(x)) = x$ and $\pi(\pi^{-1}(x)) = x$, for all $x \in \{0, 1\}^n$.
2. Suppose, x_k is the k -th query ($k \geq 1$), submitted to the oracle π , and $y \in \{0, 1\}^n$. Then, for the current query x_i :

$$\Pr \left[\pi(x_i) = y \mid \pi(x_1) = y_1, \pi(x_2) = y_2, \dots, \pi(x_{i-1}) = y_{i-1} \right]$$

$$= \begin{cases} 1, & \text{if } x_i = x_j, y = y_j, j < i. \\ 0, & \text{if } x_i = x_j, y \neq y_j, j < i, \\ 0, & \text{if } x_i \neq x_j, y = y_j, j < i, \\ \frac{1}{2^{n-i+1}}, & \text{if } x_i \neq x_j, y \neq y_j, j < i. \end{cases}$$

3. Suppose, y_k is the k -th query ($k \geq 1$), submitted to the oracle π^{-1} , and $x \in \{0, 1\}^n$. Then, for the current query y_i :

$$\Pr \left[\pi^{-1}(y_i) = x \middle| \pi^{-1}(y_1) = x_1, \pi^{-1}(y_2) = x_2, \dots, \pi^{-1}(y_{i-1}) = x_{i-1} \right]$$

$$= \begin{cases} 1, & \text{if } y_i = y_j, x = x_j, j < i. \\ 0, & \text{if } y_i = y_j, x \neq x_j, j < i, \\ 0, & \text{if } y_i \neq y_j, x = x_j, j < i, \\ \frac{1}{2^{n-i+1}}, & \text{if } y_i \neq y_j, x \neq x_j, j < i. \end{cases}$$

2.2.2 Random Function

Let the oracle $\text{rf}: \{0, 1\}^n \mapsto \{0, 1\}^n$. Then rf is called a *random function* if the following property is satisfied. Suppose, x_k is the k -th query ($k \geq 1$), submitted to the oracle rf , and $y \in \{0, 1\}^n$. Then, for the current query x_i :

$$\Pr \left[\text{rf}(x_i) = y \middle| \text{rf}(x_1) = y_1, \text{rf}(x_2) = y_2, \dots, \text{rf}(x_{i-1}) = y_{i-1} \right]$$

$$= \begin{cases} 1, & \text{if } x_i = x_j, y = y_j, j < i. \\ 0, & \text{if } x_i = x_j, y \neq y_j, j < i, \\ \frac{1}{2^n}, & \text{if } x_i \neq x_j, j < i. \end{cases}$$

2.2.3 Hash Function

Below we elaborately discuss the syntax, correctness and security definition of the *Hash function*.

2.2.3.1 Definition

Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *hash function* $\vartheta = (\vartheta, \mathcal{H})$ is an algorithm over the *setup* algorithm ϑ . **Setup**, satisfying the following conditions.

1. The PPT *setup* algorithm ϑ . **Setup**(1^λ) outputs parameter $\text{params}^{(\vartheta)}$, *message-space* $\mathcal{M}^{(\vartheta)} \subseteq \{0, 1\}^*$ and *digest-space* $\mathcal{T}^{(\vartheta)} \subseteq \{0, 1\}^*$.
2. The *hash* algorithm ϑ . $\mathcal{H}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\vartheta)}$ and message $M \in \mathcal{M}^{(\vartheta)}$, and returns hash value $H := \vartheta$. $\mathcal{H}(\text{params}^{(\vartheta)}, M)$, where $H \in \mathcal{T}^{(\vartheta)}$.

Correctness

This condition requires that the hash value generated from two identical messages will always be identical. Mathematically, the correctness of a *hash function* can be defined as follows.

The *correctness* of ϑ requires that for all $(\lambda, M, M') \in \mathbb{N} \times \mathcal{M}^{(\vartheta)} \times \mathcal{M}^{(\vartheta)}$, if $M = M'$, then $\vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M) = \vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M')$.

Security definitions

In Figure 2.7, we define the security game **HF-CR** for the *hash function* $\vartheta = (\vartheta, \mathcal{H})$. The game is written using the challenger-adversary framework.

Game **HF-CR** $_{\vartheta}^{\mathcal{A}}(1^\lambda)$

 $(\text{params}^{(\vartheta)}, \mathcal{M}^{(\vartheta)}, \mathcal{T}^{(\vartheta)}) := \vartheta.\text{Setup}(1^\lambda);$
 $(M_0, M_1) := \mathcal{A}(1^\lambda);$
 $H_0 := \vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M_0);$
 $H_1 := \vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M_1);$
If $(H_0 = H_1) \wedge (M_0 \neq M_1)$, **then** return 1;
Else return 0;

Figure 2.7: Security game **HF-CR** for the *hash function* ϑ .

HF-CR SECURITY. This security is dedicated to *collision-resistance* property. The adversarial advantage is the probability of the event when the adversary generates two unidentical messages with identical hash values, and it is desired to be negligible.

According to the **HF-CR** game, as defined in Figure 2.7, the adversary first returns two messages M_0 and M_1 , and then the following operations are performed: first it computes the hash value $H_0 := \vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M_0)$ for M_0 ; and then it computes the hash value $H_1 := \vartheta.\mathcal{H}(\text{params}^{(\vartheta)}, M_1)$ for M_1 . The game returns 1, if $H_0 = H_1$ under $M_0 \neq M_1$.

The advantage of the **HF-CR** adversary \mathcal{A} against ϑ is defined as follows:

$$\text{Adv}_{\vartheta, \mathcal{A}}^{\text{HF-CR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{HF-CR}_{\vartheta}^{\mathcal{A}}(1^\lambda) = 1].$$

The ϑ is **HF-CR** secure, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\vartheta, \mathcal{A}}^{\text{HF-CR}}(1^\lambda)$ is *negligible*.

2.2.3.2 FP hash mode of operation

FP hash mode of operation was proposed by Paul, Homsirikamol and Gaj in *Indocrypt 2012* [PHG12]. FP mode is derived from the Fast Wide Pipe (FWP) hash mode of operation, by replacing the *hard-to-invert function* of the FWP mode by an *easy-to-invert permutation* [NP10]. This results in improved efficiency and easier design.

The pictorial description and pseudocode for the algorithm in $\text{FP} = (\text{FP}, \mathcal{H})$ over the *setup* algorithm $\text{FP}.\text{Setup}$ are given in Figures 2.8 and 2.9; all wires are λ -bit long. Below we give the textual description.

- The *setup* algorithm $\text{FP}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\text{FP})}$, message-space $\mathcal{M}^{(\text{FP})} = \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and digest-space $\mathcal{T}^{(\text{FP})} = \{0, 1\}^\lambda$.

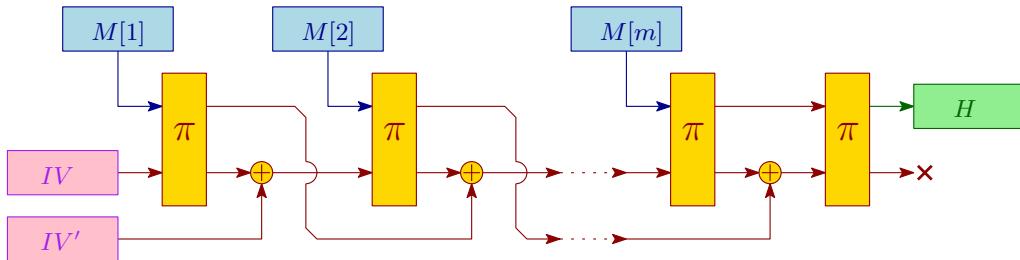


Figure 2.8: Pictorial description of the *hash* function $\text{FP}.\mathcal{H}$.

$\text{FP}.\mathcal{H}(\text{params}^{(\text{FP})}, M)$

```

#Initialisation.
 $IV := IV' := 0^\lambda; \quad s := IV; \quad t := IV';$ 
 $m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$ 

#Processing message blocks.
for ( $j := 1, 2, \dots, m$ )
   $r\|s := \pi(M[j]\|s); \quad s := s \oplus t; \quad t := r;$ 
   $H\|s := \pi(r\|s);$ 

#Computing final output.
return  $H;$ 
  
```

Figure 2.9: Algorithmic description of the *hash* function $\text{FP}.\mathcal{H}$.

- The *hash* algorithm $\text{FP}.\mathcal{H}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\text{FP})}$ and message M , and outputs hash value H . The pictorial description and pseudocode are given in Figures 2.8 and 2.9. Very briefly, the algorithm works as follows: first it initialises s and t with IV and IV' (here, $IV := IV' := 0^\lambda$); and then it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$. Now, the value of H is computed iteratively, as j runs through $1, 2, \dots, m$ in the following way: first it computes $r \parallel s := \pi(M[j] \parallel s)$; then it computes $s := s \oplus t$; and finally it assigns $t := r$. After this, it computes $H \parallel s := \pi(r \parallel s)$.

Now, the hash value H , computed above, is returned.

Security of FP construction

FP mode is secure against *collision*, *second-preimage*, and *herding attacks*, since it is indistinguishable from *random oracle* up to approximately $2^{\lambda/2}$ queries. For the HF-CR security defined in Figure 2.7, we have:

$$\text{Adv}_{\text{FP}, \mathcal{A}}^{\text{HF-CR}}(1^\lambda, m) \leq \frac{28 \cdot m^2}{2^\lambda}.$$

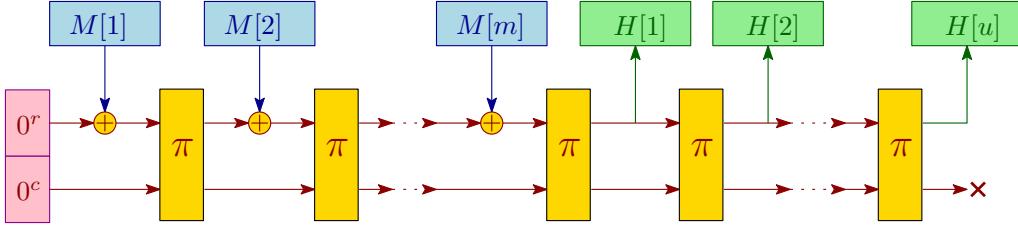
where, m is total number of block queries that the adversary makes.

2.2.3.3 Sponge hash function

Sponge hash function was introduced by Bertoni *et al.* in *ECRYPT Hash Workshop 2007*, that uses permutation as the basic building block [BDPA07]. It takes a message of arbitrary length as input, and returns a hash value of desired length. Here, for simplicity, we are considering the messages and hash values that are multiple of r -bit block-length.

The pictorial description and pseudocode for the algorithm in $\text{Sponge} = (\text{Sponge}.\mathcal{H})$ over the *setup* algorithm $\text{Sponge}.\text{Setup}$ are given in Figures 2.10 and 2.11. Below we give the textual description.

- The *setup* algorithm $\text{Sponge}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\text{Sponge})}$, *message-space* $\mathcal{M}^{(\text{Sponge})} = \bigcup_{i \geq 1} \{0, 1\}^{ir}$ and *digest-space* $\mathcal{T}^{(\text{Sponge})} = \bigcup_{i \geq 1} \{0, 1\}^{ir}$. The variables rate $r \in \mathbb{N}$ and capacity $c \in \mathbb{N}$ are included in parameter $\text{params}^{(\text{Sponge})}$.
- The *hash* algorithm $\text{Sponge}.\mathcal{H}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\text{Sponge})}$ and message M , and outputs hash value H . The pictorial description and pseudocode are given in Figures 2.10

Figure 2.10: Pictorial description of the hash function Sponge. \mathcal{H} .

```

Sponge.  $\mathcal{H}(params^{(Sponge)}, M)$ 
#Initialisation.
 $V_r := 0^r; V_c := 0^c;$ 
 $m := |M|/r; M[1]\|M[2]\|\dots\|M[m] := M;$ 

#Processing message blocks.
for ( $j := 1, 2, \dots, m$ )
 $V_r\|V_c := \pi((V_r \oplus M[i])\|V_c);$ 

#Computing hash blocks.
for ( $j := 1, 2, \dots, u$ )
 $H[j] := V_r; V_r\|V_c := \pi(V_r\|V_c);$ 

#Computing final output.
return  $H := H[1]\|H[2]\|\dots\|H[u];$ 

```

Figure 2.11: Algorithmic description of the hash function Sponge. \mathcal{H} .

and 2.11. Very briefly, the algorithm works as follows: first it initialises V_r and V_c with 0^r and 0^c ; then it parses M into r -bit blocks $M[1]\|M[2]\|\dots\|M[m] := M$; after that the value of H is computed iteratively, as j runs through $1, 2, \dots, m$ by computing $V_r\|V_c := \pi((V_r \oplus M[i])\|V_c)$; and finally it generates the ur -bit hash value iteratively, as j runs through $1, 2, \dots, u$, by assigning the $H[j] := V_r$ and computing $V_r\|V_c := \pi(V_r\|V_c)$.

Now, the hash value $H := H[1]\|H[2]\|\dots\|H[u]$.

Security of Sponge construction

Sponge construction is already proven to be secure against *collision*, *second-preimage*, and *herding* attacks [ADMA15, BDPA07, BDPA08, BDPA09, BDPA11, BDP+14]. For the HF-CR game (given in Figure 2.7), for a ur -bit

hash value, we have:

$$\text{Adv}_{\text{Sponge}, \mathcal{A}}^{\text{HF-CR}}(1^\lambda) \leq \frac{m^2}{2^{ur+1}}.$$

where, m is total number of block queries that the adversary makes.

2.2.4 Symmetric Encryption (SE)

The definition of *symmetric encryption (SE)* has been discussed in [BDJR97]. We revisit the definition below, with a few suitable modifications to suit the present context.

2.2.4.1 Definition

Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *SE* scheme $\rho = (\rho.\mathcal{GEN}, \rho.\mathcal{E}, \rho.\mathcal{D})$ is a 3-tuple of algorithms over the *setup* algorithm $\rho.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\rho.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\rho)}$, key-space $\mathcal{K}^{(\rho)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\rho)} \subseteq \{0, 1\}^*$ and ciphertext-space $\mathcal{C}^{(\rho)} \subseteq \{0, 1\}^*$.
2. The PPT *key generation* algorithm $\rho.\mathcal{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\rho)}$, and returns a key $K := \rho.\mathcal{GEN}(\text{params}^{(\rho)})$, where $K \in \mathcal{K}^{(\rho)}$.
3. The PPT *encryption* algorithm $\rho.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\rho)}$, key $K \in \mathcal{K}^{(\rho)}$ and message $M \in \mathcal{M}^{(\rho)}$, and returns ciphertext $C := \rho.\mathcal{E}(\text{params}^{(\rho)}, K, M)$, where $C \in \mathcal{C}^{(\rho)}$.
4. The *decryption* algorithm $\rho.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\rho)}$, key $K \in \mathcal{K}^{(\rho)}$ and ciphertext $C \in \mathcal{C}^{(\rho)}$, and returns message $M := \rho.\mathcal{D}(\text{params}^{(\rho)}, K, C)$, where $M \in \mathcal{M}^{(\rho)}$.

Note: We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\Psi)}$.

Correctness

This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the correctness of an *SE* can be defined as follows.

Suppose $K := \rho. \mathcal{GEN}(\text{params}^{(\rho)})$. Then, the correctness of ρ requires that for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\rho)}$, we have:

$$\rho. \mathcal{D}(\text{params}^{(\rho)}, K, \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M)) = M.$$

Security definitions

In Figures 2.12 and 2.13, we define the security games **SE-KR** and **SE-IND**, respectively, for the *SE* scheme $\rho = (\rho. \mathcal{GEN}, \rho. \mathcal{E}, \rho. \mathcal{D})$. As usual, the games are written using the challenger-adversary framework.

Game $\text{SE-KR}_\rho^A(1^\lambda)$

 $(\text{params}^{(\rho)}, \mathcal{K}^{(\rho)}, \mathcal{M}^{(\rho)}, \mathcal{C}^{(\rho)}) := \rho. \text{Setup}(1^\lambda);$
 $K := \rho. \mathcal{GEN}(\text{params}^{(\rho)});$
 $M := \mathcal{A}_1^{\rho, \mathcal{E}(\text{params}^{(\rho)}, K, \cdot)}(1^\lambda);$
 $C := \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M);$
 $K' := \mathcal{A}_2(1^\lambda, C);$
If $(K' = K)$, **then** return 1;
Else return 0;

Figure 2.12: Security game **SE-KR** for the *SE* scheme ρ .

SE-KR SECURITY. This captures the intuitive notion of *key recovery* attack, as defined in Figure 2.12. The adversarial advantage is the probability of the event when the adversary is able to deduce the encryption key, and it is desired to be negligible.

According to the **SE-KR** game, the adversary first returns a message M , and then the following operations are performed: first a key $K := \rho. \mathcal{GEN}(\text{params}^{(\rho)})$ is generated; and then message M is encrypted using K to obtain the ciphertext $C := \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M)$. Given access to C , the adversary returns a key K' . The game returns 1, if $K = K'$.

The advantage of the **SE-KR** adversary \mathcal{A} against ρ is defined as follows:

$$Adv_{\rho, \mathcal{A}}^{\text{SE-KR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{SE-KR}_\rho^A(1^\lambda) = 1].$$

The ρ is **SE-KR** secure, if, for all PPT adversaries \mathcal{A} , $Adv_{\rho, \mathcal{A}}^{\text{SE-KR}}(1^\lambda)$ is negligible.

```

Game SE-IND $_{\rho}^{\mathcal{A}}(1^\lambda, b)$ 
( $params^{(\rho)}, \mathcal{K}^{(\rho)}, \mathcal{M}^{(\rho)}, \mathcal{C}^{(\rho)}$ ) :=  $\rho$ .Setup( $1^\lambda$ );
 $K := \rho$ .GEN( $params^{(\rho)}$ );
 $M := \mathcal{A}_1^{\rho, \mathcal{E}(params^{(\rho)}, K, \cdot)}(1^\lambda)$ ;
 $C_1 := \rho$ .E( $params^{(\rho)}, K, M$ );
 $C_0 \xleftarrow{\$} \{0, 1\}^{|C_1|}$ ;
 $b' := \mathcal{A}_2(1^\lambda, C_b)$ ;
return  $b'$ ;

```

Figure 2.13: Security game **SE-IND** for the *SE* scheme ρ .

SE-IND SECURITY. This captures the notion of *indistinguishability privacy* attack, as defined in Figure 2.13. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a message from a random string, and it is desired to be negligible.

According to the **SE-IND** game, the adversary first returns a message M . Then the following operations are performed: first a key $K := \rho$.**GEN**($params^{(\rho)}$) is generated; then message M is encrypted using K to obtain the ciphertext $C_1 := \rho$.**E**($params^{(\rho)}, K, M$); and finally a random string C_0 of length $|C_1|$ is computed. Given access to C_b , where $b \in \{0, 1\}$, the adversary returns a bit b' .

The advantage of the **SE-IND** adversary \mathcal{A} against ρ is defined as follows:

$$Adv_{\rho, \mathcal{A}}^{\text{SE-IND}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{SE-IND}_{\rho}^{\mathcal{A}}(1^\lambda, b = 1) = 1] - \Pr[\text{SE-IND}_{\rho}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|.$$

The ρ is **SE-IND** secure, if, for all PPT adversaries \mathcal{A} , $Adv_{\rho, \mathcal{A}}^{\text{SE-IND}}(1^\lambda)$ is negligible.

2.2.5 Authenticated Encryption (AE)

Below we elaborately discuss the syntax, correctness and security definition of the *AE*.

2.2.5.1 Definition

Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *AE* scheme $\varrho = (\varrho.\mathcal{GEN}, \varrho.\mathcal{E}, \varrho.\mathcal{D})$ is a 3-tuple of algorithms over the *setup* algorithm $\varrho.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\varrho.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\varrho)}$, key-space $\mathcal{K}^{(\varrho)} \subseteq \{0,1\}^*$, message-space $\mathcal{M}^{(\varrho)} \subseteq \{0,1\}^*$, ciphertext-space $\mathcal{C}^{(\varrho)} \subseteq \{0,1\}^*$ and tag-space $\mathcal{T}^{(\varrho)} \subseteq \{0,1\}^*$.
2. The PPT *key generation* algorithm $\varrho.\mathcal{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\varrho)}$, and returns a key $K := \varrho.\mathcal{GEN}(\text{params}^{(\varrho)})$, where $K \in \mathcal{K}^{(\varrho)}$.
3. The PPT *encryption* algorithm $\varrho.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\varrho)}$, key $K \in \mathcal{K}^{(\varrho)}$ and message $M \in \mathcal{M}^{(\varrho)}$, and returns a pair $(C, T) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, K, M)$, where ciphertext $C \in \mathcal{C}^{(\varrho)}$ and tag $T \in \mathcal{T}^{(\varrho)}$. It is possible that the tag is incorporated in the ciphertext itself, in this case, T is an empty string.
4. The *decryption* algorithm $\varrho.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\varrho)}$, key $K \in \mathcal{K}^{(\varrho)}$, ciphertext $C \in \mathcal{C}^{(\varrho)}$ and tag $T \in \mathcal{T}^{(\varrho)}$, and returns message $M := \varrho.\mathcal{D}(\text{params}^{(\varrho)}, K, C, T)$, where $M \in \mathcal{M}^{(\varrho)} \cup \{\perp\}$. The decryption algorithm $\varrho.\mathcal{D}$ returns \perp , if the ciphertext C and tag T are not generated using the key K .

Note: When the tag is incorporated in the ciphertext itself, we observe an obvious and intuitive expansion of the ciphertext. Therefore, we restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\varrho)}$.

Correctness

DECRYPTION CORRECTNESS. This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the *decryption correctness* of an *AE* can be defined as follows.

Suppose:

- $K := \varrho.\mathcal{GEN}(\text{params}^{(\varrho)})$, and
- $(C, T) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, K, M)$.

Then, the *decryption correctness* of ϱ requires that $\varrho.\mathcal{D}(\text{params}^{(\varrho)}, K, C, T) = M$, for all $(\lambda, K, M) \in \mathbb{N} \times \mathcal{K}^{(\varrho)} \times \mathcal{M}^{(\varrho)}$.

Security definitions

In Figures 2.14 and 2.15, we define the security games **AE-IND** and **AE-INT**, respectively, for the *AE* scheme $\varrho = (\varrho.\mathcal{GEN}, \varrho.\mathcal{E}, \varrho.\mathcal{D})$. As usual, the games are written using the challenger-adversary framework.

```

Game AE-IND $_{\varrho}^{\mathcal{A}}(1^\lambda, b)$ 
_____
 $(\text{params}^{(\varrho)}, \mathcal{K}^{(\varrho)} \mathcal{M}^{(\varrho)}, \mathcal{C}^{(\varrho)}, \mathcal{T}^{(\varrho)}) := \varrho.\text{Setup}(1^\lambda);$ 
 $K := \varrho.\mathcal{GEN}(\text{params}^{(\varrho)});$ 
 $(M_0, M_1) := \mathcal{A}_1(1^\lambda);$ 
If ( $|M_0| \neq |M_1|$ ), then return Error;
 $(C, T) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, K, M_b);$ 
 $b' := \mathcal{A}_2(1^\lambda, C, T, M_0, M_1);$ 
return  $b'$ ;

```

Figure 2.14: Security game **AE-IND** for the *AE* scheme ϱ .

AE-IND SECURITY. This captures the notion of *indistinguishability privacy* attack, as defined in Figure 2.14. The adversarial advantage is the probability of the event when the adversary is able to distinguish between the encryptions of two messages, and it is desired to be negligible.

According to the **AE-IND** game, the adversary first returns two messages M_0 and M_1 , such that $|M_0| = |M_1|$. Then the following operations are performed: first a key $K := \varrho.\mathcal{GEN}(\text{params}^{(\varrho)})$ is generated; and then the message M_b , where $b \in \{0, 1\}$, is encrypted using key K to obtain the pair $(C, T) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, K, M_b)$. Given access to C and T , the adversary returns a bit b' .

The advantage of the **AE-IND** adversary \mathcal{A} against ϱ is defined as follows:

$$\text{Adv}_{\varrho, \mathcal{A}}^{\text{AE-IND}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{AE-IND}_{\varrho}^{\mathcal{A}}(1^\lambda, b = 1) = 1] - \Pr[\text{AE-IND}_{\varrho}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|.$$

The ϱ is **AE-IND** secure, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\varrho, \mathcal{A}}^{\text{AE-IND}}(1^\lambda)$ is negligible.

AE-INT SECURITY. This captures the notion of *tag consistency* attack, as defined in Figure 2.15. The adversarial advantage is the probability of the

```

Game  $\text{AE-INT}_{\varrho}^{\mathcal{A}}(1^\lambda, \sigma)$ 
 $(params^{(\varrho)}, \mathcal{K}^{(\varrho)} \mathcal{M}^{(\varrho)}, \mathcal{C}^{(\varrho)}, \mathcal{T}^{(\varrho)}) := \varrho.\text{Setup}(1^\lambda);$ 
 $K := \varrho.\mathcal{GEN}(params^{(\varrho)});$ 
 $(C_0, C_1, T) := \mathcal{A}^{\varrho, \mathcal{E}(params^{(\varrho)}, K, \cdot)}(1^\lambda, \sigma);$ 
If  $(C_0 = C_1)$ , then return 0;
 $M_0 := \varrho.\mathcal{D}(params^{(\varrho)}, K, C_0, T);$ 
 $M_1 := \varrho.\mathcal{D}(params^{(\varrho)}, K, C_1, T);$ 
If  $(M_0 \neq \perp) \wedge (M_1 \neq \perp)$ , then return 1;
Else return 0;

```

Figure 2.15: Security game AE-INT for the AE scheme ϱ .

event when the adversary is able to compute two unidentical ciphertexts and a common tag, such that the decryption of each ciphertext with the tag results into a valid message, and it is desired to be negligible.

According to the AE-INT game, first we generate a key $K := \varrho.\mathcal{GEN}(params^{(\varrho)})$. Given access to encryption oracle $\varrho.\mathcal{E}(params^{(\varrho)}, K, \cdot)$ and the limit to number of queries to the oracle σ , the adversary returns two ciphertexts C_0 and C_1 , and a tag T , such that $C_0 \neq C_1$. Finally, the two ciphertexts are decrypted to obtain $M_0 := \varrho.\mathcal{D}(params^{(\varrho)}, K, C_0, T)$ and $M_1 := \varrho.\mathcal{D}(params^{(\varrho)}, K, C_1, T)$. The game returns 1, if $M_0 \neq \perp$ and $M_1 \neq \perp$.

The advantage of the AE-INT adversary \mathcal{A} against ϱ is defined as follows:

$$Adv_{\varrho, \mathcal{A}}^{\text{AE-INT}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{AE-INT}_{\varrho}^{\mathcal{A}}(1^\lambda, \sigma) = 1].$$

The ϱ is AE-INT secure, if, for all PT adversaries \mathcal{A} , $Adv_{\varrho, \mathcal{A}}^{\text{AE-INT}}(1^\lambda)$ is negligible.

2.2.5.2 APE authenticated encryption

Authenticated Permutation-based Encryption (APE) is an $AEAD$ scheme (short-hand for *authenticated encryption with associated data*) proposed by Andreeva *et al.* in *DIAC 2012* and *FSE 2014* [ABB⁺14]. APE has a very unique property of *reverse decryption*, meaning that the direction of decryption is *opposite* to that of the encryption. It uses an *easy-to-invert* permutation as the basic building block.

$AEAD$ is a special kind of AE (described in Section 2.2.5), where an additional data, called *associated data (AD)*, is also supplied during the

encryption/decryption. AD could be like a meta-data for the file, which is sent across the insecure channel without any encryption. Therefore, the inclusion of *associated data* does not result in any ciphertext expansion.

APE also has a provision for including a *nonce*, along with the other inputs. A nonce is considered as the last block of AD , whenever present.

The pictorial description and pseudocode for the 3-tuple of algorithms in $\text{APE} = (\text{APE.GEN}, \text{APE.E}, \text{APE.D})$ over the *setup* algorithm $\text{APE}.\text{Setup}$ are given in Figures 2.16, 2.17, 2.18, 2.19 and 2.20. Below we give the textual description.

- The *setup* algorithm $\text{APE}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\text{APE})}$, key-space $\mathcal{K}^{(\text{APE})} = \{0, 1\}^c$, associated-data-space $\mathcal{AD}^{(\text{APE})} = \bigcup_{i \geq 0} \{0, 1\}^{ir}$, message-space $\mathcal{M}^{(\text{APE})} = \bigcup_{i \geq 1} \{0, 1\}^{ir}$, ciphertext-space $\mathcal{C}^{(\text{APE})} = \bigcup_{i \geq 1} \{0, 1\}^{ir}$ and tag-space $\mathcal{T}^{(\text{APE})} = \{0, 1\}^c$. The variables rate $r \in \mathbb{N}$ and capacity $c \in \mathbb{N}$ are included in parameter $\text{params}^{(\text{APE})}$.

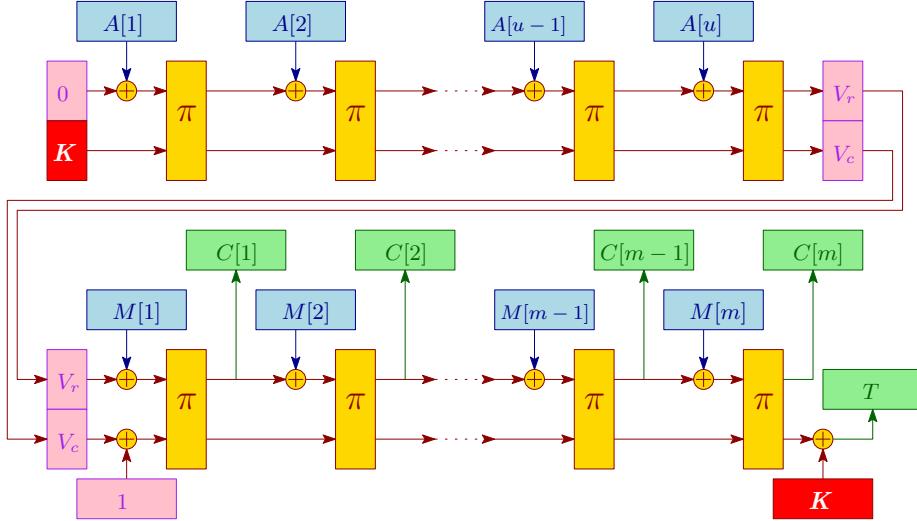
```

APE.GEN(params(APE))
#Choosing a random  $\lambda$ -bit key.
 $K \xleftarrow{\$} \{0, 1\}^c;$ 
return  $K$ ;

```

Figure 2.16: Algorithmic description of the *key generation* function APE.GEN .

- The PPT *key generation* algorithm $\text{APE}.\text{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\text{APE})}$, and outputs a key K randomly chosen from the key-space $\mathcal{K}^{(\text{APE})}$. The pseudocode is given in Figure 2.16.
- The *encryption* algorithm $\text{APE}.\text{E}(\cdot)$ is randomized that takes as input parameter $\text{params}^{(\text{APE})}$, key K , associated data A and message M , and outputs a pair of ciphertext C and tag T . The pictorial description and pseudocode are given in Figures 2.17 and 2.18. Very briefly, the algorithm works as follows: first it initialises $V_r := 0^r$ and $V_c := K$; then it parses the associated data into r -bit strings $A[1]||A[2]||\dots||A[u] := A$; next it processes the associated data blocks $A[j]$ iteratively, as j runs through $1, 2, \dots, u$, by computing $V_r||V_c := \pi((V_r \oplus A[j])||V_c)$; after that it computes $V_c := V_c \oplus 1$; then it parses M into r -bit strings $M[1]||M[2]||\dots||M[m] := M$; and finally encrypts

Figure 2.17: Pictorial description of the *encryption* function APE. \mathcal{E} .

```

APE.  $\mathcal{E}$ (params(APE),  $K$ ,  $A$ ,  $M$ )
#Initialisation.
 $V_r := 0^r$ ;  $V_c := K$ ;

#Processing associated data.
If ( $A \neq \epsilon$ )
 $u := |A|/r$ ;  $A[1]\|A[2]\|\cdots\|A[u] := A$ ;
for ( $j := 1, 2, \dots, u$ )
 $V_r\|V_c := \pi((V_r \oplus A[j])\|V_c)$ ;

#Processing message blocks.
 $V_c := V_c \oplus 1$ ;  $m := |M|/r$ ;
 $M[1]\|M[2]\|\cdots\|M[m] := M$ ;
for ( $j := 1, 2, \dots, m$ )
 $V_r\|V_c := \pi((V_r \oplus M[j])\|V_c)$ ;  $C[j] := V_r$ ;

#Computing final output.
 $C := C[1]\|C[2]\|\cdots\|C[m]$ ;  $T := V_c \oplus K$ ;
return  $(C, T)$ ;

```

Figure 2.18: Algorithmic description of the *encryption* function APE. \mathcal{E} .

the message blocks $M[j]$ iteratively, as j runs through $1, 2, \dots, m$, by computing $V_r\|V_c := \pi((V_r \oplus M[j])\|V_c)$ and $C[j] := V_r$.

Now, the ciphertext $C := C[1] \parallel C[2] \parallel \cdots \parallel C[m]$, and tag $T := V_c \oplus K$.

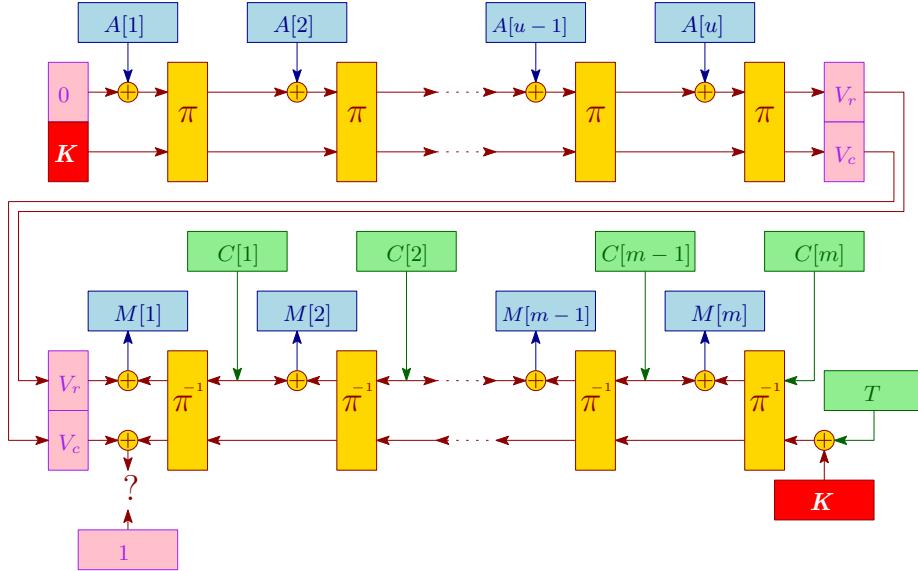


Figure 2.19: Pictorial description of the *decryption* function $\text{APE}.\mathcal{D}$.

- The *decryption* algorithm $\text{APE}.\mathcal{D}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\text{APE})}$, key K , associated data A , ciphertext C and tag T , and outputs message M or an *invalid* symbol \perp . See Figures 2.19 and 2.20 for the pictorial description and pseudocode. The algorithm returns \perp , if the ciphertext C and tag T are not generated using the key K . Very briefly, the algorithm works as follows: first it initialises $V_r := 0^r$ and $V_c := K$; then it parses the associated data into r -bit strings $A[1] \parallel A[2] \parallel \cdots \parallel A[u] := A$; next it processes the associated data blocks $A[j]$ iteratively, as j runs through $1, 2, \dots, u$, by computing $V_r \parallel V_c$ as $\pi((V_r \oplus A[j]) \parallel V_c)$; after that it assigns $\text{temp} := V_c$; next it initialises $C[0] := V_r$; after that it parses C into r -bit strings $C[1] \parallel C[2] \parallel \cdots \parallel C[m] := C$; next it assigns $V_r := C[m]$; then it computes $V_c := K \oplus T$; and finally it decrypts the ciphertext blocks $C[j]$ iteratively, as j runs through $m, m-1, \dots, 1$, by first computing $V_r \parallel V_c$ as $\pi^{-1}(V_r \parallel V_c)$, then computing $M[j] := C[j-1] \oplus V_r$ and finally assigning $V_r := C[j-1]$.

Now, the message $M := M[1] \parallel M[2] \parallel \cdots \parallel M[m]$. If $(V_c \oplus 1) = \text{temp}$, then it returns M , otherwise it returns an invalid symbol \perp .

```

APE.  $\mathcal{D}(params^{\text{APE}}, K, A, C, T)$ 
#Initialisation.
 $V_r := 0^r; \quad V_c := K;$ 

#Processing associated data.
If ( $A \neq \epsilon$ )
 $u := |A|/r; \quad A[1]\|A[2]\|\cdots\|A[u] := A;$ 
for ( $j := 1, 2, \dots, u$ )
 $V_r\|V_c := \pi((V_r \oplus A[j])\|V_c);$ 

#Processing ciphertext blocks.
 $temp := V_c; \quad m := |C|/r;$ 
 $C[1]\|C[2]\|\cdots\|C[m] := C;$ 
 $C[0] := V_r; \quad V_r := C[m]; \quad V_c := K \oplus T;$ 
for ( $j := m, m-1, \dots, 1$ )
 $V_r\|V_c := \pi^{-1}(V_r\|V_c); \quad M[j] := C[j-1] \oplus V_r;$ 
 $V_r := C[j-1];$ 

#Computing final output.
 $M := M[1]\|M[2]\|\cdots\|M[m];$ 
If ( $V_c \oplus 1 = temp$ ), then return  $M$ ; Else return  $\perp$ ;

```

Figure 2.20: Algorithmic description of the *decryption* function $\text{APE. } \mathcal{D}$.

Security of APE construction

APE provides the *privacy* security under *chosen plaintext attack (CPA)* and the *integrity* security up to about $c/2$ bits. For the *authenticated encryption (AE)* scheme APE , we have:

$$\text{Adv}_{\text{APE}, \mathcal{A}}^{\text{AE-IND}}(1^\lambda, m) \leq \frac{m^2}{2^{r+c}} + \frac{m(m+1)}{2^c}.$$

$$\text{Adv}_{\text{APE}, \mathcal{A}}^{\text{AE-INT}}(1^\lambda, m) \leq \frac{m^2}{2^{r+c}} + \frac{2m(m+1)}{2^c}.$$

where, m is the total number of block queries.

2.3 Useful Techniques

In this section, we discuss a few cryptographic tools and techniques used to determine the security bounds of various cryptographic schemes.

- **Code-based game playing technique.** In the *code-based game playing technique*, we design a series of hybrid games, where every pair of successive games is identical until the *bad* flag is set. Then the *advantage* of the adversary, distinguishing among the two successive games, is bound by the probability of the event when the *bad* flag is set [BR06].
- **Triangle inequality (of security bounds).** According to the *triangle inequality*, the *advantage* of the adversary \mathcal{A} in distinguishing between the games G_S and G_L is upper-bounded by the sum of *advantages* of \mathcal{A} in: distinguishing between G_S and G_1 ; and distinguishing between G_1 and G_L , where G_1 is an intermediate game [BR06]. Mathematically, it can be represented as:

$$\text{Adv}_{G_S, G_L, \mathcal{A}}^{\text{IND}}(1^\lambda) \leq \text{Adv}_{G_S, G_1, \mathcal{A}}^{\text{IND}}(1^\lambda) + \text{Adv}_{G_1, G_L, \mathcal{A}}^{\text{IND}}(1^\lambda).$$

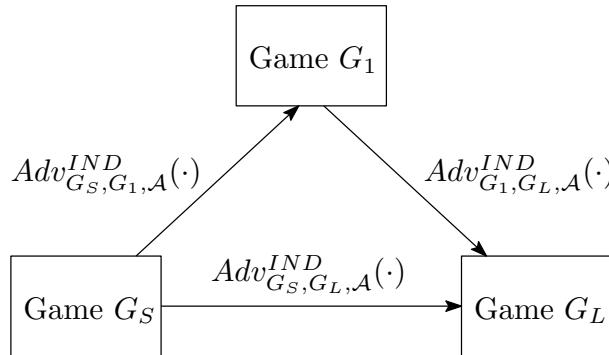


Figure 2.21: Pictorial representation of the *triangle inequality*.

- **PRP/PRF switching lemma.** The *PRP/PRF switching lemma* says that the *advantage* of an adversary distinguishing between λ -bit *pseudo-random permutation* π and λ -bit *pseudo-random function* rf , when at most σ queries are allowed to π or rf , is bounded by:

$$\text{Adv}_{\pi,\text{rf},\mathcal{A}}^{\text{IND}}(1^\lambda, \sigma) \leq \frac{\sigma(\sigma - 1)}{2^\lambda}.$$

- **Negligible function.** A function $\text{negl}(\cdot)$ is *negligible*, if, for some constant $c > 0$, it holds that $\text{negl}(n) < n^{-c}$, for all sufficiently large n 's.

CHAPTER 3

Advanced Tools and Techniques

3.1 Introduction

In Chapter 2, we have discussed the basics of cryptology, some fundamental cryptographic primitives and a few elementary tools. In this chapter, we will discuss two high-level cryptographic primitives – *message-locked encryption (MLE)* and *key assignment scheme (KAS)* – and the real-life protocols they are used for; these protocols are *deduplication* and *hierarchical access control (HAC)*.

ORGANIZATION. In Section 3.2, we begin with a detailed discussion on the *graphs* and graph-related algorithms, then we move on to an elaborate discussion on very special categories of graph, called *partially ordered set (poset)* and *totally ordered set (chain)*, and finally conclude with some advanced graph algorithms that we will be using in the subsequent chapters of the thesis. In Section 3.3, we discuss a cryptographic problem in the *Cloud* storage, known as *deduplication* protocol. In Section 3.4, we elaborately discuss how to solve the *deduplication* protocol using *MLE* and its variants. In Section 3.5, we describe another cryptographic problem, called *hierarchical access control (HAC)*. We elaborately discuss the solution to *HAC* problem using *key assignment scheme (KAS)* and its variants, in Section 3.6.

3.2 Graphs

A *graph* $G = (V, E)$ is a pair of sets, where the *vertex/node set* V is finite, and the *edge set* E is a binary relation on V [CLRS09]. An edge $e = (u, v)$ is a pair, where $u, v \in V$.

Directed and Undirected graphs

In a *directed* graph, the edge e is an ordered pair, while, in an *undirected* graph, e is unordered. The *degree* of node u , denoted $\deg(u)$, is the number of edges *incident from* node u .

Weighted and Unweighted graphs

In a *weighted* graph, each edge e is associated with weight $wt[e]$, while, in an *unweighted* graph, the edges have no weights (or, equivalently, each edge has the *identical* weight).

Paths and cycles

A *path* from u_{i_1} to u_{i_2} of length ℓ is a sequence of nodes $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_{\ell-1}}, u_{i_2})$ such that $(u_{i_1}, u_{j_1}), (u_{j_1}, u_{j_2}), \dots, (u_{j_{\ell-1}}, u_{i_2}) \in E$; the *length of the path* is the number of edges in the path. To find a path from a fixed node u_{i_1} to node u_{i_2} , we utilize the existing *single-source shortest path* algorithms, such as *Bellman-Ford* algorithm and *Dijkstra's* algorithm. To find the paths between all pair of nodes, we apply *all-pair shortest-path* algorithms, such as *Floyd-Warshall* algorithm and *Johnson's* algorithm.

A *cycle* is a path from u_{i_1} to u_{i_1} of length $\ell \geq 1$ for the sequence of nodes $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_{\ell-1}}, u_{i_1})$ such that $(u_{i_1}, u_{j_1}), (u_{j_1}, u_{j_2}), \dots, (u_{j_{\ell-1}}, u_{i_1}) \in E$. A graph G is called *cyclic*, if it contains one or more *cycles*, while a graph is *acyclic*, if it has no *cycles*.

Adjacency lists and adjacency matrices

Throughout the thesis, we use (*unweighted*) *directed acyclic graph (DAG)*. The edges of a *DAG* can be stored by using either of the two data structures – *adjacency list* and *adjacency matrix*. In the storage based on *adjacency list*, a list is associated to each vertex $u_{i_1} \in V$ containing the list of vertices $u_{i_2} \in V$ such that $(u_{i_1}, u_{i_2}) \in E$. Usage of *adjacency list* is preferred in the case of *sparse* matrix, that is, when the number of edges is much less than $|V|^2$. In the storage based on *adjacency matrix*, a $|V| \times |V|$ matrix T stores the value; the value $T[i_1, i_2] = 1$, if $(u_{i_1}, u_{i_2}) \in E$, otherwise $T[i_1, i_2] = 0$.

Usage of *adjacency matrix* is preferred in the case of *dense* matrix, that is, when the number of edges is close to $|V|^2$.

In the subsequent subsections: we first discuss *Dijkstra's* single-source shortest-path algorithm in Section 3.2.1; then we discuss *partially ordered set (poset)*, *totally ordered set (chain)* and *access graphs* in Section 3.2.2; and after that we describe the graph algorithms used in this thesis in Section 3.2.3.

In the graph algorithms to be discussed in the subsequent sections, we use two *data structures* – a *queue*, and its special variant, *min-priority queue*. The *queue* data structure works on the principle of *first-in, first-out (FIFO)* policy, where the element that has been in the set for the longest time, will be the first one to be deleted. There are two operations on *queue* – **ENQUEUE** and **DEQUEUE**. The **ENQUEUE(·)** operation takes as input queue Q and an element *item*, and adds *item* to Q . The **DEQUEUE(·)** operation takes as input Q , and it removes and returns the element *item* that has been in the queue for the longest time. *Min-priority queue* is a special variant of *queue*, where the function **ENQUEUE(·)** is identical to that of *queue*, but the function **DEQUEUE(·)** is modified to **EXTRACT-MIN(·)**, where the element removed from the queue is the one with the lowest value of a predefined attribute (*e.g.*, *index*, *distance* or *value*).

3.2.1 Dijkstra's Algorithm

In this section, we revisit the *Dijkstra's* algorithm proposed by Edsger W. Dijkstra in 1956, that determines the single-source shortest paths in a weighted DAG [Dij59]. *Dijkstra's* algorithm is an improvement over the *Bellman-Ford* algorithm, subject to the condition that the weights on the edges cannot be *negative*.¹ The *Dijkstra's* algorithm also works for unweighted DAG, by assuming the weight of each edge to be equal, say 1 unit.

Since the graphs used in this thesis are unweighted DAG, our main focus lies in the *Dijkstra's* algorithm for unweighted DAG. The pseudocode is given in Figure 3.1. Below we give the textual description.

The PPT *Dijkstra's* algorithm **DIJKSTRA-SSSP(·)** takes as input graph G and node u_i , and outputs array of distances $dist[]$ and array of parents $par[]$. Very briefly, the algorithm works as follows: first it initializes $dist[u_{i_1}]$ and $par[u_{i_1}]$ for all the vertices $u_{i_1} \in V$ with values ∞ and **NIL**, respectively;

¹*Bellman-Ford* algorithm was published by Lester Ford Jr. and Richard Bellman in 1956 and 1958 respectively to solve the problem of single-source shortest-path for a weighted directed graph that may have *negative* weight on edges [Bel58, For56].

```

DIJKSTRA-SSSP( $G, u_i$ )


---


#Initialization.
for all  $u_{i_1} \in V$ 
   $dist[u_{i_1}] := \infty$ ;  $par[u_{i_1}] := \text{NIL}$ ;
   $dist[u_i] := 0$ ;  $Q := V$ ;

#Processing each node.
while ( $Q \neq \emptyset$ )
   $u_{i_1} := \text{EXTRACT-MIN}(Q)$ ;
  for all  $(u_{i_1}, u_{i_2}) \in E$ 
    If ( $dist[u_{i_2}] > dist[u_{i_1}] + 1$ )
       $dist[u_{i_2}] := dist[u_{i_1}] + 1$ ;  $par[u_{i_2}] := u_{i_1}$ ;

#Returning final output.
  return ( $dist[]$ ,  $par[]$ );

```

Figure 3.1: Algorithmic description of the *Dijkstra's* algorithm.

then it assigns $dist[u_i] := 0$; and finally initializes *min-priority queue* Q with set of vertices V .

Now, to compute the values of $dist[]$ and $par[]$, the following operations are performed iteratively for $|V|$ times: first it extracts (or *dequeues*) a node $u_{i_1} := \text{EXTRACT-MIN}(Q)$ that has minimum *distance* value from the *min-priority queue* Q ; and after that for all the nodes u_{i_2} , such that (u_{i_1}, u_{i_2}) is an edge, it checks if ($dist[u_{i_2}] > dist[u_{i_1}] + 1$), then it updates $dist[u_{i_2}] := dist[u_{i_1}] + 1$ and $par[u_{i_2}] := u_{i_1}$. Now, the array of distances $dist[]$ and array of parents $par[]$, computed above, are returned.

The running-time complexity of *Dijkstra's* algorithm is $\mathcal{O}(|E| + |V| \log |V|)$ when *min-priority queue* is implemented with Fibonacci heap [CLRS09].²

3.2.2 Posets, Chains and Access Graphs

Suppose the users in an organisation are grouped into a set of pairwise disjoint classes $V = \{u_1, u_2, \dots, u_n\}$; in our case, the u_i 's are various *security classes*. Suppose $u, v \in V$; let $v \leq u$ imply that u can access all the data which can be accessed by v (this forms the *hierarchical access rule* for the *security classes*). Therefore, (V, \leq) is a *partially ordered set (poset)*, since

²The running-time complexity of *Dijkstra's* algorithm is $\mathcal{O}(|V|^2)$ with linear search implementation for *min-priority queue*.

\leq can be easily shown to be reflexive, anti-symmetric and transitive. We say:

- $v < u$, if: u and v are two distinct classes; and $v \leq u$.
- $v \lessdot u$, if: $v < u$; and $\nexists c \in V$ such that $v < c < u$.
- (V, \leq) is a *totally ordered set* or a *chain*, if: $\forall u, v \in V$, either $v \leq u$ or $u \leq v$.
- $A \subseteq V$ is an *anti-chain* in V , if, for all $u, v \in A$ such that $u \neq v$: we have $v \not\leq u$ and $u \not\leq v$. The cardinality of the largest *anti-chain* in V is called the *width* of V , denoted w .

Access graphs

An *access graph* is a representation of a *poset* (V, \leq) by a *directed acyclic graph* $G = (V, E)$, where the vertices represent the *security classes*, and, if $v \lessdot u$, then there is an edge from u to v . So, for all $u, v \in V$, where $v < u$, there is either a directed edge or a directed path from u to v .

Partitions and chains

A *partition* of set V is a collection of sets $\{V_1, V_2, \dots, V_s\}$ such that:

- $V_i \subseteq V, \forall i \in [s]$;
- $V_1 \cup V_2 \cup \dots \cup V_s = V$; and
- $i \neq j \Rightarrow V_i \cap V_j = \emptyset, \forall i, j \in [s]$.

According to Dilworth's Theorem, every poset (V, \leq) can be partitioned into w *chains*, where w is the *width* of V [Dil50]. The partition may not be unique. Let the set of *chains* $\{C_1, C_2, \dots, C_w\}$ denote a partition of V , $l_i = |C_i|$ (for $i \in [w]$), and $l_{max} = \max_{i \in [w]} l_i$. The *maximum* node of C_i is denoted u_1^i (i.e. $\forall v \in C_i, v \leq u_1^i$); and the *minimum* node of C_i is denoted $u_{l_i}^i$ (i.e. $\forall v \in C_i, u_{l_i}^i \leq v$). If $C_i = \{u_{l_i}^i, u_{l_i-1}^i, \dots, u_1^i\}$ and $u_{l_i}^i \lessdot u_{l_i-1}^i \lessdot \dots \lessdot u_1^i$, then $u_{l_i}^i \lessdot u_{l_i-1}^i \lessdot \dots \lessdot u_j^i$ is said to be a suffix of C_i , where $j \in [l_i]$. We say that: v is a successor of u , if $v \leq u$; and v is an ancestor of u , if $u \leq v$. For all $u \in V$, the set of all ancestors (and successors) of u is denoted $\uparrow u := \{v \in V : u \leq v\}$ (and $\downarrow u := \{v \in V : v \leq u\}$). Note that $\downarrow u$ has a non-empty intersection with one or more *chains* C_1, C_2, \dots, C_w , and, therefore, $\downarrow u \cap C_i$ is either a suffix of C_i or an empty set \emptyset . Since, $\{C_1, C_2, \dots, C_w\}$ is a disjoint partition of V , $\{\downarrow u \cap C_1, \downarrow u \cap C_2, \dots, \downarrow u \cap C_w\}$ is also a collection of pairwise disjoint sets. The *maximum* node of $\downarrow u \cap C_i$ is denoted \hat{u}_i . If $\downarrow u \cap C_i = \emptyset$, then $\hat{u}_i = \perp$.

3.2.3 Graph Algorithms

In this thesis, we frequently use some graph-based algorithms; we will describe them in detail shortly. In the access graph $G = (V, E)$ for the poset (V, \leq) , we represent the *security classes* by nodes $u_1, u_2, \dots, u_n \in V$, where $n = |V|$. The pseudocodes for the graph algorithms are given in Figures 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8 and 3.9.

Here: `ENQUEUE(·)` and `DEQUEUE(·)` functions have already been discussed in Section 3.2; the *Dijkstra's* algorithm `DIJKSTRA-SSSP(·)` has been described in Section 3.2.1; `SORT(·)` function takes as input a list of elements, and returns a sorted sequence of elements in the ascending order, based on their index values; the `FIND_ROOT(·)` function takes as input a graph G , and outputs the root node of G (that is, it returns the node that has no incoming edges).

```

all_succ( $u_i, G$ )

#Initialization.
for all  $u_{i_1} \in V$ 
     $\text{colour}[u_{i_1}] := \text{WHITE};$ 
     $\text{colour}[u_i] := \text{GREY}; \quad Q := \emptyset; \quad U := \{u_i\}; \quad \text{ENQUEUE}(Q, u_i);$ 

#Processing each node.
while ( $Q \neq \emptyset$ )
     $u_{i_1} := \text{DEQUEUE}(Q);$ 
    for all  $(u_{i_1}, u_{i_2}) \in E$ 
        If ( $\text{colour}[u_{i_2}] = \text{WHITE}$ )
             $\text{colour}[u_{i_2}] := \text{GREY}; \quad \text{ENQUEUE}(Q, u_{i_2}); \quad U := U \cup \{u_{i_2}\};$ 
             $\text{colour}[u_{i_1}] := \text{BLACK};$ 

#Returning final output.
return  $U;$ 

```

Figure 3.2: Algorithmic description of the `all_succ` function.

- **all_succ(u_i, G):** Given node u_i and graph G as input, this function outputs the set of all the successor nodes $\downarrow u_i = \{u_{i_1} \in V \mid u_{i_1} \leq u_i\}$. The pseudocode is given in Figure 3.2. This can be implemented by using *Breadth First Search (BFS)* (or *Depth First Search (DFS)*) traversal on the graph G with u_i as the *source/root* node. Very briefly, the algorithms works as follows: first it initializes $\text{colour}[u_{i_1}] := \text{WHITE}$

for all nodes $u_{i_1} \in V$; next it assigns $\text{colour}[u_i] := \text{GREY}$, queue $Q := \emptyset$ and set $U := \{u_i\}$; and after that it adds the node u_i to queue Q by invoking $\text{ENQUEUE}(Q, u_i)$. Now, to compute the set U , the following operations are performed iteratively while Q is not empty: first it extracts (or *dequeues*) a node $u_{i_1} := \text{DEQUEUE}(Q)$ from Q ; and after that for all nodes $u_{i_2} \in V$ such that (u_{i_1}, u_{i_2}) is an edge, it checks if $\text{colour}[u_{i_2}] = \text{WHITE}$, then it assigns $\text{colour}[u_{i_2}] := \text{GREY}$, adds u_{i_2} to Q by invoking $\text{ENQUEUE}(Q, u_{i_2})$, and adds u_{i_2} to set U ; and finally it assigns $\text{colour}[u_{i_1}] := \text{BLACK}$. Now, the set U is returned.

The running time of $\text{all_succ}(u_i, G)$ is $\mathcal{O}(|V| + |E|)$.

```

ch_seq( $u_i, G$ )
#Initialization.
 $U := \emptyset$ ;

#Processing each node.
for all  $(u_i, u_{i_1}) \in E$ 
     $U := U \cup \{u_{i_1}\}$ ;
     $U' := \text{SORT}(U)$ ;

#Returning final output.
return  $U'$ ;

```

Figure 3.3: Algorithmic description of the **ch_seq** function.

- **ch_seq(u_i, G):** Given node u_i and graph G as input, this function outputs a sequence of nodes $\tilde{u}_i = (u_{j_1}, u_{j_2}, \dots, u_{j_d})$ – that are children of node u_i in G – in the ascending order of their indices. Therefore, u_{j_1} has the lowest index, u_{j_2} the second lowest, and so on. We say that \tilde{u}_i is **NULL**, if u_i is a leaf node. The pseudocode is given in Figure 3.3. The algorithm works in the following way: first it initializes set U with an empty set \emptyset ; then for all nodes $u_{i_1} \in V$ such that (u_i, u_{i_1}) is an edge, the set U is updated as $U := U \cup \{u_{i_1}\}$; and finally U is sorted as $U' := \text{SORT}(U)$. Now, the sequence U' is returned.

The running time of $\text{ch_seq}(u_i, G)$ is $\mathcal{O}(|V| + \deg(u_i) \log(\deg(u_i)))$.

- **height(G):** Given a (directed acyclic) graph G as input, this function first assigns to $\text{level}[u_i]$ the maximum level of node u_i for all $u_i \in V$, and then returns $\text{level}[\cdot]$ and $h = \max_{u_i \in V} \text{level}[u_i]$. Note that there is exactly one root in a connected DAG. Also, the graph G is

```

height( $G$ )

#Initialization.
 $u_i := \text{FIND\_ROOT}(G); \quad h := 1; \quad Q := \emptyset; \quad level[u_i] := 1;$ 
 $\text{ENQUEUE}(Q, u_i);$ 

#Processing each node.
while ( $Q \neq \emptyset$ )
   $u_{i_1} := \text{DEQUEUE}(Q);$ 
  for all  $(u_{i_1}, u_{i_2}) \in E$ 
     $level[u_{i_2}] := level[u_{i_1}] + 1; \quad \text{ENQUEUE}(Q, u_{i_2});$ 
    If ( $h < level[u_{i_2}]$ ), then  $h := level[u_{i_2}];$ 

#Returning final output.
return ( $level[ ], h$ );

```

Figure 3.4: Algorithmic description of the `height` function.

acyclic; therefore, the value of $level[u_i]$, for all $u_i \in V$, can be at most n . The pseudocode is given in Figure 3.4. The algorithm works as follows: first it computes the root node $u_i := \text{FIND_ROOT}(G)$; then it initializes $h := 1$ and $Q := \emptyset$; next it assigns $level[u_i] := 1$; and finally it adds u_i to Q by invoking `ENQUEUE(Q, u_i)`. Now, to compute $level[]$, the following operations are performed iteratively while Q is not empty: first it extracts (or *dequeues*) a node u_{i_1} from Q ; and then for all nodes $u_{i_2} \in V$ such that (u_{i_1}, u_{i_2}) is an edge, it assigns $level[u_{i_2}] := level[u_{i_1}] + 1$, adds u_{i_2} to Q as `ENQUEUE(Q, u_{i_2})`, and checks if $(h < level[u_{i_2}])$, then assigns $h := level[u_{i_2}]$. Now, the array of levels $level[]$ and height of graph h is returned.

The running time of `height(G)` is $\mathcal{O}(|V|^2 + |V| + |E|)$.

- **max_isect(u_i, C, G):** Given node u_i , *chain* C and graph G as input, this function outputs the *maximal element* of $\downarrow u_i \cap C$. The pseudocode is given in Figure 3.5. Very briefly, the algorithm works as follows: first it computes set of all successors of u_i in G as $\downarrow u_i := \text{all_succ}(u_i, G)$; then it computes intersection of $\downarrow u_i$ and C as $(u_{i_1}, u_{i_2}, \dots, u_{i_j}) := \downarrow u_i \cap C$; next it initializes the value $max := 1$; and finally it computes the maximal element iteratively, as k runs through $2, 3, \dots, j$, by checking if $(u_{i_{max}} \leq u_{i_k})$, then assigns $max := k$. Now, the maximal element $u_{i_{max}}$ is returned.

The running time of `max_isect(u, C)` is $\mathcal{O}(|V| + |E|)$.

```

max_isect( $u_i, C, G$ )
#Initialization.
 $\downarrow u_i := \text{all\_succ}(u_i, G); \quad max := 1;$ 
 $(u_{i_1}, u_{i_2}, \dots, u_{i_j}) := \downarrow u_i \cap C;$ 

#Processing each node.
for ( $k := 2, 3, \dots, j$ )
  If ( $u_{i_{max}} \leq u_{i_k}$ ), then  $max := k$ ;

#Returning final output.
  return  $u_{i_{max}}$ ;

```

Figure 3.5: Algorithmic description of the **max_isect** function.

```

max_isect_chs( $u_i, G$ )
#Initialization.
 $\downarrow u_i := \text{all\_succ}(u_i, G); \quad W := \emptyset;$ 

#Processing each chain.
for ( $m := 1, 2, \dots, w$ )
   $(u_{i_1}, u_{i_2}, \dots, u_{i_j}) := \downarrow u_i \cap C_m; \quad max := 1;$ 
  for ( $k := 2, 3, \dots, j$ )
    If ( $u_{i_{max}} \leq u_{i_k}$ ), then  $max := k$ ;
     $W := (W \circ u_{i_{max}});$ 

#Returning final output.
  return  $W$ ;

```

Figure 3.6: Algorithmic description of the **max_isect_chs** function.

- **max_isect_chs(u_i, G)**: Given node u_i and graph G as input, this function outputs sequence of nodes $(\hat{u}_{i_1}, \hat{u}_{i_2}, \dots, \hat{u}_{i_w})$ who are the *maximum elements* of $\downarrow u_i \cap C_1, \downarrow u_i \cap C_2, \dots, \downarrow u_i \cap C_w$. The pseudocode is given in Figure 3.6. Very briefly, the algorithm works as follows: first it computes set of all successors of u_i in G as $\downarrow u_i := \text{all_succ}(u_i, G)$; and then it initializes sequence W with an empty set \emptyset . Now, for each chain C_m , where $m \in [w]$, the following operations are performed: first it computes intersection of $\downarrow u_i$ and C_m as $(u_{i_1}, u_{i_2}, \dots, u_{i_j}) := \downarrow u_i \cap C_m$; next it initializes value of max with 1; after that it computes the maximal element iteratively, as

k runs through $2, 3, \dots, j$, by checking if $(u_{i_{max}} \leq u_{i_k})$, then assigns $max := k$; and finally it updates W as $W := (W \circ u_{i_{max}})$. Now, the sequence of nodes W is returned.

The running time of $\text{max_isect_chs}(u, G)$ is $\mathcal{O}(|V| + |E|)$.

```

nodes_at_level( $V, level[], x$ )
#Initialization.
 $U := \emptyset$ ;

#Processing each node.
For all  $u_i \in V$ 
  If ( $level[u_i] = x$ ), then  $U := U \cup \{u_i\}$ ;

#Returning final output.
return  $U$ ;

```

Figure 3.7: Algorithmic description of the `nodes_at_level` function.

- **nodes_at_level**($V, level[], x$): Given set of vertices V , array $level[]$ storing levels of nodes and a level x as input, this function outputs set of nodes in G that are at level x . We have already assigned values to the array $level[]$ during execution of `height(G)` function. Now, we need to compare the values, and build the set of those elements whose level value is x . The pseudocode is given in Figure 3.7. The algorithm works as follows: first it initializes the set U with an empty set \emptyset ; and after that for all nodes $u_i \in V$, it checks if $level[u_i] = x$, then it updates U as $U := U \cup \{u_i\}$. Now, the set U is returned.

The running time of $\text{nodes_at_level}(V, level[], x)$ is $\mathcal{O}(|V|)$.

- **partition**(G): Given graph G as input, this function outputs the number of partitions w and the set of *chains* C_1, C_2, \dots, C_w (as used by Freire *et al.* [FPP13]). The running time of `partition`(G) is $\text{poly}(n)$.
- **path**(G, u_{i_1}, u_{i_2}): Given graph G and two nodes u_{i_1} and u_{i_2} (such that $u_{i_2} \leq u_{i_1}$) as input, this function outputs a sequence of nodes $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2})$ such that $u_{j_1} \lessdot u_{i_1}, u_{j_2} \lessdot u_{j_1}, \dots, u_{i_2} \lessdot u_{j_\ell}$. The pseudocode is given in Figure 3.8. Very briefly, the algorithm works as follows: first it initializes p and v both with u_{i_2} ; then it invokes *Dijkstra's* algorithm to compute arrays of distances and parents $(dist[], par[]) := \text{DIJKSTRA-SSSP}(G, u_{i_1})$; and finally it computes

```

path( $G, u_{i_1}, u_{i_2}$ )
#Initialization.
 $p := u_{i_2}; \quad v := u_{i_2};$ 

#Computing shortest path.
 $(dist[], par[]) := \text{DIJKSTRA-SSSP}(G, u_{i_1});$ 
while ( $v \neq u_{i_1}$ )
     $p := par[v] \circ p; \quad v := par[v];$ 

#Returning final output.
return  $p;$ 

```

Figure 3.8: Algorithmic description of the **path** function.

the path iteratively until $v = u_{i_1}$, by computing $p := par[v] \circ p$ and updating $v := par[v]$. Now, the path p is returned.

The running time of $\text{path}(G, u_{i_1}, u_{i_2})$ is $\mathcal{O}(|E| + |V| \cdot \log |V| + |V|)$.

```

vertex_in_order( $G$ )
#Initialization.
 $u_1 := \text{FIND\_ROOT}(G);$ 

#Processing the nodes.
for ( $i := 1, 2, \dots, m - 1$ )
    Find  $u_{i+1} \in V$  such that  $(u_i, u_{i+1}) \in E$ ;

#Returning final output.
return  $(u_1, u_2, \dots, u_m);$ 

```

Figure 3.9: Algorithmic description of the **vertex_in_order** function.

- **vertex_in_order(G)**: Given a graph G – corresponding to a *totally ordered set* – as input, this function outputs a sequence of nodes (u_1, u_2, \dots, u_m) such that $u_m \lessdot u_{m-1} \lessdot \dots \lessdot u_1$, where $m = |V|$. The pseudocode is given in Figure 3.9. The algorithm works as follows: first it computes the root node $u_1 := \text{FIND_ROOT}(G)$; and then it computes the sequence of nodes iteratively, as i runs through $1, 2, \dots, m-1$, by determining the node $u_{i+1} \in V$ such that $(u_i, u_{i+1}) \in E$. Now, the sequence of nodes (u_1, u_2, \dots, u_m) is returned.

The running time of `vertex_in_order`(G) is $\mathcal{O}(|V|^2 + |E|)$.

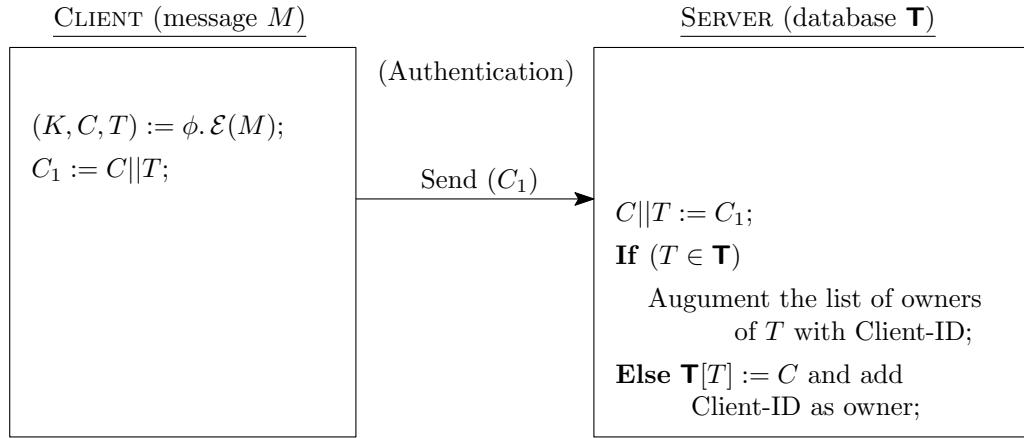
3.3 Deduplication

In a shared memory, such as the *Cloud*, several users store their files, which often turn out to be identical. Storing multiple copies of a file results in wastage of memory space. This memory wastage can be reduced by: identifying identical files (by simple comparison), then deleting all copies but one, and finally attaching a special file containing the list of owners to each distinct file. The situation worsens when a file is stored after encryption by several keys owned by as many users. *Deduplication* protocol aims to resolve this issue by improving the utilization of storage space, without compromising on the *privacy*. For each unique file F in the memory, the protocol works as follows: first it identifies all the files identical to F ; after that it creates and associates to F a special file called *list of owners*; and finally it deletes all the replicas of F . The *list of owners* is a file specifically meant to preserve the meta-data about the owners of all the replicas of F .

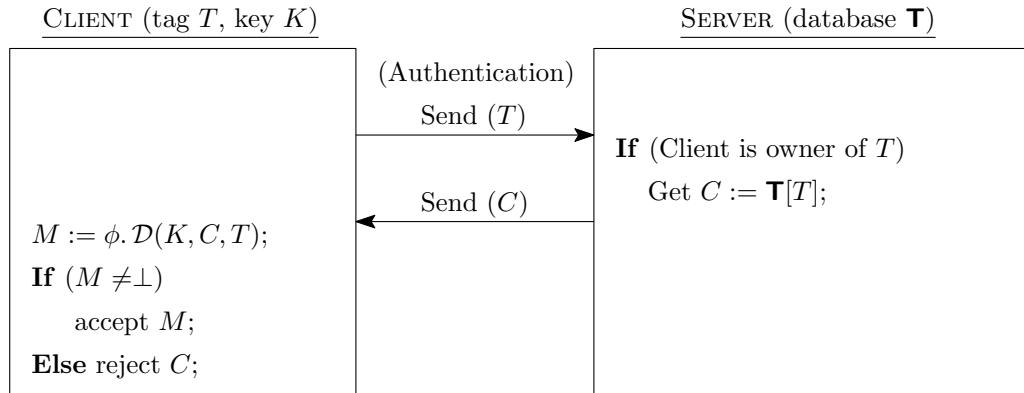
Deduplication protocol has two independent protocols – (1) *UPLOAD*, and (2) *DOWNLOAD*. The *UPLOAD* protocol specifies the steps undertaken while a client uploads a file to the server, while the *DOWNLOAD* protocol specifies those to be followed while a client wishes to download an already uploaded file from the server. One of the most popular techniques to implement *deduplication* protocol is by employing the cryptographic primitive *message-locked encryption (MLE)*; *MLE* is a pair of algorithms $\phi = (\phi.\mathcal{E}, \phi.\mathcal{D})$ and will be elaborately discussed in Section 3.4. Below we give the full textual description of the *MLE*-based *deduplication* protocol.

Remark. A *deduplication* protocol always begins with an independent *user authentication*; it could be a password-based or biometric-based mechanism.

- **UPLOAD.** This protocol governs the sequence of operations executed when the client uploads a file M to the server; the pictorial description is given in Figure 3.10. After *user authentication*, the client encrypts M to compute key K , ciphertext C and tag T using $\phi.\mathcal{E}(\cdot)$. The tag serves as a unique identifier for file M as well as ciphertext C . Then the client securely saves K and T in its local memory and sends $C_1 := C\|T$ to the server. The server computes C and T from C_1 . Then it checks if C are already stored in the database at index T , if true, the client-ID is appended in the existing *list of owners* file for C ,

Figure 3.10: UPLOAD protocol for MLE-based *deduplication*.

otherwise, C is stored in the database at index T , and a *list of owners* file is created corresponding to C , and client-ID is written onto it.

Figure 3.11: DOWNLOAD protocol for MLE-based *deduplication*.

- DOWNLOAD. This protocol governs the sequence of operations executed when the client downloads an already uploaded ciphertext from the server; the pictorial description is given in Figure 3.11. After *user authentication*, the client sends tag T corresponding to the ciphertext C it requires. The server checks if the client is an owner of C , if true, then it extracts C from its database at index T , and sends it to the client. On receiving C , the client decrypts it using K and T to obtain the file M , and accepts M only if $M \neq \perp$.

Besides application on *file-level*, *deduplication* protocol may also be employed on *block-level*, where the entire file is first fragmented into small

strings of fixed-size, called *blocks*, and then the *deduplication* is implemented on these blocks. There is a performance trade-off in deduplication at *file-level* vis-à-vis at *block-level*.

The *file-level deduplication* has the following advantages: a constant amount of memory is required for storing the file tag; and only one key for the entire file has to be securely stored. Its main drawback is when multiple files – with difference in only a few bits – have to be stored; in this case each file will be considered unique and be separately stored.

On the other hand, the *block-level deduplication* has following advantages: the master key has to be securely stored; and if multiple files with difference in only a few bits have to be stored, a huge amount of storage space will be saved. It has following disadvantages: a large number of block tags need to be stored besides the file tag; block keys either have to be stored with the master key or need to be processed and stored with ciphertext/tag; and metadata about the blocks in the file and their sequence needs to be stored separately.

3.4 Message-Locked Encryption (MLE)

The cryptographic primitive *message-locked encryption (MLE)* is a special kind of *symmetric encryption*, where key is generated from the message itself. This property of *MLE* is useful in several applications, such as *deduplication* in cloud and authenticated encryption.

To solve the problem of *deduplication*, *MLE* can be employed in the following way: the *key* generated by *MLE* is message-dependent, so, all the users possessing copies of a file will always generate identical key; the *encryption* function of *MLE* generates a deterministic *tag*, which can be used as an identifier for the file; and in *MLE*, if key, ciphertext and tag are not generated from a valid message, then the *decryption* function will always return an *invalid* string, which acts as a verifier for the correctness of the stored file.

In this section, we begin our discussion with the message-source, which is essential for defining the security properties of *MLE*. Then we describe the formal definition, followed by the existing constructions of *MLE*. Next we discuss the *proof-or-ownership* function, which is another requirement necessitated by the increased adversarial activities. After this, we describe a special variant of *MLE*, called *updatable block-level message-locked encryption (UMLE)*, that includes the functions of *update* and *proof-of-ownership*, in addition to the properties of *MLE*. Lastly, we discuss a special kind of

attack, known as *dictionary attack*, applicable on *MLE* for messages with low-entropy.

3.4.1 Message-source \mathcal{S}

In *MLE* (and its variants), the key is derived from the message itself, therefore, for a *known* message, the conventional security notions of *privacy* and *key recovery* would be meaningless.³ Therefore, we are modelling the security of *MLE* and its variants, based on an *unpredictable* message-source, rather than a *known* or *chosen* message.

A message-source $\mathcal{S}(\cdot)$ is a PPT algorithm that takes as input security parameter λ , and outputs (\mathbf{M}, Z) or $(\mathbf{M}_0, \mathbf{M}_1, Z)$, where each *vector of messages* $\mathbf{M} \in \{0, 1\}^{**}$ (or $\mathbf{M}_0, \mathbf{M}_1 \in \{0, 1\}^{**}$) and *auxiliary information* $Z \in \{0, 1\}^*$. We consider that $\mathcal{S}(\cdot)$ is a *public* message-source, that is, it is known to all parties including the adversary. Here, each vector of *message* \mathbf{M} has $m(1^\lambda)$ number of strings, i.e., $\|\mathbf{M}\| = m(1^\lambda)$ and length of each string $\mathbf{M}^{(i)}$ is $l(1^\lambda, i)$, i.e., $|\mathbf{M}^{(i)}| = l(1^\lambda, i)$ for $i \in [m(1^\lambda)]$. Here, m and l are two functions. We require that the two strings $\mathbf{M}^{(i_1)} \neq \mathbf{M}^{(i_2)}$, for $i_1 \neq i_2$ and $i_1, i_2 \in [m(1^\lambda)]$.

Associated with $\mathcal{S}(\cdot)$ is a real number $GP_{\mathcal{S}}$, namely, the *Guessing Probability of message-source*, which is the maximum of all probabilities of guessing a single string in \mathbf{M} , given the auxiliary information; its formal definition is $GP_{\mathcal{S}}(1^\lambda) \stackrel{\text{def}}{=} \max_{i \in [m(1^\lambda)]} GP(\mathbf{M}^{(i)} | Z)$. The $\mathcal{S}(\cdot)$ is *unpredictable* if the value of $GP_{\mathcal{S}}(1^\lambda)$ is *negligible*. We now define the *min-entropy* $\mu_{\mathcal{S}}(\cdot)$ of $\mathcal{S}(\cdot)$ as $\mu_{\mathcal{S}}(1^\lambda) = -\log(GP_{\mathcal{S}}(1^\lambda))$. The $\mathcal{S}(\cdot)$ is valid message-source for *MLE* scheme ϕ , if $\mathbf{M}^{(i)} \in \mathcal{M}^{(\phi)}, \forall i \in [m(1^\lambda)]$.

3.4.2 Definition

The definition of *MLE* has already been described in [BKR13b]. We briefly re-discuss it below, with a few suitable changes in the notation to suit the present context.

³An attack is possible. For *privacy* security, given a bit-string C and a message M , adversary can always compute key K from M , and then use K to decrypt C in order to verify if C was encryption of M or a random string. The *key recovery* security is jeopardized because of the fact that given a message, the adversary can always compute key.

3.4.2.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *MLE* scheme $\phi = (\phi.\mathcal{E}, \phi.\mathcal{D})$ is a pair of algorithms over the *setup* algorithm $\phi.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\phi.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\phi)}$, key-space $\mathcal{K}^{(\phi)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\phi)} \subseteq \{0, 1\}^*$, ciphertext-space $\mathcal{C}^{(\phi)} \subseteq \{0, 1\}^*$ and tag-space $\mathcal{T}^{(\phi)} \subseteq \{0, 1\}^*$.
2. The PPT *encryption* algorithm $\phi.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\phi)}$ and message $M \in \mathcal{M}^{(\phi)}$, and returns a 3-tuple $(K, C, T) := \phi.\mathcal{E}(\text{params}^{(\phi)}, M)$, where key $K \in \mathcal{K}^{(\phi)}$, ciphertext $C \in \mathcal{C}^{(\phi)}$ and tag $T \in \mathcal{T}^{(\phi)}$.
3. The *decryption* algorithm $\phi.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\phi)}$, key $K \in \mathcal{K}^{(\phi)}$, ciphertext $C \in \mathcal{C}^{(\phi)}$ and tag $T \in \mathcal{T}^{(\phi)}$, and returns message $M := \phi.\mathcal{D}(\text{params}^{(\phi)}, K, C, T)$, where $M \in \mathcal{M}^{(\phi)} \cup \{\perp\}$. The *decryption* algorithm $\phi.\mathcal{D}$ returns \perp , if key K , ciphertext C and tag T are not generated from a valid message.

Note: When tag is incorporated in the ciphertext itself, the ciphertext will be expanded. We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\phi)}$.

3.4.2.2 Correctness

KEY CORRECTNESS. This condition requires that the key generated from two identical messages will always be identical. Mathematically, the *key correctness* of an *MLE* can be defined as follows.

Suppose:

- $M_0 = M_1$,
- $(K_0, C_0, T_0) := \phi.\mathcal{E}(\text{params}^{(\phi)}, M_0)$, and
- $(K_1, C_1, T_1) := \phi.\mathcal{E}(\text{params}^{(\phi)}, M_1)$.

Then, the *key correctness* of ϕ requires that $K_0 = K_1$, for all $(\lambda, M_0, M_1) \in \mathbb{N} \times \mathcal{M}^{(\Phi)} \times \mathcal{M}^{(\Phi)}$.

DECRYPTION CORRECTNESS. This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the *decryption correctness* of an *MLE* can be defined as follows.

Suppose $(K, C, T) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M)$. The *decryption correctness* of ϕ requires that $\phi. \mathcal{D}(\text{params}^{(\phi)}, K, C, T) = M$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\phi)}$.

TAG CORRECTNESS. This condition requires that the tag generated from two identical messages will always be identical. Mathematically, the *tag correctness* of an *MLE* can be defined as follows.

Suppose:

- $M_0 = M_1$,
- $(K_0, C_0, T_0) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M_0)$, and
- $(K_1, C_1, T_1) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M_1)$.

Then, the *tag correctness* of ϕ requires that $T_0 = T_1$, for all $(\lambda, M_0, M_1) \in \mathbb{N} \times \mathcal{M}^{(\Phi)} \times \mathcal{M}^{(\Phi)}$.

3.4.2.3 Security definitions

In Figure 3.12, 3.13, 3.14 and 3.15, we define the security games **MLE-PRV**, **MLE-PRV\$**, **MLE-STC** and **MLE-TC**, and **MLE-KR**, respectively, for the *MLE* scheme $\phi = (\phi. \mathcal{E}, \phi. \mathcal{D})$. The first four games have already been described in [BKR13b]; we define a new security notion of *key recovery* useful for our purpose. As usual, all the games are written using the challenger-adversary framework.

Game **MLE-PRV** $_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b)$

 $(\text{params}^{(\phi)}, \mathcal{K}^{(\phi)}, \mathcal{M}^{(\phi)}, \mathcal{C}^{(\phi)}, \mathcal{T}^{(\phi)}) := \phi. \text{Setup}(1^\lambda);$
 $(\mathbf{M}_0, \mathbf{M}_1, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$
for $(i := 1, 2, \dots, m(1^\lambda))$
 $(\mathbf{K}^{(i)}, \mathbf{C}^{(i)}, \mathbf{T}^{(i)}) := \phi. \mathcal{E}(\text{params}^{(\phi)}, \mathbf{M}_b^{(i)});$
 $b' := \mathcal{A}(1^\lambda, \mathbf{C}, \mathbf{T}, Z);$
return b' ;

Figure 3.12: Security game **MLE-PRV** for the *MLE* scheme ϕ .

MLE-PRV SECURITY. Since no *MLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpredictable* message-source $\mathcal{S}(\cdot)$ (for details see Section 3.4.1). The **MLE-PRV** security – capturing the notion of *indistinguishability privacy against chosen distribution* attack – is defined in Figure 3.12. The adversarial advantage is the probability of the event when the adversary is able to distinguish between the encryptions of two vectors of messages, and it is desired to be negligible.

According to the **MLE-PRV** game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of two vectors of

strings $\mathbf{M}_0 = \mathbf{M}_0^{(1)} \circ \mathbf{M}_0^{(2)} \circ \cdots \circ \mathbf{M}_0^{(m(1^\lambda))}$ and $\mathbf{M}_1 = \mathbf{M}_1^{(1)} \circ \mathbf{M}_1^{(2)} \circ \cdots \circ \mathbf{M}_1^{(m(1^\lambda))}$, and auxiliary information Z . Now, for each string $\mathbf{M}_b^{(i)}$, depending upon the value of $b \in \{0, 1\}$, for all $i \in [m(1^\lambda)]$, key, ciphertext and tag are computed as $(\mathbf{K}^{(i)}, \mathbf{C}^{(i)}, \mathbf{T}^{(i)}) := \phi.\mathcal{E}(params^{(\phi)}, \mathbf{M}_b^{(i)})$. Given access to \mathbf{C} , \mathbf{T} and Z , the adversary returns a bit b' .

The advantage of the **MLE-PRV** adversary \mathcal{A} against ϕ for the message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\begin{aligned} Adv_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-PRV}}(1^\lambda) &\stackrel{def}{=} \left| \Pr[\text{MLE-PRV}_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=1) = 1] \right. \\ &\quad \left. - \Pr[\text{MLE-PRV}_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=0) = 1] \right|. \end{aligned}$$

The ϕ is **MLE-PRV** secure over a set of valid PPT message-sources for ϕ , $\bar{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \bar{\mathcal{S}}$, if $Adv_{\phi, \mathcal{S}_i, \mathcal{A}}^{\text{MLE-PRV}}(1^\lambda)$ is *negligible*. The ϕ is **MLE-PRV** secure, for all PPT adversaries \mathcal{A} , if $Adv_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-PRV}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for ϕ .

<p>Game MLE-PRV\$$_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b)$</p> <hr/> <p>$(params^{(\phi)}, \mathcal{K}^{(\phi)}, \mathcal{M}^{(\phi)}, \mathcal{C}^{(\phi)}, \mathcal{T}^{(\phi)}) := \phi.\text{Setup}(1^\lambda);$</p> <p>$(\mathbf{M}, Z) \xleftarrow{\\$} \mathcal{S}(1^\lambda);$</p> <p>for $(i := 1, 2, \dots, m(1^\lambda))$</p> <p style="padding-left: 20px;">$(\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \phi.\mathcal{E}(params^{(\phi)}, \mathbf{M}^{(i)});$</p> <p style="padding-left: 20px;">$\mathbf{C}_0^{(i)} \xleftarrow{\\$} \{0, 1\}^{ \mathbf{C}_1^{(i)} }; \quad \mathbf{T}_0^{(i)} \xleftarrow{\\$} \{0, 1\}^{ \mathbf{T}_1^{(i)} };$</p> <p style="padding-left: 20px;">$b' := \mathcal{A}(1^\lambda, \mathbf{C}_b, \mathbf{T}_b, Z);$</p> <p style="padding-left: 20px;">return b';</p>
--

Figure 3.13: Security game **MLE-PRV\$** for the *MLE* scheme ϕ .

MLE-PRV\$ SECURITY. This security notion is identical to **MLE-PRV**, except that in **MLE-PRV**, the adversary's challenge is to distinguish between the encryptions of two vectors of strings, while in **MLE-PRV\$**, the adversary's challenge is to distinguish the encryption of a vector of strings from a vector of random strings. Since no *MLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpredictable* message-source $\mathcal{S}(\cdot)$ (for details see Section 3.4.1). The **MLE-PRV\$** security – capturing the notion of *strong indistinguishability privacy against chosen distribution* attack – is defined in Figure 3.13. The adversarial advantage is the probability of the event when the adversary is able to distinguish the

encryption of a vector of messages from a vector of random strings, and it is desired to be negligible.

According to the **MLE-PRV\$** game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of a vector of strings $\mathbf{M} = \mathbf{M}^{(1)} \circ \mathbf{M}^{(2)} \circ \dots \circ \mathbf{M}^{(m(1^\lambda))}$ and auxiliary information Z . Now, for each string $\mathbf{M}^{(i)}$, for values of $i \in [m(1^\lambda)]$, following operations are performed: key, ciphertext and tag as computed as $(\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \phi.\mathcal{E}(\text{params}^{(\phi)}, \mathbf{M}^{(i)})$; a random string of length $|\mathbf{C}_1^{(i)}|$ is chosen as $\mathbf{C}_0^{(i)}$; and a random string of length $|\mathbf{T}_1^{(i)}|$ is chosen as $\mathbf{T}_0^{(i)}$. Given access to \mathbf{C}_b , \mathbf{T}_b and Z , where $b \in \{0, 1\}$, the adversary returns a bit b' .

The advantage of the **MLE-PRV\$** adversary \mathcal{A} against ϕ for the message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\begin{aligned} \text{Adv}_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-PRV\$}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{MLE-PRV\$}_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=1) = 1] \right. \\ &\quad \left. - \Pr[\text{MLE-PRV\$}_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=0) = 1] \right|. \end{aligned}$$

The ϕ is **MLE-PRV\$** secure over a set of valid PPT message-sources for ϕ , $\bar{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \bar{\mathcal{S}}$, if $\text{Adv}_{\phi, \mathcal{S}_i, \mathcal{A}}^{\text{MLE-PRV\$}}(1^\lambda)$ is *negligible*. The ϕ is **MLE-PRV\$** secure, for all PPT adversaries \mathcal{A} , if $\text{Adv}_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-PRV\$}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for ϕ .

Note: **MLE-PRV\$** is a stronger security notion than **MLE-PRV** [BKR13b].

Game MLE-STC $_{\phi}^{\mathcal{A}}(1^\lambda)$	MLE-TC $_{\phi}^{\mathcal{A}}(1^\lambda)$
$(\text{params}^{(\phi)}, \mathcal{K}^{(\phi)}, \mathcal{M}^{(\phi)}, \mathcal{C}^{(\phi)}, \mathcal{T}^{(\phi)}) := \phi.\text{Setup}(1^\lambda);$	
$(M_0, C_1, T_1) := \mathcal{A}(1^\lambda);$	
If $(M_0 = \perp) \vee (C_1 = \perp)$, then return 0;	
$(K_0, C_0, T_0) := \phi.\mathcal{E}(\text{params}^{(\phi)}, M_0);$	
$M_1 := \phi.\mathcal{D}(\text{params}^{(\phi)}, K_0, C_1, T_1);$	
If $(T_0 = T_1) \wedge (M_0 \neq M_1)$	$\wedge (M_1 \neq \perp)$
return 1;	
Else return 0;	

Figure 3.14: Security games **MLE-STC** and **MLE-TC** for the *MLE* scheme ϕ .

MLE-STC AND MLE-TC SECURITY. The **MLE-STC** and **MLE-TC** security notions – capturing the notions of *strong tag consistency* and *tag consistency*

– are defined in Figure 3.14. Both the security notions aim to provide security against *duplicate faking* attacks. In a *duplicate faking* attack, two unidentical messages – one fake message produced by an adversary and a legitimate one produced by an honest client – produce the same tag, thereby causing loss of message and hampers the integrity. In addition to *duplicate faking* attack, MLE-STC security aims to prevent *erasure* attack, where the adversary replaces the ciphertext with any ciphertext that decrypts successfully. The adversarial advantage is the probability of the event when the adversary is able to compute a message, a ciphertext and a tag, such that, (1) on encryption of the supplied message, the tag produced is identical to the supplied one, and (2) on decryption of the supplied ciphertext with supplied tag, the message computed is unidentical to the supplied one. This adversarial advantage is desired to be negligible.

According to the MLE-STC game, at the first step, the adversary returns a valid message M_0 , a valid ciphertext C_1 and a tag T_1 . Then the following operations are performed: key, ciphertext and tag for message M_0 are computed as $(K_0, C_0, T_0) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M_0)$; and ciphertext C_1 (with tag T_1) is decrypted using key K_0 as $M_1 := \phi. \mathcal{D}(\text{params}^{(\phi)}, K_0, C_1, T_1)$. The game returns 1, if $T_0 = T_1$ under $M_0 \neq M_1$. For MLE-TC game, an additional comparison for validity of M_1 (i.e. $M_1 \neq \perp$) is performed, before returning the bit 1.

The advantage of the MLE-TC adversary \mathcal{A} against ϕ is defined as follows:

$$\text{Adv}_{\phi, \mathcal{A}}^{\text{MLE-TC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{MLE-TC}_\phi^{\mathcal{A}}(1^\lambda) = 1].$$

The advantage of the MLE-STC adversary \mathcal{A} against ϕ is defined as follows:

$$\text{Adv}_{\phi, \mathcal{A}}^{\text{MLE-STC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{MLE-STC}_\phi^{\mathcal{A}}(1^\lambda) = 1].$$

The ϕ is MLE-TC (or MLE-STC) secure, if, for all PPT adversaries \mathcal{A} , the value of $\text{Adv}_{\phi, \mathcal{A}}^{\text{MLE-TC}}(1^\lambda)$ (or $\text{Adv}_{\phi, \mathcal{A}}^{\text{MLE-STC}}(1^\lambda)$) is negligible.

<p>Game MLE-KR$_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda)$</p> <hr/> $(\text{params}^{(\phi)}, \mathcal{K}^{(\phi)}, \mathcal{M}^{(\phi)}, \mathcal{C}^{(\phi)}, \mathcal{T}^{(\phi)}) := \phi. \text{Setup}(1^\lambda);$ $(M, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$ $(K, C, T) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M);$ $K' := \mathcal{A}(1^\lambda, C, T, Z);$ $\text{return } K = K';$
--

Figure 3.15: Security game MLE-KR for the MLE scheme ϕ .

MLE-KR SECURITY. Since no *MLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpredictable* message-source $\mathcal{S}(\cdot)$, (for details see Section 3.4.1). The **MLE-KR** security – capturing the notion of *key recovery against chosen distribution* attack – is defined in Figure 3.15. The adversarial advantage is the probability of the event when the adversary is able to deduce the encryption key, and it is desired to be negligible.

According to the **MLE-KR** game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of string M and auxiliary information Z ; and after that M is encrypted as $(K, C, T) := \phi \cdot \mathcal{E}(\text{params}^{(\phi)}, M)$. Given access to C , T and Z , the adversary returns a key K' . The game returns 1, if $K = K'$.

The advantage of the **MLE-KR** adversary \mathcal{A} against ϕ for the message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\text{Adv}_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-KR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{MLE-KR}_{\phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda) = 1].$$

The ϕ is **MLE-KR** secure over a set of valid PPT message-sources for ϕ , $\bar{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \bar{\mathcal{S}}$, if $\text{Adv}_{\phi, \mathcal{S}_i, \mathcal{A}}^{\text{MLE-KR}}(1^\lambda)$ is *negligible*. The ϕ is **MLE-KR** secure, for all PPT adversaries \mathcal{A} , if $\text{Adv}_{\phi, \mathcal{S}, \mathcal{A}}^{\text{MLE-KR}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for ϕ .

Note: It is easy to show that an *MLE* scheme secure against **MLE-PRV\$** attack is also secure against **MLE-KR** attack.

Remark. Note that a *block-level message-locked encryption (BL-MLE)* $\xi = (\xi \cdot \mathcal{E}, \xi \cdot \mathcal{D})$ over *setup* algorithm $\xi \cdot \text{Setup}$, is a special type of *MLE* where the messages are fixed-sized blocks rather than of arbitrary length. In other words, in *BL-MLE*, the *message-space* is $\mathcal{M}^{(\xi)} := \{0, 1\}^B$, where the *block-length* $B \in \mathbb{N}$ is fixed.

3.4.3 Existing *MLE* schemes

In this section, we discuss several constructions of *MLE* and its variants. In the first three subsections, we describe the three most popular constructions of *MLE*, namely, **Convergent Encryption (CE)**, **Hash and Convergent Encryption 2 (HCE2)** and **Randomized Convergent Encryption (RCE)**. We describe other *MLE* constructions in Section 3.4.3.4.

3.4.3.1 Convergent Encryption (CE)

The first *MLE* construction, namely, Convergent Encryption (CE), was proposed by Douceur et al. in 2002 [DAB⁺02]. CE uses the *hash function* $\vartheta = (\vartheta, \mathcal{H})$ over ϑ . *Setup* and the *symmetric encryption* $\rho = (\rho, \mathcal{GEN}, \rho, \mathcal{E}, \rho, \mathcal{D})$ over ρ . *Setup*. The pseudocode is given in Figure 3.16.

$\text{CE. } \mathcal{E}(\text{params}^{(\text{CE})}, M)$	$\text{CE. } \mathcal{D}(\text{params}^{(\text{CE})}, K, C, T)$
<i># Computing key, ciphertext, tag.</i>	<i># Verifying ciphertext and tag.</i>
$K := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, M);$	$T' := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, C);$
$C := \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M);$	If ($T' \neq T$), then return \perp ;
$T := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, C);$	<i># Computing final output.</i>
<i># Returning final output.</i>	$M := \rho. \mathcal{D}(\text{params}^{(\rho)}, K, C);$
return $(K, C, T);$	return $M;$

Figure 3.16: Algorithmic description of the *MLE* scheme CE.

Security of CE

CE is MLE-PRV, MLE-PRV\$, MLE-STC and MLE-TC secure [BKR13b]. Besides this, it is easy to show that CE construction is also secure against MLE-KR attack.

3.4.3.2 Hash and Convergent Encryption 2 (HCE2)

In 2013, Bellare, Keelvedhi and Ristenpart proposed an *MLE* construction Hash and Convergent Encryption 2 (HCE2) [BKR13b]. It utilizes the *hash function* $\vartheta = (\vartheta, \mathcal{H})$ over ϑ . *Setup* and the *symmetric encryption* $\rho = (\rho, \mathcal{GEN}, \rho, \mathcal{E}, \rho, \mathcal{D})$ over ρ . *Setup*. The pseudocode is given in Figure 3.17.

Security of HCE2

HCE2 is MLE-PRV, MLE-PRV\$ and MLE-TC secure [BKR13b]. Besides this, it is easy to show that HCE2 construction is also secure against MLE-KR attack. Also, it is proven that HCE2 is not MLE-STC secure [BKR13b].

3.4.3.3 Randomized Convergent Encryption (RCE)

In 2013, Bellare, Keelvedhi and Ristenpart proposed a randomized *MLE* construction Randomized Convergent Encryption (RCE) [BKR13b]. It utilizes

HCE2. $\mathcal{E}(params^{(HCE2)}, M)$	HCE2. $\mathcal{D}(params^{(HCE2)}, K, C, T)$
# Computing key, ciphertext, tag.	# Computing message.
$K := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$	$M := \rho \cdot \mathcal{D}(params^{(\rho)}, K, C);$
$C := \rho \cdot \mathcal{E}(params^{(\rho)}, K, M);$	
$T := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K);$	# Verifying tag.
# Returning final output.	$K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$
return $(K, C, T);$	$T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K');$
	If $(T' \neq T)$, then return \perp ;
	Else return M ;

Figure 3.17: Algorithmic description of the *MLE* scheme HCE2.

the *hash function* $\vartheta = (\vartheta \cdot \mathcal{H})$ over $\vartheta \cdot \text{Setup}$ and the *symmetric encryption* $\rho = (\rho \cdot \mathcal{GEN}, \rho \cdot \mathcal{E}, \rho \cdot \mathcal{D})$ over $\rho \cdot \text{Setup}$. The pseudocode is given in Figure 3.18.

RCE. $\mathcal{E}(params^{(RCE)}, M)$	RCE. $\mathcal{D}(params^{(RCE)}, K, C, T)$
# Computing key, ciphertext, tag.	# Computing message.
$L \xleftarrow{\$} \mathcal{K}^{(\rho)};$	$C_1 \ C_2 := C; L := C_2 \oplus K;$
$C_1 := \rho \cdot \mathcal{E}(params^{(\rho)}, L, M);$	$M := \rho \cdot \mathcal{D}(params^{(\rho)}, L, C_1);$
$K := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$	
$C_2 := L \oplus K; C := C_1 \ C_2;$	# Verifying tag.
$T := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K);$	$K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$
# Returning final output.	$T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K');$
return $(K, C, T);$	If $T' \neq T$, then return \perp ;
	Else return M ;

Figure 3.18: Algorithmic description of the *MLE* scheme RCE.

Security of RCE

RCE is MLE-PRV, MLE-PRV\$ and MLE-TC secure [BKR13b]. Besides this, it is easy to show that RCE construction is also secure against MLE-KR attack. Also, it is proven that RCE is not MLE-STC secure [BKR13b].

3.4.3.4 Other MLE schemes

Besides the *MLE* constructions CE, HCE2 and RCE, we discuss the various pieces of work done by several researchers in the field of *MLE*. In 2013,

Bellare, Keelvedhi and Ristenpart formalized the cryptographic primitive *message-locked encryption (MLE)*, besides proposing HCE2 and RCE, and providing a systematic discussion of various *MLE* schemes [BKR13b]. In the same year, in a separate paper, Bellare, Keelvedhi and Ristenpart designed a new system **DupLESS** that supports deduplication even if the message entropy is low [BKR13a].

In 2015, Beelare and Keelvedhi extended *message-locked encryption (MLE)* to design *interactive message-locked encryption (i-MLE)*, and have addressed the cases, when the messages are correlated, as well as dependent on the public parameters, leading to weakening of privacy [BK15]. Abadi *et al.* gave two new constructions for the *i-MLE*; these fully randomized schemes also supported equality-testing algorithm for finding ciphertexts derived from identical messages [ABM⁺13]. Jiang *et al.* gave an efficient logarithmic-time deduplication scheme that substantially reduces the equality-testing in the *i-MLE* schemes [JCW⁺16].

Canard, Laguillaumie and Paindavoine introduced *deduplication consistency* – a new security property – that prevents the clients from bypassing the deduplication protocol. This is accomplished by introducing a new feature named *verifiability (of the well-formation) of ciphertext at the server* [CLP16]. They also proposed a new ElGamal-based construction satisfying this property. Wang *et al.* proposed a stronger security notion PRV-CDA3 for *privacy against chosen distribution attack*, and showed that their new construction *ME* is secure in this model [WCQ⁺17].

Chen *et al.* proposed the *block-level message-locked encryption (BL-MLE)*, which is nothing but breaking a big message into smaller chunks – called *blocks* – and then applying *MLE* on the individual *blocks* [CMYG15]. Huang, Zhang and Wang showed how to integrate the functionality *proof of storage (PoS)* with *MLE* by using a new data structure *Quadruple Tags* [HZW17]. Zhao and Chow proposed the use of *BL-MLE* to design *Efficiently Updatable Block-Level Message-Locked Encryption (UMLE)* scheme which has an additional function of updating the ciphertext that costs sub-linear time [ZC17].

3.4.4 Proof-of-Ownership Function

The *proof-of-ownership (PoW)* function is an interactive protocol, where the file-owner proves the ownership of a file to the cloud storage. This protocol assumes that the adversary does not have access to the entire ciphertext which was uploaded onto the cloud by some previous (or first) owner, but he may know the tag, which is a small fraction of the entire information. The protocol works in the following way: first the cloud storage provider

generates a challenge Q corresponding to the ciphertext, and sends it to the client; then in response, the client computes the proof P corresponding to the given challenge Q and his own ciphertext, and sends it back to the cloud; and finally the cloud verifies the proof, and if the verification is successful, then the file is stored and the client is granted access, otherwise the protocol aborts.

3.4.5 Updatable block-level message-locked encryption (UMLE)

The definition of *updatable block-level message-locked encryption (UMLE)* has already been described in [ZC17]. We briefly re-discuss it below, with a few suitable changes in the notation to suit the present context.

3.4.5.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *UMLE* scheme $\zeta = (\zeta.\text{KeyGen}, \zeta.\text{Enc}, \zeta.\text{TagGen}, \zeta.\text{Dec}, \zeta.\text{Update}, \zeta.\text{UpdateTag}, \zeta.\text{PoWPrf}, \zeta.\text{PoWVer})$ is an 8-tuple of algorithms over the *setup* algorithm $\zeta.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\zeta.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\zeta)}$, key-space $\mathcal{K}^{(\zeta)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\zeta)} \subseteq \{0, 1\}^*$, ciphertext-space $\mathcal{C}^{(\zeta)} \subseteq \{0, 1\}^*$ and tag-space $\mathcal{T}^{(\zeta)} \subseteq \{0, 1\}^*$.
2. The PPT *key generation* algorithm $\zeta.\text{KeyGen}(\cdot)$ takes as input parameter $\text{params}^{(\zeta)}$ and message $M \in \mathcal{M}^{(\zeta)}$ such that $M = M[1]\|M[2]\|\cdots\|M[n]$, and returns a set of keys $(k_{mas}, k_1, k_2, \dots, k_n) := \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M)$, where $k_{mas}, k_1, k_2, \dots, k_n \in \mathcal{K}^{(\zeta)}$. It uses two routines: $\zeta.\text{B-KeyGen}(\cdot)$ that takes as input i th block of message $M[i]$, and returns the block key k_i ; and $\zeta.\text{M-KeyGen}(\cdot)$ that takes as input message M , and returns the master key k_{mas} .
3. The PPT *encryption* algorithm $\zeta.\text{Enc}(\cdot)$ takes as input parameter $\text{params}^{(\zeta)}$, set of keys $(k_{mas}, k_1, k_2, \dots, k_n)$, such that $k_{mas}, k_1, k_2, \dots, k_n \in \mathcal{K}^{(\zeta)}$ and message $M \in \mathcal{M}^{(\zeta)}$, and returns ciphertext $C := \zeta.\text{Enc}(\text{params}^{(\zeta)}, (k_{mas}, k_1, k_2, \dots, k_n), M)$, where $C \in \mathcal{C}^{(\zeta)}$. It uses two routines: $\zeta.\text{B-Enc}(\cdot)$ that takes as input i th message-block $M[i]$ and corresponding key k_i , and returns ciphertext-block $C[i]$; and $\zeta.\text{BK-Enc}(\cdot)$ that takes as input set of block keys (k_1, k_2, \dots, k_n) , and returns encrypted block keys $C[n+1]\|C[n+2]\|\cdots\|C[n']$, where $n' \in \mathcal{O}(n)$. Then, the ciphertext is $C = C[1]\|C[2]\|\cdots\|C[n']$.

4. The PPT *tag generation* algorithm $\zeta.\text{TagGen}(\cdot)$ takes as input parameter $params^{(\zeta)}$ and ciphertext $C \in \mathcal{C}^{(\zeta)}$, and returns tag $T := \zeta.\text{TagGen}(params^{(\zeta)}, C)$, where $T \in \mathcal{T}^{(\zeta)}$. It uses two routines: $\zeta.\text{B-TagGen}(\cdot)$ that takes as input i th block of ciphertext $C[i]$, and returns block tag $T[i]$; and $\zeta.\text{M-TagGen}(\cdot)$ that takes as input ciphertext C , and returns file tag $T[0]$. Then, the tag is $T = T[0]\|T[1]\|T[2]\|\cdots\|T[n']$.
5. The *decryption* algorithm $\zeta.\text{Dec}(\cdot)$ is a deterministic algorithm that takes as input parameter $params^{(\zeta)}$, master key $k_{mas} \in \mathcal{K}^{(\zeta)}$ and ciphertext $C \in \mathcal{C}^{(\zeta)}$, and returns message $M := \zeta.\text{Dec}(params^{(\zeta)}, k_{mas}, C)$, where $M \in \mathcal{M}^{(\zeta)} \cup \{\perp\}$. It uses two routines: $\zeta.\text{BK-Dec}(\cdot)$ that takes as input master key k_{mas} and encrypted block keys $C[n+1]\|C[n+2]\|\cdots\|C[n']$, and returns set of block keys $k_1, k_2, \dots, k_n \in \mathcal{K}^{(\zeta)}$; and $\zeta.\text{B-Dec}(\cdot)$ that takes as input ciphertext block $C[i]$ and corresponding key k_i , and returns file block $M[i]$. Then, the message is $M := M[1]\|M[2]\|\cdots\|M[n]$.
6. The *update ciphertext* algorithm $\zeta.\text{Update}(\cdot)$ takes as input parameter $params^{(\zeta)}$, master key $k_{mas} \in \mathcal{K}^{(\zeta)}$, index of the block to be updated $i \in \mathbb{N}$, new message block $M_{new} \in \mathcal{M}^{(\zeta)}$ and ciphertext $C \in \mathcal{C}^{(\zeta)}$, and returns $(k'_{mas}, C') := \zeta.\text{Update}(params^{(\zeta)}, (k_{mas}, i, M_{new}), C)$, where new master key $k'_{mas} \in \mathcal{K}^{(\zeta)}$ and new ciphertext $C' \in \mathcal{C}^{(\zeta)}$.
7. The *update tag* algorithm $\zeta.\text{UpdateTag}(\cdot)$ takes as input parameter $params^{(\zeta)}$, old tag $T \in \mathcal{T}^{(\zeta)}$, old ciphertext $C \in \mathcal{C}^{(\zeta)}$ and new ciphertext $C' \in \mathcal{C}^{(\zeta)}$, and returns new tag $T' := \zeta.\text{UpdateTag}(params^{(\zeta)}, T, C, C')$, where $T' \in \mathcal{T}^{(\zeta)}$.
8. The PPT *PoW algorithm for prover* algorithm $\zeta.\text{PoWPrf}(\cdot)$ takes as input parameter $params^{(\zeta)}$, challenge Q and message $M \in \mathcal{M}^{(\zeta)}$, and returns proof $P := \zeta.\text{PoWPrf}(params^{(\zeta)}, Q, M)$.
9. The PPT *PoW algorithm for verifier* algorithm $\zeta.\text{PoWVer}(\cdot)$ takes as input parameter $params^{(\zeta)}$, challenge Q , tag $T \in \mathcal{T}^{(\zeta)}$ and proof P , and returns value $val := \zeta.\text{PoWVer}(params^{(\zeta)}, Q, T, P)$, where $val \in \{\text{TRUE}, \text{FALSE}\}$.

Note: When tag is incorporated in the ciphertext itself, the ciphertext will be expanded. We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\phi)}$.

3.4.5.2 Correctness

DECRYPTION CORRECTNESS. This condition requires that if the ciphertext is generated from the correct input block, decryption will produce the correct output block. Mathematically, the *decryption correctness* of a *UMLE* can be defined as follows.

Suppose:

- $M = M[1] \| M[2] \| \cdots \| M[n]$,
- For all $i \in [n]$, $k_i := \zeta \cdot \text{B-KeyGen}(M[i])$, and
- For all $i \in [n]$, $C[i] := \zeta \cdot \text{B-Enc}(k_i, M[i])$.

Then, the *decryption correctness* of ζ requires that $\zeta \cdot \text{B-Dec}(k_i, C[i]) = M[i]$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\zeta)}$.

BLOCK KEY RETRIEVAL CORRECTNESS. This condition requires that if the master key is generated from the correct input, then the block keys can be computed from the master key. Mathematically, the *block key retrieval correctness* of a *UMLE* can be defined as follows.

Suppose:

- $M = M[1] \| M[2] \| \cdots \| M[n]$,
- For all $i \in [n]$, $k_i := \zeta \cdot \text{B-KeyGen}(M[i])$,
- $k_{mas} := \zeta \cdot \text{M-KeyGen}(M)$, and
- $C[n+1] \| C[n+2] \| \cdots \| C[n'] := \zeta \cdot \text{BK-Enc}(k_1, k_2, \dots, k_n)$.

Then, the *block key retrieval correctness* of ζ requires that $\zeta \cdot \text{BK-Dec}(k_{mas}, C[n+1] \| C[n+2] \| \cdots \| C[n']) = (k_1, k_2, \dots, k_n)$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\zeta)}$.

BLOCK TAG CORRECTNESS. This condition requires that the tag generated from two identical message-blocks will always be identical. Mathematically, the *block tag correctness* of a *UMLE* can be defined as follows.

Suppose, for all $i, j \in [n]$:

- $M = M[1] \| M[2] \| \cdots \| M[n]$,
- $M[i] = M[j]$,
- $T[i] := \zeta \cdot \text{B-TagGen}(\zeta \cdot \text{B-Enc}(\zeta \cdot \text{B-KeyGen}(M[i]), M[i]))$, and
- $T[j] := \zeta \cdot \text{B-TagGen}(\zeta \cdot \text{B-Enc}(\zeta \cdot \text{B-KeyGen}(M[j]), M[j]))$.

Then, the *block tag correctness* of ζ requires that $T[i] = T[j]$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\zeta)}$.

FILE TAG CORRECTNESS. This condition requires that the tag generated from two identical messages will always be identical. Mathematically, the *file tag correctness* of a *UMLE* can be defined as follows.

Suppose:

- $M_0 = M_1$,

- $C_0 := \zeta.\text{Enc}(\text{params}^{(\zeta)}, \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M_0), M_0)$,
- $T_0 := \zeta.\text{TagGen}(\text{params}^{(\zeta)}, C_0)$,
- $C_1 := \zeta.\text{Enc}(\text{params}^{(\zeta)}, \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M_1), M_1)$, and
- $T_1 := \zeta.\text{TagGen}(\text{params}^{(\zeta)}, C_1)$.

Then, the *file tag correctness* of ζ requires that $T_0 = T_1$, for all $(\lambda, M_0, M_1) \in \mathbb{N} \times \mathcal{M}^{(\zeta)} \times \mathcal{M}^{(\zeta)}$.

UPDATE CORRECTNESS. This condition requires that the updated ciphertext and updated master key should always produce updated message on decryption. Mathematically, the *update correctness* of a *UMLE* can be defined as follows.

Suppose:

- $M = M[1] \| M[2] \| \cdots \| M[n]$,
- $(k_{mas}, k_1, k_2, \dots, k_n) := \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M)$,
- $C := \zeta.\text{Enc}(\text{params}^{(\zeta)}, (k_{mas}, k_1, k_2, \dots, k_n), M)$, and
- $(k'_{mas}, C') := \zeta.\text{Update}(\text{params}^{(\zeta)}, (k_{mas}, i, M_{new}), C)$

Then, the *update correctness* of ζ requires that $\zeta.\text{Dec}(\text{params}^{(\zeta)}, k'_{mas}, C') = M[1] \| M[2] \| \cdots \| M[i-1] \| M_{new} \| M[i+1] \| \cdots \| M[n]$.

PoW CORRECTNESS. This condition requires that a proof – computed correctly from the message – should be able to verify successfully. Mathematically, the *PoW correctness* of a *UMLE* can be defined as follows.

Suppose:

- $M = M[1] \| M[2] \| \cdots \| M[n]$,
- $C := \zeta.\text{Enc}(\text{params}^{(\zeta)}, \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M), M)$, and
- $T := \zeta.\text{TagGen}(\text{params}^{(\zeta)}, C)$.

Then, the *PoW correctness* of ζ requires that for any challenge Q , we have probability $\Pr[\zeta.\text{PoWVer}(\text{params}^{(\zeta)}, Q, T, \zeta.\text{PoWPrf}(\text{params}^{(\zeta)}, Q, M)) = \text{TRUE}] = 1$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\zeta)}$.

3.4.5.3 Security definitions

In Figure 3.19, 3.20, 3.21 and 3.22, we define the security games **UMLE-PRV**, **UMLE-STC** and **UMLE-TC**, **UMLE-CXH** and **UMLE-UNC**, respectively, for the *UMLE* scheme $\zeta = (\zeta.\text{KeyGen}, \zeta.\text{Enc}, \zeta.\text{TagGen}, \zeta.\text{Dec}, \zeta.\text{Update}, \zeta.\text{UpdateTag}, \zeta.\text{PoWPrf}, \zeta.\text{PoWVer})$. As usual, all the games are written using the challenger-adversary framework.

UMLE-PRV SECURITY. Since no *UMLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpre-*

```

Game UMLE-PRV $_{\zeta, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b)$ 


---


 $(params^{(\zeta)}, \mathcal{K}^{(\zeta)}, \mathcal{M}^{(\zeta)}, \mathcal{C}^{(\zeta)}, \mathcal{T}^{(\zeta)}) := \zeta.\text{Setup}(1^\lambda);$ 
 $(\mathbf{M}_1, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$ 
for  $(i := 1, 2, \dots, m(1^\lambda))$ 
     $\mathbf{M}_0^{(i)} \xleftarrow{\$} \{0, 1\}^{|\mathbf{M}_1^{(i)}|};$ 
     $\mathbf{K}^{(i)} := \zeta.\text{KeyGen}(params^{(\zeta)}, \mathbf{M}_b^{(i)});$ 
     $\mathbf{C}^{(i)} := \zeta.\text{Enc}(params^{(\zeta)}, \mathbf{K}^{(i)}, \mathbf{M}_b^{(i)});$ 
     $\mathbf{T}^{(i)} := \zeta.\text{TagGen}(params^{(\zeta)}, \mathbf{C}^{(i)});$ 
 $b' := \mathcal{A}(1^\lambda, \mathbf{C}, \mathbf{T}, Z);$ 
return  $b';$ 

```

Figure 3.19: Security game UMLE-PRV for the UMLE scheme ζ .

dictable message-source $\mathcal{S}(\cdot)$ (for details see Section 3.4.1). The UMLE-PRV security – capturing the notion of *indistinguishability privacy against chosen distribution* attack – is defined in Figure 3.19. The adversarial advantage is the probability of the event when the adversary is able to distinguish between the encryptions of a computed message and a random string, and it is desired to be negligible.

According to the UMLE-PRV game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of vector of strings $\mathbf{M}_1 = \mathbf{M}_1^{(1)} \circ \mathbf{M}_1^{(2)} \circ \dots \circ \mathbf{M}_1^{(m(1^\lambda))}$ and auxiliary information Z . Now, for each string $\mathbf{M}_1^{(i)}$, for all $i \in [m(1^\lambda)]$, following operations are performed: first it computes random string of length $|\mathbf{M}_1^{(i)}|$ as $\mathbf{M}_0^{(i)}$; then it computes key as $\mathbf{K}^{(i)} := \zeta.\text{KeyGen}(params^{(\zeta)}, \mathbf{M}_b^{(i)})$, depending upon value of $b \in \{0, 1\}$; next it computes ciphertext as $\mathbf{C}^{(i)} := \zeta.\text{Enc}(params^{(\zeta)}, \mathbf{K}^{(i)}, \mathbf{M}_b^{(i)})$, depending upon value of $b \in \{0, 1\}$; and finally it computes tag as $\mathbf{T}^{(i)} := \zeta.\text{TagGen}(params^{(\zeta)}, \mathbf{C}^{(i)})$. Given access to \mathbf{C} , \mathbf{T} and Z , the adversary returns a bit b' .

The advantage of the UMLE-PRV adversary \mathcal{A} against ζ for the message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\begin{aligned} Adv_{\zeta, \mathcal{S}, \mathcal{A}}^{\text{UMLE-PRV}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{UMLE-PRV}_{\zeta, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{UMLE-PRV}_{\zeta, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|. \end{aligned}$$

The ζ is UMLE-PRV secure over a set of valid PPT message-sources for ζ , $\overline{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \overline{\mathcal{S}}$, if $Adv_{\zeta, \mathcal{S}_i, \mathcal{A}}^{\text{UMLE-PRV}}(1^\lambda)$ is negligible. The ζ is UMLE-PRV secure, for all PPT

adversaries \mathcal{A} , if $Adv_{\zeta, \mathcal{S}, \mathcal{A}}^{\text{UMLE-PRV}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for ζ .

Game $\text{UMLE-STC}_\zeta^{\mathcal{A}}(1^\lambda)$	$\text{UMLE-TC}_\zeta^{\mathcal{A}}(1^\lambda)$
$(params^{(\zeta)}, \mathcal{K}^{(\zeta)}, \mathcal{M}^{(\zeta)}, \mathcal{C}^{(\zeta)}, \mathcal{T}^{(\zeta)}) := \zeta.\text{Setup}(1^\lambda);$ $(M_0, C_1) := \mathcal{A}(1^\lambda);$ If $(M_0 = \perp) \vee (C_1 = \perp)$, then return 0; $T_1[0] := \zeta.\text{M-TagGen}(params^{(\zeta)}, C_1);$ $K_0 := \zeta.\text{KeyGen}(params^{(\zeta)}, M_0);$ $C_0 := \zeta.\text{Enc}(params^{(\zeta)}, K_0, M_0);$ $T_0[0] := \zeta.\text{M-TagGen}(params^{(\zeta)}, C_0);$ $M_1 := \zeta.\text{Dec}(params^{(\zeta)}, K_0, C_1);$ If $(T_0[0] = T_1[0]) \wedge (M_0 \neq M_1)$ $\wedge (M_1 \neq \perp)$ return 1; Else return 0;	

Figure 3.20: Security games UMLE-STC and UMLE-TC for the *UMLE* scheme ζ .

UMLE-STC AND UMLE-TC SECURITY. The **UMLE-STC** and **UMLE-TC** security notions – capturing the notions of *strong tag consistency* and *tag consistency* – are defined in Figure 3.20, in the form of two games. The adversarial advantage is the probability of the event when the adversary is able to compute a message and a ciphertext, such that, (1) the tag computed from the supplied message is identical to the tag produced by the supplied ciphertext, and (2) on decryption of the supplied ciphertext, the message computed is unidentical to the supplied one. This adversarial advantage is desired to be negligible.

According to the **UMLE-STC** game, at the first step, the adversary returns a valid message M_0 and a valid ciphertext C_1 . Then the following operations are performed: first it computes file tag for C_1 as $T_1[0] := \zeta.\text{M-TagGen}(params^{(\zeta)}, C_1)$; then it generates key for M_0 as $K_0 := \zeta.\text{KeyGen}(params^{(\zeta)}, M_0)$; next it encrypts M_0 as $C_0 := \zeta.\text{Enc}(params^{(\zeta)}, K_0, M_0)$; after that it computes file tag for C_0 as $T_0[0] := \zeta.\text{M-TagGen}(params^{(\zeta)}, C_0)$; and finally it decrypts C_1 using K_0 as $M_1 := \zeta.\text{Dec}(params^{(\zeta)}, K_0, C_1)$. The game returns 1, if $T_0[0] = T_1[0]$ under $M_0 \neq M_1$. For **UMLE-TC** game, an additional comparison for validity of M_1 (i.e. $M_1 \neq \perp$) is performed, before returning the bit 1.

The advantage of the **UMLE-TC** adversary \mathcal{A} against ζ is defined as

follows:

$$\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-TC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{UMLE-TC}_\zeta^{\mathcal{A}}(1^\lambda) = 1].$$

The advantage of the **UMLE-STC** adversary \mathcal{A} against ζ is defined as follows:

$$\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-STC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{UMLE-STC}_\zeta^{\mathcal{A}}(1^\lambda) = 1].$$

The ζ is **UMLE-TC** (or **UMLE-STC**) secure, if, for all PPT adversaries \mathcal{A} , the value of $\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-TC}}(1^\lambda)$ (or $\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-STC}}(1^\lambda)$) is *negligible*.

<pre> Game UMLE-CXH$_{\zeta}^{\mathcal{A}}(1^\lambda, b)$ _____ ($\text{params}^{(\zeta)}, \mathcal{K}^{(\zeta)}, \mathcal{M}^{(\zeta)}, \mathcal{C}^{(\zeta)}, \mathcal{T}^{(\zeta)}$) := ζ.$\text{Setup}(1^\lambda)$; (M_0, M_1, i) := $\mathcal{A}_1(1^\lambda)$; If ($M_0[j] \neq M_1[j]$) for any $j \neq i$ return Error; $K_0 := \zeta$.$\text{KeyGen}(\text{params}^{(\zeta)}, M_0)$; $C_0 := \zeta$.$\text{Enc}(\text{params}^{(\zeta)}, K_0, M_0)$; $K_1 := \zeta$.$\text{KeyGen}(\text{params}^{(\zeta)}, M_1)$; $C_1 := \zeta$.$\text{Enc}(\text{params}^{(\zeta)}, K_1, M_1)$; ($K'_1, C'_1$) := ζ.$\text{Update}(\text{params}^{(\zeta)}, (K_1, i, M_0[i]), C_1)$; If ($b = 0$), then $C^* := C_0$; Else $C^* := C'_1$; $b' := \mathcal{A}_2(1^\lambda, C^*)$; return b'; </pre>

Figure 3.21: Security game **UMLE-CXH** for the *UMLE* scheme ζ .

UMLE-CXH SECURITY. The **UMLE-CXH** security – capturing the notion of *context hiding* attack – is defined in Figure 3.21. Precisely, **UMLE-CXH** security aims to provide security against distinguishing between an updated ciphertext and a ciphertext encrypted from scratch, to ensure that the level of privacy is not compromised during the update process. This adversarial advantage is desired to be negligible.

According to the **UMLE-CXH** game, at first, the adversary returns two messages M_0 and M_1 of identical length, and index of block i where M_0 and M_1 differ. The game aborts if M_0 and M_1 differ at any block other than the i th one. Then the following operations are performed: first it generates key as $K_0 := \zeta$. $\text{KeyGen}(\text{params}^{(\zeta)}, M_0)$; then it computes ciphertext as $C_0 := \zeta$. $\text{Enc}(\text{params}^{(\zeta)}, K_0, M_0)$; next it generates key as $K_1 := \zeta$. $\text{KeyGen}(\text{params}^{(\zeta)}, M_1)$; after that it computes ciphertext as $C_1 := \zeta$. $\text{Enc}(\text{params}^{(\zeta)}, K_1, M_1)$; then i th block of C_1 is updated with i th block of M_0 as $(K'_1, C'_1) := \zeta$. $\text{Update}(\text{params}^{(\zeta)}, (K_1, i, M_0[i]), C_1)$; and finally it

checks if $b = 0$, then it assigns $C^* := C_0$, otherwise $C^* := C'_1$. Given access to C^* , the adversary returns a bit b' .

The advantage of the **UMLE-CXH** adversary \mathcal{A} for 1-block update in message, against ζ is defined as follows:

$$\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-CXH}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{UMLE-CXH}_\zeta^{\mathcal{A}}(1^\lambda, b=1) = 1] - \Pr[\text{UMLE-CXH}_\zeta^{\mathcal{A}}(1^\lambda, b=0) = 1] \right|.$$

The ζ is **UMLE-CXH** secure, for 1-block update in message, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\zeta, \mathcal{A}}^{\text{CXH}}(1^\lambda)$ is *negligible*.

<pre> Game UMLE-UNC$_{\zeta}^{\mathcal{A}}(1^\lambda)$ ($\text{params}^{(\zeta)}, \mathcal{K}^{(\zeta)}, \mathcal{M}^{(\zeta)}, \mathcal{C}^{(\zeta)}, \mathcal{T}^{(\zeta)}$) := $\zeta.\text{Setup}(1^\lambda)$; $\mathcal{S}(\cdot) := \mathcal{A}_1(1^\lambda)$; $(M, Z) \xleftarrow{\\$} \mathcal{S}(1^\lambda)$; $K := \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M)$; $C := \zeta.\text{Enc}(\text{params}^{(\zeta)}, K, M)$; $T := \zeta.\text{TagGen}(\text{params}^{(\zeta)}, C)$; $P^* := \mathcal{A}_2(1^\lambda, Q, Z)$; $P := \zeta.\text{PoWPrf}(\text{params}^{(\zeta)}, Q, M)$; If ($\zeta.\text{PoWVer}(\text{params}^{(\zeta)}, Q, T, P^*) = \text{TRUE}$) \wedge ($P^* \neq P$) return 1; Else return 0; </pre>

Figure 3.22: Security game **UMLE-UNC** for the **UMLE** scheme ζ .

UMLE-UNC SECURITY. The **UMLE-UNC** security – capturing the notion of *uncheatable chosen distribution* attack – is defined in Figure 3.22. Precisely, **UMLE-UNC** security aims to provide security against the adversaries who are trying to prove that they possess the entire file, when they actually have only a partial information about the file. This is to block the unauthorised ownership of the file. The adversarial advantage is the probability of the event when the adversary is able to compute a false proof that verifies successfully on *PoW algorithm for verifier*, and is unidentical to genuine proof. This adversarial advantage is desired to be negligible.

According to the **UMLE-UNC** game, at first, the adversary returns an *unpredictable* message-source $\mathcal{S}(\cdot)$. Then, the following operations are performed: first it invokes *unpredictable* message-source $\mathcal{S}(1^\lambda)$ for computation of string M and auxiliary information Z ; then it computes key as $K := \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M)$; next it encrypts M as $C := \zeta.\text{Enc}(\text{params}^{(\zeta)}, K,$

M); and finally it computes tag as $T := \zeta.\text{TagGen}(\text{params}^{(\zeta)}, C)$. Now, given challenge Q and Z , the adversary has to return a proof P^* . Then, the actual proof corresponding to challenge Q and message M is computed as $P := \zeta.\text{PoWPrf}(\text{params}^{(\zeta)}, Q, M)$. The game returns 1, if $\zeta.\text{PoWVer}(\text{params}^{(\zeta)}, Q, T, P^*) = \text{TRUE}$ under $P^* \neq P$, otherwise 0.

The advantage of the UMLE-UNC adversary \mathcal{A} against ζ for a message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-UNC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{UMLE-UNC}_\zeta^{\mathcal{A}}(1^\lambda) = 1].$$

The ζ is UMLE-UNC secure, for all PPT adversaries \mathcal{A} , if $\text{Adv}_{\zeta, \mathcal{A}}^{\text{UMLE-UNC}}(1^\lambda)$ is negligible.

3.4.6 Dictionary Attack

A *dictionary attack* is defined to be a *brute-force attack*, where the adversary first builds a dictionary off-line, and then processes every element of the dictionary to determine the correct solution against an online challenge. For example, suppose that the hash of a message is given as a challenge to the adversary for her to determine the correct message. If the entropy of the message is *low*, then the adversary generates the dictionary of all possible messages and their corresponding hash values off-line; and given the online challenge, she selects the message whose hash value matches the challenge. *Any deterministic MLE with low message entropy is vulnerable to dictionary attack.*

3.5 Hierarchical Access Control (HAC)

Hierarchical access control is a mechanism that allows the classes of people in an organisation with varying levels of privileges to access data based on their positions. Since, nowadays, most of the organisations operate with clearly defined hierarchies, and, also because their data is mainly stored in public servers (*e.g.* the *Cloud*), secure and efficient *HAC* solutions have gained importance.

One of the most popular techniques to implement *HAC* is by employing the cryptographic primitive *key assignment scheme (KAS)* – for generation, distribution and re-computation of keys – and by using *symmetric encryption (SE)* scheme – to impart privacy. *SE* is a 3-tuple of algorithms $\rho = (\rho.\mathcal{GEN}, \rho.\mathcal{E}, \rho.\mathcal{D})$, and has already been described in Section 2.2.4. *KAS* is a pair of algorithms $\Omega = (\Omega.\mathcal{GEN}, \Omega.\mathcal{DER})$, and will be elaborately discussed in Section 3.6. The *KAS-based HAC* protocol is composed

of three independent protocols: (1) ESTABLISH, (2) KEY DERIVE, and (3) DECRYPTION. Below we give the full textual description.

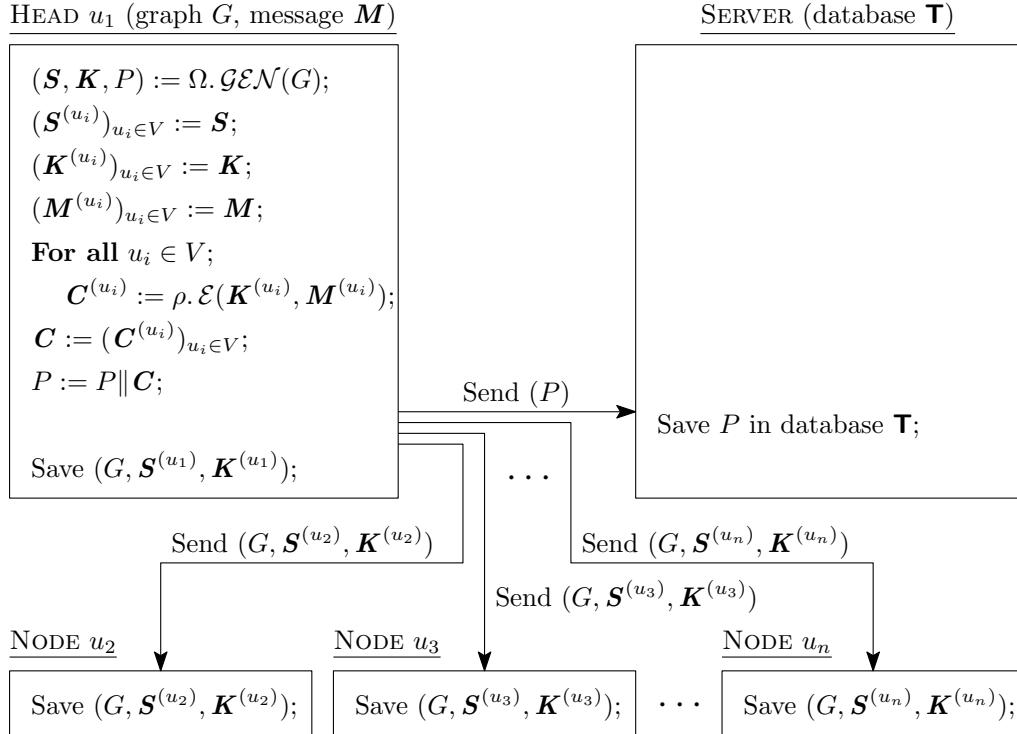
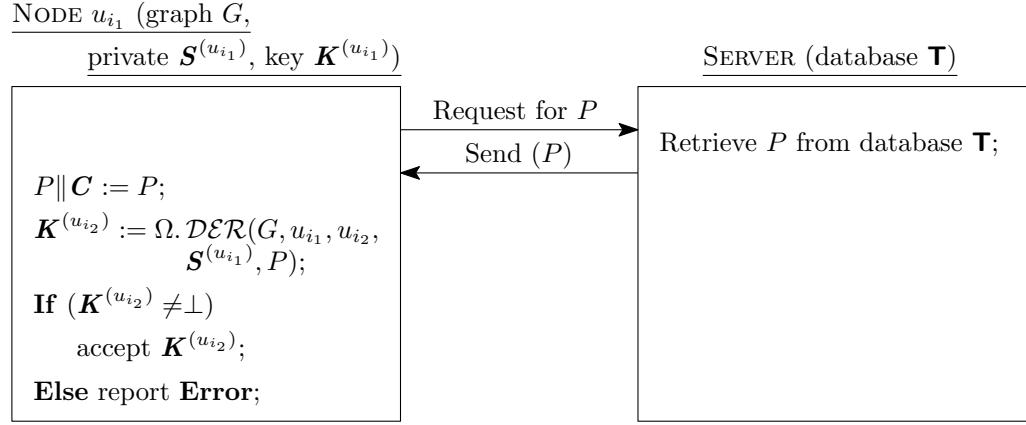
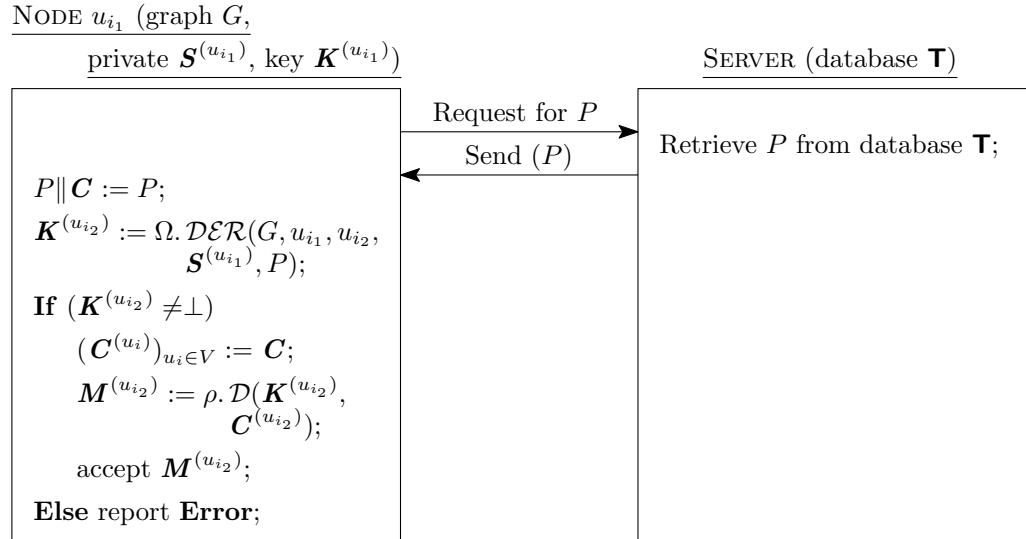


Figure 3.23: ESTABLISH protocol for KAS-based HAC.

- ESTABLISH. This protocol governs the sequence of operations executed, when the head of the organization wants to provide access to the vector of *files* M for the nodes in the access graph $G = (V, E)$, and to distributes the keys and other necessary information to all the nodes; the pictorial description is given in Figure 3.23. The head node u_1 performs the following operations: first it computes a 3-tuple $(S, K, P) := \Omega. \mathcal{GEN}(G)$; then it parses the vector of *private information* $(S^{(u_i)})_{u_i \in V} := S$, the vector of *keys* $(K^{(u_i)})_{u_i \in V} := K$, and the vector of *files* $(M^{(u_i)})_{u_i \in V} := M$; next, it computes the ciphertext $C^{(u_i)} := \rho. \mathcal{E}(K^{(u_i)}, M^{(u_i)})$ for each node $u_i \in V$; after that it computes the vector of *ciphertexts* $C := (C^{(u_i)})_{u_i \in V}$, and appends it to the public information P ; then it sends the *public information* P to the server; and finally it sends a 3-tuple $(G, S^{(u_i)}, K^{(u_i)})$ to each node $u_i \in V$. The nodes $u_i \in V$ store the supplied values of $(G, S^{(u_i)}, K^{(u_i)})$ corresponding to the *public information* P .

Figure 3.24: KEY DERIVE protocol for *KAS*-based *HAC*.

- **KEY DERIVE.** This protocol takes care of the operations performed when a node u_{i_1} computes the decryption key of a node $u_{i_2} \leq u_{i_1}$; the pictorial description is given in Figure 3.24. The node u_{i_1} requests the server to supply the *public information* P . In response, the server retrieves P from its database and sends it to u_{i_1} . After receiving P , node u_{i_1} performs the following operations: first it parses $P \parallel C := P$; then it computes the key $K^{(u_{i_2})} := \Omega.DER(G, u_{i_1}, u_{i_2}, S^{(u_{i_1})}, P)$ using the stored G and $S^{(u_{i_1})}$; and accepts $K^{(u_{i_2})}$ only if $K^{(u_{i_2})} \neq \perp$.

Figure 3.25: DECRYPTION protocol for *KAS*-based *HAC*.

- **DECRYPTION.** This protocol takes care of the operations performed

when a node u_{i_1} computes the file corresponding to node $u_{i_2} \leq u_{i_1}$; the pictorial description is given in Figure 3.25. The node u_{i_1} requests the server to supply the *public information* P . In response, the server retrieves P from its database and sends it to u_{i_1} . After receiving P , node u_{i_1} performs the following operations: first it parses $P \parallel \mathbf{C} := P$; then it computes the key $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ using the stored G and $\mathbf{S}^{(u_{i_1})}$; and if $\mathbf{K}^{(u_{i_2})} \neq \perp$, then it parses $(\mathbf{C}^{(u_i)})_{u_i \in V} := \mathbf{C}$, next it computes message $\mathbf{M}^{(u_{i_2})} := \rho \cdot \mathcal{D}(\mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})})$ and finally, accepts $\mathbf{M}^{(u_{i_2})}$.

3.6 Key Assignment Scheme (KAS)

From a high level, so far, the *HAC* problem has been solved using the following two-step methodology: distribute the secret keys to various classes of people in the organization such that the people in the higher class can derive the secret keys owned by the classes below it; after the distribution of keys, all the data is encrypted using *symmetric encryption* (data authentication may also be incorporated in some way). Loosely speaking, the secure generation of these secret keys, as done in the first step of the above *HAC* methodology, is known as *key assignment scheme (KAS)*. The idea of *KAS* and its practical construction was introduced by Akl and Taylor in 1983 [AT83]. Since then, for over three decades, a large number of *KAS* constructions have been proposed in the literature with extensive study of their security properties [ABFF09, AFB05, CC02, CCM15, CDM10, CMW06, CSM16a, CSM⁺16b, DSFM10, FP11, FPP13, HL90, SC02, SFM07a, WC01, YCN03].

In this section, we first describe the formal definition of *KAS*. Then we discuss its various existing constructions, along with the classification and construction methodology of *KAS*.

3.6.1 Definition

The definition of *KAS* has already been described in [FPP13], and the security notions in [ABFF09, ASFM06, DSFM10, FPP13, SFM07a]. We briefly re-discuss it below, with a few suitable changes in the notation to suit the present context.

3.6.1.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *KAS* $\Omega = (\Omega \cdot \mathcal{GEN}, \Omega \cdot \mathcal{DER})$ is a pair of algorithms over the *setup* algorithm $\Omega \cdot \text{Setup}$, satisfying the

following conditions.

1. The PPT *setup* algorithm $\Omega.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\Omega)}$, a set of access graphs $\Gamma^{(\Omega)}$ and *key-space* $\mathcal{K}^{(\Omega)} \subseteq \{0, 1\}^*$.
2. The PPT *key generation* algorithm $\Omega.\mathcal{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\Omega)}$ and graph $G = (V, E) \in \Gamma^{(\Omega)}$, and returns a 3-tuple $(\mathbf{S}, \mathbf{K}, P) := \Omega.\mathcal{GEN}(\text{params}^{(\Omega)}, G)$, where vector of *private information* $\mathbf{S} = (\mathbf{S}^{(u_i)})_{u_i \in V}$, vector of *keys* $\mathbf{K} = (\mathbf{K}^{(u_i)})_{u_i \in V}$ and *public information* $P \in \{0, 1\}^*$.

Note that $\mathbf{S}^{(u_i)} \in \{0, 1\}^*$ and $\mathbf{K}^{(u_i)} \in \mathcal{K}^{(\Omega)}$, for all $u_i \in V$, and all $G \in \Gamma^{(\Omega)}$.

3. The *key derivation* $\Omega.\mathcal{DER}(\cdot)$ is a deterministic *poly-time* algorithm such that $\mathbf{K}^{(u_{i_2})} := \Omega.\mathcal{DER}(\text{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$, where $u_{i_2} \leq u_{i_1}$ are two nodes of access graph $G \in \Gamma^{(\Omega)}$, $\mathbf{S}^{(u_{i_1})}$ is u_{i_1} 's *private information*, P is *public information*, and $\mathbf{K}^{(u_{i_2})}$ is u_{i_2} 's decryption key.

Note that $\mathbf{S}^{(u_{i_1})} \in \{0, 1\}^*$, $P \in \{0, 1\}^*$ and $\mathbf{K}^{(u_{i_2})} \in \mathcal{K}^{(\Omega)} \cup \{\perp\}$.

3.6.1.2 Correctness

KEY DERIVATION CORRECTNESS. This condition requires that if the vector of *keys*, vector of *private information* and *public information* are generated correctly, then the keys can be recomputed correctly. Mathematically, the *key derivation correctness* of a *KAS* can be defined as follows.

Suppose:

- $(\mathbf{S}, \mathbf{K}, P) := \Omega.\mathcal{GEN}(\text{params}^{(\Omega)}, G)$.

Then, the *key derivation correctness* of Ω requires that $\Omega.\mathcal{DER}(\text{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P) = \mathbf{K}^{(u_{i_2})}$, for all $(\lambda, G) \in \mathbb{N} \times \Gamma^{(\Omega)}$, and all nodes $u_{i_2} \leq u_{i_1}$.

3.6.1.3 Security

In Figures 3.26, 3.27 and 3.28, we define the security games **KAS-KI**, **KAS-SKI** and **KAS-KR**, respectively, for the *KAS* $\Omega = (\Omega.\mathcal{GEN}, \Omega.\mathcal{DER})$. As usual, all the games are written using the challenger-adversary framework.

KAS-KI AND KAS-SKI SECURITY. The **KAS-KI** and **KAS-SKI** security notions – capturing the notions of *key indistinguishability with respect to static adversary* and *strong key indistinguishability with respect to static adversary*

```

Game KAS-KI $_{\Omega}^{\mathcal{A}}(1^{\lambda}, G, b)$ 
( $params^{(\Omega)}, \Gamma^{(\Omega)}, \mathcal{K}^{(\Omega)}$ ) :=  $\Omega$ . Setup( $1^{\lambda}$ );
 $u_{i_1} := \mathcal{A}_1(1^{\lambda}, G)$ ;
( $\mathbf{S}, \mathbf{K}, P$ ) :=  $\Omega$ .  $\mathcal{GEN}$ ( $params^{(\Omega)}, G$ );
 $SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} | u_{i_2} < u_{i_1}\}$ ;
If ( $b = 1$ ), then  $T := \mathbf{K}^{(u_{i_1})}$ ;
Else  $T \xleftarrow{\$} \{0, 1\}^{|\mathbf{K}^{(u_{i_1})}|}$ ;
 $b' := \mathcal{A}_2(1^{\lambda}, G, P, SV^{(u_{i_1})}, T)$ ;
return  $b'$ ;

```

Figure 3.26: Security game KAS-KI for the KAS Ω .

```

Game KAS-SKI $_{\Omega}^{\mathcal{A}}(1^{\lambda}, G, b)$ 
( $params^{(\Omega)}, \Gamma^{(\Omega)}, \mathcal{K}^{(\Omega)}$ ) :=  $\Omega$ . Setup( $1^{\lambda}$ );
 $u_{i_1} := \mathcal{A}_1(1^{\lambda}, G)$ ;
( $\mathbf{S}, \mathbf{K}, P$ ) :=  $\Omega$ .  $\mathcal{GEN}$ ( $params^{(\Omega)}, G$ );
 $SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} | u_{i_2} < u_{i_1}\}$ ;
 $KV^{(u_{i_1})} := \{\mathbf{K}^{(u_{i_2})} \in \mathbf{K} | u_{i_1} < u_{i_2}\}$ ;
If ( $b = 1$ ), then  $T := \mathbf{K}^{(u_{i_1})}$ ;
Else  $T \xleftarrow{\$} \{0, 1\}^{|\mathbf{K}^{(u_{i_1})}|}$ ;
 $b' := \mathcal{A}_2(1^{\lambda}, G, P, SV^{(u_{i_1})}, KV^{(u_{i_1})}, T)$ ;
return  $b'$ ;

```

Figure 3.27: Security game KAS-SKI for the KAS Ω .

– are defined in Figures 3.26 and 3.27, in the form of two games. The adversarial advantage is the probability of the event when the adversary is able to distinguish between a genuine key and a random string, and it is desired to be negligible.

According to the KAS-KI game, at the first step, given access to graph G , the adversary returns a security class $u_{i_1} \in V$, that \mathcal{A} chooses to attack. Then, the following operations are performed: first it computes $(\mathbf{S}, \mathbf{K}, P)$ using Ω . \mathcal{GEN} ($params^{(\Omega)}, G$); then it computes $SV^{(u_{i_1})}$ as set of *private information* $\mathbf{S}^{(u_{i_2})}$ for classes $u_{i_2} \in V$ such that $u_{i_2} < u_{i_1}$; and finally it checks if $b = 1$, then T is assigned value of key $\mathbf{K}^{(u_{i_1})}$, otherwise it randomly chooses a $|\mathbf{K}^{(u_{i_1})}|$ -bit string T . Given access to G , P , $SV^{(u_{i_1})}$ and T , the adversary returns a bit b' . In KAS-SKI game, an additional knowledge of all keys of nodes u_{i_2} , such that $u_{i_1} < u_{i_2}$, is given, i.e. $KV^{(u_{i_1})} := \{\mathbf{K}^{(u_{i_2})} \in \mathbf{K} | u_{i_1} < u_{i_2}\}$ is given.

The advantage of the **KAS-KI** adversary \mathcal{A} against Ω on a graph $G \in \Gamma^{(\Omega)}$ is defined as follows:

$$\text{Adv}_{\Omega, \mathcal{A}, G}^{\text{KAS-KI}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{KAS-KI}_\Omega^{\mathcal{A}}(1^\lambda, G, b = 1) = 1] - \Pr[\text{KAS-KI}_\Omega^{\mathcal{A}}(1^\lambda, G, b = 0) = 1] \right|.$$

The advantage of the **KAS-SKI** adversary \mathcal{A} against Ω on a graph $G \in \Gamma^{(\Omega)}$ is defined as follows:

$$\text{Adv}_{\Omega, \mathcal{A}, G}^{\text{KAS-SKI}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{KAS-SKI}_\Omega^{\mathcal{A}}(1^\lambda, G, b = 1) = 1] - \Pr[\text{KAS-SKI}_\Omega^{\mathcal{A}}(1^\lambda, G, b = 0) = 1] \right|.$$

The Ω is **KAS-KI** (or **KAS-SKI**) secure, if, for all PPT static adversaries \mathcal{A} , the value of $\text{Adv}_{\Omega, \mathcal{A}, G}^{\text{KAS-KI}}(1^\lambda)$ (or $\text{Adv}_{\Omega, \mathcal{A}, G}^{\text{KAS-SKI}}(1^\lambda)$) is *negligible*.

Game KAS-KR$_{\Omega}^{\mathcal{A}}(1^\lambda, G)$ $(params^{(\Omega)}, \Gamma^{(\Omega)}, \mathcal{K}^{(\Omega)}) := \Omega.\text{Setup}(1^\lambda);$ $u_{i_1} := \mathcal{A}_1(1^\lambda, G);$ $(\mathbf{S}, \mathbf{K}, P) := \Omega.\mathcal{GEN}(params^{(\Omega)}, G);$ $SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} u_{i_2} < u_{i_1}\};$ $K' := \mathcal{A}_2(1^\lambda, G, P, SV^{(u_{i_1})});$ return $(\mathbf{K}^{(u_{i_1})} = K');$
--

Figure 3.28: Security game **KAS-KR** for the *KAS* Ω .

KAS-KR SECURITY. The **KAS-KR** security – capturing the notion of *key recovery with respect to static adversary* – is defined in Figure 3.28, in the form of a game. The adversarial advantage is the probability of the event when the adversary is able to deduce the encryption key, and it is desired to be negligible.

According to the **KAS-KR** game, given access to graph G , the adversary, first, returns a security class $u_{i_1} \in V$, that \mathcal{A} chooses to attack. Then, the following operations are performed: first it computes $(\mathbf{S}, \mathbf{K}, P) := \Omega.\mathcal{GEN}(params^{(\Omega)}, G)$; and then $SV^{(u_{i_1})}$ is computed as set of *private information* $\mathbf{S}^{(u_{i_2})}$ for classes $u_{i_2} \in V$ such that $u_{i_2} < u_{i_1}$. Given access to *public information* P and set $SV^{(u_{i_1})}$, the adversary returns a key K' . The game returns 1, if $\mathbf{K}^{(u_{i_1})} = K'$.

The advantage of the **KAS-KR** adversary \mathcal{A} against Ω on a graph $G \in \Gamma^{(\Omega)}$ is defined as follows:

$$\text{Adv}_{\Omega, \mathcal{A}, G}^{\text{KAS-KR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{KAS-KR}_\Omega^{\mathcal{A}}(1^\lambda, G) = 1].$$

The Ω is **KAS-KR** secure, if, for all PPT static adversaries \mathcal{A} , $Adv_{\Omega, \mathcal{A}, G}^{\text{KAS-KR}}(1^\lambda)$ is *negligible*.

Remark. Note that a *KAS-chain* scheme, $\omega = (\omega.\mathcal{GEN}, \omega.\mathcal{DER})$ over $\omega.\text{Setup}$, is a special type of *KAS* where the access graph is a *totally ordered set*.

3.6.2 Existing *KAS* schemes

KAS has been studied for over three decades. In 1983, the first *KAS* scheme was proposed by Akl and Taylor, in which each user stores one secret key, and derives the other keys using *public information* [AT83]. MacKinnon *et al.* have attempted to optimize the solution proposed by Akl and Taylor [MTMA85]. Since then, a large number of *KAS* constructions have been proposed in the literature [ABFF09, AFB05, CCM15, CSM16a, CSM⁺16b, CC02, CDM10, CMW06, DSFM10, FP11, FPP13, HL90, SFM07a, SC02, WC01, YCN03]. Crampton *et al.* have extensively studied the existing *KAS* constructions, and classified them into five generic schemes; we discuss them in detail in Section 3.6.2.1 [CMW06]. They have also highlighted the advantages and disadvantages of each generic scheme.

Crampton, Daud and Martin have discussed procedure for designing *KAS* constructions by using *KAS-chains* and an innovative *chain partition* construction; we discuss this construction in Section 3.6.2.2 [CDM10]. This scheme was also used to construct *KAS* with useful performance-security trade-offs [FPP13]. A special type of *KAS* with expiry date and/or time for key, called *time-bound key assignment schemes*, has also been studied, and various schemes of this type have been proposed [ABF07, ASFM06, ASFM12, ASFM13, BSJ08, Chi04, HC04, PWCW15a, PWCW15b, SFM07b, SFM08, Tze02, Tze06, WL06, Yeh05, Yi05, YY03].

3.6.2.1 Classification of KAS constructions

Crampton, Martin and Wild have classified *KAS* constructions into five *generic* schemes [CMW06]. These schemes differ in:

1. The way encryption key $\mathbf{K}^{(u_{i_1})}$ (for file $\mathbf{M}^{(u_{i_1})}$) corresponding to node $u_{i_1} \in V$ is selected;
2. The method for generation and distribution of *secret information* $\mathbf{S} = (\mathbf{S}^{(u_{i_1})})_{u_{i_1} \in V}$ and *public information* P ; and

3. The working of *key derivation* algorithm where node u_{i_1} recomputes the key corresponding to node $u_{i_2} \leq u_{i_1}$.

These five *generic* schemes are as follows:

- **Scheme 1: TKAS.** A *trivial key assignment scheme (TKAS)* has the following properties:
 - For all $u_{i_1} \in V$, $\mathbf{K}^{(u_{i_1})}$'s are chosen independently;
 - $\mathbf{S}^{(u_{i_1})} := (\mathbf{K}^{(u_{i_2})})_{u_{i_2} \leq u_{i_1}}$;
 - $pub := \emptyset$; and
 - $\mathbf{K}^{(u_{i_2})} \in \mathbf{S}^{(u_{i_1})} \forall u_{i_2} \leq u_{i_1}$, so deriving the key $\mathbf{K}^{(u_{i_2})}$ is trivial.
- **Scheme 2: TKEKAS.** A *trivial key-encrypting-key assignment scheme (TKEKAS)* has the following properties:
 - For all $u_{i_1} \in V$, $\mathbf{K}^{(u_{i_1})}$'s and $\mathbf{key_val}^{(u_{i_1})}$'s are chosen independently, where $\mathbf{key_val}^{(u_{i_1})}$ is a key used to encrypt $\mathbf{K}^{(u_{i_1})}$;
 - $\mathbf{S}^{(u_{i_1})} := (\mathbf{key_val}^{(u_{i_2})})_{u_{i_2} \leq u_{i_1}}$;
 - $pub := (\mathcal{E}_{\mathbf{key_val}^{(u_{i_1})}}(\mathbf{K}^{(u_{i_1})}))_{u_{i_1} \in V}$; and
 - $\mathbf{K}^{(u_{i_2})}$ is obtained by decrypting $\mathcal{E}_{\mathbf{key_val}^{(u_{i_2})}}(\mathbf{K}^{(u_{i_2})}) \in pub$ using the key $\mathbf{key_val}^{(u_{i_2})} \in \mathbf{S}^{(u_{i_1})}$.
- **Scheme 3: DKEKAS.** A *direct key-encrypting-key assignment scheme (DKEKAS)* has the following properties:
 - For all $u_{i_1} \in V$, $\mathbf{K}^{(u_{i_1})}$'s are chosen independently;
 - $\mathbf{S}^{(u_{i_1})} := \mathbf{K}^{(u_{i_1})}$;
 - $pub := (\mathcal{E}_{\mathbf{K}^{(u_{i_1})}}(\mathbf{K}^{(u_{i_2})}))_{u_{i_2} < u_{i_1}, u_{i_1} \in V}$; and
 - $\mathbf{K}^{(u_{i_2})}$ is obtained by decrypting $\mathcal{E}_{\mathbf{K}^{(u_{i_1})}}(\mathbf{K}^{(u_{i_2})}) \in pub$ using $\mathbf{K}^{(u_{i_1})} \in \mathbf{S}^{(u_{i_1})}$.
- **Scheme 4: IKEKAS.** An *iterative key-encrypting-key assignment scheme (IKEKAS)* has the following properties:
 - For all $u_{i_1} \in V$, $\mathbf{K}^{(u_{i_1})}$'s are chosen independently;
 - $\mathbf{S}^{(u_{i_1})} := \mathbf{K}^{(u_{i_1})}$;
 - $pub := (\mathcal{E}_{\mathbf{K}^{(u_{i_1})}}(\mathbf{K}^{(u_{i_2})}))_{u_{i_2} < u_{i_1}, u_{i_1} \in V}$; and

- There exists a path $(u_{i_1}, u_{j_1}), (u_{j_1}, u_{j_2}) \dots (u_{j_m}, u_{i_2})$ and we sequentially calculate $\mathbf{K}^{(u_{j_1})} := \mathcal{D}_{\mathbf{K}^{(u_{i_1})}}(\mathcal{E}_{\mathbf{K}^{(u_{i_1})}}(\mathbf{K}^{(u_{j_1})})), \mathbf{K}^{(u_{j_2})} := \mathcal{D}_{\mathbf{K}^{(u_{j_1})}}(\mathcal{E}_{\mathbf{K}^{(u_{j_1})}}(\mathbf{K}^{(u_{j_2})})), \dots, \mathbf{K}^{(u_{i_2})} := \mathcal{D}_{\mathbf{K}^{(u_{j_m})}}(\mathcal{E}_{\mathbf{K}^{(u_{j_m})}}(\mathbf{K}^{(u_{i_2})}))$, to obtain $\mathbf{K}^{(u_{i_2})}$.
- **Scheme 5: NBKAS.** A *node-based key assignment scheme (NBKAS)* has the following properties:
 - $\mathbf{K}^{(u_{i_1})} := f(\mathbf{e}^{(u_{i_1})})$ are keys such that $g(f(\mathbf{e}^{(u_{i_1})}), \mathbf{e}^{(u_{i_1})}, \mathbf{e}^{(u_{i_2})}) = \mathbf{K}^{(u_{i_2})}$;
 - $\mathbf{S}^{(u_{i_1})} := \mathbf{K}^{(u_{i_1})}$;
 - $pub := (\mathbf{e}^{(u_{i_1})})_{u_{i_1} \in V}$; and
 - $\mathbf{K}^{(u_{i_2})} := g(\mathbf{K}^{(u_{i_1})}, \mathbf{e}^{(u_{i_1})}, \mathbf{e}^{(u_{i_2})})$ can be calculated using $\mathbf{e}^{(u_{i_1})}, \mathbf{e}^{(u_{i_2})} \in pub$ and $\mathbf{K}^{(u_{i_1})} \in \mathbf{S}^{(u_{i_1})}$.

3.6.2.2 Building KAS

A *key assignment scheme* is usually built in following two ways:

1. **Constructing KAS from scratch:** We refer the reader to [AT83, ABFF09, AFB05, CHW92, CC02, CH05, Gud80, HL90, SFM07a, SC02, TC95, YL04, ZRM01] to know about the various existing *KAS* constructions in detail.
2. **Chain Partition Construction:** The *chain partition* construction provides a framework to design a *KAS* construction from *KAS-chain* construction(s). The *chain partition* construction $\vec{\Omega} = (\vec{\Omega}. \mathcal{GEN}, \vec{\Omega}. \mathcal{DER})$ uses *KAS-chain* $\omega = (\omega. \mathcal{GEN}, \omega. \mathcal{DER})$ as a subroutine. We refer the reader to [CDM10, FP11, FPP13] to know about the various existing *KAS* constructions build from *KAS-chain* in detail.

Building *KAS-chain* from scratch

Crampton *et al.* described two *KAS-chain*, one based on *iterated hashing* and the other based on *RSA* [CDM10]. Freire and Paterson also gave a *KAS-chain* based on Factoring problem in [FP11]. Freire *et al.* described two *KAS-chain* schemes, one based on *pseudorandom functions* and the other based on *forward-secure pseudorandom generators* [FPP13].

Chain Partition construction

This framework builds a *KAS* for an arbitrary access graph, from the *KAS-chains*. Crampton, Daud and Martin have discussed procedures for designing efficient *KAS* schemes, from *KAS-chains*, using an innovative *chain partition* construction in [CDM10]. The main idea behind the *chain partition* construction is the following: partition the access graph into *disjoint chains*, and design *KAS-chains* corresponding to these *chains*; finally, securely join these *KAS-chains* to form the *KAS* for the full access graph. The detailed description of *chain partition* construction is given below, with a few suitable changes in the notation to suit the present context.

Let $\lambda \in \mathbb{N}$ be the security parameter, (V, \leq) be a poset represented by the access graph $G = (V, E)$, and $\omega = (\omega.\mathcal{GEN}, \omega.\mathcal{DER})$ be the *KAS-chain* for a totally ordered set of length at most l_{max} . Suppose the set of *chains* $\{C_1, C_2, \dots, C_w\}$ is a partition of G . The *chain partition* construction $\vec{\Omega} = (\vec{\Omega}.\mathcal{GEN}, \vec{\Omega}.\mathcal{DER})$ is a pair of algorithms over the *setup* algorithm $\vec{\Omega}.\text{Setup}$. The pseudocode for $\vec{\Omega}$ is given in Figures 3.29 and 3.30. Below we give the textual description. The subroutines used by the algorithm are described in Section 3.2.3. These subroutines are identical to the subroutines used in [FPP13], but we reproduce them for the sake of completeness.

- The *setup* algorithm $\vec{\Omega}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\vec{\Omega})}$, set of access graphs $\Gamma^{(\vec{\Omega})}$ and *key-space* $\mathcal{K}^{(\vec{\Omega})}$. The $\vec{\Omega}.\text{Setup}(1^\lambda)$ is designed in the following way: first it invokes $\omega.\text{Setup}(1^\lambda)$; then it computes $params^{(\vec{\Omega})}$ such that it includes $params^{(\omega)}$; next it computes set of *access graphs* $\Gamma^{(\vec{\Omega})}$ that contains all the graphs for chains in $\Gamma^{(\omega)}$; and finally it computes *key-space* $\mathcal{K}^{(\vec{\Omega})} = \mathcal{K}^{(\omega)}$.
- The *key generation* algorithm $\vec{\Omega}.\mathcal{GEN}(\cdot)$ is a randomized algorithm that takes as input parameter $params^{(\vec{\Omega})}$ and access graph $G \in \Gamma^{(\vec{\Omega})}$, and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pseudocode is given in Figure 3.29. The algorithm works as follows: first it computes *chain partition* C_1, C_2, \dots, C_w of graph G as $(w, C[\]) := \text{partition}(G)$; and then for each chain C_j , where $j \in [w]$, it computes $(\mathbf{S}_j, \mathbf{K}_j, P_j) := \omega.\mathcal{GEN}(params^{(\omega)}, C_j)$. Then, for every node $u_i \in V$, the following operations are performed: first it computes maximal nodes of $\downarrow u_i \cap C_g$ for all chains as $(\hat{u}_{i_1}, \hat{u}_{i_2}, \dots, \hat{u}_{i_w}) := \text{max_isect_chs}(u_i, G)$; and finally it computes *private information* of u_i as $\mathbf{S}^{(u_i)} := \mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})}$.

```

 $\vec{\Omega} \cdot \mathcal{GEN}(params^{(\vec{\Omega})}, G)$ 

#Initialization.
 $(w, C[ ]) := \text{partition}(G);$ 

#Generating  $(\mathbf{S}, \mathbf{K}, P)$  for each chain using KAS-chain.
for  $(j := 1, 2, \dots, w)$ 
     $(\mathbf{S}_j, \mathbf{K}_j, P_j) := \omega \cdot \mathcal{GEN}(params^{(\omega)}, C_j);$ 

#Computing private information for each node.
for  $u_i \in V$ 
     $(\hat{u}_{i_1}, \hat{u}_{i_2}, \dots, \hat{u}_{i_w}) := \text{max\_isect\_chs}(u_i, G);$ 
     $\mathbf{S}^{(u_i)} := \mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})};$ 

#Computing final output.
 $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}; \quad \mathbf{K} := (\mathbf{K}^{(u_i)})_{u_i \in V};$ 
 $P := P_1 \circ P_2 \circ \dots \circ P_w;$ 
return  $(\mathbf{S}, \mathbf{K}, P);$ 

```

The functions `partition` and `max_isect_chs` have been described in Section 3.2.3.

Figure 3.29: Algorithmic description of the *key generation* function $\vec{\Omega} \cdot \mathcal{GEN}$.

Now, the vector of *private information* $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}$, vector of *keys* $\mathbf{K} := (\mathbf{K}^{(u_i)})_{u_i \in V}$, and *public information* $P := P_1 \circ P_2 \circ \dots \circ P_w$.

```

 $\vec{\Omega} \cdot \mathcal{DER}(params^{(\vec{\Omega})}, G, u_j^i, u_h^g, \mathbf{S}^{(u_j^i)}, P)$ 

#Initialization.
 $\hat{u}_{i_g} := \text{max\_isect}(u_j^i, C_g, G);$ 
 $\mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_g^{(\hat{u}_{i_g})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})} := \mathbf{S}^{(u_j^i)};$ 
 $P_1 \circ P_2 \circ \dots \circ P_w := P;$ 

#Computing key.
 $\mathbf{K}^{(u_h^g)} := \omega \cdot \mathcal{DER}(params^{(\omega)}, C_g, \hat{u}_{i_g}, u_h^g, \mathbf{S}_g^{(\hat{u}_{i_g})}, P_g);$ 
return  $\mathbf{K}^{(u_h^g)};$ 

```

The function `max_isect` is described in Section 3.2.3.

Figure 3.30: Algorithmic description of the *key derivation* function $\vec{\Omega} \cdot \mathcal{DER}$.

- The *key derivation* algorithm $\vec{\Omega}.\mathcal{DER}(\cdot)$ is a deterministic algorithm that takes as input parameter $params^{(\vec{\Omega})}$, graph G , two nodes u_j^i and u_h^g such that $u_h^g \leq u_j^i$, node u_j^i 's *private information* $\mathbf{S}^{(u_j^i)}$ and *public information* P , and outputs key $\mathbf{K}^{(u_h^g)}$ corresponding to u_h^g . See Figure 3.30 for the pseudocode. The algorithm works as follows: first it computes maximal node of $\downarrow u_j^i \cap C_g$ as $\hat{u}_{i_g} := \max_isect(u_j^i, C_g, G)$; then it computes *private information* for \hat{u}_{i_g} as $\mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_g^{(\hat{u}_{i_g})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})} := \mathbf{S}^{(u_j^i)}$; next it parses *public information* into $P_1 \circ P_2 \circ \dots \circ P_w$; and finally it computes key as $\mathbf{K}^{(u_h^g)} := \omega.\mathcal{DER}(params^{(\omega)}, C_g, \hat{u}_{i_g}, u_h^g, \mathbf{S}_g^{(\hat{u}_{i_g})}, P_g)$.

Now, the key $\mathbf{K}^{(u_h^g)}$, computed above, is returned.

CHAPTER 4

Message-Locked Encryption with File Update

4.1 Introduction

We have given an elaborate discussion of the cryptographic primitive *MLE* in Section 3.4, and of its special variant, *UMLE*, in Section 3.4.5. In this chapter, we introduce a new cryptographic primitive *file-updatable message-locked encryption (FMLE)*, which can be regarded as a special variant of *MLE*, as well as a generalization of *UMLE*.

MOTIVATION FOR STUDYING FMLE. From a high level, both *UMLE* and *FMLE* have the identical functions; the main difference, however, is that the former is necessarily based on *block-level message-locked encryption (BL-MLE)*, but the latter may or may not be (see Section 3.4.2 for the definition of *BL-MLE*). Therefore, the definition of *FMLE* can be viewed as a generalisation of *UMLE*, where we remove the constraint of it being *always* built using a *BL-MLE*. Below we summarize our motivation:

Does there exist an *FMLE* scheme which is *not* based on
BL-MLE? If such a construction exists, is it more efficient
than *UMLE*?

Studying *FMLE* is a futile exercise, if both the answers are in the negative. A moment's reflection suggests that the answer to the first question is actually 'Yes'. A trivial *FMLE* construction – not based on *BL-MLE* – always exists, where *file update* function is designed in the following way:

apply decryption to the ciphertext to recover original plaintext; edit/modify the original message; and finally encrypt the updated message. Note that this trivial *file update* function does not need any *BL-MLE*, therefore, by definition, it is an *FMLE*, but certainly not a *UMLE* scheme. The main drawback of this *FMLE* scheme is that this is several orders of magnitude slower than a *UMLE* scheme. Therefore, the main challenge is:

Does there exist an *FMLE* scheme more efficient than *UMLE*?

Searching for such a construction is the main motivation of this chapter.

4.2 Contribution of this Chapter

Our first contribution is formalizing the new cryptographic notion *file-updatable message-locked encryption (FMLE)*. We also propose two efficient *FMLE* constructions $\hat{\Phi}$ and $\tilde{\Phi}$: their *update* functions are at least 50% faster (on average). Also, our constructions are more space efficient than the so-far best *MLE* variants; in particular, the *ciphertext expansion* and *tag storage* in $\hat{\Phi}$ and $\tilde{\Phi}$ are constant, while they are logarithmic and linear (or worse) for others. In order to obtain this improvement in the performance, our constructions critically exploit a very unique feature – what we call *reverse decryption* – of the authenticated encryption *APE* and the hash function *FP*. We also present proofs of security of our constructions. Extensive comparison of our constructions with the others – in terms of time complexity, storage requirements and security properties – has also been provided in Tables 4.1 and 4.2. Being randomized, our constructions are secure against *dictionary attacks*, however, they lack the *STC* security (see Section 4.3), like any other randomized *MLEs*.

4.3 FMLE: A New Cryptographic Primitive

We now elaborately discuss the syntax, correctness and security definition of the new notion *FMLE*.

4.3.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *FMLE* scheme $\Phi = (\Phi, \mathcal{E}, \Phi, \mathcal{D}, \Phi, \mathcal{U}, \Phi, \mathcal{P}, \Phi, \mathcal{V})$ is a 5-tuple of algorithms over the *setup* algorithm Φ, Setup , satisfying the following conditions.

1. The PPT *setup* algorithm $\Phi.\text{Setup}(1^\lambda)$ outputs parameter $params^{(\Phi)}$, key-space $\mathcal{K}^{(\Phi)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\Phi)} \subseteq \{0, 1\}^*$, ciphertext-space $\mathcal{C}^{(\Phi)} \subseteq \{0, 1\}^*$ and tag-space $\mathcal{T}^{(\Phi)} \subseteq \{0, 1\}^*$.
2. The PPT *encryption* algorithm $\Phi.\mathcal{E}(\cdot)$ takes as input parameter $params^{(\Phi)}$ and message $M \in \mathcal{M}^{(\Phi)}$, and returns a 3-tuple $(K, C, T) := \Phi.\mathcal{E}(params^{(\Phi)}, M)$, where key $K \in \mathcal{K}^{(\Phi)}$, ciphertext $C \in \mathcal{C}^{(\Phi)}$ and tag $T \in \mathcal{T}^{(\Phi)}$.
3. The *decryption* algorithm $\Phi.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $params^{(\Phi)}$, key $K \in \mathcal{K}^{(\Phi)}$, ciphertext $C \in \mathcal{C}^{(\Phi)}$ and tag $T \in \mathcal{T}^{(\Phi)}$, and returns message $M := \Phi.\mathcal{D}(params^{(\Phi)}, K, C, T)$, where $M \in \mathcal{M}^{(\Phi)} \cup \{\perp\}$. The *decryption* algorithm $\Phi.\mathcal{D}$ returns \perp , if key K , ciphertext C and tag T are not generated from a valid message.
4. The PPT *update* algorithm $\Phi.\mathcal{U}(\cdot)$ takes as input parameter $params^{(\Phi)}$, index of starting and ending bits $i_{st}, i_{end} \in \mathbb{N}$, new message bits $M_{\text{new}} \in \{0, 1\}^*$, decryption key $K \in \mathcal{K}^{(\Phi)}$, old ciphertext $C \in \mathcal{C}^{(\Phi)}$, old tag $T \in \mathcal{T}^{(\Phi)}$ and bit $app \in \{0, 1\}$, indicating the change in the length of the new message, and finally returns a 3-tuple $(K', C', T') := \Phi.\mathcal{U}(params^{(\Phi)}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$, where $K' \in \mathcal{K}^{(\Phi)}$, $C' \in \mathcal{C}^{(\Phi)}$ and $T' \in \mathcal{T}^{(\Phi)}$ are the updated key, ciphertext and tag, respectively.
5. The PPT *proof-of-ownership (PoW) algorithm for prover* $\Phi.\mathcal{P}(\cdot)$ takes as input parameter $params^{(\Phi)}$, challenge $Q \in \{0, 1\}^*$, message $M \in \mathcal{M}^{(\Phi)}$, decryption key $K \in \mathcal{K}^{(\Phi)}$, ciphertext $C \in \mathcal{C}^{(\Phi)}$ and tag $T \in \mathcal{T}^{(\Phi)}$, and returns proof $P := \Phi.\mathcal{P}(params^{(\Phi)}, Q, M, K, C, T)$, where $P \in \{0, 1\}^*$.
6. The PPT *proof-of-ownership (PoW) algorithm for verifier* $\Phi.\mathcal{V}(\cdot)$ takes as input parameter $params^{(\Phi)}$, challenge $Q \in \{0, 1\}^*$, ciphertext $C \in \mathcal{C}^{(\Phi)}$, tag $T \in \mathcal{T}^{(\Phi)}$ and proof $P \in \{0, 1\}^*$, and returns value $val := \Phi.\mathcal{V}(params^{(\Phi)}, Q, C, T, P)$, where $val \in \{\text{TRUE}, \text{FALSE}\}$.

Note: When tag is incorporated in the ciphertext itself, the ciphertext will be expanded. We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\Phi)}$.

4.3.2 Correctness

KEY CORRECTNESS. This condition requires that the key generated from two identical messages will always be identical. Mathematically, the *key correctness* of an *FMLE* can be defined as follows.

Suppose:

- $M_0 = M_1$,
- $(K_0, C_0, T_0) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M_0)$, and
- $(K_1, C_1, T_1) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M_1)$.

Then, the *key correctness* of Φ requires that $K_0 = K_1$, for all $(\lambda, M_0, M_1) \in \mathbb{N} \times \mathcal{M}^{(\Phi)} \times \mathcal{M}^{(\Phi)}$.

DECRYPTION CORRECTNESS. This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the *decryption correctness* of an *FMLE* can be defined as follows.

Suppose $(K, C, T) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M)$. The *decryption correctness* of Φ requires that $\Phi \cdot \mathcal{D}(\text{params}^{(\Phi)}, K, C, T) = M$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\Phi)}$.

TAG CORRECTNESS. This condition requires that the tag generated from two identical messages will always be identical. Mathematically, the *tag correctness* of an *FMLE* can be defined as follows.

Suppose:

- $M_0 = M_1$,
- $(K_0, C_0, T_0) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M_0)$, and
- $(K_1, C_1, T_1) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M_1)$.

Then, the *tag correctness* of Φ requires that $T_0 = T_1$, for all $(\lambda, M_0, M_1) \in \mathbb{N} \times \mathcal{M}^{(\Phi)} \times \mathcal{M}^{(\Phi)}$.

UPDATE CORRECTNESS. This condition requires that the updated key, ciphertext and tag should always produce updated message on decryption. Mathematically, the *update correctness* of an *FMLE* can be defined as follows.

Suppose:

- $m = |M|$,
- $(K, C, T) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, M)$, and
- $(K', C', T') := \Phi \cdot \mathcal{U}(\text{params}^{(\Phi)}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$.

Then, the *update correctness* of Φ requires that:

- for $app = 1$, $\Phi \cdot \mathcal{D}(\text{params}^{(\Phi)}, K', C', T') = M[1] \| M[2] \| \dots \| M[i_{st} - 1] \| M_{\text{new}}$, and

- for $app = 0$, $\Phi.\mathcal{D}(params^{(\Phi)}, K', C', T') = M[1]\|M[2]\|\cdots\|M[i_{st} - 1]\|M_{\text{new}}\|M[i_{end} + 1]\|M[i_{end} + 2]\|\cdots\|M[m]$,
for all $(\lambda, M, M_{\text{new}}) \in \mathbb{N} \times \mathcal{M}^{(\Phi)} \times \{0, 1\}^{i_{end}-i_{st}+1}$, $1 \leq i_{st} \leq m$ and $i_{st} \leq i_{end}$.

PoW CORRECTNESS. This condition requires that a proof – computed correctly from the message – should be able to verify successfully. Mathematically, the *PoW correctness* of an *FMLE* can be defined as follows.

Suppose:

- $(K, C, T) := \Phi.\mathcal{E}(params^{(\Phi)}, M)$,
- Q is any challenge, and
- $P := \Phi.\mathcal{P}(params^{(\Phi)}, Q, M, K, C, T)$.

Then, the *PoW correctness* of Φ requires that $\Pr[\Phi.\mathcal{V}(params^{(\Phi)}, Q, C, T, P) = \text{TRUE}] = 1$, for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\Phi)}$.

4.3.3 Security definitions

Security definitions of *FMLE* are naturally adapted from those of *UMLE*, with some modifications. For the sake of completeness, we describe them below in full detail. As usual, all the games are written using the challenger-adversary framework. In Figure 4.1, 4.2, 4.3, 4.4 and 4.5, we define the security games **FMLE-PRV**, **FMLE-PRV\$**, **FMLE-STC** and **FMLE-TC**, **FMLE-UNC** and **FMLE-CXH**, respectively, for the *FMLE* scheme $\Phi = (\Phi.\mathcal{E}, \Phi.\mathcal{D}, \Phi.\mathcal{U}, \Phi.\mathcal{P}, \Phi.\mathcal{V})$.

Game FMLE-PRV $_{\Phi,\mathcal{S}}^{\mathcal{A}}(1^\lambda, b)$ $(params^{(\Phi)}, \mathcal{K}^{(\Phi)}, \mathcal{M}^{(\Phi)}, \mathcal{C}^{(\Phi)}, \mathcal{T}^{(\Phi)}) := \Phi.\text{Setup}(1^\lambda);$ $(\mathbf{M}_0, \mathbf{M}_1, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$ for $(i := 1, 2, \dots, m(1^\lambda))$ $\quad (\mathbf{K}^{(i)}, \mathbf{C}^{(i)}, \mathbf{T}^{(i)}) := \Phi.\mathcal{E}(params^{(\Phi)}, \mathbf{M}_b^{(i)});$ $b' := \mathcal{A}(1^\lambda, \mathbf{C}, \mathbf{T}, Z);$ return b' ;
--

Figure 4.1: Security game **FMLE-PRV** for the *FMLE* scheme Φ .

FMLE-PRV SECURITY. Since no *FMLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpredictable* message-source $\mathcal{S}(\cdot)$ (for details see Section 3.4.1). The **FMLE-PRV** security – capturing the notion of *indistinguishability privacy against chosen distribution attack* – is defined in Figure 4.1. The adversarial advantage

is the probability of the event when the adversary is able to distinguish between the encryptions of two vectors of messages, and it is desired to be negligible.

According to the **FMLE-PRV** game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of two vectors of strings $\mathbf{M}_0 = \mathbf{M}_0^{(1)} \circ \mathbf{M}_0^{(2)} \circ \cdots \circ \mathbf{M}_0^{(m(1^\lambda))}$ and $\mathbf{M}_1 = \mathbf{M}_1^{(1)} \circ \mathbf{M}_1^{(2)} \circ \cdots \circ \mathbf{M}_1^{(m(1^\lambda))}$, and auxiliary information Z . Now, for each string $\mathbf{M}_b^{(i)}$, depending upon the value of $b \in \{0, 1\}$, for all $i \in [m(1^\lambda)]$, key, ciphertext and tag are computed as $(\mathbf{K}^{(i)}, \mathbf{C}^{(i)}, \mathbf{T}^{(i)}) := \Phi.\mathcal{E}(\text{params}^{(\Phi)}, \mathbf{M}_b^{(i)})$. Given access to \mathbf{C} , \mathbf{T} and Z , the adversary returns a bit b' .

The advantage of the **FMLE-PRV** adversary \mathcal{A} against Φ for the message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$\begin{aligned} \text{Adv}_{\Phi, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{FMLE-PRV}_{\Phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=1) = 1] \right. \\ &\quad \left. - \Pr[\text{FMLE-PRV}_{\Phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=0) = 1] \right|. \end{aligned}$$

The Φ is **FMLE-PRV** secure over a set of valid PPT message-sources for Φ , $\bar{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \bar{\mathcal{S}}$, if $\text{Adv}_{\Phi, \mathcal{S}_i, \mathcal{A}}^{\text{FMLE-PRV}}(1^\lambda)$ is *negligible*. The Φ is **FMLE-PRV** secure, for all PPT adversaries \mathcal{A} , if $\text{Adv}_{\Phi, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for Φ .

Game $\text{FMLE-PRV\\$}_{\Phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b)$ $(\text{params}^{(\Phi)}, \mathcal{K}^{(\Phi)}, \mathcal{M}^{(\Phi)}, \mathcal{C}^{(\Phi)}, \mathcal{T}^{(\Phi)}) := \Phi.\text{Setup}(1^\lambda);$ $(\mathbf{M}, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$ for $(i := 1, 2, \dots, m(1^\lambda))$ $\quad (\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \Phi.\mathcal{E}(\text{params}^{(\Phi)}, \mathbf{M}^{(i)});$ $\quad \mathbf{C}_0^{(i)} \xleftarrow{\$} \{0, 1\}^{ \mathbf{C}_1^{(i)} }; \quad \mathbf{T}_0^{(i)} \xleftarrow{\$} \{0, 1\}^{ \mathbf{T}_1^{(i)} };$ $\quad b' := \mathcal{A}(1^\lambda, \mathbf{C}_b, \mathbf{T}_b, Z);$ return b' ;
--

Figure 4.2: Security game **FMLE-PRV\$** for the *FMLE* scheme Φ .

FMLE-PRV\$ SECURITY. This security notion is identical to **FMLE-PRV**, except that in **FMLE-PRV**, the adversary's challenge is to distinguish between the encryptions of two vectors of strings, while in **FMLE-PRV\$**, the adversary's challenge is to distinguish the encryption of a vector of strings

from a vector of random strings. Since no *FMLE* scheme can provide security for predictable messages, we are modelling the security based on the *unpredictable* message-source $\mathcal{S}(\cdot)$ (for details see Section 3.4.1). The **FMLE-PRV\$** security – capturing the notion of *strong indistinguishability privacy against chosen distribution attack* – is defined in Figure 4.2. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a vector of messages from a vector of random strings, and it is desired to be negligible.

According to the **FMLE-PRV\$** game, at the first step, the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked for the computation of a vector of strings $\mathbf{M} = \mathbf{M}^{(1)} \circ \mathbf{M}^{(2)} \circ \dots \circ \mathbf{M}^{(m(1^\lambda))}$ and auxiliary information Z . Now, for each string $\mathbf{M}^{(i)}$, for values of $i \in [m(1^\lambda)]$, following operations are performed: key, ciphertext and tag are computed as $(\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \Phi \cdot \mathcal{E}(\text{params}^{(\Phi)}, \mathbf{M}^{(i)})$; a random string of length $|\mathbf{C}_1^{(i)}|$ is chosen as $\mathbf{C}_0^{(i)}$; and a random string of length $|\mathbf{T}_1^{(i)}|$ is chosen as $\mathbf{T}_0^{(i)}$. Given access to \mathbf{C}_b , \mathbf{T}_b and Z , the adversary returns a bit b' .

The advantage of the **FMLE-PRV\$** adversary \mathcal{A} against Φ for the message source $\mathcal{S}(\cdot)$ is defined as follows:

$$\begin{aligned} \text{Adv}_{\Phi, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{FMLE-PRV\$}_{\Phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{FMLE-PRV\$}_{\Phi, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|. \end{aligned}$$

The Φ is **FMLE-PRV\$** secure over a set of valid PPT message-sources for Φ , $\overline{\mathcal{S}} = \{\mathcal{S}_1, \mathcal{S}_2, \dots\}$, for all PPT adversaries \mathcal{A} and for all $\mathcal{S}_i \in \overline{\mathcal{S}}$, if $\text{Adv}_{\Phi, \mathcal{S}_i, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda)$ is *negligible*. The Φ is **FMLE-PRV\$** secure, for all PPT adversaries \mathcal{A} , if $\text{Adv}_{\Phi, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda)$ is *negligible*, for all valid PPT message-source \mathcal{S} for Φ .

Note: **FMLE-PRV\$** is a stronger security notion than **FMLE-PRV** [BKR13b].

FMLE-STC AND FMLE-TC SECURITY. The **FMLE-STC** and **FMLE-TC** security notions – capturing the notions of *strong tag consistency* and *tag consistency* – are defined in Figure 4.3. The adversarial advantage is the probability of the event when the adversary is able to compute a message, a ciphertext and a tag, such that, (1) on encryption of the supplied message, the tag produced is identical to the supplied one, and (2) on decryption of the supplied ciphertext with supplied tag, the message computed is unidentical to the supplied one. This adversarial advantage is desired to be negligible.

Game $\mathbf{FMLE-STC}_{\Phi}^{\mathcal{A}}(1^{\lambda})$	$\mathbf{FMLE-TC}_{\Phi}^{\mathcal{A}}(1^{\lambda})$
$(params^{(\Phi)}, \mathcal{K}^{(\Phi)}, \mathcal{M}^{(\Phi)}, \mathcal{C}^{(\Phi)}, \mathcal{T}^{(\Phi)}) := \Phi.\text{Setup}(1^{\lambda});$	
$(M_0, C_1, T_1) := \mathcal{A}(1^{\lambda});$	
If $(M_0 = \perp) \vee (C_1 = \perp)$	
return 0;	
$(K_0, C_0, T_0) := \Phi.\mathcal{E}(params^{(\Phi)}, M_0);$	
$M_1 := \Phi.\mathcal{D}(params^{(\Phi)}, K_0, C_1, T_1);$	
If $(T_0 = T_1) \wedge (M_0 \neq M_1)$	$\wedge(M_1 \neq \perp)$
return 1;	
Else return 0;	

Figure 4.3: Security games $\mathbf{FMLE-STC}$ and $\mathbf{FMLE-TC}$ for the $FMLE$ scheme Φ .

According to the $\mathbf{FMLE-STC}$ game, at the first step, the adversary returns a valid message M_0 , a valid ciphertext C_1 and a tag T_1 . Then the following operations are performed: key, ciphertext and tag for message M_0 are computed as $(K_0, C_0, T_0) := \Phi.\mathcal{E}(params^{(\Phi)}, M_0)$; and ciphertext C_1 (with tag T_1) is decrypted using key K_0 as $M_1 := \Phi.\mathcal{D}(params^{(\Phi)}, K_0, C_1, T_1)$. The game returns 1, if $T_0 = T_1$ under $M_0 \neq M_1$. For $\mathbf{FMLE-TC}$ game, an additional comparision for validity of M_1 (i.e. $M_1 \neq \perp$) is performed, before returning the bit 1.

The advantage of the $\mathbf{FMLE-TC}$ adversary \mathcal{A} against Φ is defined as follows:

$$Adv_{\Phi, \mathcal{A}}^{\mathbf{FMLE-TC}}(1^{\lambda}) \stackrel{def}{=} \Pr[\mathbf{FMLE-TC}_{\Phi}^{\mathcal{A}}(1^{\lambda}) = 1].$$

The advantage of the $\mathbf{FMLE-STC}$ adversary \mathcal{A} against Φ is defined as follows:

$$Adv_{\Phi, \mathcal{A}}^{\mathbf{FMLE-STC}}(1^{\lambda}) \stackrel{def}{=} \Pr[\mathbf{FMLE-STC}_{\Phi}^{\mathcal{A}}(1^{\lambda}) = 1].$$

The Φ is $\mathbf{FMLE-TC}$ (or $\mathbf{FMLE-STC}$) secure, if, for all PPT adversaries \mathcal{A} , the value of $Adv_{\Phi, \mathcal{A}}^{\mathbf{FMLE-TC}}(1^{\lambda})$ (or $Adv_{\Phi, \mathcal{A}}^{\mathbf{FMLE-STC}}(1^{\lambda})$) is *negligible*.

FMLE-UNC SECURITY. This captures the notion of *uncheatable chosen distribution* attack, as defined in Figure 4.4. Precisely, $\mathbf{FMLE-UNC}$ aims to provide resistance against the adversaries who are trying to prove that they possess the entire file, when, in reality, they have only a partial information of the file. This is to prevent unauthorised ownership of the file. The adversarial advantage is the probability of the event when the adversary is able to compute a false proof that verifies successfully on *PoW algorithm*

```

Game FMLE-UNC $_{\Phi}^{\mathcal{A}}(1^\lambda)$ 
_____
( $params^{(\Phi)}, \mathcal{K}^{(\Phi)}, \mathcal{M}^{(\Phi)}, \mathcal{C}^{(\Phi)}, \mathcal{T}^{(\Phi)}$ ) :=  $\Phi$ .Setup( $1^\lambda$ );
 $\mathcal{S}(\cdot) := \mathcal{A}_1(1^\lambda)$ ;
 $(M, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda)$ ;
 $(K, C, T) := \Phi$ .E( $params^{(\Phi)}$ ,  $M$ )
 $P^* := \mathcal{A}_2(1^\lambda, Q, Z)$ ;
 $P := \Phi$ .P( $params^{(\Phi)}$ ,  $Q, M, K, C, T$ );
If  $(\Phi$ .V( $params^{(\Phi)}$ ,  $Q, C, T, P^*$ ) = TRUE)  $\wedge (P^* \neq P)$ 
    return 1;
Else return 0;

```

Figure 4.4: Security game **FMLE-UNC** for the *FMLE* scheme Φ .

for verifier, and is unidentical to genuine proof. This adversarial advantage is desired to be negligible.

According to the **FMLE-UNC** game, at first, the adversary returns an *unpredictable* message-source $\mathcal{S}(\cdot)$. Then the following operations are performed: the *unpredictable* message-source $\mathcal{S}(1^\lambda)$ is invoked that outputs a string M and the auxiliary information Z ; and message M is encrypted as $(K, C, T) := \Phi$.**E**($params^{(\Phi)}$, M). Now, given challenge Q and auxiliary information Z , the adversary has to return a proof P^* . Then, the actual proof – corresponding to challenge Q , message M , key K , ciphertext C and tag T – is computed as $P := \Phi$.**P**($params^{(\Phi)}$, Q, M, K, C, T). The game returns 1, if Φ .**V**($params^{(\Phi)}$, Q, C, T, P^*) = TRUE under $P^* \neq P$, otherwise 0.

The advantage of the **FMLE-UNC** adversary \mathcal{A} against Φ for a message-source $\mathcal{S}(\cdot)$ is defined as follows:

$$Adv_{\Phi, \mathcal{A}}^{\text{FMLE-UNC}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{FMLE-UNC}_{\Phi}^{\mathcal{A}}(1^\lambda) = 1].$$

The Φ is **FMLE-UNC** secure, for all PPT adversaries \mathcal{A} , if $Adv_{\Phi, \mathcal{A}}^{\text{FMLE-UNC}}(1^\lambda)$ is *negligible*.

FMLE-CXH SECURITY. This captures the notion of *context hiding* attack, as defined in Figure 4.5. Precisely, **FMLE-CXH** security aims to provide security against distinguishing attack that attempt to tell apart an updated ciphertext from a ciphertext encrypted from scratch. This ensures that the level of privacy is not compromised during the update process. This adversarial advantage is desired to be negligible.

According to the **FMLE-CXH** game, at first, the adversary returns two messages M_0 and M_1 of identical length. Then the bit-positions i_1, i_2, \dots, i_m

```

Game  $\text{FMLE-CXH}_{\Phi}^{\mathcal{A}}(1^{\lambda}, \sigma, b)$ 
 $(params^{(\Phi)}, \mathcal{K}^{(\Phi)}, \mathcal{M}^{(\Phi)}, \mathcal{C}^{(\Phi)}, \mathcal{T}^{(\Phi)}) := \Phi.\text{Setup}(1^{\lambda});$ 
 $(M_0, M_1) := \mathcal{A}_1(1^{\lambda}, \sigma);$ 
Determine bit-positions  $i_1, i_2, \dots, i_m$  where  $M_0$  and  $M_1$  differ;
If ( $m > \sigma$ ), then return Error;
 $(K_0, C_0, T_0) := \Phi.\mathcal{E}(params^{(\Phi)}, M_0);$ 
 $(K_1, C_1, T_1) := \Phi.\mathcal{E}(params^{(\Phi)}, M_1);$ 
 $(K'_1, C'_1, T'_1) := \Phi.\mathcal{U}(params^{(\Phi)}, i_1, i_m, M_0[i_1, i_1 + 1, \dots, i_m],$ 
 $K_1, C_1, T_1, 0);$ 
If ( $b = 0$ ), then  $C^* := C_0$ ; Else  $C^* := C'_1$ ;
 $b' := \mathcal{A}_2(1^{\lambda}, C^*);$ 
return  $b'$ ;

```

Figure 4.5: Security game FMLE-CXH for the FMLE scheme Φ .

are determined, where messages M_0 and M_1 differ. The game aborts, if the messages M_0 and M_1 differ at more than σ bits. Then the following operations are performed: key, ciphertext and tag for message M_0 are computed as $(K_0, C_0, T_0) := \Phi.\mathcal{E}(params^{(\Phi)}, M_0)$; key, ciphertext and tag for message M_1 are computed as $(K_1, C_1, T_1) := \Phi.\mathcal{E}(params^{(\Phi)}, M_1)$; ciphertext C_1 and tag T_1 are updated with message bits $M_0[i_1, i_1 + 1, \dots, i_m]$ as $(K'_1, C'_1, T'_1) := \Phi.\mathcal{U}(params^{(\Phi)}, i_1, i_m, M_0[i_1, i_1 + 1, \dots, i_m], K_1, C_1, T_1, 0)$; and if the value of b is 0, C^* is assigned the value of C_0 , otherwise, C^* is assigned the value of C'_1 . Given ciphertext C^* , the adversary returns the bit b' .

The advantage of the FMLE-CXH adversary \mathcal{A} for σ -bit update in message, against the FMLE construction Φ is defined as follows:

$$\begin{aligned} Adv_{\Phi, \mathcal{A}}^{\text{FMLE-CXH}}(1^{\lambda}, \sigma) \stackrel{\text{def}}{=} & \left| \Pr[\text{FMLE-CXH}_{\Phi}^{\mathcal{A}}(1^{\lambda}, \sigma, b = 1) = 1] \right. \\ & \left. - \Pr[\text{FMLE-CXH}_{\Phi}^{\mathcal{A}}(1^{\lambda}, \sigma, b = 0) = 1] \right|. \end{aligned}$$

The Φ is FMLE-CXH secure, for σ -bit update in message, if, for all PPT adversaries \mathcal{A} , $Adv_{\Phi, \mathcal{A}}^{\text{CXH}}(1^{\lambda}, \sigma)$ is *negligible*.

4.4 Deduplication: An Application of FMLE

In Section 3.3, we have discussed elaborately the necessity of a *deduplication* protocol. We also knew that *MLE* could be used to design a *dedupli-*

cation protocol. In this section, we construct a *deduplication* protocol using *FMLE* (instead of *MLE*). The advantage is that the *deduplication* protocol now inherits the *update* and *PoW* functions of *FMLE*. The *FMLE-based deduplication* protocol is composed of three independent protocols: (1) *UPLOAD*, (2) *DOWNLOAD*, and (3) *UPDATE*. Below we give the full textual description.

Remark. A *deduplication* protocol is always preceded by an additional *user authentication*, which needs to be implemented separately. Some of the well-known *user authentication* schemes are password-based and biometric ones.

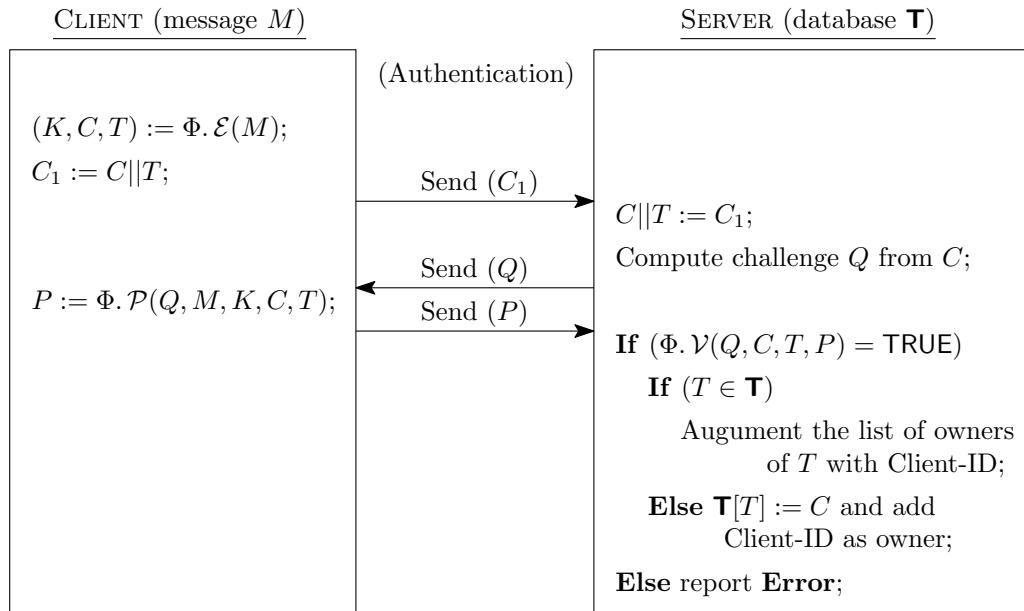


Figure 4.6: *UPLOAD* protocol for *FMLE-based deduplication*.

- **UPLOAD.** This protocol is responsible for the series of operations undertaken when the client uploads a file to the server; the pictorial description is given in Figure 4.6. After *user authentication*, the client computes key K , ciphertext C and tag T using $\Phi.\mathcal{E}(\cdot)$, and sends $C_1 := C||T$ to the server. The server first computes C and T from C_1 , then it generates a challenge Q for C , and finally sends Q to the client. In response, the client computes proof P corresponding to challenge Q using $\Phi.\mathcal{P}(\cdot)$, and sends it to the server. After successful verification of P by the server using $\Phi.\mathcal{V}(\cdot)$, if T and C are already stored in the database, then client-ID is appended in the *list of owners*, otherwise

the server stores C in the database at index T , and creates a file *list of owners* for C with client-ID as the owner.

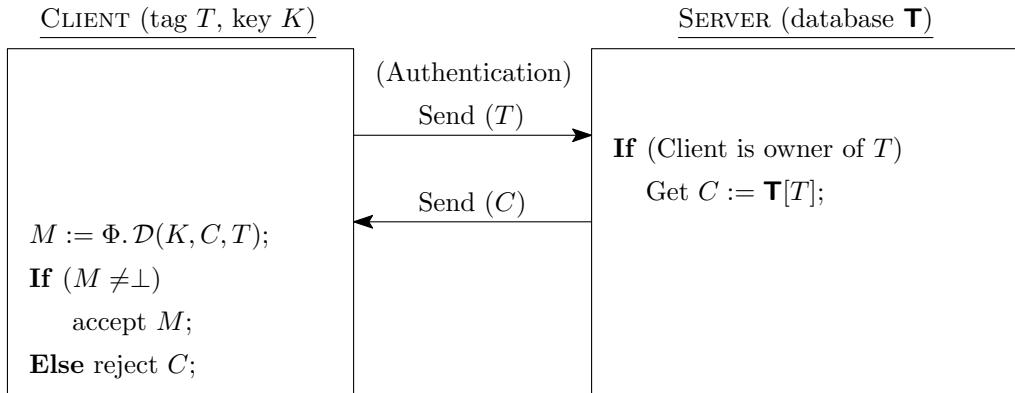
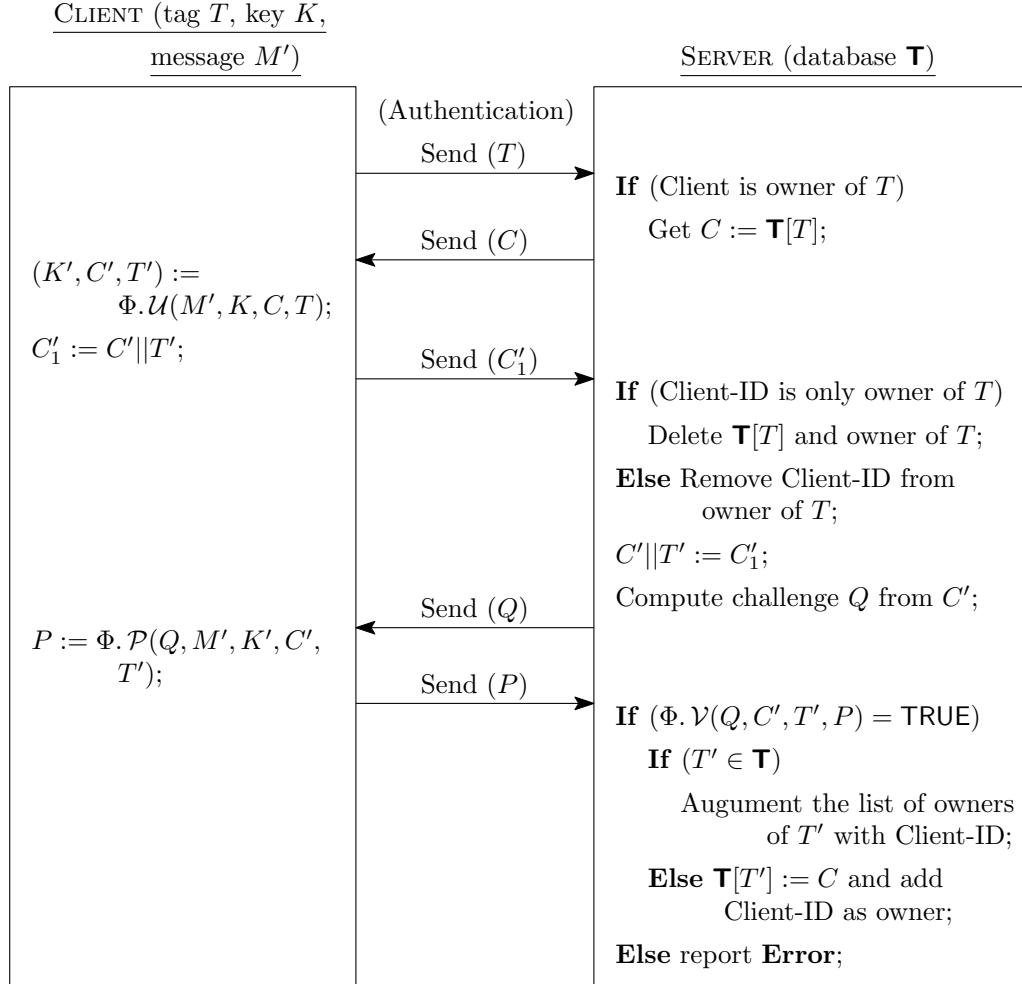


Figure 4.7: DOWNLOAD protocol for *FMLE*-based *deduplication*.

- **DOWNLOAD.** This protocol takes care of the operations performed when the client downloads a previously stored file from the server; the pictorial description is given in Figure 4.7. After authenticating its identity to the server, the client sends tag T corresponding to the required file C . If the server discovers that the client is a genuine owner of C , it then sends ciphertext C to the client. The client decrypts C to obtain M , and accepts M only if $M \neq \perp$.
- **UPDATE.** This protocol is responsible for the operations carried out while the client downloads (a part of) the previously uploaded file, and uploads the modified file; the pictorial description is given in Figure 4.8. The client first sends tag T corresponding to the required file C . After verifying client's ownership, the server sends C to the client. The client updates C using $\Phi.U(\cdot)$, to obtain new K' , C' and T' , and sends $C'_1 := C' \| T'$ to the server. Now, the server performs the following operations: first it revokes the ownership of client from C ; then it computes C' and T' from C'_1 , then it generates a challenge Q for C' , and finally sends Q to the client. In response, the client computes proof P corresponding to challenge Q using $\Phi.P(\cdot)$, and sends it to the server. After successful verification of P by the server using $\Phi.V(\cdot)$, if T' and C' are already stored in the database, then client-ID is appended in the *list of owners*, otherwise the server stores C' in the database at index T' , and creates a file *list of owners* for C' with client-ID as the owner.

Figure 4.8: UPDATE protocol for *FMLE*-based *deduplication*.

4.5 Practical *FMLE* Constructions from existing *MLE* schemes

In this section, we give three *FMLE* schemes – F-CE, F-HCE2 and F-RCE – built by augmenting the existing *MLE* schemes – CE, HCE2 and RCE. These *MLE* constructions are built from a hash function family $\vartheta = (\vartheta, \mathcal{H})$ over ϑ . $\text{Setup}(1^\lambda)$, and a *symmetric encryption* scheme $\rho = (\rho, \mathcal{GEN}, \rho, \mathcal{E}, \rho, \mathcal{D})$ over ρ . $\text{Setup}(1^\lambda)$, as discussed in Sections 2.2.3 and 2.2.4 respectively.

4.5.1 Construction F-CE

The *MLE* construction *CE*, proposed by Douceur *et al.* (described in Section 3.4.3.1), can be converted into an *FMLE* scheme, denoted *F-CE*, by adding three new algorithms to the existing set of two algorithms, as shown in Figures 4.9, 4.10, 4.11, 4.12 and 4.13. Below, we give the textual description.

- The *setup* algorithm $\text{F-CE}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\text{F-CE})}$, *key-space* $\mathcal{K}^{(\text{F-CE})}$, *message-space* $\mathcal{M}^{(\text{F-CE})}$, *ciphertext-space* $\mathcal{C}^{(\text{F-CE})}$ and *tag-space* $\mathcal{T}^{(\text{F-CE})}$. The $\text{F-CE}.\text{Setup}(\cdot)$ is designed in the following way: first it invokes $\vartheta.\text{Setup}(1^\lambda)$ and $\rho.\text{Setup}(1^\lambda)$; then it computes $\text{params}^{(\text{F-CE})}$ that includes, among others, $\text{params}^{(\vartheta)}$ and $\text{params}^{(\rho)}$; and finally it computes *key-space* $\mathcal{K}^{(\text{F-CE})} = \mathcal{T}^{(\vartheta)} = \mathcal{K}^{(\rho)}$, *message-space* $\mathcal{M}^{(\text{F-CE})} = \mathcal{M}^{(\vartheta)} = \mathcal{M}^{(\rho)} \cup \mathcal{C}^{(\rho)}$, *ciphertext-space* $\mathcal{C}^{(\text{F-CE})} = \mathcal{M}^{(\vartheta)} = \mathcal{M}^{(\rho)} \cup \mathcal{C}^{(\rho)}$ and *tag-space* $\mathcal{T}^{(\text{F-CE})} = \mathcal{T}^{(\vartheta)}$.

```

F-CE.  $\mathcal{E}(\text{params}^{(\text{F-CE})}, M)$ 
#Computing key, ciphertext and tag.
 $K := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, M);$ 
 $C := \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M);$ 
 $T := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, C);$ 

#Returning final output.
return  $(K, C, T);$ 

```

Figure 4.9: Algorithmic description of the *encryption* function $\text{F-CE}.\mathcal{E}$.

- The *encryption* algorithm $\text{F-CE}.\mathcal{E}(\cdot)$ is a randomised algorithm that takes as input parameter $\text{params}^{(\text{F-CE})}$ and message M , and outputs key K , ciphertext C and tag T . The pseudocode is given in Figure 4.9. Very briefly, the algorithm works as follows: first it computes key $K := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, M)$; then it computes ciphertext $C := \rho. \mathcal{E}(\text{params}^{(\rho)}, K, M)$; and finally tag $T := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, C)$ is computed.
- The *decryption* algorithm $\text{F-CE}.\mathcal{D}(\cdot)$ is a deterministic algorithm that takes as input parameter $\text{params}^{(\text{F-CE})}$, key K , ciphertext C and tag T . The pseudocode is given in Figure 4.10. The algorithm works as follows: first it computes tag $T' := \vartheta. \mathcal{H}(\text{params}^{(\vartheta)}, C)$; and if

F-CE. $\mathcal{D}(params^{(\text{F-CE})}, K, C, T)$

#Verifying ciphertext and tag.

$T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, C);$

If ($T' \neq T$), **then** return \perp ;

#Computing final output.

$M := \rho \cdot \mathcal{D}(params^{(\rho)}, K, C);$

return M ;

Figure 4.10: Algorithmic description of the *decryption* function F-CE. \mathcal{D} .

$T' \neq T$, then it returns \perp , otherwise it computes and returns message $M := \rho \cdot \mathcal{D}(params^{(\rho)}, K, C)$.

F-CE. $\mathcal{U}(params^{(\text{F-CE})}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$

#Computing original message.

$M := \rho \cdot \mathcal{D}(params^{(\rho)}, K, C);$

#Computing new message.

$M' := M[1] \| M[2] \| \cdots \| M[i_{st} - 1] \| M_{\text{new}};$

if ($app = 0$)

$m := |M|;$

$M' := M' \| M[i_{end} + 1] \| M[i_{end} + 2] \| \cdots \| M[m];$

#Computing key, ciphertext and tag.

$K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M');$

$C' := \rho \cdot \mathcal{E}(params^{(\rho)}, K', M');$

$T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, C');$

#Returning final output.

return (K', C', T') ;

Figure 4.11: Algorithmic description of the *update* function F-CE. \mathcal{U} .

- The *update* algorithm F-CE. $\mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $params^{(\text{F-CE})}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' , updated ciphertext C' and updated tag T' . The pseudocode is given in Figure 4.11. Very briefly, the algorithm works as follows:

first it computes original message $M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, K, C)$; then it computes new message $M' := M[1] \| M[2] \| \cdots \| M[i_{st} - 1] \| M_{\text{new}}$; if $app = 0$, that is, length of message is not to be changed, then at first, $m := |M|$ is computed, and then new message $M' := M' \| M[i_{end} + 1] \| M[i_{end} + 2] \| \cdots \| M[m]$ is updated; after that it computes updated key $K' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, M')$; next it encrypts new message $C' := \rho \cdot \mathcal{E}(\text{params}^{(\rho)}, K', M')$; and finally it computes updated tag $T' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, C')$.

F-CE. $\mathcal{P}(\text{params}^{(\text{F-CE})}, Q = \{i_1, i_2, \dots, i_\sigma\}, M, K, C, T)$

#Computing the proof.

```

 $P := C[i_1] \| C[i_2] \| \cdots \| C[i_\sigma];$ 
return  $P;$ 
```

Figure 4.12: Algorithmic description of the *PoW algorithm for prover F-CE*. \mathcal{P} .

- The PPT *PoW algorithm for prover F-CE*. $\mathcal{P}(\cdot)$ takes as input parameter $\text{params}^{(\text{F-CE})}$, challenge $Q = \{i_1, i_2, \dots, i_\sigma\}$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$, message M , decryption key K , ciphertext C and tag T , and outputs proof $P \in \{0, 1\}^\sigma$ as sequence of ciphertext bits. The pseudocode is given in Figure 4.12. The proof P is computed $P := C[i_1] \| C[i_2] \| \cdots \| C[i_\sigma]$.

F-CE. $\mathcal{V}(\text{params}^{(\text{F-CE})}, Q = \{i_1, i_2, \dots, i_\sigma\}, C', T', P)$

#Verifying bit-by-bit.

```

for ( $j := 1, 2, \dots, \sigma$ )
  If ( $C'[i_j] \neq P[j]$ ), then return FALSE;
  return TRUE;
```

Figure 4.13: Algorithmic description of the *PoW algorithm for verifier F-CE*. \mathcal{V} .

- The PPT *PoW algorithm for verifier F-CE*. $\mathcal{V}(\cdot)$ takes as input parameter $\text{params}^{(\text{F-CE})}$, challenge $Q = \{i_1, i_2, \dots, i_\sigma\}$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$, ciphertext C' , tag T' and proof P , and outputs value $val \in \{\text{TRUE}, \text{FALSE}\}$. The pseudocode is given in Figure 4.13. The algorithm returns **TRUE**, if $C'[i_j] = P[j]$, for all $j \in [\sigma]$. Otherwise, the algorithm returns **FALSE**.

Security of **F-CE**

The *FMLE* scheme **F-CE** is constructed by adding functions to the *MLE* scheme **CE**. Since **CE** is *MLE-PRV\$*, *MLE-TC* and *MLE-STC* secure, the *FMLE* scheme **F-CE** is also *FMLE-PRV\$*, *FMLE-TC* and *FMLE-STC* secure. The *FMLE-CXH* and *FMLE-UNC* security of the **F-CE** follows from the fact that it is deterministic.

4.5.2 Construction **F-HCE2**

The *MLE* construction **HCE2**, proposed by Bellare, Keelveedhi and Ristenpart (described in Section 3.4.3.2), can be modified into an *FMLE* scheme, denoted **F-HCE2**, by adding three new algorithms to the existing set of two algorithms, as shown in Figures 4.14, 4.15, 4.16, 4.17 and 4.18. Below, we give the textual description.

- The *setup* algorithm **F-HCE2. Setup**(\cdot) takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\text{F-HCE2})}$, key-space $\mathcal{K}^{(\text{F-HCE2})}$, message-space $\mathcal{M}^{(\text{F-HCE2})}$, ciphertext-space $\mathcal{C}^{(\text{F-HCE2})}$ and tag-space $\mathcal{T}^{(\text{F-HCE2})}$. The **F-HCE2. Setup**(\cdot) is designed in the following way: first it invokes $\vartheta.\text{Setup}(1^\lambda)$ and $\rho.\text{Setup}(1^\lambda)$; then it computes $params^{(\text{F-HCE2})}$ that includes, among others, $params^{(\vartheta)}$ and $params^{(\rho)}$; and finally it computes key-space $\mathcal{K}^{(\text{F-HCE2})} = \mathcal{T}^{(\vartheta)} = \mathcal{K}^{(\rho)}$, message-space $\mathcal{M}^{(\text{F-HCE2})} = \mathcal{M}^{(\vartheta)} = \mathcal{M}^{(\rho)} = \mathcal{K}^{(\rho)} \cup \mathcal{M}^{(\rho)}$, ciphertext-space $\mathcal{C}^{(\text{F-HCE2})} = \mathcal{C}^{(\rho)}$ and tag-space $\mathcal{T}^{(\text{F-HCE2})} = \mathcal{T}^{(\vartheta)}$.

```

F-HCE2. E( $params^{(\text{F-HCE2})}, M$ )
#Computing key, ciphertext and tag.
     $K := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$ 
     $C := \rho \cdot \mathcal{E}(params^{(\rho)}, K, M);$ 
     $T := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K);$ 

#Returning final output.
    return ( $K, C, T$ );

```

Figure 4.14: Algorithmic description of the *encryption* function **F-HCE2. E**.

- The *encryption* algorithm **F-HCE2. E**(\cdot) is a randomised algorithm that takes as input parameter $params^{(\text{F-HCE2})}$ and message M , and outputs key K , ciphertext C and tag T . The pseudocode is given in Figure 4.14. The algorithm works as follows: first it computes

key $K := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, M)$; then it computes ciphertext $C := \rho \cdot \mathcal{E}(\text{params}^{(\rho)}, K, M)$; and finally tag $T := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, K)$ is computed.

```
F-HCE2.  $\mathcal{D}(\text{params}^{(\text{F-HCE2})}, K, C, T)$ 
#Computing message.
 $M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, K, C);$ 

#Verifying tag.
 $K' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, M);$ 
 $T' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, K');$ 
If ( $T' \neq T$ ), then return  $\perp$ ; Else return  $M$ ;
```

Figure 4.15: Algorithmic description of the *decryption* function $\text{F-HCE2. } \mathcal{D}$.

- The *decryption* algorithm $\text{F-HCE2. } \mathcal{D}(\cdot)$ is a deterministic algorithm that takes as input parameter $\text{params}^{(\text{F-HCE2})}$, key K , ciphertext C and tag T . The pseudocode is given in Figure 4.15. Very briefly, the algorithm works as follows: first it computes message $M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, K, C)$; then it computes key $K' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, M)$; after that it computes tag $T' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, K')$; and finally if $T' \neq T$, then it returns \perp , otherwise it returns message M .
- The *update* algorithm $\text{F-HCE2. } \mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $\text{params}^{(\text{F-HCE2})}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' , updated ciphertext C' and updated tag T' . The pseudocode is given in Figure 4.16. The algorithm works as follows: first it computes original message $M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, K, C)$; then it computes new message $M' := M[1] \| M[2] \| \cdots \| M[i_{st}-1] \| M_{\text{new}}$; it checks if $app = 0$, that is, length of message is not to be changed, then at first, $m := |M|$ is computed, and then new message $M' := M' \| M[i_{end}+1] \| M[i_{end}+2] \| \cdots \| M[m]$ is updated; after that it computes updated key $K' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, M')$; next it encrypts new message $C' := \rho \cdot \mathcal{E}(\text{params}^{(\rho)}, K', M')$; and finally it computes updated tag $T' := \vartheta \cdot \mathcal{H}(\text{params}^{(\vartheta)}, K')$.
- The PPT *PoW algorithm for prover* $\text{F-HCE2. } \mathcal{P}(\cdot)$ takes as input parameter $\text{params}^{(\text{F-HCE2})}$, challenge $Q = \{i_1, i_2, \dots, i_\sigma\}$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$, message M , decryption key K , ciphertext C and

```

F-HCE2. $\mathcal{U}(params^{(\text{F-HCE2})}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$ 

#Computing original message.
 $M := \rho \cdot \mathcal{D}(params^{(\rho)}, K, C);$ 

#Computing new message.
 $M' := M[1] \| M[2] \| \cdots \| M[i_{st} - 1] \| M_{\text{new}};$ 
If ( $app = 0$ )
 $m := |M|;$ 
 $M' := M' \| M[i_{end} + 1] \| M[i_{end} + 2] \| \cdots \| M[m];$ 

#Computing key, ciphertext and tag.
 $K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M');$ 
 $C' := \rho \cdot \mathcal{E}(params^{(\rho)}, K', M');$ 
 $T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K');$ 

#Returning final output.
return  $(K', C', T');$ 

```

Figure 4.16: Algorithmic description of the *update* function **F-HCE2.** \mathcal{U} .

```

F-HCE2. $\mathcal{P}(params^{(\text{F-HCE2})}, Q = \{i_1, i_2, \dots, i_\sigma\}, M, K, C, T)$ 

#Computing the proof.
 $P := C[i_1] \| C[i_2] \| \cdots \| C[i_\sigma];$ 
return  $P;$ 

```

Figure 4.17: Algorithmic description of the *PoW algorithm for prover* **F-HCE2.** \mathcal{P} .

tag T , and outputs proof $P \in \{0, 1\}^\sigma$ as sequence of ciphertext bits. The pseudocode is given in Figure 4.17. The proof P is computed $P := C[i_1] \| C[i_2] \| \cdots \| C[i_\sigma]$.

- The PPT *PoW algorithm for verifier* **F-HCE2.** $\mathcal{V}(\cdot)$ takes as input parameter $params^{(\text{F-HCE2})}$, challenge $Q = \{i_1, i_2, \dots, i_\sigma\}$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$, ciphertext C' , tag T' and proof P , and outputs value $val \in \{\text{TRUE}, \text{FALSE}\}$. The pseudocode is given in Figure 4.18. The algorithm returns **TRUE**, if $C'[i_j] = P[j]$, for all $j \in [\sigma]$. Otherwise, the algorithm returns **FALSE**.

F-HCE2. $\mathcal{V}(params^{(F\text{-HCE2})}, Q = \{i_1, i_2, \dots, i_\sigma\}, C', T', P)$

```
#Verifying bit-by-bit.
for (j := 1, 2, ..., σ)
    If (C'[i_j] ≠ P[j]), then, return FALSE;
return TRUE;
```

Figure 4.18: Algorithmic description of the *PoW algorithm for verifier F-HCE2. \mathcal{V} .*

Security of F-HCE2

The *MLE* scheme F-HCE2 is constructed by adding functions to the *MLE* scheme HCE2. Since HCE2 is MLE-PRV\$ and MLE-TC secure, the *FMLE* scheme F-HCE2 is also FMLE-PRV\$ and FMLE-TC secure. The FMLE-CXH and FMLE-UNC security of the F-HCE2 follows from the fact that it is deterministic.

4.5.3 Construction F-RCE

The *MLE* construction RCE, proposed by Bellare, Keelvedhi and Ristenpart (described in Section 3.4.3.3), can be converted into an *FMLE* scheme, denoted F-RCE, by adding three new algorithms to the existing set of two algorithms, as shown in Figures 4.19, 4.20, 4.21, 4.22 and 4.23. Below, we give the textual description.

- The *setup* algorithm F-RCE. $\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(F\text{-RCE})}$, key-space $\mathcal{K}^{(F\text{-RCE})}$, message-space $\mathcal{M}^{(F\text{-RCE})}$, ciphertext-space $\mathcal{C}^{(F\text{-RCE})}$ and tag-space $\mathcal{T}^{(F\text{-RCE})}$. The F-RCE. $\text{Setup}(\cdot)$ is designed in the following way: first it invokes $\vartheta.\text{Setup}(1^\lambda)$ and $\rho.\text{Setup}(1^\lambda)$; then it computes $params^{(F\text{-RCE})}$ that includes, among others, $params^{(\vartheta)}$ and $params^{(\rho)}$; and finally it computes key-space $\mathcal{K}^{(F\text{-RCE})} = \mathcal{T}^{(\vartheta)} = \mathcal{K}^{(\rho)}$, message-space $\mathcal{M}^{(F\text{-RCE})} = \mathcal{M}^{(\vartheta)} = \mathcal{K}^{(\rho)} \cup \mathcal{M}^{(\rho)}$, ciphertext-space $\mathcal{C}^{(F\text{-RCE})} = \mathcal{C}^{(\rho)}$ and tag-space $\mathcal{T}^{(F\text{-RCE})} = \mathcal{T}^{(\vartheta)}$.
- The *encryption* algorithm F-RCE. $\mathcal{E}(\cdot)$ is a randomised algorithm that takes as input parameter $params^{(F\text{-RCE})}$ and message M , and outputs key K , ciphertext C and tag T . The pseudocode is given in Figure 4.19. Very briefly, the algorithm works as follows: first it computes key $K := \vartheta.\mathcal{H}(params^{(\vartheta)}, M)$; then it randomly chooses a string L from $\mathcal{K}^{(\rho)}$; next it computes $C_1 := \rho.\mathcal{E}(params^{(\rho)}, L, M)$; after that

```

F-RCE.  $\mathcal{E}(params^{(\text{F-RCE})}, M)$ 


---


#Computing key, ciphertext and tag.

$$K := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M); \quad L \xleftarrow{\$} \mathcal{K}^{(\rho)};$$


$$C_1 := \rho \cdot \mathcal{E}(params^{(\rho)}, L, M); \quad C_2 := L \oplus K;$$


$$C := C_1 \| C_2; \quad T := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K);$$

#Returning final output.
return  $(K, C, T);$ 

```

Figure 4.19: Algorithmic description of the *encryption* function F-RCE. \mathcal{E} .

it computes $C_2 := L \oplus K$; next it computes ciphertext $C := C_1 \| C_2$; and finally tag $T := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K)$ is computed.

```

F-RCE.  $\mathcal{D}(params^{(\text{F-RCE})}, K, C, T)$ 


---


#Computing message.

$$C_1 \| C_2 := C; \quad L := C_2 \oplus K;$$


$$M := \rho \cdot \mathcal{D}(params^{(\rho)}, L, C_1);$$

#Verifying tag.

$$K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M);$$


$$T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K');$$

If  $(T' \neq T)$ , then return  $\perp$ ; Else return  $M$ ;

```

Figure 4.20: Algorithmic description of the *decryption* function F-RCE. \mathcal{D} .

- The *decryption* algorithm F-RCE. $\mathcal{D}(\cdot)$ is a deterministic algorithm that takes as input parameter $params^{(\text{F-RCE})}$, key K , ciphertext C and tag T . The pseudocode is given in Figure 4.20. The algorithm works as follows: first it parses $C_1 \| C_2 := C$; then it computes $L := C_2 \oplus K$; next it computes message $M := \rho \cdot \mathcal{D}(params^{(\rho)}, L, C_1)$; then it computes key $K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M)$; after that it computes tag $T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K')$; and finally, if $T' \neq T$, then it returns \perp , otherwise it returns message M .
- The *update* algorithm F-RCE. $\mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $params^{(\text{F-RCE})}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' , updated ciphertext C' and updated tag T' . The pseudocode

```

F-RCE.U( $params^{(F\text{-RCE})}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app$ )


---


#Computing original message.
 $C_1 \| C_2 := C;$      $L := C_2 \oplus K;$ 
 $M := \rho \cdot \mathcal{D}(params^{(\rho)}, L, C_1);$ 

#Computing new message.
 $M' := M[1] \| M[2] \| \cdots \| M[i_{st} - 1] \| M_{\text{new}};$ 
If ( $app = 0$ )
   $m := |M|;$ 
   $M' := M' \| M[i_{end} + 1] \| M[i_{end} + 2] \| \cdots \| M[m];$ 

#Computing key, ciphertext and tag.
 $K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M');$ 
 $C'_1 := \rho \cdot \mathcal{E}(params^{(\rho)}, L, M');$      $C'_2 := L \oplus K';$ 
 $C' := C'_1 \| C'_2;$      $T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K');$ 

#Returning final output.
  return  $(K', C', T');$ 

```

Figure 4.21: Algorithmic description of the *update* function $F\text{-RCE.U}$.

is given in Figure 4.21. Very briefly, the algorithm works as follows: first it parses $C_1 \| C_2 := C$; then it computes $L := C_2 \oplus K$; next it computes original message $M := \rho \cdot \mathcal{D}(params^{(\rho)}, L, C_1)$; after that it computes new message $M' := M[1] \| M[2] \| \cdots \| M[i_{st} - 1] \| M_{\text{new}}$; if $app = 0$, that is, length of message is not to be changed, then, at first, $m := |M|$ is computed, and then new message $M' := M' \| M[i_{end} + 1] \| M[i_{end} + 2] \| \cdots \| M[m]$ is updated; after that it computes updated key $K' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, M')$; next it encrypts new message $C'_1 := \rho \cdot \mathcal{E}(params^{(\rho)}, L, M')$; then it computes $C'_2 := L \oplus K'$; after that it computes ciphertext $C' := C'_1 \| C'_2$; and finally it computes updated tag $T' := \vartheta \cdot \mathcal{H}(params^{(\vartheta)}, K')$.

- The PPT *PoW algorithm for prover* $F\text{-RCE.P}(\cdot)$ takes as input parameter $params^{(F\text{-RCE})}$, challenge $Q = (\{i_1, i_2, \dots, i_\sigma\}, C'_2)$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$ and the last block of already stored ciphertext C'_2 , message M , decryption key K , ciphertext C and tag T , and outputs proof $P \in \{0, 1\}^\sigma$ as sequence of ciphertext bits. The pseudocode is given in Figure 4.22. Very briefly, the algorithm works as follows: first it parses $C_1 \| C_2 := C$; then it computes $L :=$

F-RCE. $\mathcal{P}(\text{params}^{(\text{F-RCE})}, Q = (\{i_1, i_2, \dots, i_\sigma\}, C'_2), M, K, C, T)$

#Computing proof.

$C_1 \| C_2 := C;$

$L := C_2 \oplus K; \quad L' := C'_2 \oplus K;$

$M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, L, C_1); \quad C'_1 := \rho \cdot \mathcal{E}(\text{params}^{(\rho)}, L', M);$

$C' := C'_1 \| C'_2;$

$P := C'[i_1] \| C'[i_2] \| \dots \| C'[i_\sigma];$

#Returning final output.

return $P;$

Figure 4.22: Algorithmic description of the *PoW algorithm for prover F-RCE*. \mathcal{P} .

$C_2 \oplus K$; after that it computes $L' := C'_2 \oplus K$; next it computes message $M := \rho \cdot \mathcal{D}(\text{params}^{(\rho)}, L, C_1)$; after that it encrypts message $C'_1 := \rho \cdot \mathcal{E}(\text{params}^{(\rho)}, L', M)$; then it computes $C' := C'_1 \| C'_2$; and finally it computes proof $P := C'[i_1] \| C'[i_2] \| \dots \| C'[i_\sigma]$.

F-RCE. $\mathcal{V}(\text{params}^{(\text{F-RCE})}, Q = (\{i_1, i_2, \dots, i_\sigma\}, C'_2), C', T', P)$

#Verifying bit-by-bit.

for ($j := 1, 2, \dots, \sigma$)

If ($C'[i_j] \neq P[j]$), **then** return FALSE;

Else return TRUE;

Figure 4.23: Algorithmic description of the *PoW algorithm for verifier F-RCE*. \mathcal{V} .

- The PPT *PoW algorithm for verifier F-RCE*. $\mathcal{V}(\cdot)$ takes as input parameter $\text{params}^{(\text{F-RCE})}$, challenge $Q = (\{i_1, i_2, \dots, i_\sigma\}, C'_2)$ as set of indices of bits $i_1, i_2, \dots, i_\sigma$ and the last block of already stored ciphertext C'_2 , ciphertext C' , tag T' and proof P , and outputs value $\text{val} \in \{\text{TRUE}, \text{FALSE}\}$. The pseudocode is given in Figure 4.23. The algorithm returns TRUE, if $C'[i_j] = P[j]$, for all $j \in [\sigma]$. Otherwise, the algorithm returns FALSE.

Security of F-RCE

The *FMLE* scheme F-RCE is constructed by adding functions to the *MLE* scheme RCE. Since RCE is *MLE-PRV\$* and *MLE-TC* secure, the *FMLE*

scheme F-RCE is also FMLE-PRV\$ and FMLE-TC secure (the reduction between them is obvious).

The FMLE-CXH security of the scheme F-RCE follows from the fact that the new decryption key is generated for the updated message, and the whole updated message is re-encrypted (as described in the Figures 4.19 and 4.21). Therefore, the advantage of the adversary is *exactly* zero.

The FMLE-UNC security is achieved in the F-RCE scheme, because the proof P^* produced by the adversary has to be identical to the real proof P , which is already stored as the ciphertext in the cloud, and verification algorithm only checks equality.

4.6 Practical *FMLE* Construction from existing *UMLE* scheme

In this section, we give one *FMLE* scheme, denoted F-UMLE, built by augmenting the existing *UMLE* scheme.

4.6.1 Construction F-UMLE

The F-UMLE scheme is constructed from a *UMLE* scheme $\zeta = (\zeta.\text{KeyGen}, \zeta.\text{Enc}, \zeta.\text{TagGen}, \zeta.\text{Dec}, \zeta.\text{Update}, \zeta.\text{UpdateTag}, \zeta.\text{PoWPrf}, \zeta.\text{PoWVer})$ over the *setup* algorithm $\zeta.\text{Setup}$ (described in Section 3.4.5). The pseudocode for the 5-tuple of algorithms of F-UMLE are given in Figures 4.24, 4.25, 4.26, 4.27 and 4.28. Below, we give the textual description.

- The *setup* algorithm $\text{F-UMLE}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\text{F-UMLE})}$, *key-space* $\mathcal{K}^{(\text{F-UMLE})}$, *message-space* $\mathcal{M}^{(\text{F-UMLE})}$, *ciphertext-space* $\mathcal{C}^{(\text{F-UMLE})}$ and *tag-space* $\mathcal{T}^{(\text{F-UMLE})}$. The $\text{F-UMLE}.\text{Setup}(\cdot)$ is designed in the following way: first it invokes $\zeta.\text{Setup}(1^\lambda)$; then it computes $\text{params}^{(\text{F-UMLE})}$ that includes, among others, $\text{params}^{(\zeta)}$; and finally it computes *key-space* $\mathcal{K}^{(\text{F-UMLE})} = \mathcal{K}^{(\zeta)}$, *message-space* $\mathcal{M}^{(\text{F-UMLE})} = \mathcal{M}^{(\zeta)}$, *ciphertext-space* $\mathcal{C}^{(\text{F-UMLE})} = \mathcal{C}^{(\zeta)}$ and *tag-space* $\mathcal{T}^{(\text{F-UMLE})} = \mathcal{T}^{(\zeta)}$.
- The *encryption* algorithm $\text{F-UMLE}.\mathcal{E}(\cdot)$ is a randomised algorithm that takes as input parameter $\text{params}^{(\text{F-UMLE})}$ and message M , and outputs key K , ciphertext C and tag T . The pseudocode is given in Figure 4.24. Very briefly, the algorithm works as follows: first it computes keys $(k_{mas}, k_1, k_2, \dots, k_n) := \zeta.\text{KeyGen}(\text{params}^{(\zeta)}, M)$; then it assigns value of k_{mas} to K ; next it computes ciphertext $C :=$

```

F-UMLE.  $\mathcal{E}(params^{(F-UMLE)}, M)$ 


---


# Computing key, ciphertext and tag.
 $(k_{mas}, k_1, k_2, \dots, k_n) := \zeta \cdot \text{KeyGen}(params^{(\zeta)}, M);$ 
 $K := k_{mas};$ 
 $C := \zeta \cdot \text{Enc}(params^{(\zeta)}, (k_{mas}, k_1, k_2, \dots, k_n), M);$ 
 $T := \zeta \cdot \text{TagGen}(params^{(\zeta)}, C);$ 

# Returning final output.
return  $(K, C, T);$ 

```

Figure 4.24: Algorithmic description of the *encryption* function $\text{F-UMLE. } \mathcal{E}$.

$\zeta \cdot \text{Enc}(params^{(\zeta)}, (k_{mas}, k_1, k_2, \dots, k_n), M)$; and finally tag $T := \zeta \cdot \text{TagGen}(params^{(\zeta)}, C)$ is computed.

```

F-UMLE.  $\mathcal{D}(params^{(F-UMLE)}, K, C, T)$ 


---


# Verifying tag.
 $T' := \zeta \cdot \text{TagGen}(params^{(\zeta)}, C);$ 
If  $(T' \neq T)$ , then return  $\perp$ ;

# Computing final output.
 $M := \zeta \cdot \text{Dec}(params^{(\zeta)}, K, C);$ 
return  $M;$ 

```

Figure 4.25: Algorithmic description of the *decryption* function $\text{F-UMLE. } \mathcal{D}$.

- The *decryption* algorithm $\text{F-UMLE. } \mathcal{D}(\cdot)$ is a deterministic algorithm that takes as input parameter $params^{(F-UMLE)}$, key K , ciphertext C and tag T . The pseudocode is given in Figure 4.25. The algorithm works as follows: first it computes tag $T' := \zeta \cdot \text{TagGen}(params^{(\zeta)}, C)$; and finally checks if $T' \neq T$, then it returns \perp , otherwise it computes and returns message $M := \zeta \cdot \text{Dec}(params^{(\zeta)}, K, C)$.
- The *update* algorithm $\text{F-UMLE. } \mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $params^{(F-UMLE)}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' , updated ciphertext C' and updated tag T' . The pseudocode is given in Figure 4.26. Very briefly, the algorithm works as follows: first it computes index of block containing i_{st} th bit $i := \lfloor i_{st}/\lambda \rfloor$; and then it

```

F-UMLE.  $\mathcal{U}(params^{(F\text{-}UMLE)}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$ 
#Computing new key and ciphertext.
 $i := \lfloor i_{st}/\lambda \rfloor; \quad K' := K; \quad C' := C;$ 
for( $j := i, i + 1, \dots, \lfloor i_{end}/\lambda \rfloor$ )
   $(K', C') := \zeta.\text{Update}(params^{(\zeta)}, (K', j, M_{\text{new}}[j - i + 1]), C');$ 

#Computing new tag.
 $T' := \zeta.\text{UpdateTag}(params^{(\zeta)}, T, C, C');$ 

#Returning final output.
return  $(K', C', T');$ 

```

Figure 4.26: Algorithmic description of the *update* function **F-UMLE. \mathcal{U}** .

initializes K' and C' with K and C . Now, values of K' and C' are computed iteratively, as j runs through $i, i + 1, \dots, \lfloor i_{end}/\lambda \rfloor$, by updating values $(K', C') := \zeta.\text{Update}(params^{(\zeta)}, (K', j, M_{\text{new}}[j - i + 1]), C')$ in every iteration. Finally, new tag $T' := \zeta.\text{UpdateTag}(params^{(\zeta)}, T, C, C')$ is computed.

```

F-UMLE.  $\mathcal{P}(params^{(F\text{-}UMLE)}, Q, M, K, C, T)$ 
#Computing final output.
 $P := \zeta.\text{PoWPrf}(params^{(\zeta)}, Q, M);$ 
return  $P;$ 

```

Figure 4.27: Algorithmic description of the *PoW algorithm for prover* **F-UMLE. \mathcal{P}** .

- The PPT *PoW algorithm for prover* **F-UMLE. $\mathcal{P}(\cdot)$** takes as input parameter $params^{(F\text{-}UMLE)}$, challenge Q , message M , decryption key K , ciphertext C and tag T , and outputs proof P . The pseudocode is given in Figure 4.27. The proof P is computed $P := \zeta.\text{PoWPrf}(params^{(\zeta)}, Q, M)$.
- The PPT *PoW algorithm for verifier* **F-UMLE. $\mathcal{V}(\cdot)$** takes as input parameter $params^{(F\text{-}UMLE)}$, challenge Q , ciphertext C' , tag T' and proof P , and outputs value $val \in \{\text{TRUE}, \text{FALSE}\}$. The pseudocode is given in Figure 4.28. The val is computed $val := \zeta.\text{PoWVer}(params^{(\zeta)}, Q, T', P)$.

F-UMLE. $\mathcal{V}(params^{(F\text{-}UMLE)}, Q, C', T', P)$

#Computing final output.

```
val :=  $\zeta$ . PoWVer(params $^{(\zeta)}$ , Q, T', P);
return val;
```

Figure 4.28: Algorithmic description of the *PoW algorithm for verifier F-UMLE*. \mathcal{V} .

Security of **F-UMLE**

We note that the **F-UMLE** scheme is a *UMLE* as well as an *FMLE* scheme. Also, the four security targets – **FMLE-STC**, **FMLE-TC**, **FMLE-UNC** and **FMLE-CXH** – are identical for both *UMLE* and *FMLE*. Since **F-UMLE** satisfies the *UMLE* security targets, hence **F-UMLE** is a secure *FMLE* scheme against **FMLE-STC**, **FMLE-TC**, **FMLE-UNC** and **FMLE-CXH** security properties.

Since *UMLE* scheme is secure against **UMLE-PRV** security, it can be easily proven that the **F-UMLE** scheme is also secure against **FMLE-PRV**.

Since the *tag generation* algorithm of a *UMLE* scheme takes only ciphertext as input, **F-UMLE** is not **FMLE-PRV\$** secure.¹

4.7 New Efficient *FMLE* Schemes

In this section, we present two new efficient constructions for *FMLE* – denoted $\hat{\Phi}$ and $\tilde{\Phi}$ – which are based on a 2λ -bit *easy-to-invert* permutation π . We assume that the message-length is a multiple of λ ; λ is the security parameter.

4.7.1 Construction $\hat{\Phi}$

This construction is motivated by the design of APE authenticated encryption (described in Section 2.2.5.2).

¹The following attack against **FMLE-PRV\$** of **F-UMLE** is possible: the adversary can always compute the tag from the supplied ciphertext; and then compare it with the supplied tag. If the two values are identical, then the ciphertext and tag correspond to a real message with probability 1.

4.7.1.1 Description of $\hat{\Phi}$

The pictorial description and pseudocode for the 5-tuple of algorithms in $\hat{\Phi} = (\hat{\Phi}.E, \hat{\Phi}.D, \hat{\Phi}.U, \hat{\Phi}.P, \hat{\Phi}.V)$ over the *setup* algorithm $\hat{\Phi}.Setup$ are given in Figures 4.29, 4.30, 4.31, 4.32, 4.33, 4.34, 4.35, 4.36 and 4.37; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\hat{\Phi}.Setup(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\hat{\Phi})}$, *key-space* $\mathcal{K}^{(\hat{\Phi})} := \{0, 1\}^\lambda$, *message-space* $\mathcal{M}^{(\hat{\Phi})} := \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$, *ciphertext-space* $\mathcal{C}^{(\hat{\Phi})} := \bigcup_{i \geq 2} \{0, 1\}^{i\lambda}$, and *tag-space* $\mathcal{T}^{(\hat{\Phi})} := \{0, 1\}^\lambda$.

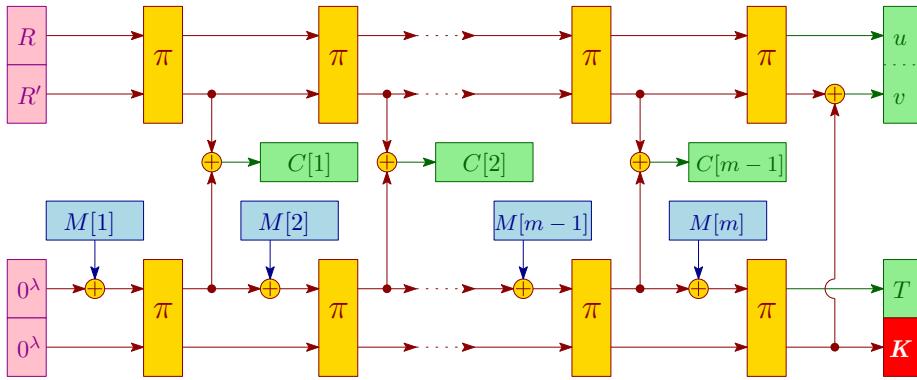


Figure 4.29: Pictorial description of the *encryption* function $\hat{\Phi}.E$.

- The *encryption* algorithm $\hat{\Phi}.E(\cdot)$ is randomized that takes as input parameter $params^{(\hat{\Phi})}$ and message M , and outputs key K , ciphertext C , and tag T . The pictorial description and pseudocode are given in Figures 4.29 and 4.30. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then it initializes both the strings r and s with 0^λ ; and finally it randomly chooses two λ -bit strings u and v . The encryption of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$. First $r \parallel s := \pi((r \oplus M[j]) \parallel s)$ is computed; then $u \parallel v := \pi(u \parallel v)$ is computed; and finally ciphertext block $C[j] := v \oplus r$ is computed.

For the encryption of last block $M[m]$, we perform the following operations: first $r \parallel s := \pi((r \oplus M[m]) \parallel s)$ is computed; then $u \parallel v := \pi(u \parallel v)$ is computed; and finally $v := v \oplus s$ is updated.

```

 $\hat{\Phi} \cdot \mathcal{E}(params^{(\hat{\Phi})}, M)$ 
#Initialization.
 $m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$ 
 $r := 0^\lambda; \quad s := 0^\lambda; \quad u \xleftarrow{\$} \{0, 1\}^\lambda; \quad v \xleftarrow{\$} \{0, 1\}^\lambda;$ 

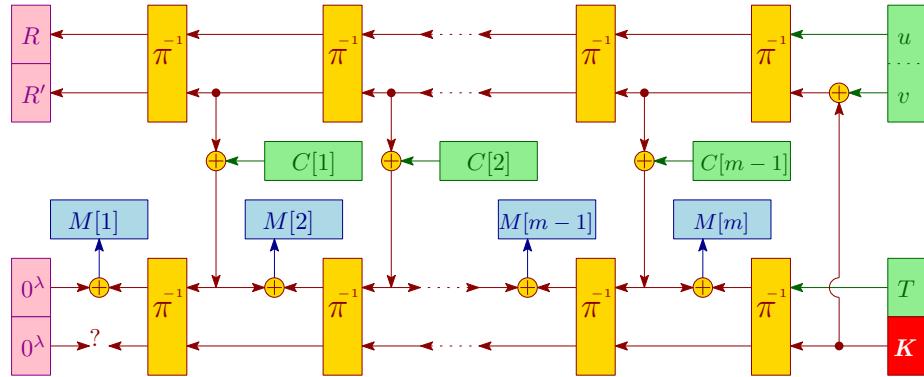
#Processing message blocks.
for ( $j := 1, 2, \dots, m$ )
   $r\|s := \pi((r \oplus M[j])\|s); \quad u\|v := \pi(u\|v);$ 
  If ( $j \neq m$ ), then  $C[j] := v \oplus r;$ 
   $v := v \oplus s;$ 

#Computing final outputs.
 $K := s; \quad C := C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v; \quad T := r;$ 
return ( $K, C, T$ );

```

Figure 4.30: Algorithmic description of the *encryption* function $\hat{\Phi} \cdot \mathcal{E}$.

Now, the key $K := s$, the ciphertext $C := C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v$, and the tag $T := r$.

Figure 4.31: Pictorial description of the *decryption* function $\hat{\Phi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\hat{\Phi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\Phi})}$, key K , ciphertext C and tag T , and outputs message M or an *invalid* symbol \perp . See Figures 4.31 and 4.32 for the pictorial description and pseudocode. The algorithm works as follows: first it parses C into λ -bit blocks $C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C$; next it initializes the value of r and s with T and K ; and it finally

```

 $\hat{\Phi} \cdot \mathcal{D}(params^{(\hat{\Phi})}, K, C, T)$ 


---


#Validating the length.
If ( $|C| < 2\lambda$ ) OR ( $|C| \bmod \lambda \neq 0$ ), then return Error;  

#Initialization.
 $m := (|C|/\lambda) - 1; \quad C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C;$ 
 $r := T; \quad s := K; \quad v := v \oplus s;$   

#Processing ciphertext blocks.
for ( $j := m, m-1, \dots, 1$ )
   $r\|s := \pi^{-1}(r\|s); \quad u\|v := \pi^{-1}(u\|v); \quad M[j] := r;$ 
  If ( $j \neq 1$ ), then  $r := C[j-1] \oplus v; \quad M[j] := M[j] \oplus r;$   

#Computing final outputs.
 $M := M[1]\|M[2]\|\cdots\|M[m];$ 
If ( $s = 0^\lambda$ ), then return  $M$ ; Else return  $\perp$ ;  


```

Figure 4.32: Algorithmic description of the *decryption* function $\hat{\Phi} \cdot \mathcal{D}$.

updates $v := v \oplus s$. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

Now, we give the details of decryption of ciphertext block $C[j]$, for all $j \in [m]$. First $r\|s := \pi^{-1}(r\|s)$ is computed; then $u\|v := \pi^{-1}(u\|v)$ is computed; next the value of r is assigned to $M[j]$; and finally if $j \neq 1$, $r := C[j-1] \oplus v$ is computed and $M[j] := M[j] \oplus r$ is updated.

Now, the message $M := M[1]\|M[2]\|\cdots\|M[m]$. If $s = 0^\lambda$ after the execution of the last iteration, then M is returned, otherwise the *invalid* symbol is returned.

- The *update* algorithm $\hat{\Phi} \cdot \mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $params^{(\hat{\Phi})}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' , updated ciphertext C' and updated tag T' . The pictorial description and pseudocode are given in Figures 4.33 and 4.34. The high level idea is: we start decrypting ciphertext blocks from the end until we reach the block containing i_{st} th bit of the message, and we store the values of r , s , u and v at this point; then we compute new message blocks starting from the first bit of the block containing the i_{st} th bit;

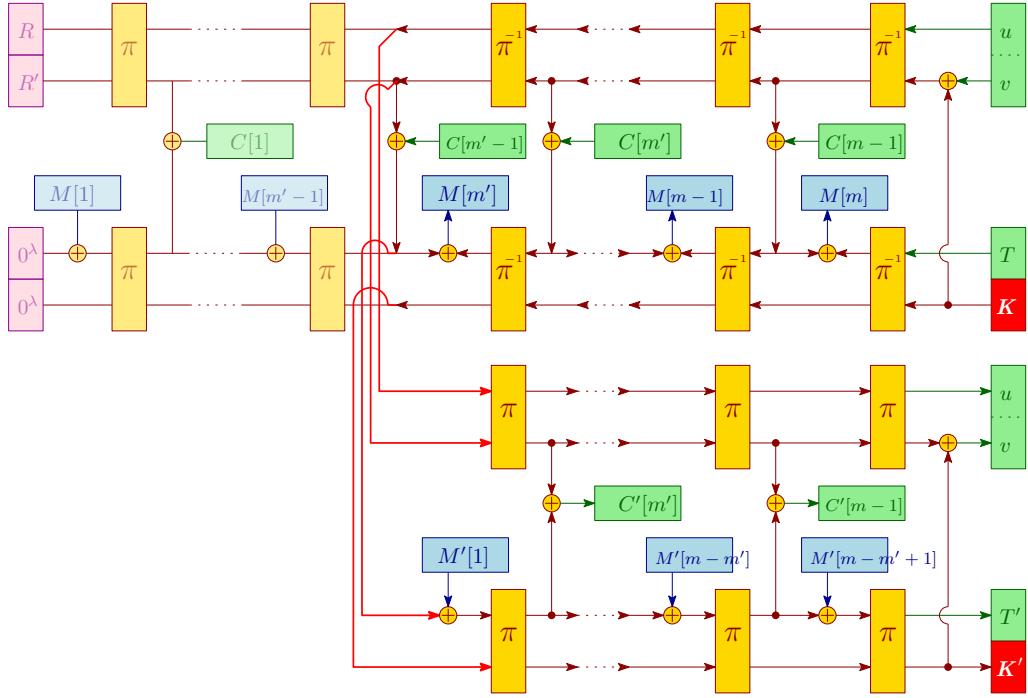


Figure 4.33: Pictorial description of the *update* function $\hat{\Phi}(\mathcal{U})$.

and finally, using the values of r , s , u and v (stored in the previous step), we begin the encryption of the new message blocks.

Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \| C[2] \| \dots \| C[m-1] \| u \| v := C$; then it computes $m' := \lceil i_{st}/\lambda \rceil$; next it initializes the values of r and s with T and K ; and updates $v := v \oplus s$. Now, ciphertext blocks $C[m], C[m-1], \dots, C[m']$ are decrypted, as explained in the decryption algorithm $\hat{\Phi}(\mathcal{D}(\cdot))$. The values of r , s , u and v obtained after decryption of $C[m']$, are preserved for future use.

Now, to compute new message, the following operations are performed: first the position of i_{st} th bit of the message in block $M[m']$ is computed $j := ((i_{st} - 1) \bmod \lambda) + 1$; then new message $M' := M[m'][1] \| M[m'][2] \| \dots \| M[m'][j-1] \| M_{\text{new}}$ is computed; if $app = 0$, that is, length of message is not to be changed, then at first, the block containing $(i_{end} + 1)$ th bit is computed $n := \lceil (i_{end} + 1)/\lambda \rceil$, then position of $(i_{end} + 1)$ th bit of the message in block $M[n]$ is computed $j := (i_{end} \bmod \lambda) + 1$, and finally, new message $M' := M' \| M[n][j] \| M[n][j+1] \| \dots \| M[n][\lambda] \| M[n+1] \| M[n+2] \| \dots \| M[m]$ is updated; if $app = 1$, that is, length of message is to be changed, then

```

 $\hat{\Phi}.\mathcal{U}(params^{(\hat{\Phi})}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$ 

#Initialization.
 $m := (|C|/\lambda) - 1; \quad C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C;$ 
 $r := T; \quad s := K; \quad v := v \oplus s; \quad m' := \lceil i_{st}/\lambda \rceil;$ 

#Processing ciphertext blocks.
for ( $j := m, m-1, \dots, m'$ )
   $r\|s := \pi^{-1}(r\|s); \quad u\|v := \pi^{-1}(u\|v); \quad M[j] := r;$ 
  If ( $j \neq 1$ ), then  $r := C[j-1] \oplus v; M[j] := M[j] \oplus r;$ 

#Computing new message.
 $j := ((i_{st} - 1) \bmod \lambda) + 1;$ 
 $M' := M[m'][1]\|M[m'][2]\|\cdots\|M[m'][j-1]\|M_{\text{new}};$ 
If ( $app = 0$ )
   $n := \lceil (i_{end} + 1)/\lambda \rceil; \quad j := (i_{end} \bmod \lambda) + 1;$ 
   $M' := M'\|M[n][j]\|M[n][j+1]\|\cdots\|M[n][\lambda]\|M[n+1]\|$ 
   $M[n+2]\|\cdots\|M[m];$ 
Else  $m := (m - 1) + (|M'|/\lambda);$ 
 $M'[1]\|M'[2]\|\cdots\|M'[m-m'+1] := M';$ 

#Processing new message blocks.
for ( $j := m', m'+1, \dots, m$ )
   $r\|s := \pi((r \oplus M'[j-m'+1])\|s); \quad u\|v := \pi(u\|v);$ 
  If ( $j \neq m$ ), then  $C'[j] := v \oplus r;$ 
   $v := v \oplus s;$ 

#Computing final outputs.
 $K' := s; \quad T' := r;$ 
 $C' := C[1]\|C[2]\|\cdots\|C[m'-1]\|C'[m']\|\cdots\|C'[m-1]\|u\|v;$ 
return ( $K', C', T'$ );

```

Figure 4.34: Algorithmic description of the $\hat{\Phi}.\mathcal{U}$.

new length $m := (m-1) + (|M'|/\lambda)$ is computed; and finally new message M' is parsed into λ -bit blocks $M'[1]\|M'[2]\|\cdots\|M'[m-m'+1] := M'$.

The new message M' is encrypted after $(m-1)$ th message (or ciphertext) block (using the values of r , s , u and v obtained after decryption of $C[m']$) as explained in the encryption algorithm $\hat{\Phi}.\mathcal{E}(\cdot)$.

Now, the updated key $K' := s$, the updated ciphertext $C' := C[1] \parallel C[2] \parallel \cdots \parallel C[m-1] \parallel C'[m'] \parallel C'[m'+1] \parallel \cdots \parallel C'[m-1] \parallel u \parallel v$, and updated tag $T' := r$.

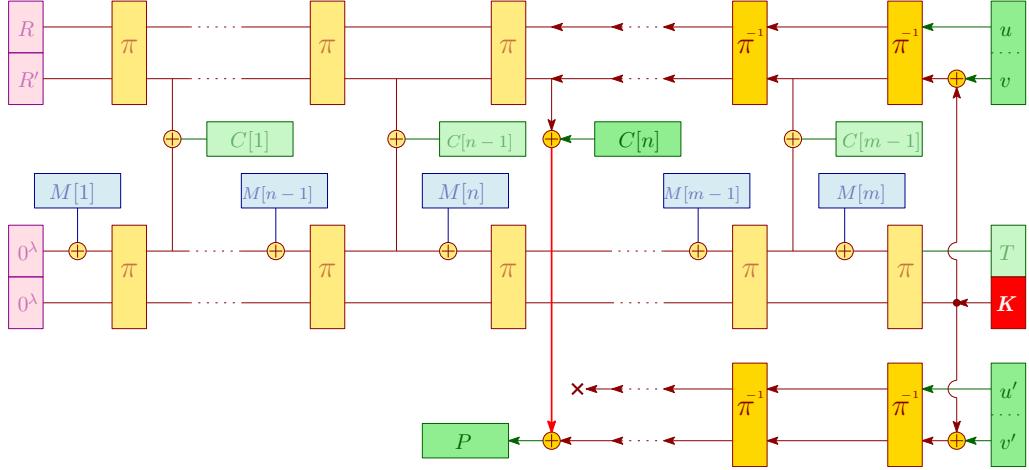


Figure 4.35: Pictorial description of the *PoW algorithm for prover* $\hat{\Phi} \cdot \mathcal{P}$.

$\hat{\Phi} \cdot \mathcal{P}(\text{params}^{(\hat{\Phi})}, Q = (n, u', v'), M, K, C, T)$

#Initialization.

$$\begin{aligned} m &:= (|C|/\lambda) - 1; & C[1] \parallel C[2] \parallel \cdots \parallel C[m-1] \parallel u \parallel v &:= C; \\ s &:= K; & v &:= v \oplus s; & v' &:= v' \oplus s; \end{aligned}$$

#Processing blocks.

$$\begin{aligned} \textbf{for } (j := m, m-1, \dots, n+1) \\ u \parallel v &:= \pi^{-1}(u \parallel v); & u' \parallel v' &:= \pi^{-1}(u' \parallel v'); \end{aligned}$$

#Computing final output.

$$\text{return } P := C[n] \oplus v \oplus v';$$

Figure 4.36: Algorithmic description of the *PoW algorithm for prover* $\hat{\Phi} \cdot \mathcal{P}$.

- The PPT *PoW algorithm for prover* $\hat{\Phi} \cdot \mathcal{P}(\cdot)$ takes as input parameter $\text{params}^{(\hat{\Phi})}$, challenge $Q = (n, u', v')$ as the block number n of ciphertext C and last two blocks u' and v' of the already stored ciphertext C' , message M , decryption key K and tag T , and outputs proof $P \in \{0, 1\}^\lambda$ as one block of the ciphertext. The pictorial description and pseudocode are given in Figures 4.35 and 4.36. Very briefly, the algorithm works as follows: first it parses C into λ -bit

blocks $C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C$; then it assigns value of K to s ; next it updates $v := v \oplus s$; and finally updates $v' := v' \oplus s$.

Now, to process u , v , u' and v' , the following operations are performed iteratively for $m - n$ times: first $u\|v := \pi^{-1}(u\|v)$ and then $u'\|v' := \pi^{-1}(u'\|v')$ are computed.

Now, the proof $P := C[n] \oplus v \oplus v'$.

$\hat{\Phi}.\mathcal{V}(\text{params}^{(\hat{\Phi})}, Q = (n, u', v'), C, T, P)$

Verifying ciphertext block.

If $(C[n] = P)$, then return TRUE;
Else return FALSE;

Figure 4.37: Algorithmic description of the *PoW algorithm for verifier* $\hat{\Phi}.\mathcal{V}$.

- The PPT *PoW algorithm for verifier* $\hat{\Phi}.\mathcal{V}(\cdot)$ takes as input parameter $\text{params}^{(\hat{\Phi})}$, challenge $Q = (n, u', v')$ as the block number n of the ciphertext and last two blocks u' and v' of the already stored ciphertext C , tag T and proof P , and outputs value $\text{val} \in \{\text{TRUE}, \text{FALSE}\}$. The algorithm returns **TRUE**, if $C[n] = P$, otherwise it returns **FALSE**. See Figure 4.37 for the pseudocode.

4.7.1.2 Security of $\hat{\Phi}$

Theorem 1. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\hat{\Phi}$ has been defined in Section 4.7.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\hat{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) \leq \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2(\ell+1)^2m^2}{2^\mu} + \frac{2(\ell+1)^2m^2}{2^\lambda}.$$

Here, \mathcal{S} is a message-source that outputs the message vector \mathbf{M} such that $\|\mathbf{M}\| \stackrel{\text{def}}{=} m = m(1^\lambda)$, with min-entropy $\mu = \mu_{\mathcal{S}}(1^\lambda)$ and $\ell = \max_{i \in \{1, 2, \dots, m\}} |\mathbf{M}_i|$. The **FMLE-PRV\$** game is defined in Figure 4.2.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{FMLE-PRV\$}_{\hat{\Phi}, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 1)$; our last game is $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$, which is identical to $\text{FMLE-PRV\$}_{\hat{\Phi}, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b = 0)$; and our intermediate games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$. Then, we

compute the advantages between the successive games. We wish to bound the adversarial advantage:

$$\begin{aligned}
Adv_{\hat{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{FMLE-PRV\$}_{\hat{\Phi}, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=1) = 1] \right. \\
&\quad \left. - \Pr[\text{FMLE-PRV\$}_{\hat{\Phi}, \mathcal{S}}^{\mathcal{A}}(1^\lambda, b=0) = 1] \right| \\
&= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right| \\
&= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right. \\
&\quad + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \\
&\quad \left. + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned}
Adv_{\hat{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

Therefore, $Adv_{\hat{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3$, (4.1)

$$\begin{aligned}
\text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\
\Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

In the following, we compute Δ_1 , Δ_2 and Δ_3 (see Figure 4.38 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\hat{\Phi}^{(1)}$ instead of $\hat{\Phi}$. In $\hat{\Phi}^{(1)}$ (as described in Figure 4.39), the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the message \mathbf{M} (where $\|\mathbf{M}\| = m$), we obtain:

$$\Delta_1 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (4.2)$$

Game \mathbf{G}_2 : This game is identical to **Game \mathbf{G}_1** , except that it uses $\hat{\Phi}^{(2)}$ instead of $\hat{\Phi}^{(1)}$. In $\hat{\Phi}^{(2)}$ (as described in Figure 4.39), the bad_1 and bad_2 flags are set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random function rf . From Figure 4.39, using

the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the message \mathbf{M} (where $\|\mathbf{M}\| = m$), we obtain:

$$\Delta_2 \leq \Pr[\text{bad}_1 \text{ is set in } \mathbf{G}_2] + \Pr[\text{bad}_2 \text{ is set in } \mathbf{G}_2]. \quad (4.3)$$

The event that the flag bad_1 is set in \mathbf{G}_2 happens when s matches with one of the previous lower λ -bit component of the 2λ -bit input to the random function rf , that is, s matches with v or s , previously generated.

The event that the flag bad_2 is set in \mathbf{G}_2 happens when v matches with one of the previous lower λ -bit component of the 2λ -bit input to the random function rf , that is, v matches with v or s , previously generated.

Calculation of $\Pr[\text{bad}_1 \text{ is set in } \mathbf{G}_2]$: Suppose that we are using the construction $\hat{\Phi}^{(2)}$, and we define the following events:

- A is the event that the flag bad_1 is set in \mathbf{G}_2 , that is, at least one of the s values collides with a previously generated s value (or a v value).
- $A_{i,j}$ is the event that all the s and v values (that are generated until the j th block of the i th string is processed) are all distinct.
- $B_{i,j}$ is the event that current value of s collides with a s or v value, that have been previously generated.

So, we calculate the probability of event A as follows:

$$\begin{aligned} \Pr[A] &\leq \Pr[B_{1,2}|A_{1,2}] + \Pr[B_{1,3}|A_{1,3}] + \cdots + \Pr[B_{m,\ell+1}|A_{m,\ell+1}] \\ &\leq \left(\frac{1}{2^\mu} + \frac{1}{2^\lambda}\right) + \left(\frac{2}{2^\mu} + \frac{2}{2^\lambda}\right) + \cdots + \left(\frac{m \times (\ell+1)}{2^\mu} + \frac{m \times (\ell+1)}{2^\lambda}\right) \\ &\leq \frac{(\ell+1)^2 m^2}{2^\mu} + \frac{(\ell+1)^2 m^2}{2^\lambda} \\ \Pr[\text{bad}_1 \text{ is set in } \mathbf{G}_2] &\leq \frac{(\ell+1)^2 m^2}{2^\mu} + \frac{(\ell+1)^2 m^2}{2^\lambda} \end{aligned} \quad (4.4)$$

Similarly, we can calculate:

$$\Pr[\text{bad}_2 \text{ is set in } \mathbf{G}_2] \leq \frac{(\ell+1)^2 m^2}{2^\mu} + \frac{(\ell+1)^2 m^2}{2^\lambda} \quad (4.5)$$

Using the Equations 4.3, 4.4 and 4.5, we obtain the following:

$$\Delta_2 \leq \frac{2(\ell+1)^2 m^2}{2^\mu} + \frac{2(\ell+1)^2 m^2}{2^\lambda} \quad (4.6)$$

In $\hat{\Phi}^{(2)}$ (as described in Figure 4.39), the flags bad_1 and bad_2 are set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random function rf . Since the events that the $\text{bad}_1 = 0$ and

$bad_2 = 0$ make sure that there is no collision among the lower λ -bit components of the 2λ -bit inputs to the random function rf , this implies that $C[1], C[2], \dots, C[m - 1]$ are uniformly distributed and independent. Since, $\hat{\Phi}^{(2)}. \mathcal{E}(\text{params}^{(\hat{\Phi}^{(2)})}, M)$ is uniformly distributed over $\{0, 1\}^{|M|+\lambda}$, the two games \mathbf{G}_2 and \mathbf{G}_L are indistinguishable. Hence, we obtain:

$$\Delta_3 = 0. \quad (4.7)$$

Using (4.1), (4.2), (4.6) and (4.7), we get:

$$\begin{aligned} Adv_{\hat{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 \\ &\leq \frac{2m(2m - 1)}{2^{2\lambda}} + \frac{2(\ell + 1)^2 m^2}{2^\mu} + \frac{2(\ell + 1)^2 m^2}{2^\lambda} + 0 \\ &\leq \frac{2m(2m - 1)}{2^{2\lambda}} + \frac{2(\ell + 1)^2 m^2}{2^\mu} + \frac{2(\ell + 1)^2 m^2}{2^\lambda}. \end{aligned} \quad (4.8)$$

■

Theorem 2. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\hat{\Phi}$ has been defined in Section 4.7.1. For all poly-time FMLE-TC adversaries \mathcal{A} against $\hat{\Phi}$, there exists a poly-time HF-CR adversary \mathcal{B} against Sponge, such that:

$$Adv_{\hat{\Phi}, \mathcal{A}}^{\text{FMLE-TC}}(1^\lambda) \leq Adv_{\text{Sponge}, \mathcal{B}}^{\text{HF-CR}}(1^\lambda).$$

Here, the FMLE-TC game is defined in Figure 4.3, and the HF-CR game is defined in Figure 2.7.

Proof. We poly-reduce any FMLE-TC adversary \mathcal{A} against $\hat{\Phi}$ into a HF-CR adversary \mathcal{B} against Sponge. The explicit reduction is shown in Figure 4.40 and the proof readily follows from it. ■

Theorem 3. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\hat{\Phi}$ has been defined in Section 4.7.1. Then for all adversaries \mathcal{A} ,

$$Adv_{\hat{\Phi}, \mathcal{A}}^{\text{FMLE-UNC}}(1^\lambda) = 0$$

Here, the FMLE-UNC game is defined in Figure 4.4.

Proof. The FMLE-UNC game requires the proof P^* generated by adversary \mathcal{A} to be different from the actual proof P , and then P^* needs to be correctly verified by the function $\hat{\Phi}. \mathcal{V}(\cdot)$. However, this is not possible in $\hat{\Phi}$, because for each challenge Q , the corresponding unique proof P is already stored in the cloud (as a ciphertext-block) and some other value of $P^* \neq P$ will never be accepted by function $\hat{\Phi}. \mathcal{V}(\cdot)$. ■

Game $G_S(\mathcal{A}, 1^\lambda)$
 $(params^{(\hat{\Phi})}, \mathcal{K}^{(\hat{\Phi})}, \mathcal{M}^{(\hat{\Phi})}, \mathcal{C}^{(\hat{\Phi})}, \mathcal{T}^{(\hat{\Phi})}) := \hat{\Phi}.\text{Setup}(1^\lambda);$
 $(\mathbf{M}, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$
for $(i := 1, 2, \dots, m(1^\lambda))$
 $(\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \hat{\Phi}.\mathcal{E}(params^{(\hat{\Phi})}, \mathbf{M}^{(i)});$
 $b' := \mathcal{A}(1^\lambda, \mathbf{C}_1, \mathbf{T}_1, Z);$

 return b' ;
Game $G_i(\mathcal{A}, 1^\lambda)$
 $(params^{(\hat{\Phi}^{(i)})}, \mathcal{K}^{(\hat{\Phi}^{(i)})}, \mathcal{M}^{(\hat{\Phi}^{(i)})}, \mathcal{C}^{(\hat{\Phi}^{(i)})}, \mathcal{T}^{(\hat{\Phi}^{(i)})}) := \hat{\Phi}^{(i)}.\text{Setup}(1^\lambda);$
 $(\mathbf{M}, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$
for $(i := 1, 2, \dots, m(1^\lambda))$
 $(\mathbf{K}_1^{(i)}, \mathbf{C}_1^{(i)}, \mathbf{T}_1^{(i)}) := \hat{\Phi}^{(i)}.\mathcal{E}(params^{(\hat{\Phi}^{(i)})}, \mathbf{M}^{(i)});$
 $b' := \mathcal{A}(1^\lambda, \mathbf{C}_1, \mathbf{T}_1, Z);$

 return b' ;
Game $G_L(\mathcal{A}, 1^\lambda)$
 $(params^{(\hat{\Phi})}, \mathcal{K}^{(\hat{\Phi})}, \mathcal{M}^{(\hat{\Phi})}, \mathcal{C}^{(\hat{\Phi})}, \mathcal{T}^{(\hat{\Phi})}) := \hat{\Phi}.\text{Setup}(1^\lambda);$
 $(\mathbf{M}, Z) \xleftarrow{\$} \mathcal{S}(1^\lambda);$
for $(i := 1, 2, \dots, m(1^\lambda))$
 $\mathbf{C}_0^{(i)} \xleftarrow{\$} \{0, 1\}^{|\mathbf{M}^{(i)}|+\lambda}; \quad \mathbf{T}_0^{(i)} \xleftarrow{\$} \{0, 1\}^\lambda;$
 $b' := \mathcal{A}(1^\lambda, \mathbf{C}_0, \mathbf{T}_0, Z);$

 return b' ;

Here, in **Game G_i** , $i \in [2]$.

Figure 4.38: Games used in the proof of Theorem 1.

Theorem 4. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\hat{\Phi}$ has been defined in Section 4.7.1. Then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\hat{\Phi}, \mathcal{A}}^{\text{FMLE-CXH}}(1^\lambda) = 0.$$

Here, the FMLE-CXH game is defined in Figure 4.5.

Proof. The function $\hat{\Phi}.\mathcal{E}(params^{(\hat{\Phi})}, M)$ can be rewritten as $\hat{\Phi}.\mathcal{E}_r(params^{(\hat{\Phi})}, M)$ because internally it generates and uses a random number r . So, the function $\hat{\Phi}.\mathcal{U}(params^{(\hat{\Phi})}, i_1, i_\rho, M_0[i_1, i_1 + 1, \dots, i_\rho], \hat{\Phi}.\mathcal{E}_{r_2}(params^{(\hat{\Phi})},$

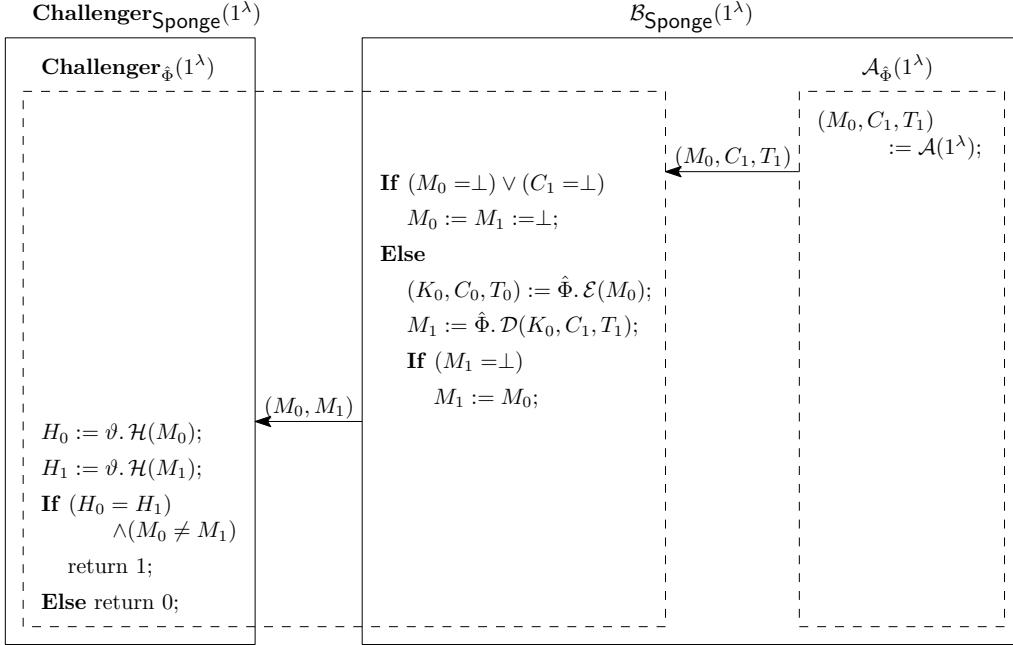
$\hat{\Phi}^{(1)}. \mathcal{E}(params^{(\hat{\Phi}^{(1)})}, M)$	$\hat{\Phi}^{(2)}. \mathcal{E}(params^{(\hat{\Phi}^{(2)})}, M)$
<i>#Initialization.</i> $m := M /\lambda; \quad M[1]\ M[2]\ \cdots\ M[m] := M;$ $r := 0^\lambda; \quad s := 0^\lambda; \quad u \xleftarrow{\$} \{0, 1\}^\lambda; \quad v \xleftarrow{\$} \{0, 1\}^\lambda; \quad U := \emptyset;$	
<i>#Processing message blocks.</i> for ($j := 1, 2, \dots, m$) If ($s \in U$), then $bad_1 := 1; \quad r\ s \xleftarrow{\$} \{0, 1\}^{2\lambda};$ $U := U \cup \{s\};$ If ($v \in U$), then $bad_2 := 1; \quad u\ v \xleftarrow{\$} \{0, 1\}^{2\lambda};$ $U := U \cup \{v\};$ $r\ s := \text{rf}((r \oplus M[j])\ s); \quad u\ v := \text{rf}(u\ v);$ If ($j \neq m$), then $C[j] := v \oplus r;$ $v := v \oplus s;$	
<i>#Computing final outputs.</i> $K := s; \quad C := C[1]\ C[2]\ \cdots\ C[m-1]\ u\ v; \quad T := r;$ return (K, C, T);	
<u>proc. $\text{rf}(x)$</u> If ($x \notin \text{Dom}(D_{\text{rf}})$), then $D_{\text{rf}}[x] \xleftarrow{\$} \{0, 1\}^{2\lambda};$ return $D_{\text{rf}}[x];$	

Figure 4.39: Algorithmic description of $\hat{\Phi}^{(1)}$ and $\hat{\Phi}^{(2)}$.

$M_1), 0)$ is equivalent to $\hat{\Phi}. \mathcal{E}_{r_2}(params^{(\hat{\Phi})}, M_0)$ (from the algorithms described in Figures 4.30 and 4.34). Since, $\hat{\Phi}. \mathcal{E}_{r_1}(params^{(\hat{\Phi})}, M_0)$ and $\hat{\Phi}. \mathcal{E}_{r_2}(params^{(\hat{\Phi})}, M_0)$ differ in the choice of random numbers only. Therefore, $\hat{\Phi}. \mathcal{E}_{r_1}(params^{(\hat{\Phi})}, M_0)$ is indistinguishable from $\hat{\Phi}. \mathcal{U}(params^{(\hat{\Phi})}, i_1, i_\rho, M_0[i_1, i_1 + 1, \dots, i_\rho], \hat{\Phi}. \mathcal{E}_{r_2}(params^{(\hat{\Phi})}, M_1), 0)$. ■

4.7.2 Construction $\tilde{\Phi}$

This construction is motivated by the design of **FP** hash mode of operation (described in Section 2.2.3.2).



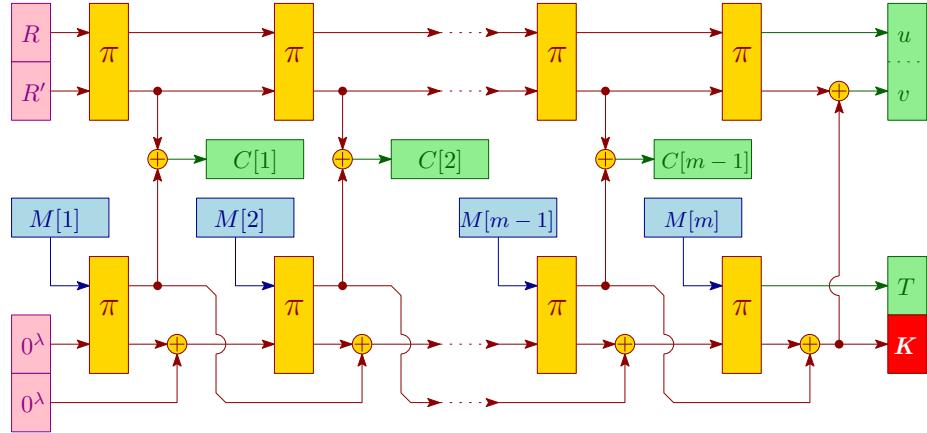
Reducing an **FMLE-TC** adversary \mathcal{A} against $\hat{\Phi}$ into an **HF-CR** adversary \mathcal{B} against **Sponge** (see the **FMLE-TC** game for $\hat{\Phi}$ in Figure 4.3 and **HF-CR** game for **Sponge** in Figure 2.7).

Figure 4.40: Reduction used in Theorem 2.

4.7.2.1 Description of $\tilde{\Phi}$

The pictorial description and pseudocode for the 5-tuple of algorithms in $\tilde{\Phi} = (\tilde{\Phi}.E, \tilde{\Phi}.D, \tilde{\Phi}.U, \tilde{\Phi}.P, \tilde{\Phi}.V)$ over the *setup* algorithm $\tilde{\Phi}.\text{Setup}$ are given in Figures 4.41, 4.42, 4.43, 4.44, 4.45, 4.46, 4.47, 4.48 and 4.49; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\tilde{\Phi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\tilde{\Phi})}$, key-space $\mathcal{K}^{(\tilde{\Phi})} := \{0, 1\}^\lambda$, message-space $\mathcal{M}^{(\tilde{\Phi})} := \bigcup^{i \geq 1} \{0, 1\}^{i\lambda}$, ciphertext-space $\mathcal{C}^{(\tilde{\Phi})} := \bigcup^{i \geq 2} \{0, 1\}^{i\lambda}$, and tag-space $\mathcal{T}^{(\tilde{\Phi})} := \{0, 1\}^\lambda$.
- The *encryption* algorithm $\tilde{\Phi}.E(\cdot)$ is randomized that takes as input parameter $\text{params}^{(\tilde{\Phi})}$ and message M , and outputs key K , ciphertext C , and tag T . The pictorial description and pseudocode are given in Figures 4.41 and 4.42. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then

Figure 4.41: Pictorial description of the *encryption* function $\tilde{\Phi} \cdot \mathcal{E}$.

$\tilde{\Phi} \cdot \mathcal{E}(\text{params}^{(\tilde{\Phi})}, M)$

#Initialization.

$$m := |M|/\lambda; \quad M[1] \| M[2] \| \cdots \| M[m] := M; \\ s := 0^\lambda; \quad t := 0^\lambda; \quad u \xleftarrow{\$} \{0, 1\}^\lambda; \quad v \xleftarrow{\$} \{0, 1\}^\lambda;$$

#Processing message blocks.

```
for (j := 1, 2, ..., m)
    r \| s :=  $\pi(M[j] \| s)$ ;  $u \| v := \pi(u \| v)$ ;  $s := s \oplus t$ ;  $t := r$ ;
    If ( $j \neq m$ ), then  $C[j] := v \oplus r$ ;
     $v := v \oplus s$ ;
```

#Computing final outputs.

```
 $K := s$ ;  $C := C[1] \| C[2] \| \cdots \| C[m - 1] \| u \| v$ ;  $T := r$ ;
return ( $K, C, T$ );
```

Figure 4.42: Algorithmic description of the *encryption* function $\tilde{\Phi} \cdot \mathcal{E}$.

it initializes both the strings s and t with 0^λ ; and finally it randomly chooses two λ -bit strings u and v . The encryption of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$. First $r \| s := \pi(M[j] \| s)$ is computed; next $u \| v := \pi(u \| v)$ is computed; then $s := s \oplus t$ is updated; after that t is assigned the value of r ; and finally ciphertext block $C[j] := v \oplus r$ is computed.

For the encryption of last block $M[m]$, we perform the following op-

erations: first $r\|s := \pi(M[m]\|s)$ is computed; then $u\|v := \pi(u\|v)$ is computed; next $s := s \oplus t$ is updated; after that t is assigned value of r ; and finally $v := v \oplus s$ is updated.

Now, the key $K := s$, the ciphertext $C := C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v$, and the tag $T := r$.

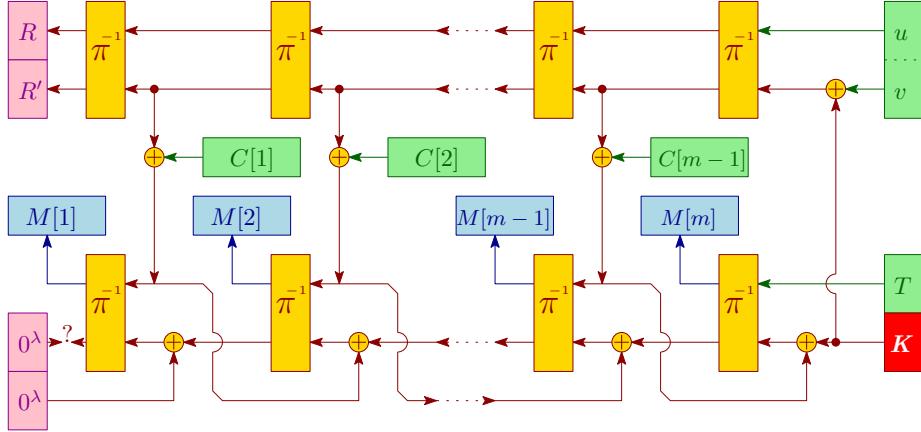


Figure 4.43: Pictorial description of the *decryption* function $\tilde{\Phi}. \mathcal{D}$.

$\tilde{\Phi}. \mathcal{D}(params^{(\tilde{\Phi})}, K, C, T)$

#Validating the length.
If ($|C| < 2\lambda$) OR ($|C| \bmod \lambda \neq 0$), **then return Error**;

#Initialization.
 $m := (|C|/\lambda) - 1$; $C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C$;
 $s := K$; $t := T$; $v := v \oplus s$;

#Processing ciphertext blocks.
for ($j := m, m-1, \dots, 1$)
 $r := t$; $u\|v := \pi^{-1}(u\|v)$;
 If ($j = 1$), **then** $t := 0^\lambda$; **Else** $t := C[j-1] \oplus v$;
 $s := s \oplus t$; $M[j]\|s := \pi^{-1}(r\|s)$;

#Computing final outputs.
 $M := M[1]\|M[2]\|\cdots\|M[m]$;
If ($s = 0^\lambda$), **then return** M ; **Else return** \perp ;

Figure 4.44: Algorithmic description of the *decryption* function $\tilde{\Phi}. \mathcal{D}$.

- The *decryption* algorithm $\tilde{\Phi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\tilde{\Phi})}$, key K , ciphertext C and tag T , and outputs message M or an *invalid* symbol \perp . The pictorial description and pseudocode are given in Figures 4.43 and 4.44. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \| C[2] \| \cdots \| C[m - 1] \| u \| v := C$; then it initializes the value of s and t with K and T ; and it finally updates $v := v \oplus s$. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

Now, we give the details of decryption of ciphertext block $C[j]$, for all $j \in [m]$. First r is assigned the value of t ; then $u \| v := \pi^{-1}(u \| v)$ is computed; if value of $j = 1$, t is assigned 0^λ , otherwise, $t := C[j-1] \oplus v$ is computed; next $s := s \oplus t$ is updated; and finally $M[j] \| s := \pi^{-1}(r \| s)$ is computed.

Now, the message $M := M[1] \| M[2] \| \cdots \| M[m]$. If $s = 0^\lambda$ after the execution of the last iteration, then M is returned, otherwise the *invalid* symbol is returned.

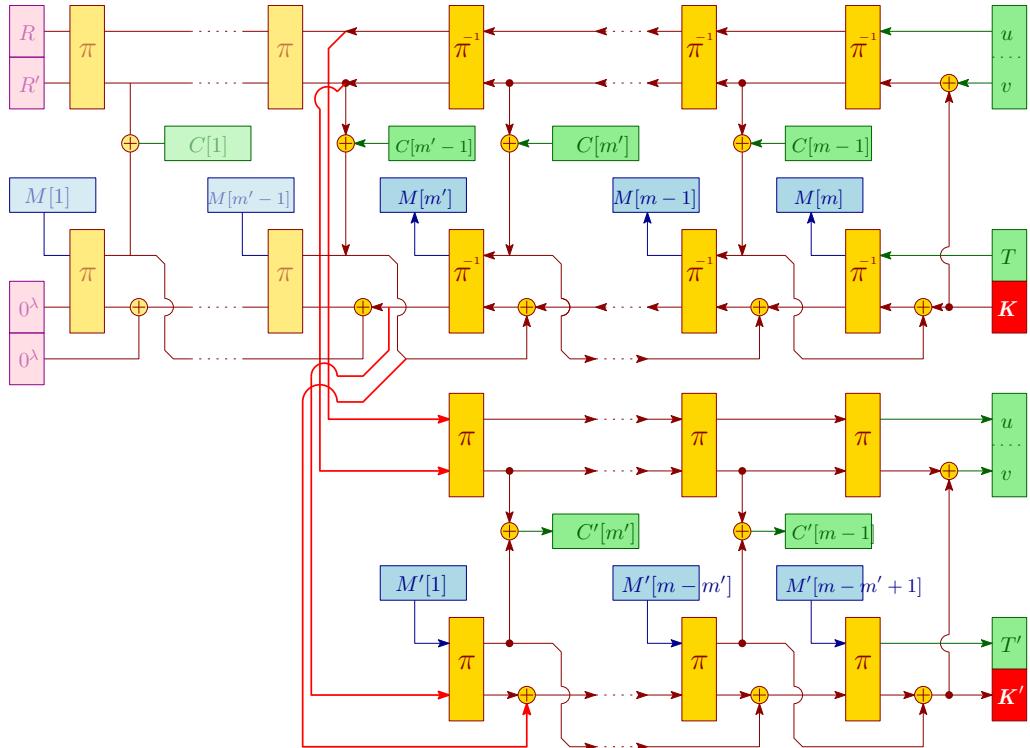


Figure 4.45: Pictorial description of the *update* function $\tilde{\Phi} \cdot \mathcal{U}$.

```

 $\tilde{\Phi} \cdot \mathcal{U}(params^{(\tilde{\Phi})}, i_{st}, i_{end}, M_{\text{new}}, K, C, T, app)$ 

#Initialization.
 $m := (|C|/\lambda) - 1; \quad C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C;$ 
 $m' := \lceil i_{st}/\lambda \rceil; \quad s := K; \quad t := T; \quad v := v \oplus s;$ 

#Processing ciphertext blocks.
for ( $j := m, m-1, \dots, m'$ )
   $r := t; \quad u\|v := \pi^{-1}(u\|v);$ 
  If ( $j = 1$ ), then  $t := 0^\lambda$ ; Else  $t := C[j-1] \oplus v;$ 
   $s := s \oplus t; \quad M[j]\|s := \pi^{-1}(r\|s);$ 

#Computing new message.
 $j := ((i_{st} - 1) \bmod \lambda) + 1;$ 
 $M' := M[m'][1]\|M[m'][2]\|\cdots\|M[m'][j-1]\|M_{\text{new}};$ 
If ( $app = 0$ )
   $n := \lceil (i_{end} + 1)/\lambda \rceil; \quad j := (i_{end} \bmod \lambda) + 1;$ 
   $M' := M'\|M[n][j]\|M[n][j+1]\|\cdots\|M[n][\lambda]\|M[n+1]\|$ 
   $M[n+2]\|\cdots\|M[m];$ 
Else  $m := (m - 1) + (|M'|/\lambda);$ 
 $M'[1]\|M'[2]\|\cdots\|M'[m-m'+1] := M';$ 

#Processing message blocks.
for ( $j := m', m'+1, \dots, m$ )
   $r\|s := \pi(M'[j-m'+1]\|s); \quad u\|v := \pi(u\|v);$ 
  If ( $j \neq m$ ), then  $C'[j] := v \oplus r;$ 
   $s := s \oplus t; \quad t := r;$ 
   $v := v \oplus s;$ 

#Computing final outputs.
 $K' := s; \quad T' := r;$ 
 $C' := C[1]\|C[2]\|\cdots\|C[m'-1]\|C'[m']\|\cdots\|C'[m-1]\|u\|v;$ 
return ( $K', C', T'$ );

```

Figure 4.46: Algorithmic description of the $\tilde{\Phi} \cdot \mathcal{U}$.

- The $update$ algorithm $\tilde{\Phi} \cdot \mathcal{U}(\cdot)$ is a PPT algorithm that takes as input parameter $params^{(\tilde{\Phi})}$, index of starting and ending bits i_{st} and i_{end} , new message bits $M_{\text{new}} \in \{0, 1\}^{i_{end}-i_{st}+1}$, old decryption key K , old ciphertext C , old tag T and bit app , and outputs updated key K' ,

updated ciphertext C' and updated tag T' . The pictorial description and pseudocode are given in Figures 4.45 and 4.46. The high level idea is: we start decrypting ciphertext blocks from the end until we reach the block containing i_{st} th bit of the message, and we store the values of s , t , u and v at this point; then we compute new message blocks starting from the first bit of the block containing the i_{st} th bit; and finally, using the values of s , t , u and v (stored in the previous step), we begin the encryption of the new message blocks.

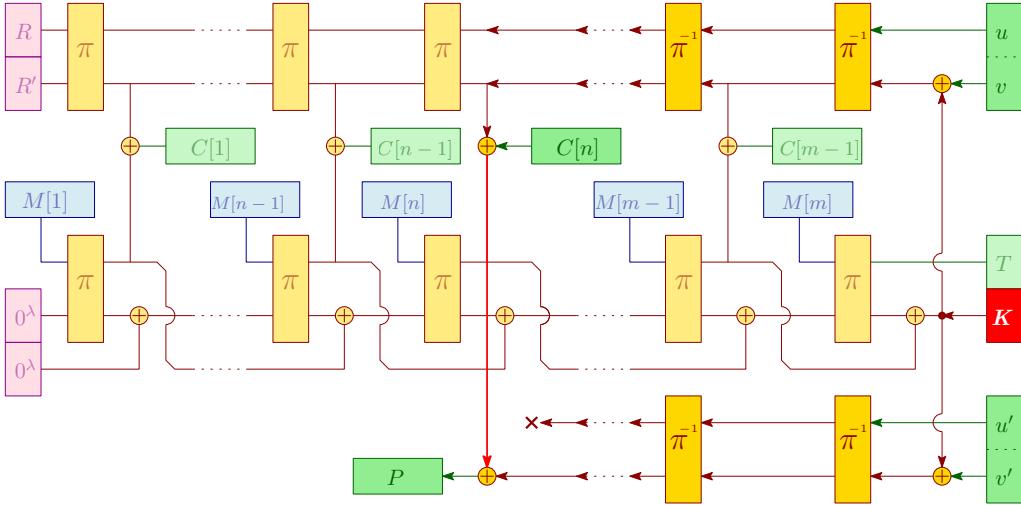
Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \parallel C[2] \parallel \dots \parallel C[m-1] \parallel u \parallel v := C$; then it computes $m' := \lceil i_{st}/\lambda \rceil$; next it initializes the values of s and t with K and T ; and updates $v := v \oplus s$. Now, ciphertext blocks $C[m], C[m-1], \dots, C[m']$ are decrypted, as explained in the decryption algorithm $\tilde{\Phi} \cdot \mathcal{D}(\cdot)$. The value of s , t , u and v obtained after decryption of $C[m']$, are preserved for future use.

Now, to compute new message, the following operations are performed: first the position of i_{st} th bit of the message in block $M[m']$ is computed $j := ((i_{st}-1) \bmod \lambda) + 1$; then new message $M' := M[m'][1] \parallel M[m'][2] \parallel \dots \parallel M[m'][j-1] \parallel M_{\text{new}}$ is computed; if $app = 0$, that is, length of message is not to be changed, then at first, the block containing $(i_{end}+1)$ th bit is computed $n := \lceil (i_{end}+1)/\lambda \rceil$, then position of $(i_{end}+1)$ th bit of the message in block $M[n]$ is computed $j := (i_{end} \bmod \lambda) + 1$, and finally, new message $M' := M' \parallel M[n][j] \parallel M[n][j+1] \parallel \dots \parallel M[n][\lambda] \parallel M[n+1] \parallel M[n+2] \parallel \dots \parallel M[m]$ is updated; if $app = 1$, that is, length of message is to be changed, then new length $m := (m-1) + (|M'|/\lambda)$ is computed; and finally new message M' is parsed into λ -bit blocks $M'[1] \parallel M'[2] \parallel \dots \parallel M'[m-m'+1] := M'$.

The new message M' is encrypted after $(m-1)$ th message (or ciphertext) block (using the values of s , t , u and v obtained after decryption of $C[m']$) as explained in the encryption algorithm $\tilde{\Phi} \cdot \mathcal{E}(\cdot)$.

Now, the updated key $K' := s$, the updated ciphertext $C' := C[1] \parallel C[2] \parallel \dots \parallel C[m'-1] \parallel C'[m'] \parallel C'[m'+1] \parallel \dots \parallel C'[m-1] \parallel u \parallel v$, and updated tag $T' := r$.

- The PPT *PoW algorithm for prover* $\tilde{\Phi} \cdot \mathcal{P}(\cdot)$ takes as input parameter $params^{(\tilde{\Phi})}$, challenge $Q = (n, u', v')$ as the block number n of ciphertext C and last two blocks u' and v' of the already stored ciphertext C' , message M , decryption key K and tag T , and outputs

Figure 4.47: Pictorial description of the *PoW algorithm for prover $\tilde{\Phi}. \mathcal{P}$* .

$\tilde{\Phi}. \mathcal{P}(\text{params}^{(\tilde{\Phi})}, Q = (n, u', v'), M, K, C, T)$

#Initialization.

$$\begin{aligned} m &:= (|C|/\lambda) - 1; & C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v &:= C; \\ s &:= K; & v &:= v \oplus s; & v' &:= v' \oplus s; \end{aligned}$$

#Processing blocks.

$$\begin{aligned} \textbf{for } (j := m, m-1, \dots, n+1) \\ u\|v &:= \pi^{-1}(u\|v); & u'\|v' &:= \pi^{-1}(u'\|v'); \end{aligned}$$

#Computing final output.

$$\text{return } P := C[n] \oplus v \oplus v';$$

Figure 4.48: Algorithmic description of the *PoW algorithm for prover $\tilde{\Phi}. \mathcal{P}$* .

proof $P \in \{0, 1\}^\lambda$ as one block of the ciphertext. The pictorial description and pseudocode are given in Figures 4.47 and 4.48. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1]\|C[2]\|\cdots\|C[m-1]\|u\|v := C$; then it assigns value of K to s ; next it updates $v := v \oplus s$; and finally updates $v' := v' \oplus s$.

Now, to process u, v, u' and v' , the following operations are performed iteratively for $m-n$ times: first $u\|v := \pi^{-1}(u\|v)$ and then $u'\|v' := \pi^{-1}(u'\|v')$ are computed.

Now, the proof $P := C[n] \oplus v \oplus v'$.

$\tilde{\Phi} \cdot \mathcal{V}(params^{(\tilde{\Phi})}, Q = (n, u', v'), C, T, P)$

Verifying ciphertext block.

If $(C[n] = P)$, then return TRUE;

Else return FALSE;

Figure 4.49: Algorithmic description of the *PoW algorithm for verifier* $\tilde{\Phi} \cdot \mathcal{V}$.

- The PPT *PoW algorithm for verifier* $\tilde{\Phi} \cdot \mathcal{V}(\cdot)$ takes as input parameter $params^{(\tilde{\Phi})}$, challenge $Q = (n, u', v')$ as the block number n of the ciphertext and last two blocks u' and v' of the already stored ciphertext C , tag T and proof P , and outputs value $val \in \{\text{TRUE}, \text{FALSE}\}$. The algorithm returns TRUE, if $C[n] = P$, otherwise it returns FALSE. See Figure 4.49 for the pseudocode.

4.7.2.2 Security of $\tilde{\Phi}$

Theorem 5. Let $\lambda \in \mathbb{N}$ be the security parameter. The *FMLE scheme* $\tilde{\Phi}$ has been defined in Section 4.7.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\tilde{\Phi}, \mathcal{S}, \mathcal{A}}^{\text{FMLE-PRV\$}}(1^\lambda) \leq \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2(\ell+1)^2 m^2}{2^\mu} + \frac{2(\ell+1)^2 m^2}{2^\lambda}.$$

Here, \mathcal{S} is a message-source that outputs the message vector \mathbf{M} such that $\|\mathbf{M}\| \stackrel{\text{def}}{=} m = m(1^\lambda)$, with min-entropy $\mu = \mu_{\mathcal{S}}(1^\lambda)$ and $\ell = \max_{i \in \{1, 2, \dots, m\}} |\mathbf{M}_i|$. The *FMLE-PRV\$* game is defined in Figure 4.2.

Proof. The proof is similar to that of Theorem 1. ■

Theorem 6. Let $\lambda \in \mathbb{N}$ be the security parameter. The *FMLE scheme* $\tilde{\Phi}$ has been defined in Section 4.7.2. For all poly-time *FMLE-TC* adversaries \mathcal{A} against $\tilde{\Phi}$, there exists a poly-time *HF-CR* adversary \mathcal{B} against *FP*, such that:

$$Adv_{\tilde{\Phi}, \mathcal{A}}^{\text{FMLE-TC}}(1^\lambda) \leq Adv_{\text{FP}, \mathcal{B}}^{\text{HF-CR}}(1^\lambda).$$

Here, the *FMLE-TC* game is defined in Figure 4.3, and the *HF-CR* game is defined in Figure 2.7.

Proof. The proof is similar to that of Theorem 2. ■

Theorem 7. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\tilde{\Phi}$ has been defined in Section 4.7.2. Then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Phi}, \mathcal{A}}^{\text{FMLE-UNC}}(1^\lambda) = 0$$

Here, the FMLE-UNC game is defined in Figure 4.4.

Proof. The proof is similar to that of Theorem 3. \blacksquare

Theorem 8. Let $\lambda \in \mathbb{N}$ be the security parameter. The FMLE scheme $\tilde{\Phi}$ has been defined in Section 4.7.2. Then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Phi}, \mathcal{A}}^{\text{FMLE-CXH}}(1^\lambda) = 0.$$

Here, the FMLE-CXH game is defined in Figure 4.5.

Proof. The proof is similar to that of Theorem 4. \blacksquare

4.7.3 Resistance of $\hat{\Phi}$ and $\tilde{\Phi}$ against Dictionary Attack

Since, $\hat{\Phi}$ and $\tilde{\Phi}$ have randomized encryption algorithms (where the random numbers are generated in the active phase of the execution), they are not vulnerable to dictionary attacks (see Section 3.4.6 for a definition of dictionary attack).

4.8 Comparison of FMLE schemes

In this section, we compare various FMLE schemes: F-CE (Section 4.5.1), F-HCE2 (Section 4.5.2), F-RCE (Section 4.5.3), F-UMLE (Section 4.6), $\hat{\Phi}$ (Section 4.7.1) and $\tilde{\Phi}$ (Section 4.7.2).

Here, we assume that the message-length $|M|$ and the value of security parameter λ are identical for all the schemes. In our comparison tables: B denotes block-size for *BL-MLE* scheme used in *F-UMLE* scheme; M' is the shortest suffix of M containing all modified bits; k is the index of the first bit in challenge in *PoW* protocol; $c_{\vartheta, H_{\lambda, |M|}}$, $c_{\vartheta, H_{\lambda, B}}$ and $c_{\vartheta, H_{\lambda, \lambda}}$ denote the costs of hash operation to produce λ -bit message-digest on inputs having lengths $|M|$, B and λ respectively; $c_{\rho, E_{|M|}}$ (or $c_{\rho, D_{|M|}}$) denotes the cost of encrypting (or decrypting) a $|M|$ -bit message (or ciphertext) in a symmetric encryption; c_{ρ, E_B} (or c_{ρ, D_B}) denotes the cost of encrypting (or decrypting) a B -bit message (or ciphertext) in a *BL-MLE* scheme employing

Constructions \rightarrow	F-CE	F-HCE2	F-RCE	F-UMLE	$\hat{\Phi}$ (or $\tilde{\Phi}$)
Parameters \downarrow	[DAB ⁺ 02]	[BKR13b]	[BKR13b]	[ZC17]	Section 4.7
ENCRYPTION					
• Key Gen.	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$(M /B + \log_{B/\lambda} M) \cdot C_{\vartheta, \mathcal{H}_{\lambda, B}}$	0
• Tag Gen.	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, \lambda}}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }} + (M /B + \log_{B/\lambda} M) \cdot C_{\vartheta, \mathcal{H}_{\lambda, B}}$	0
• Ciphertext Comp.					
No. of Passes	$C_{\rho, \mathcal{E}_{ M }}$	$C_{\rho, \mathcal{E}_{ M }}$	$C_{\rho, \mathcal{E}_{ M }}$	$(M /B + \log_{B/\lambda} M) \cdot C_{\rho, \mathcal{E}_B}$	$2 \cdot M /\lambda \cdot C_{\pi_{2\lambda}}$
DECRYPTION					
• Plaintext Comp.	$C_{\rho, \mathcal{D}_{ M }}$	$C_{\rho, \mathcal{D}_{ M }}$	$C_{\rho, \mathcal{D}_{ M }}$	$(M /B + \log_{B/\lambda} M) \cdot C_{\rho, \mathcal{D}_B}$	$2 \cdot M /\lambda \cdot C_{\pi_{2\lambda}^{-1}}$
• Tag Verif.	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }} + C_{\vartheta, \mathcal{H}_{\lambda, \lambda}}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }} + C_{\vartheta, \mathcal{H}_{\lambda, \lambda}}$	$(M /B + \log_{B/\lambda} M) \cdot C_{\vartheta, \mathcal{H}_{\lambda, B}}$	0
No. of Passes	3	2	1	$1 + \log_{B/\lambda} M $	1
UPDATE					
• Decryption	$C_{\rho, \mathcal{D}_{ M }}$	$C_{\rho, \mathcal{D}_{ M }}$	$C_{\rho, \mathcal{D}_{ M }}$	$\log_{B/\lambda} M \cdot C_{\rho, \mathcal{D}_B}$	$2 \cdot M' /\lambda \cdot C_{\pi_{2\lambda}^{-1}}$
• Encryption	$C_{\rho, \mathcal{E}_{ M }}$	$C_{\rho, \mathcal{E}_{ M }}$	$C_{\rho, \mathcal{E}_{ M }}$	$\log_{B/\lambda} M \cdot C_{\rho, \mathcal{E}_B}$	$2 \cdot M' /\lambda \cdot C_{\pi_{2\lambda}}$
• Key Gen.	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$\log_{B/\lambda} M \cdot C_{\vartheta, \mathcal{H}_{\lambda, B}}$	0
• Tag Gen.	$C_{\vartheta, \mathcal{H}_{\lambda, M }}$	$C_{\vartheta, \mathcal{H}_{\lambda, \lambda}}$	$C_{\vartheta, \mathcal{H}_{\lambda, \lambda}}$	$\log_{B/\lambda} M \cdot C_{\vartheta, \mathcal{H}_{\lambda, B}} + C_{\vartheta, \mathcal{H}_{\lambda, M }}$	0
No. of Passes	4	3	1	$1 + \log_{B/\lambda} M / M $	$2 \cdot M' / M $
PoW FOR PROVER					
• Proof Gen.	0	0	$C_{\rho, \mathcal{D}_{ M }} + C_{\rho, \mathcal{E}_{ M }}$	0	$2 \cdot M ^{-k}/\lambda \cdot C_{\pi_{2\lambda}^{-1}}$
No. of Passes	0	0	1	0	$1 - k/ M $

Table 4.1: Comparison of running time complexities of *FMLE* schemes.

Constructions → Parameters ↓	F-CE [DAB ⁺ 02]	F-HCE2 [BKR13b]	F-RCE [BKR13b]	F-UMLE [ZC17]	$\hat{\Phi}$ (or $\tilde{\Phi}$) Section 4.7
STORAGE REQUIREMENT					
• Key	λ	λ	λ	λ	λ
• Ciphertext	$ M $	$ M $	$ M + \lambda$	$ M + \log_{B/\lambda} M \cdot B$	$ M + \lambda$
• Tag	λ	λ	λ	$(M /B + \log_{B/\lambda} M)\lambda$	λ
SECURITY PROPERTIES					
• FMLE-PRV	✓	✓	✓	✓	✓
• FMLE-PRV\$	✓	✓	✓	✗	✓
• FMLE-TC	✓	✓	✓	✓	✓
• FMLE-STC	✓	✗	✗	✗	✗
• FMLE-UNC	✓	✓	✓	✓	✓
• FMLE-CXH	✓	✓	✓	✓	✓
• Dictionary Attack	✗	✗	✓	✗	✓

Table 4.2: Comparison of storage requirement and security properties of *FMLE* schemes.

symmetric encryption; and $c_{\pi_{2\lambda}}$ (or $c_{\pi_{2\lambda}^{-1}}$) denotes the cost of invocation of 2λ -bit permutation (or 2λ -bit inverse permutation).

In Table 4.1, we compare the running-times of all the algorithms of the *FMLE* constructions. In Table 4.2, we compare the storage requirements and security properties.

4.9 Conclusion and Future Work

In this chapter, we proposed a new cryptographic primitive *FMLE* and its two efficient constructions, denoted $\hat{\Phi}$ and $\tilde{\Phi}$; these constructions possess the randomization property of *MLE* construction *RCE*, and the efficient update property of *UMLE*. We showed that these constructions outperform all the existing ones, both with respect to time and memory. The high performance is attributed to a unique property named *reverse decryption* of the *APE* authenticated encryption and the *FP* hash mode of operation, on which these new constructions are based. The only disadvantage is, perhaps, that these constructions are not *FMLE-STC* secure. We leave as an open problem construction of an *FMLE-STC* secure efficient *FMLE*.

CHAPTER 5

Key Assignment Scheme with Authenticated Encryption

5.1 Introduction

We have given an elaborate discussion of the challenges in *hierarchical access control* and the cryptographic primitive *KAS* in Sections 3.5 and 3.6. In this chapter, we introduce a new cryptographic primitive *key assignment scheme with authenticated encryption (KAS-AE)*, which can be regarded as a *KAS* endowed with an additional *authenticated encryption*.

MOTIVATION FOR STUDYING KAS-AE. In all the existing applications, *hierarchical access control* with *encryption* is achieved by following the same design paradigm: first execute *KAS* for the generation of keys; and then encrypt the messages using *SE*. Since its inception, the *HAC* protocol has been implemented using efficient *KAS* constructions. To the best of our knowledge, no attempt has been made so far to build efficient *HAC* protocol by combining *KAS* and *AE*, in a non-trivial way. Our main motivation in this chapter is to combine *KAS* and *AE* into a single primitive, which is different from *KAS* and *AE* executed separately, and solve the *HAC* problem more efficiently. At this point, it is essential to note that a new cryptographic primitive combining *KAS* and *AE*, such as *KAS-AE*, as discussed in this chapter, makes little sense, if it does not permit constructions that are significantly more efficient than their trivial combination. Therefore, we summarize our main challenge below:

Can we integrate *KAS* and *AE* into a novel primitive

KAS-AE, so that this new primitive solves the *HAC* problem more efficiently than achieved by separate applications of *KAS* and *AE*?

In the remainder of this chapter, we search for answers to the above question, and analyze them.

5.2 Contribution of this Chapter

Our first contribution is introducing a new primitive *KAS-AE*. To develop, motivate, analyze and understand this new idea easily, we propose a total of nine *KAS-AE* constructions – except one all are proven secure – with varying degrees of efficiencies and construction subtleties: (1) in the first construction Ψ , we show that the most natural combination of *KAS* and *AE* to generate *KAS-AE* is prone to attacks; (2) in the second construction Ψ_Ω , we obviate this attack, and show a secure way of combining *KAS* and *AE* to build a *KAS-AE* scheme; (3-6) our next four *KAS-AE* constructions $\bar{\Psi}$, $\check{\Psi}$, $\dot{\Psi}$ and $\ddot{\Psi}$ are based on building *KAS-AE* schemes for linear graphs (or totally ordered sets) followed by combining them to support arbitrary access graphs (*i.e.* partially ordered sets); (7-9) these last three constructions $\hat{\Psi}$, $\hat{\Psi}$ and $\tilde{\Psi}$ are the most efficient *KAS-AE* constructions which are based on three different primitives *MLE* [BKR13b], *APE* authenticated encryption [ABB⁺14] and *FP* hash mode [PHG12], respectively.

We have given a detailed comparison of all the proposed nine *KAS-AE* constructions in Tables 5.1, 5.2 and 5.3. Our best three constructions (see Table 5.3) outperform all other conventional *HAC* solutions (based on *KAS* and *AE* individually, as opposed to on the single primitive *KAS-AE*) with respect to running time, by a factor of *at least 2 (or 3)* for any reasonable parameter choices; also, the *private storage* of our best performing constructions is linear, while they are quadratic (or cubic) in the simple combination of *KAS* and *AE*.

In order to obtain this improvement in performance, our constructions exploit, among others, a very unique feature – what we call *reverse decryption* – supported by the *APE* authenticated encryption and *FP* hash mode of operation. It turns out that the *reverse decryption* property can also be obtained by a clever use of *MLE* schemes. Besides this, our constructions also benefit from a novel key management technique to optimize the storage requirements in the very challenging scenarios, where organizational access structure is non-linear (*i.e.*, a poset, rather than a totally ordered set).

Note that the very unique *reverse decryption* property has also been used in Chapter 4 to construct efficient *FMLE* schemes. However, our focus in this chapter is on finding an efficient solution for the very different and fairly old *HAC* problem that, unlike the *FMLE* solution, involves a significant amount of intricate graph theoretic algorithms and tools (e.g. root-finding algorithm, shortest path algorithm, etc.) to overcome crucial key management challenges.

5.3 *KAS-AE*: A New Cryptographic Primitive

We have already discussed the *key assignment scheme* (*KAS*) in Section 3.6. This new primitive *KAS-AE* can, loosely, be viewed as a *KAS* plugged with an additional function, namely, *authenticated encryption*. We observe that *KAS* consists of two algorithms, namely, *key generation* and *key derivation*. The keys generated by *KAS* are later used to encrypt messages in various use-cases. The motivation for *KAS-AE* is to combine the *KAS* and (*authenticated*) *encryption* together, and view them as a single cryptographic primitive. Therefore, in *KAS-AE*, we target three goals: a combined key generation and authenticated encryption algorithm; a key derivation algorithm, which is identical to the one in *KAS*; and a decryption algorithm, which is necessitated by the authenticated encryption already included in the scheme. As outlined earlier, this new cryptographic primitive allows us to construct schemes that are more efficient than the trivial execution of *KAS* followed by *AE*. In this chapter, we will be discussing this issue in great detail.

Now we elaborately discuss the syntax, correctness and security definition of the new notion *KAS-AE*.

5.3.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *KAS-AE* scheme $\Psi = (\Psi.\mathcal{E}, \Psi.\mathcal{DER}, \Psi.\mathcal{D})$ is a 3-tuple of algorithms over the *setup* algorithm $\Psi.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\Psi.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\Psi)}$, a set of *access graphs* $\Gamma^{(\Psi)}$, *key-space* $\mathcal{K}^{(\Psi)} \subseteq \{0, 1\}^*$ and *message-space* $\mathcal{M}^{(\Psi)} \subseteq \{0, 1\}^*$.

2. The PPT *encryption* algorithm $\Psi.\mathcal{E}(\cdot)$ takes as input parameter *params*^(Ψ), graph $G = (V, E)$ and vector of *files* $\mathbf{M} = (\mathbf{M}^{(u_i)})_{u_i \in V}$, and returns a 3-tuple $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M})$, where vector of *private information* $\mathbf{S} = (\mathbf{S}^{(u_i)})_{u_i \in V}$, vector of *keys* $\mathbf{K} = (\mathbf{K}^{(u_i)})_{u_i \in V}$ and $P \in \{0, 1\}^*$ being the *public information*.

Note that $\mathbf{M}^{(u_i)} \in \mathcal{M}^{(\Psi)}$, $\mathbf{S}^{(u_i)} \in \{0, 1\}^*$ and $\mathbf{K}^{(u_i)} \in \mathcal{K}^{(\Psi)}$, for all $u_i \in V$, and all $G \in \Gamma^{(\Psi)}$.

3. The *key derivation* algorithm $\Psi.\mathcal{DER}(\cdot)$ is a deterministic *poly-time* algorithm, such that $\mathbf{K}^{(u_{i_2})} := \Psi.\mathcal{DER}(\text{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$, where $u_{i_2} \leq u_{i_1}$ are two nodes of access graph $G \in \Gamma^{(\Psi)}$, $\mathbf{S}^{(u_{i_1})}$ is u_{i_1} 's *private information*, P is *public information*, and $\mathbf{K}^{(u_{i_2})}$ is u_{i_2} 's decryption key.

Note that $\mathbf{S}^{(u_{i_1})} \in \{0, 1\}^*$, $P \in \{0, 1\}^*$ and $\mathbf{K}^{(u_{i_2})} \in \mathcal{K}^{(\Psi)} \cup \{\perp\}$.

4. The *decryption* algorithm $\Psi.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm, such that $\mathbf{M}^{(u_{i_2})} := \Psi.\mathcal{D}(\text{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$, where u_{i_1} decrypts the ciphertext corresponding to node u_{i_2} , such that $u_{i_2} \leq u_{i_1}$ in access graph $G \in \Gamma^{(\Psi)}$, $\mathbf{S}^{(u_{i_1})}$ is u_{i_1} 's *private information*, P is *public information*, and $\mathbf{M}^{(u_{i_2})}$ is u_{i_2} 's decrypted file.

Note that $\mathbf{S}^{(u_{i_1})} \in \{0, 1\}^*$, $P \in \{0, 1\}^*$ and $\mathbf{M}^{(u_{i_2})} \in \mathcal{M}^{(\Psi)} \cup \{\perp\}$. A special case $\mathbf{M}^{(u_{i_2})} = \perp$ occurs, if ciphertext of u_{i_2} is not valid, that is, ciphertext is not generated by encrypting a valid message.

5.3.2 Correctness

KEY DERIVATION CORRECTNESS. This condition requires that if the vector of *keys*, vector of *private information* and *public information* are generated correctly, then the keys can be recomputed correctly. Mathematically, the *key derivation correctness* of a *KAS-AE* can be defined as follows.

Suppose:

- $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M})$.

Then, the *key derivation correctness* of Ψ requires that $\Psi.\mathcal{DER}(\text{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P) = \mathbf{K}^{(u_{i_2})}$, for all $(\lambda, G, \mathbf{M}) \in \mathbb{N} \times \Gamma^{(\Psi)} \times (\mathcal{M}^{(\Psi)})^{|V|}$, and all nodes $u_{i_2} \leq u_{i_1}$.

DECRYPTION CORRECTNESS. This condition requires that if the vector of *keys*, vector of *private information* and *public information* are generated correctly, then the messages can be decrypted correctly. Mathematically, the *decryption correctness* of a *KAS-AE* can be defined as follows.

Suppose:

- $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M})$.

Then, the *decryption correctness* of Ψ requires that $\Psi.\mathcal{D}(\text{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P) = \mathbf{M}^{(u_{i_2})}$, for all $(\lambda, G, \mathbf{M}) \in \mathbb{N} \times \Gamma^{(\Psi)} \times (\mathcal{M}^{(\Psi)})^{|V|}$, and all nodes $u_{i_2} \leq u_{i_1}$.

5.3.3 Security definitions

The security notions of *KAS-AE* are influenced by those of *KAS* [FPP13] and *AE* [BN08, BRW03, Rog02]. So, we should have four security notions, namely, *key indistinguishability*, *key recovery resistance*, *privacy* and *tag consistency*. However, the notion of *key indistinguishability*, as described in [FPP13], is not relevant for *KAS-AE*; this is because of the fact that a user's *decryption key* is included in the user's *private information* itself; and *public information* P contains the entire ciphertext. Thus, distinguishing a random string from a genuine key is trivial. Taking these scenarios into consideration, we target three security goals: *key recovery*, *privacy* and *integrity*. As usual, all the games are written using the challenger-adversary framework. In Figures 5.1, 5.2 and 5.3, we define the security games **KASAE-KR**, **KASAE-IND** and **KASAE-INT**, respectively, for the *KAS-AE* scheme $\Psi = (\Psi.\mathcal{E}, \Psi.\mathcal{DER}, \Psi.\mathcal{D})$.

Game KASAE-KR$_{\Psi}^{\mathcal{A}}(1^{\lambda}, G)$ $(\text{params}^{(\Psi)}, \Gamma^{(\Psi)}, \mathcal{K}^{(\Psi)}, \mathcal{M}^{(\Psi)}) := \Psi.\text{Setup}(1^{\lambda});$ If $(G \notin \Gamma^{(\Psi)})$, then return Error ; $(u_{i_1}, \mathbf{M}) := \mathcal{A}_1(1^{\lambda}, G);$ $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M});$ $SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} u_{i_2} < u_{i_1}\};$ $K' := \mathcal{A}_2(1^{\lambda}, G, P, SV^{(u_{i_1})});$ return $(\mathbf{K}^{(u_{i_1})} = K')$;

Figure 5.1: Security game **KASAE-KR** for the *KAS-AE* scheme Ψ .

KASAE-KR SECURITY. This captures the intuitive notion of *key recovery with respect to static adversary*¹, as defined in Figure 5.1. The adversarial advantage is the probability of the event when the adversary is able to deduce the encryption key, and it is desired to be negligible.

According to the **KASAE-KR** game, given access to graph G , the adversary first returns a security class $u_{i_1} \in V$, that he/she chooses to attack,

¹The dynamic adversary is different from a static adversary in the way that, unlike the latter, the former can make adaptive queries to gather information from the nodes [FPP13]. However, a dynamic adversary is *poly-reducible* to a static adversary.

and a vector of *files* \mathbf{M} . Then, the following operations are performed: first $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M})$ is computed; and then the set $SV^{(u_{i_1})}$ of *private information* $\mathbf{S}^{(u_{i_2})}$ for classes $u_{i_2} \in V$, such that $u_{i_2} < u_{i_1}$ is computed. Given access to *public information* P and set $SV^{(u_{i_1})}$, the adversary returns a key K' . The game returns 1, if $\mathbf{K}^{(u_{i_1})} = K'$.

The advantage of the **KASAE-KR** adversary \mathcal{A} against Ψ on a graph $G \in \Gamma^{(\Psi)}$ is defined as follows:

$$Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{KASAE-KR}_\Psi^{\mathcal{A}}(1^\lambda, G) = 1].$$

The scheme Ψ is **KASAE-KR** secure, if, for all PPT static adversaries \mathcal{A} , the value of $Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda)$ is negligible.

Game KASAE-IND $_\Psi^{\mathcal{A}}(1^\lambda, G, b)$ $(\text{params}^{(\Psi)}, \Gamma^{(\Psi)}, \mathcal{K}^{(\Psi)}, \mathcal{M}^{(\Psi)}) := \Psi.\text{Setup}(1^\lambda);$ If $(G \notin \Gamma^{(\Psi)})$, then return Error ; $(\mathbf{M}_0, \mathbf{M}_1) := \mathcal{A}_1(1^\lambda, G);$ $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M}_b);$ $b' := \mathcal{A}_2(1^\lambda, G, P, \mathbf{M}_0, \mathbf{M}_1);$ return b' ;

Figure 5.2: Security game **KASAE-IND** for the *KAS-AE* scheme Ψ .

KASAE-IND SECURITY. This captures the notion of *indistinguishability privacy* attack, as defined in Figure 5.2. The adversarial advantage is the probability of the event when the adversary is able to distinguish between the encryptions of two vectors of *files*, and it is desired to be negligible.

According to the **KASAE-IND** game, given access to graph G , the adversary first returns two vectors of *files* \mathbf{M}_0 and \mathbf{M}_1 , such that $|\mathbf{M}_0^{(u_i)}| = |\mathbf{M}_1^{(u_i)}|$ for all $u_i \in V$. Then, $(\mathbf{S}, \mathbf{K}, P) := \Psi.\mathcal{E}(\text{params}^{(\Psi)}, G, \mathbf{M}_b)$ is computed, where $b \in \{0, 1\}$. Given access to graph G , *public information* P and two vectors of *files* \mathbf{M}_0 and \mathbf{M}_1 , the adversary returns a bit b' .

The advantage of the **KASAE-IND** adversary \mathcal{A} against Ψ on a graph $G \in \Gamma^{(\Psi)}$ is defined as follows:

$$\begin{aligned} Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{KASAE-IND}_\Psi^{\mathcal{A}}(1^\lambda, G, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{KASAE-IND}_\Psi^{\mathcal{A}}(1^\lambda, G, b = 0) = 1] \right|. \end{aligned}$$

The scheme Ψ is **KASAE-IND** secure, if, for all PPT adversaries \mathcal{A} , the value of $Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda)$ is negligible.

```

Game KASAE-INT $_{\Psi}^{\mathcal{A}}(1^{\lambda}, G)$ 
( $params^{(\Psi)}, \Gamma^{(\Psi)}, \mathcal{K}^{(\Psi)}, \mathcal{M}^{(\Psi)}$ ) :=  $\Psi$ .Setup( $1^{\lambda}$ );
If ( $G \notin \Gamma^{(\Psi)}$ ), then return Error;
( $u_{i_1}, P_0, P_1, \mathbf{S}, \mathbf{K}$ ) :=  $\mathcal{A}(1^{\lambda}, G)$ ;
 $\mathbf{M}_0^{(u_{i_1})} := \Psi$ . $\mathcal{D}(params^{(\Psi)}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_0)$ ;
 $\mathbf{M}_1^{(u_{i_1})} := \Psi$ . $\mathcal{D}(params^{(\Psi)}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_1)$ ;
If ( $\mathbf{M}_0^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_1^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_0^{(u_{i_1})} \neq \mathbf{M}_1^{(u_{i_1})}$ )
    return 1;
Else return 0;

```

Figure 5.3: Security game KASAE-INT for the *KAS-AE* scheme Ψ .

KASAE-INT SECURITY. This captures the notion of *tag consistency* attack, as defined in Figure 5.3. The adversarial advantage is the probability of the event when the adversary is able to compute two *public informations*, a vector of *private information* and a vector of *keys* for a target node, such that, on decryption of the supplied *public informations* using the supplied vector of *private information*, the two messages for the target node are valid and unidentical. This adversarial advantage is desired to be negligible.

According to the KASAE-INT game, given access to graph G , the adversary first returns security class $u_{i_1} \in V$, two *public information* values P_0 and P_1 , vector of *private information* \mathbf{S} and vector of *keys* \mathbf{K} . Then, files corresponding to user u_{i_1} in P_0 and P_1 are computed as $\mathbf{M}_0^{(u_{i_1})} := \Psi$. $\mathcal{D}(params^{(\Psi)}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_0)$ and $\mathbf{M}_1^{(u_{i_1})} := \Psi$. $\mathcal{D}(params^{(\Psi)}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_1)$. The game returns 1, if files $\mathbf{M}_0^{(u_{i_1})}$ and $\mathbf{M}_1^{(u_{i_1})}$ are valid (i.e. $\mathbf{M}_0^{(u_{i_1})} \neq \perp$ and $\mathbf{M}_1^{(u_{i_1})} \neq \perp$) under $\mathbf{M}_0^{(u_{i_1})} \neq \mathbf{M}_1^{(u_{i_1})}$.

The advantage of the KASAE-INT adversary \mathcal{A} against Ψ on a graph $G \in \Gamma^{(\Psi)}$ is defined as follows:

$$Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-INT}}(1^{\lambda}) \stackrel{def}{=} \Pr[\text{KASAE-INT}_{\Psi}^{\mathcal{A}}(1^{\lambda}, G) = 1].$$

The scheme Ψ is KASAE-INT secure, if, for all PPT adversaries \mathcal{A} , the value of $Adv_{\Psi, \mathcal{A}, G}^{\text{KASAE-INT}}(1^{\lambda})$ is negligible.

Remark. Note that a *KAS-AE-chain* $\psi = (\psi, \mathcal{E}, \psi, \mathcal{DER}, \psi, \mathcal{D})$ is a special type of *KAS-AE* where the access graph is a *totally ordered set*.

5.4 *HAC*: An Application of *KAS-AE*

In Section 3.5, we have discussed elaborately the necessity of a *HAC* protocol. We also knew that *KAS* could be used to design a *HAC* protocol. In this section, we construct a *HAC* protocol using *KAS-AE* (instead of *KAS*). The advantage is that the *HAC* protocol now inherits the *authenticated encryption* function of *KAS-AE*. The *KAS-AE*-based *HAC* protocol is composed of three independent protocols: (1) ESTABLISH, (2) KEY DERIVE, and (3) DECRYPTION. Below we give the full textual description.

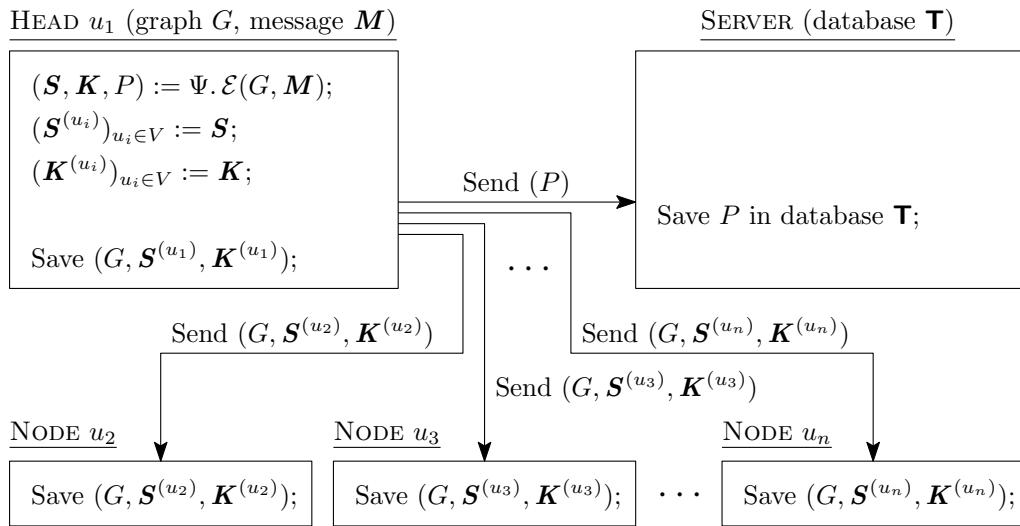


Figure 5.4: ESTABLISH protocol for *KAS-AE*-based *HAC*.

- **ESTABLISH.** This protocol is responsible for the series of operations undertaken, when the head of the organization encrypts the vector of *files* \mathbf{M} for the nodes in the access graph $G = (V, E)$, and then distributes the keys and other necessary information to all the nodes; the pictorial description is given in Figure 5.4. The head node u_1 performs the following operations: first it computes a 3-tuple $(\mathbf{S}, \mathbf{K}, P) := \Psi \cdot \mathcal{E}(G, \mathbf{M})$; then it parses the vector of *private information* $(\mathbf{S}^{(u_i)})_{u_i \in V} := \mathbf{S}$ and the vector of *keys* $(\mathbf{K}^{(u_i)})_{u_i \in V} := \mathbf{K}$; next, it sends the *public information* P to the server; and finally it sends a 3-tuple $(G, \mathbf{S}^{(u_i)}, \mathbf{K}^{(u_i)})$ to each node $u_i \in V$. The nodes $u_i \in V$ store the supplied values of $(G, \mathbf{S}^{(u_i)}, \mathbf{K}^{(u_i)})$ corresponding to the *public information* P .
- **KEY DERIVE.** This protocol takes care of the operations performed when a node u_{i_1} computes the decryption key of a node $u_{i_2} \leq u_{i_1}$;

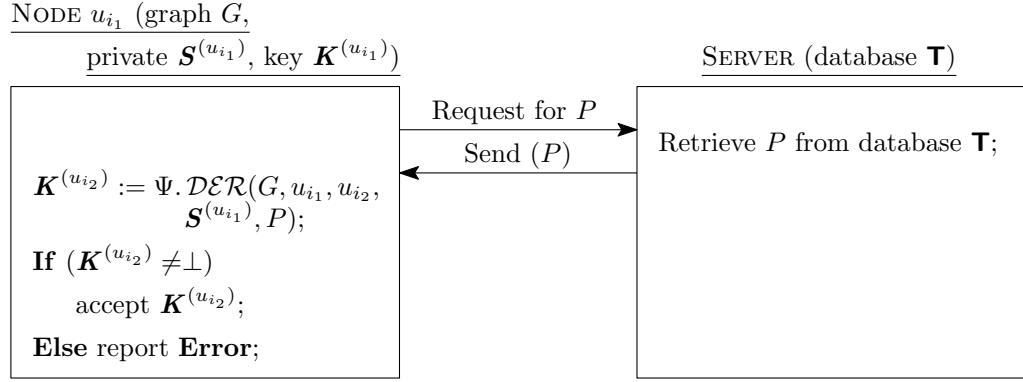


Figure 5.5: KEY DERIVE protocol for KAS-AE-based HAC.

the pictorial description is given in Figure 5.5. The node u_{i_1} requests the server to supply the *public information* P . In response, the server retrieves P from its database and sends it to u_{i_1} . After receiving P , node u_{i_1} performs the following operations: first it computes the key $\mathbf{K}^{(u_{i_2})} := \Psi \cdot \mathcal{DER}(G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ using the stored G and $\mathbf{S}^{(u_{i_1})}$, and the supplied P ; and accepts $\mathbf{K}^{(u_{i_2})}$ only if $\mathbf{K}^{(u_{i_2})} \neq \perp$.

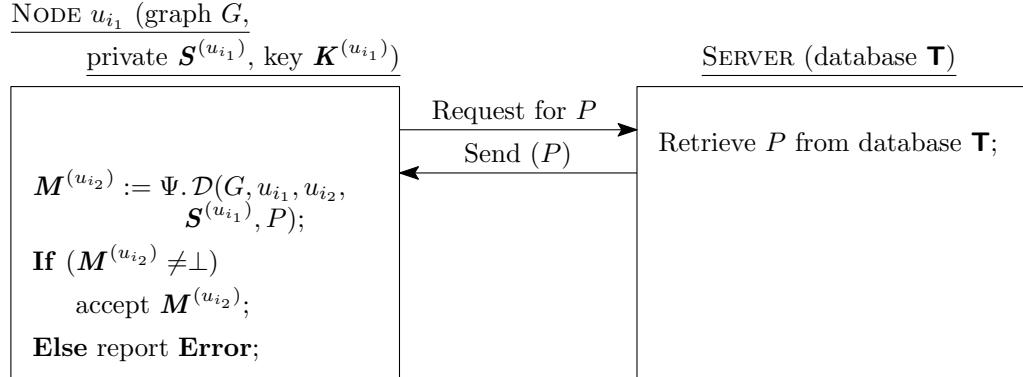


Figure 5.6: DECRYPTION protocol for KAS-AE-based HAC.

- DECRYPTION. This protocol takes care of the operations performed when a node u_{i_1} computes the file corresponding to node $u_{i_2} \leq u_{i_1}$; the pictorial description is given in Figure 5.6. The node u_{i_1} requests the server to supply the *public information* P . In response, the server retrieves P from its database and sends it to u_{i_1} . After receiving P , node u_{i_1} performs the following operations: first it computes the file $\mathbf{M}^{(u_{i_2})} := \Psi \cdot \mathcal{D}(G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ using the stored G and $\mathbf{S}^{(u_{i_1})}$, and the supplied P ; and accepts $\mathbf{M}^{(u_{i_2})}$ only if $\mathbf{M}^{(u_{i_2})} \neq \perp$.

5.5 $KAS-AE$ from KAS and AE

In this section, we design two $KAS-AE$ constructions – denoted $\acute{\Psi}$ and $\grave{\Psi}_\Omega$ – from a KAS construction $\Omega = (\Omega.\mathcal{GEN}, \Omega.\mathcal{DER})$ over $\Omega.\text{Setup}$, and an AE scheme $\varrho = (\varrho.\mathcal{GEN}, \varrho.\mathcal{E}, \varrho.\mathcal{D})$ over $\varrho.\text{Setup}$, as discussed in Sections 3.6 and 2.2.5, respectively.

5.5.1 A natural construction $\acute{\Psi}$ and an attack

We first attempt to design a $KAS-AE$ construction $\acute{\Psi}$ from KAS in the most intuitive way. Later we show that how $\acute{\Psi}$ is vulnerable to an attack.

A $KAS-AE$ scheme guarantees *authentication* of encrypted messages, in addition to the security properties of a KAS (note that KAS security properties alone do not guarantee *authenticated encryption*). A natural way to include this property in KAS could have been to use an AE scheme to encrypt messages of nodes, using the keys distributed to them by the KAS . Such a natural $KAS-AE$ scheme $\acute{\Psi}$ is constructed below.

5.5.1.1 Description of $\acute{\Psi}$

The pseudocode for the 3-tuple of algorithms in $\acute{\Psi}$ are given in Figures 5.7, 5.8 and 5.9. Below, we give the textual description.

- The *setup* algorithm $\acute{\Psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\acute{\Psi})}$, a set of *access graphs* $\Gamma^{(\acute{\Psi})}$, *key-space* $\mathcal{K}^{(\acute{\Psi})}$ and *message-space* $\mathcal{M}^{(\acute{\Psi})}$. The $\acute{\Psi}.\text{Setup}(1^\lambda)$ is designed in the following way: first it invokes $\Omega.\text{Setup}(1^\lambda)$ and $\varrho.\text{Setup}(1^\lambda)$; then it computes $\text{params}^{(\acute{\Psi})}$ that includes, among others, $\text{params}^{(\Omega)}$ and $\text{params}^{(\varrho)}$; and finally it computes set of *access graphs* $\Gamma^{(\acute{\Psi})} = \Gamma^{(\Omega)}$, *key-space* $\mathcal{K}^{(\acute{\Psi})} = \mathcal{K}^{(\Omega)} = \mathcal{K}^{(\varrho)}$, and *message-space* $\mathcal{M}^{(\acute{\Psi})} = \mathcal{M}^{(\varrho)}$.
- The *encryption* algorithm $\acute{\Psi}.\mathcal{E}(\cdot)$ is randomised that takes as input parameter $\text{params}^{(\acute{\Psi})}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pseudocode is given in Figure 5.7. Very briefly, the algorithm works as follows: first it computes $(\mathbf{S}, \mathbf{K}, P) := \Omega.\mathcal{GEN}(\text{params}^{(\Omega)}, G)$; then for all nodes $u_i \in V$, it computes a pair of ciphertext and tag $(\mathbf{C}^{(u_i)}, \mathbf{T}^{(u_i)}) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, \mathbf{K}^{(u_i)}, \mathbf{M}^{(u_i)})$; and finally it computes $\mathbf{C} := (\mathbf{C}^{(u_i)} \parallel \mathbf{T}^{(u_i)})_{u_i \in V}$.

Now, the vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* $P := \mathbf{C} \parallel P$.

```

 $\Psi \cdot \mathcal{E}(\textit{params}^{(\Psi)}, G, \mathbf{M})$ 


---


#Generating keys using KAS.
 $(\mathbf{S}, \mathbf{K}, P) := \Omega \cdot \mathcal{GEN}(\textit{params}^{(\Omega)}, G);$ 

#Computing ciphertext for each user.
for all  $u_i \in V$ 
 $(\mathbf{C}^{(u_i)}, \mathbf{T}^{(u_i)}) := \varrho \cdot \mathcal{E}(\textit{params}^{(\varrho)}, \mathbf{K}^{(u_i)}, \mathbf{M}^{(u_i)});$ 

#Computing final outputs.
 $\mathbf{C} := (\mathbf{C}^{(u_i)} \| \mathbf{T}^{(u_i)})_{u_i \in V}; \quad P := \mathbf{C} \| P;$ 
return  $(\mathbf{S}, \mathbf{K}, P);$ 

```

Figure 5.7: Algorithmic description of the *encryption* function $\Psi \cdot \mathcal{E}$.

```

 $\Psi \cdot \mathcal{DER}(\textit{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 


---


#Deriving key using KAS.
 $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 
return  $\mathbf{K}^{(u_{i_2})};$ 

```

Figure 5.8: Algorithmic description of the *key derivation* function $\Psi \cdot \mathcal{DER}$.

- The *key derivation* algorithm $\Psi \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $\textit{params}^{(\Psi)}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . See Figure 5.8 for the pseudocode. The key is computed as follows: $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$.

```

 $\Psi \cdot \mathcal{D}(\textit{params}^{(\Psi)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 


---


#Deriving key using KAS.
 $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 

#Computing message.
 $\mathbf{C} \| P := P; \quad (\mathbf{C}^{(u_i)} \| \mathbf{T}^{(u_i)})_{u_i \in V} := \mathbf{C};$ 
 $\mathbf{M}^{(u_{i_2})} := \varrho \cdot \mathcal{D}(\textit{params}^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})});$ 
return  $\mathbf{M}^{(u_{i_2})};$ 

```

Figure 5.9: Algorithmic description of the *decryption* function $\Psi \cdot \mathcal{D}$.

- The *decryption* algorithm $\dot{\Psi}.\mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\dot{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . See Figure 5.9 for the pseudocode. Very briefly, the algorithm works as follows: first it computes key $\mathbf{K}^{(u_{i_2})} := \Omega.\mathcal{DER}(params^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$; then it parses P into $\mathbf{C} \| P := P$; next it extracts ciphertext and tag $(\mathbf{C}^{(u_i)} \| \mathbf{T}^{(u_i)})_{u_i \in V} := \mathbf{C}$; and finally it computes the message $\mathbf{M}^{(u_{i_2})} := \varrho.\mathcal{D}(params^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})})$.

5.5.1.2 A simple attack on the *tag consistency* security of $\dot{\Psi}$.

The attack works as follows: a node u_{i_1} replaces the original ciphertext $\mathbf{C}^{(u_{i_1})} \| \mathbf{T}^{(u_{i_1})}$ in the *public information* P with a different ciphertext $\mathbf{C}'^{(u_{i_1})} \| \mathbf{T}'^{(u_{i_1})}$ computed under the original key $\mathbf{K}^{(u_{i_1})}$ and file $\mathbf{M}'^{(u_{i_1})} \neq \mathbf{M}^{(u_{i_1})}$; now, any node u_{i_2} , such that $u_{i_1} \leq u_{i_2}$, decrypts $\mathbf{C}'^{(u_{i_1})}$ without any error message.

5.5.2 Construction $\dot{\Psi}_\Omega$: A secure (yet inefficient) *KAS-AE*

In Section 5.5.1, we have shown an attack on the most intuitive *KAS-AE* construction $\dot{\Psi}$ built from a *KAS* and an *AE* scheme. In this section, we design a *KAS-AE* construction $\dot{\Psi}_\Omega$ by combining a generic *KAS* construction Ω and an *AE* scheme ϱ in a different way than done in construction $\dot{\Psi}$, so that the attack on $\dot{\Psi}$ is avoided.

As opposed to considering the authentication tag being a part of the ciphertext, here, we assume that tag and ciphertext are distinct. The core idea behind this construction is that tag is a secret value, and that every node stores a set of tags – for its own file as well as those for the files of its successors – in its *private information*. Although, $\dot{\Psi}_\Omega$ is a secure *KAS-AE* scheme, the high memory requirements make it unsuitable for many practical applications.

5.5.2.1 Description of $\dot{\Psi}_\Omega$

The pseudocode for the 3-tuple of algorithms in $\dot{\Psi}_\Omega$ are given in Figures 5.10, 5.11 and 5.12. Below, we give the textual description.

- The *setup* algorithm $\dot{\Psi}_\Omega.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\dot{\Psi}_\Omega)}$, set of *access graphs* $\Gamma^{(\dot{\Psi}_\Omega)}$, *key-space* $\mathcal{K}^{(\dot{\Psi}_\Omega)}$ and *message-space* $\mathcal{M}^{(\dot{\Psi}_\Omega)}$. The $\dot{\Psi}_\Omega.\text{Setup}(1^\lambda)$

is designed in the following way: first it invokes $\Omega.\text{Setup}(1^\lambda)$ and $\varrho.\text{Setup}(1^\lambda)$; then it computes $params^{(\Psi_\Omega)}$ that includes, among others, $params^{(\Omega)}$ and $params^{(\varrho)}$; and finally it computes a set of *access graphs* $\Gamma^{(\Psi_\Omega)} = \Gamma^{(\Omega)}$, *key-space* $\mathcal{K}^{(\Psi_\Omega)} = \mathcal{K}^{(\Omega)} = \mathcal{K}^{(\varrho)}$ and *message-space* $\mathcal{M}^{(\Psi_\Omega)} = \mathcal{M}^{(\varrho)}$.

```

 $\dot{\Psi}_\Omega \cdot \mathcal{E}(params^{(\Psi_\Omega)}, G, \mathbf{M})$ 
#Generating keys using KAS.
 $(\mathbf{S}, \mathbf{K}, P) := \Omega \cdot \mathcal{GEN}(params^{(\Omega)}, G);$ 

#Computing ciphertext for each user.
for all  $u_i \in V$ 
 $(\mathbf{C}^{(u_i)}, \mathbf{T}^{(u_i)}) := \varrho \cdot \mathcal{E}(params^{(\varrho)}, \mathbf{K}^{(u_i)}, \mathbf{M}^{(u_i)});$ 

#Appending tags in private information.
for all  $u_{i_1} \in V$ 
 $\downarrow u_{i_1} := \text{all\_succ}(u_{i_1}, G);$ 
for all  $u_{i_2} \in \downarrow u_{i_1}$ 
 $\mathbf{S}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_2})};$ 

#Computing final outputs.
 $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}; \quad \mathbf{C} := (\mathbf{C}^{(u_i)})_{u_i \in V}; \quad P := P \| \mathbf{C};$ 
return  $(\mathbf{S}, \mathbf{K}, P);$ 

```

Figure 5.10: Algorithmic description of the *encryption* function $\dot{\Psi}_\Omega \cdot \mathcal{E}$.

- The *encryption* algorithm $\dot{\Psi}_\Omega \cdot \mathcal{E}(\cdot)$ is randomised that takes as input parameter $params^{(\Psi_\Omega)}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pseudocode is given in Figure 5.10. The algorithm works as follows: first it computes $(\mathbf{S}, \mathbf{K}, P) := \Omega \cdot \mathcal{GEN}(params^{(\Omega)}, G)$; and then for all nodes $u_i \in V$, it computes a pair of ciphertext and tag $(\mathbf{C}^{(u_i)}, \mathbf{T}^{(u_i)}) := \varrho \cdot \mathcal{E}(params^{(\varrho)}, \mathbf{K}^{(u_i)}, \mathbf{M}^{(u_i)})$. Now, for all nodes $u_{i_1} \in V$, the following operations are performed: first the set of all the successor nodes $\downarrow u_{i_1} := \text{all_succ}(u_{i_1}, G)$ is computed; and then the tag $\mathbf{T}^{(u_{i_2})}$ of each node $u_{i_2} \in \downarrow u_{i_1}$ is appended in $\mathbf{S}^{(u_{i_1})}$.

Now, the vector of *private information* $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}$, vector of *keys* \mathbf{K} and *public information* $P := P \| (\mathbf{C}^{(u_i)})_{u_i \in V}$.

```

 $\dot{\Psi}_\Omega \cdot \mathcal{DER}(\textit{params}^{(\dot{\Psi}_\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 
#Deriving key using KAS.
 $P \| (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 
 $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{j_1})} \circ \mathbf{T}^{(u_{j_2})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_{j_d})} := \mathbf{S}^{(u_{i_1})};$ 
 $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 
return  $\mathbf{K}^{(u_{i_2})};$ 

```

Figure 5.11: Algorithmic description of the *key derivation* function $\dot{\Psi}_\Omega \cdot \mathcal{DER}$.

- The *key derivation* algorithm $\dot{\Psi}_\Omega \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $\textit{params}^{(\dot{\Psi}_\Omega)}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . The pseudocode is given in Figure 5.11. Very briefly, the algorithm works as follows: first it parses the *public information* P into $P \| (\mathbf{C}^{(u_i)})_{u_i \in V} := P$; next it parses the *private information* of u_{i_1} into $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{j_1})} \circ \mathbf{T}^{(u_{j_2})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_{j_d})} := \mathbf{S}^{(u_{i_1})}$; and finally it computes key $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$.

```

 $\dot{\Psi}_\Omega \cdot \mathcal{D}(\textit{params}^{(\dot{\Psi}_\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 
#Deriving key using KAS.
 $P \| (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 
 $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{j_1})} \circ \mathbf{T}^{(u_{j_2})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_{j_d})} := \mathbf{S}^{(u_{i_1})};$ 
 $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 

#Computing message.
 $\mathbf{M}^{(u_{i_2})} := \varrho \cdot \mathcal{D}(\textit{params}^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})});$ 
return  $\mathbf{M}^{(u_{i_2})};$ 

```

Figure 5.12: Algorithmic description of the *decryption* function $\dot{\Psi}_\Omega \cdot \mathcal{D}$.

- The *decryption* algorithm $\dot{\Psi}_\Omega \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $\textit{params}^{(\dot{\Psi}_\Omega)}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pseudocode is given in Figure 5.12. Very briefly, the algorithm works as follows: first it parses the *public information* P into $P \| (\mathbf{C}^{(u_i)})_{u_i \in V} := P$; next it parses the *private information* of u_{i_1} into $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{j_1})} \circ \mathbf{T}^{(u_{j_2})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_{j_d})} := \mathbf{S}^{(u_{i_1})}$; and finally it computes key $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\textit{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$.

$\mathbf{T}^{(u_{j_2})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_{j_d})} := \mathbf{S}^{(u_{i_1})}$; after that it computes key $\mathbf{K}^{(u_{i_2})} := \Omega \cdot \mathcal{DER}(\text{params}^{(\Omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$; and finally it computes message $\mathbf{M}^{(u_{i_2})} := \varrho \cdot \mathcal{D}(\text{params}^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})})$.

By plugging an existing KAS construction $\Omega \in \{\text{TKAS}, \text{TKEKAS}, \text{DKEKAS}, \text{IKEKAS}, \text{NBKAS}\}$ (described in Section 3.6.2.1) into the $\dot{\Psi}_\Omega$, we get a concrete KAS-AE construction.

5.5.2.2 Security of $\dot{\Psi}_\Omega$

Theorem 9. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\dot{\Psi}_\Omega$ has been defined in Section 5.5.2. For all poly-time KASAE-KR adversaries \mathcal{A} against $\dot{\Psi}_\Omega$, there exists a poly-time KAS-KR adversary \mathcal{B} against KAS scheme Ω , such that:*

$$\text{Adv}_{\dot{\Psi}_\Omega, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \text{Adv}_{\Omega, \mathcal{B}, G}^{\text{KAS-KR}}(1^\lambda).$$

Here, the KASAE-KR game is defined in Figure 5.1, and the KAS-KR game is defined in Figure 3.28.

Proof. We poly-reduce any KASAE-KR adversary \mathcal{A} against $\dot{\Psi}_\Omega$ into a KAS-KR adversary \mathcal{B} against Ω . The proof readily follows from it. ■

Theorem 10. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\dot{\Psi}_\Omega$ has been defined in Section 5.5.2. For all poly-time KASAE-IND adversaries \mathcal{A} against $\dot{\Psi}_\Omega$, there exists a poly-time AE-IND adversary \mathcal{B} against AE scheme ϱ , such that:*

$$\text{Adv}_{\dot{\Psi}_\Omega, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \text{Adv}_{\varrho, \mathcal{B}}^{\text{AE-IND}}(1^\lambda).$$

Here, the KASAE-IND game is defined in Figure 5.2, and the AE-IND game is defined in Figure 2.14.

Proof. We poly-reduce any KASAE-IND adversary \mathcal{A} against $\dot{\Psi}_\Omega$ into an AE-IND adversary \mathcal{B} against ϱ . The proof readily follows from it. ■

Theorem 11. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\dot{\Psi}_\Omega$ has been defined in Section 5.5.2. For all poly-time KASAE-INT adversaries \mathcal{A} against $\dot{\Psi}_\Omega$, there exists a poly-time AE-INT adversary \mathcal{B} against AE scheme ϱ , such that:*

$$\text{Adv}_{\dot{\Psi}_\Omega, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \text{Adv}_{\varrho, \mathcal{B}}^{\text{AE-INT}}(1^\lambda).$$

Here, the KASAE-INT game is defined in Figure 5.3, and the AE-INT game is defined in Figure 2.15.

Proof. We poly-reduce any **KASAE-INT** adversary \mathcal{A} against $\dot{\Psi}_\Omega$ into an **AE-INT** adversary \mathcal{B} against ϱ . The proof readily follows from it. ■

5.6 *Modified Chain Partition* Construction $\vec{\Psi}$: **KAS-AE** from **KAS-AE-chain**

In this section, we design *modified chain partition* construction $\vec{\Psi}$, which is a **KAS-AE** scheme, built from a **KAS-AE-chain** scheme $\psi = (\psi.\mathcal{E}, \psi.\mathcal{DER}, \psi.\mathcal{D})$ over $\psi.\text{Setup}$. The definition of **KAS-AE-chain** has already been described in Section 5.3. The construction $\vec{\Psi}$ will be described in detail shortly.

5.6.1 **KAS-AE-chain** constructions

In this section, we first construct four **KAS-AE-chain** schemes – denoted $\bar{\psi}$, $\check{\psi}$, $\dot{\psi}$ and $\ddot{\psi}$; they are based on **KAS-chain** [FPP13] and **AE** [BN08, BRW03, Rog02], **MLE** [BKR13b], **APE** [ABB⁺14] and **FP** [PHG12], respectively. These four schemes will be used to construct *modified chain partition* construction in Section 5.6.2.

5.6.1.1 Construction $\bar{\psi}$: Based on **KAS-chain** and **AE**

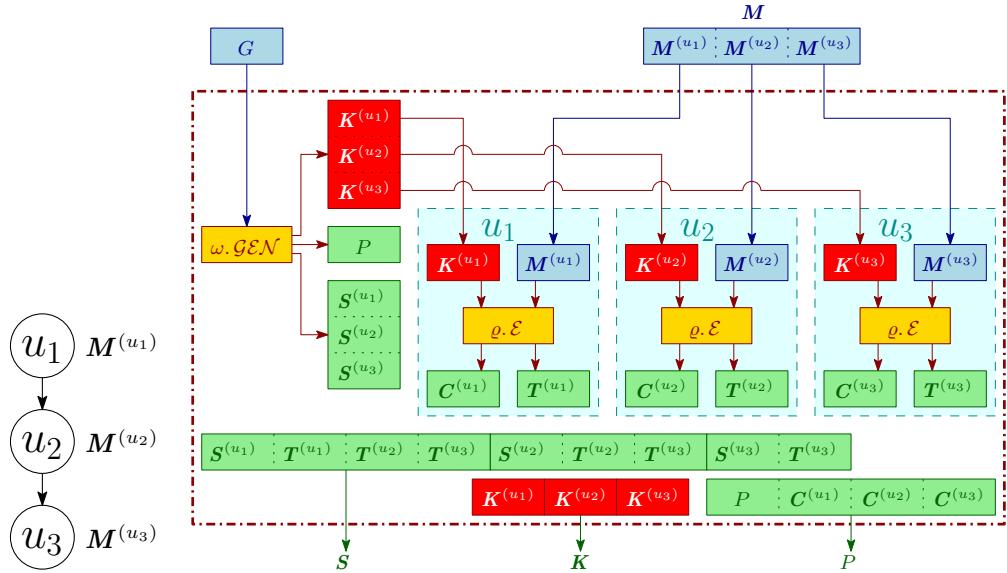
The **KAS-AE-chain** construction $\bar{\psi}$ is built from a **KAS-chain** $\omega = (\omega.\mathcal{GEN}, \omega.\mathcal{DER})$ over $\omega.\text{Setup}$ and an **AE** scheme $\varrho = (\varrho.\mathcal{GEN}, \varrho.\mathcal{E}, \varrho.\mathcal{D})$ over $\varrho.\text{Setup}$. The design methodology of $\bar{\psi}$ is identical to that of **KAS-AE** construction $\dot{\Psi}_\Omega$ (described in Section 5.5.2). In $\bar{\psi}$, we first compute vector of *private information*, vector of *keys* and *public information* using **KAS-chain** construction ω for a totally ordered set, followed by computation of ciphertext and tag for each node in the graph, and finally, for each user, tags of all the successor nodes are appended to *private information* of the user.

Description of $\bar{\psi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\bar{\psi}$ are given in Figures 5.13, 5.14, 5.15, 5.16, 5.17 and 5.18. Below, we give the textual description.

- The *setup* algorithm $\bar{\psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\bar{\psi})}$, a set of *access graphs* $\Gamma^{(\bar{\psi})}$, *key-*

space $\mathcal{K}^{(\bar{\psi})}$ and message-space $\mathcal{M}^{(\bar{\psi})}$. The $\bar{\psi}.\text{Setup}(1^\lambda)$ is designed in the following way: first it invokes $\omega.\text{Setup}(1^\lambda)$ and $\varrho.\text{Setup}(1^\lambda)$; then it computes $\text{params}^{(\bar{\psi})}$ that includes, among others, $\text{params}^{(\omega)}$ and $\text{params}^{(\varrho)}$; and finally it computes a set of access graphs $\Gamma^{(\bar{\psi})} = \Gamma^{(\omega)}$, key-space $\mathcal{K}^{(\bar{\psi})} = \mathcal{K}^{(\omega)} = \mathcal{K}^{(\varrho)}$ and message-space $\mathcal{M}^{(\bar{\psi})} = \mathcal{M}^{(\varrho)}$.



(a) The access graph G is shown in Figure 5.13(a) and vector of files $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \mathbf{M}^{(u_3)}$.

Figure 5.13: Pictorial description of the encryption function $\bar{\psi}.\mathcal{E}$.

- The encryption algorithm $\bar{\psi}.\mathcal{E}(\cdot)$ is randomised that takes as input parameter $\text{params}^{(\bar{\psi})}$, graph G and vector of files \mathbf{M} , and outputs vector of private information \mathbf{S} , vector of keys \mathbf{K} and public information P . The pictorial description and pseudocode are given in Figures 5.13 and 5.14. The algorithm works as follows: first it computes a sequence of vertices $(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G)$, such that $u_m \lessdot u_{m-1}, u_{m-1} \lessdot u_{m-2}, \dots, u_2 \lessdot u_1$; and then it computes $(\mathbf{S}, \mathbf{K}, P) := \omega.\mathcal{GEN}(\text{params}^{(\omega)}, G)$. Now, the ciphertext is computed iteratively, as j runs through $m, m-1, \dots, 1$, in the following way: first it computes a pair of ciphertext and tag $(\mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)}) := \varrho.\mathcal{E}(\text{params}^{(\varrho)}, \mathbf{K}^{(u_j)}, \mathbf{M}^{(u_j)})$; and then it updates $\mathbf{S}^{(u_j)} := \mathbf{S}^{(u_j)} \circ \mathbf{T}^{(u_j)} \circ \mathbf{T}^{(u_{j+1})} \circ \dots \circ \mathbf{T}^{(u_m)}$.

Now, the vector of private information $\mathbf{S} := \mathbf{S}^{(u_1)} \circ \mathbf{S}^{(u_2)} \circ \dots \circ \mathbf{S}^{(u_m)}$, and public information $P := P \| (\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)})$.

```

 $\bar{\psi} \cdot \mathcal{E}(\text{params}^{(\bar{\psi})}, G, M)$ 

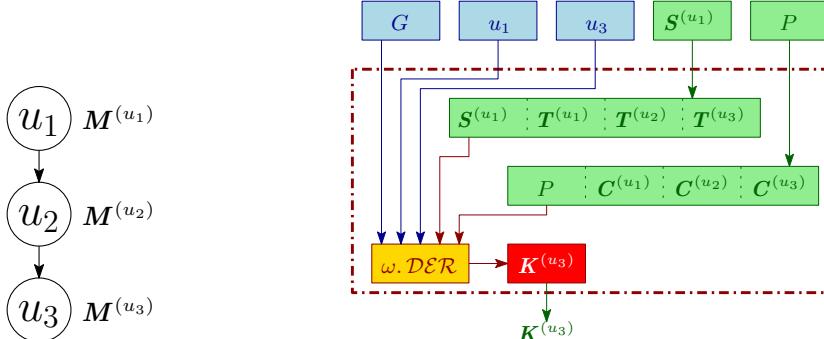
#Initialization.
 $(u_1, u_2, \dots, u_m) := \text{vertex\_in\_order}(G);$ 
 $M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_m)} := M;$ 

#Generating keys using KAS-chain.
 $(S, K, P) := \omega \cdot \mathcal{GEN}(\text{params}^{(\omega)}, G);$ 

#Computing ciphertext and private information for each user  $u_i$ .
for  $(j := m, m - 1, \dots, 1)$ 
     $(C^{(u_j)}, T^{(u_j)}) := \varrho \cdot \mathcal{E}(\text{params}^{(\varrho)}, K^{(u_j)}, M^{(u_j)});$ 
     $S^{(u_j)} := S^{(u_j)} \circ T^{(u_j)} \circ T^{(u_{j+1})} \circ \dots \circ T^{(u_m)};$ 

#Computing final outputs.
 $S := S^{(u_1)} \circ S^{(u_2)} \circ \dots \circ S^{(u_m)}; \quad C := C^{(u_1)} \circ C^{(u_2)} \circ \dots \circ C^{(u_m)};$ 
 $P := P \| C;$ 
return  $(S, K, P);$ 

```

Figure 5.14: Algorithmic description of the *encryption* function $\bar{\psi} \cdot \mathcal{E}$.Figure 5.15: Pictorial description of the *key derivation* function $\bar{\psi} \cdot \mathcal{DERR}$.

- The *key derivation* algorithm $\bar{\psi} \cdot \mathcal{DERR}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\bar{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $S^{(u_{i_1})}$ and *public information* P , and outputs key $K^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.15 and 5.16. Very

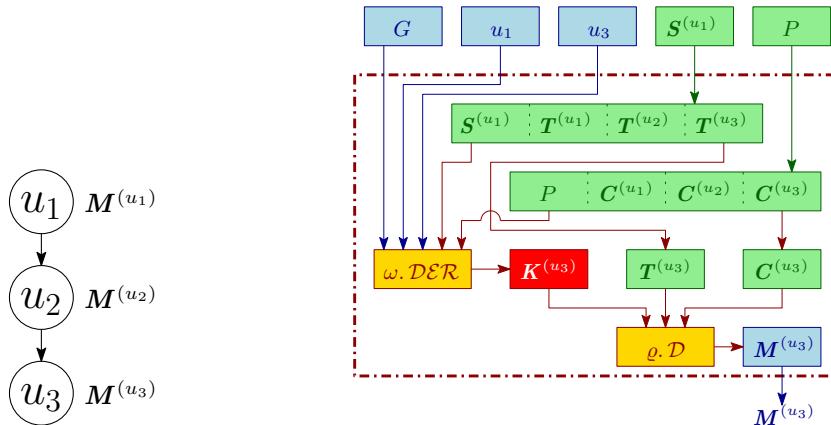
```

 $\bar{\psi} \cdot \mathcal{DER}(\text{params}^{(\bar{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 
#Deriving key using KAS-chain.
 $P \| (\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}) := P;$ 
 $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_m)} := \mathbf{S}^{(u_{i_1})};$ 
 $\mathbf{K}^{(u_{i_2})} := \omega \cdot \mathcal{DER}(\text{params}^{(\omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 
return  $\mathbf{K}^{(u_{i_2})};$ 

```

Figure 5.16: Algorithmic description of the *key derivation* function $\bar{\psi} \cdot \mathcal{DER}$.

briefly, the algorithm works as follows: first it parses *public information* P into $P \| (\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}) := P$; next it parses the *private information* of u_{i_1} into $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_m)} := \mathbf{S}^{(u_{i_1})}$; and finally it computes key $\mathbf{K}^{(u_{i_2})} := \omega \cdot \mathcal{DER}(\text{params}^{(\omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$.



(a) The access graph G . (b) Pictorial description of $\bar{\psi} \cdot \mathcal{D}(\text{params}^{(\bar{\psi})}, G, u_1, u_3, \mathbf{S}^{(u_1)}, P)$, where graph G is shown in Figure 5.17(a).

Figure 5.17: Pictorial description of the *decryption* function $\bar{\psi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\bar{\psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\bar{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.17 and 5.18. Very briefly, the algorithm works as follows: first it parses *public information* P into $P \| (\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}) := P$; next it parses the *private information* of u_{i_1} into $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \circ \mathbf{T}^{(u_m)} := \mathbf{S}^{(u_{i_1})}$; and finally it computes key $\mathbf{K}^{(u_{i_2})} := \omega \cdot \mathcal{DER}(\text{params}^{(\omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$.

```

 $\bar{\psi} \cdot \mathcal{D}(\text{params}^{(\bar{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 
#Deriving key using KAS-chain.
 $P \parallel (\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}) := P;$ 
 $\mathbf{S}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1})} \circ \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{T}^{(u_{i_2})} \circ \dots \mathbf{T}^{(u_m)} := \mathbf{S}^{(u_{i_1})};$ 
 $\mathbf{K}^{(u_{i_2})} := \omega \cdot \mathcal{DER}(\text{params}^{(\omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P);$ 

#Computing message.
 $\mathbf{M}^{(u_{i_2})} := \varrho \cdot \mathcal{D}(\text{params}^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})});$ 
return  $\mathbf{M}^{(u_{i_2})};$ 

```

Figure 5.18: Algorithmic description of the *decryption* function $\bar{\psi} \cdot \mathcal{D}$.

$\mathbf{T}^{(u_{i_2})} \circ \dots \mathbf{T}^{(u_m)} := \mathbf{S}^{(u_{i_1})}$; after that it computes key $\mathbf{K}^{(u_{i_2})} := \omega \cdot \mathcal{DER}(\text{params}^{(\omega)}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$; and finally it computes message $\mathbf{M}^{(u_{i_2})} := \varrho \cdot \mathcal{D}(\text{params}^{(\varrho)}, \mathbf{K}^{(u_{i_2})}, \mathbf{C}^{(u_{i_2})}, \mathbf{T}^{(u_{i_2})})$.

5.6.1.2 Security of $\bar{\psi}$

Theorem 12. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\bar{\psi}$ has been defined in Section 5.6.1.1. For all poly-time KASAE-KR adversaries \mathcal{A} against $\bar{\psi}$, there exists a poly-time KAS-KR adversary \mathcal{B} against KAS-chain scheme ω , such that:

$$\text{Adv}_{\bar{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \text{Adv}_{\omega, \mathcal{B}, G}^{\text{KAS-KR}}(1^\lambda).$$

Here, the KASAE-KR game is defined in Figure 5.1, and the KAS-KR game is defined in Figure 3.28.

Proof. We poly-reduce any KASAE-KR adversary \mathcal{A} against $\bar{\psi}$ into a KAS-KR adversary \mathcal{B} against ω . The proof readily follows from it. ■

Theorem 13. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\bar{\psi}$ has been defined in Section 5.6.1.1. For all poly-time KASAE-IND adversaries \mathcal{A} against $\bar{\psi}$, there exists a poly-time AE-IND adversary \mathcal{B} against AE scheme ϱ , such that:

$$\text{Adv}_{\bar{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \text{Adv}_{\varrho, \mathcal{B}}^{\text{AE-IND}}(1^\lambda).$$

Here, the KASAE-IND game is defined in Figure 5.2, and the AE-IND game is defined in Figure 2.14.

Proof. We poly-reduce any KASAE-IND adversary \mathcal{A} against $\bar{\psi}$ into an AE-IND adversary \mathcal{B} against ϱ . The proof readily follows from it. ■

Theorem 14. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\vec{\psi}$ has been defined in Section 5.6.1.1. For all poly-time KASAE-INT adversaries \mathcal{A} against $\vec{\psi}$, there exists a poly-time AE-INT adversary \mathcal{B} against AE scheme ϱ , such that:

$$Adv_{\vec{\psi}, \mathcal{A}, G}^{KASAE-INT}(1^\lambda) \leq Adv_{\varrho, \mathcal{B}}^{AE-INT}(1^\lambda).$$

Here, the KASAE-INT game is defined in Figure 5.3, and the AE-INT game is defined in Figure 2.15.

Proof. We poly-reduce any KASAE-INT adversary \mathcal{A} against $\vec{\psi}$ into an AE-INT adversary \mathcal{B} against ϱ . The proof readily follows from it. ■

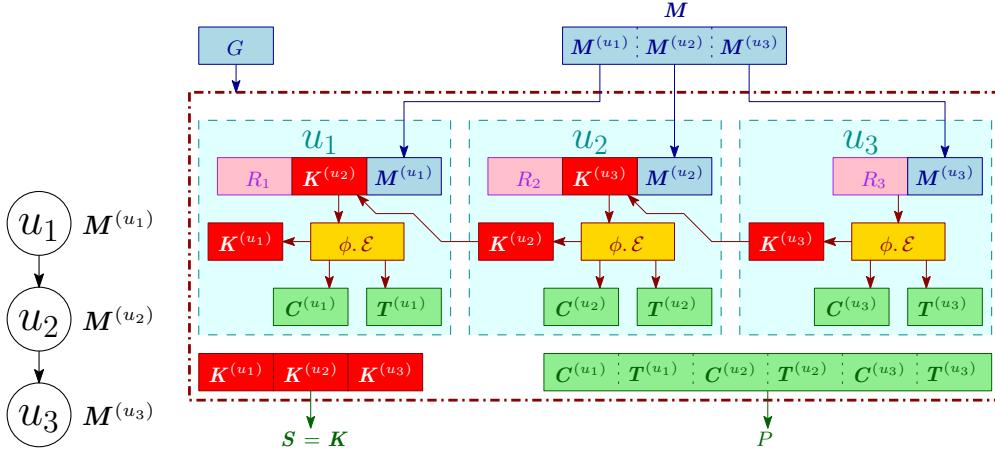
5.6.1.3 Construction $\check{\psi}$: Based on MLE

The KAS-AE-chain construction $\check{\psi}$ is built using an MLE scheme $\phi = (\phi, \mathcal{E}, \phi, \mathcal{D})$ over ϕ . **Setup.** Here, the core idea is: for every node, the key of its *immediate successor* node is prepended to the file, and then this new file is encrypted using MLE to generate key, ciphertext and tag. Here, *private information* for each user is the *key* itself; ciphertexts and tags are a part of *public information*.

Description of $\check{\psi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\check{\psi}$ are given in Figures 5.19, 5.20, 5.21, 5.22, 5.23 and 5.24. Below, we give the textual description.

- The *setup* algorithm $\check{\psi}.Setup(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\check{\psi})}$, a set of *access graphs* $\Gamma^{(\check{\psi})}$, *key-space* $\mathcal{K}^{(\check{\psi})}$ and *message-space* $\mathcal{M}^{(\check{\psi})}$. The $\check{\psi}.Setup(1^\lambda)$ is designed in the following way: first it invokes $\phi.Setup(1^\lambda)$; then it computes $params^{(\check{\psi})}$ that includes, among others, $params^{(\phi)}$; and finally it computes *key-space* $\mathcal{K}^{(\check{\psi})} = \mathcal{K}^{(\phi)}$, and *message-space* $\mathcal{M}^{(\check{\psi})} = \mathcal{M}^{(\phi)}$.
- The *encryption* algorithm $\check{\psi}.E(\cdot)$ is randomized that takes as input parameter $params^{(\check{\psi})}$, graph G and vector of *files* \mathbf{M} , and outputs a vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pictorial description and pseudocode are given in Figures 5.19 and 5.20. The algorithm works as follows: first it computes a sequence of vertices $(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G)$, such that $u_m \lessdot u_{m-1}, u_{m-1} \lessdot u_{m-2}, \dots, u_2 \lessdot u_1$; and then it initializes $\mathbf{K}^{(u_{m+1})}$ with empty string ϵ . Now, the ciphertext is computed



(a) The access graph G . (b) Pictorial description of $\check{\psi}. E(\text{params}^{(\check{\psi})}, G, M)$, where G is shown in Figure 5.19(a) and vector of files $M = M^{(u_1)} \circ M^{(u_2)} \circ M^{(u_3)}$.

Figure 5.19: Pictorial description of the *encryption* function $\check{\psi}. E$.

$\check{\psi}. E(\text{params}^{(\check{\psi})}, G, M)$

#Initialization.

$(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G);$
 $M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_m)} := M; \quad K^{(u_{m+1})} := \epsilon;$

#Computing ciphertext and private information for each user u_i .

for ($j := m, m - 1, \dots, 1$)
 $\quad temp \xleftarrow{\$} \{0, 1\}^\lambda; \quad temp := temp \| K^{(u_{j+1})} \| M^{(u_j)};$
 $\quad (K^{(u_j)}, C^{(u_j)}, T^{(u_j)}) := \phi. E(\text{params}^{(\phi)}, temp);$

#Computing final outputs.

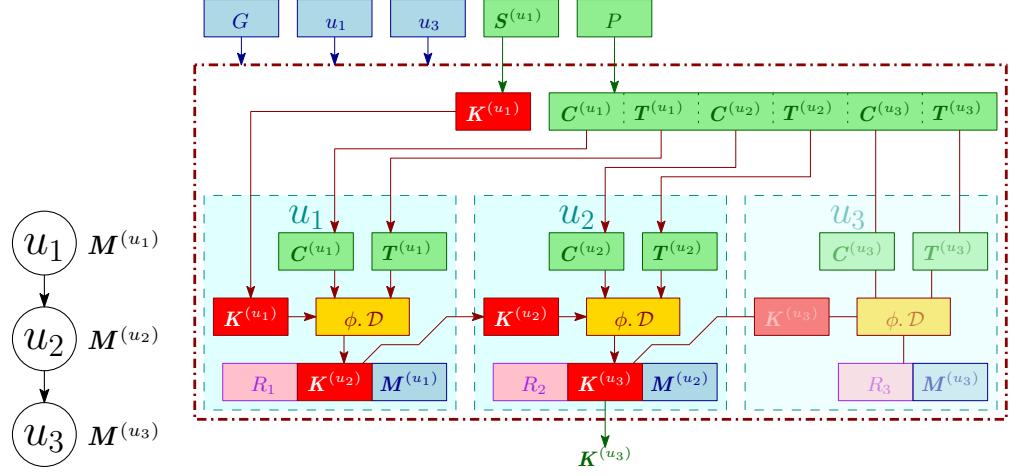
$S := K := K^{(u_1)} \circ K^{(u_2)} \circ \dots \circ K^{(u_m)};$
 $P := C^{(u_1)} \| T^{(u_1)} \circ C^{(u_2)} \| T^{(u_2)} \circ \dots \circ C^{(u_m)} \| T^{(u_m)};$
return $(S, K, P);$

Figure 5.20: Algorithmic description of the *encryption* function $\check{\psi}. E$.

iteratively, as j runs through $m, m - 1, \dots, 1$, in the following way: first it randomly chooses a λ -bit string $temp$; then it updates $temp := temp \| K^{(u_{j+1})} \| M^{(u_j)}$; and finally it computes $(K^{(u_j)}, C^{(u_j)}, T^{(u_j)}) := \phi. E(\text{params}^{(\phi)}, temp)$.

Now, the vectors of *private information* and *keys* $S := K := K^{(u_1)} \circ$

$\mathbf{K}^{(u_2)} \circ \dots \circ \mathbf{K}^{(u_m)}$, and public information $P := \mathbf{C}^{(u_1)} \| \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \| \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} \| \mathbf{T}^{(u_m)}$.



(a) The access graph G . (b) Pictorial description of $\psi.DER(params^{(\check{\psi})}, G, u_1, u_3, S^{(u_1)}, P)$, where G is shown in Figure 5.21(a).

Figure 5.21: Pictorial description of the key derivation function $\psi.DER$.

$\psi.DER(params^{(\check{\psi})}, G, u_{i_1}, u_{i_2}, S^{(u_{i_1})}, P)$

#Initialization.

$$\mathbf{K}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})};$$

If $(u_{i_1} < u_{i_2})$, then return \perp ;

If $(u_{i_1} = u_{i_2})$, then return $\mathbf{K}^{(u_{i_2})} := \mathbf{K}^{(u_{i_1})}$;

Computing key.

$$\mathbf{C}^{(u_1)} \| \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \| \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} \| \mathbf{T}^{(u_m)} := P;$$

for $(j := i_1, i_1 + 1, \dots, i_2 - 1)$

$$temp := \phi.D(params^{(\phi)}, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)});$$

If $(temp = \perp)$, then return \perp ;

Else $temp \| \mathbf{K}^{(u_{j+1})} \| \mathbf{M}^{(u_j)} := temp$;

#Computing final output.

return $\mathbf{K}^{(u_{i_2})}$;

Figure 5.22: Algorithmic description of the key derivation function $\psi.DER$.

- The *key derivation* algorithm $\check{\psi} \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $params^{(\check{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.21 and 5.22. Very briefly, the algorithm works as follows: first it initializes $\mathbf{K}^{(u_{i_1})}$ with $\mathbf{S}^{(u_{i_1})}$; next it returns $\mathbf{K}^{(u_{i_2})} := \mathbf{K}^{(u_{i_1})}$ if $u_{i_1} = u_{i_2}$; and after that it parses P into $\mathbf{C}^{(u_1)} \parallel \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \parallel \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} \parallel \mathbf{T}^{(u_m)} := P$. Now, the key is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, i_2 - 1$, in the following way: first it decrypts $\mathbf{C}^{(u_j)}$ to compute $temp := \phi \cdot \mathcal{D}(params^{(\phi)}, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)})$; and after that it checks if $temp = \perp$, then it returns an *invalid* string \perp , otherwise, it parses $temp$ into $temp \parallel \mathbf{K}^{(u_{j+1})} \parallel \mathbf{M}^{(u_j)} := temp$.

Now, the key $\mathbf{K}^{(u_{i_2})}$, computed above, is returned.

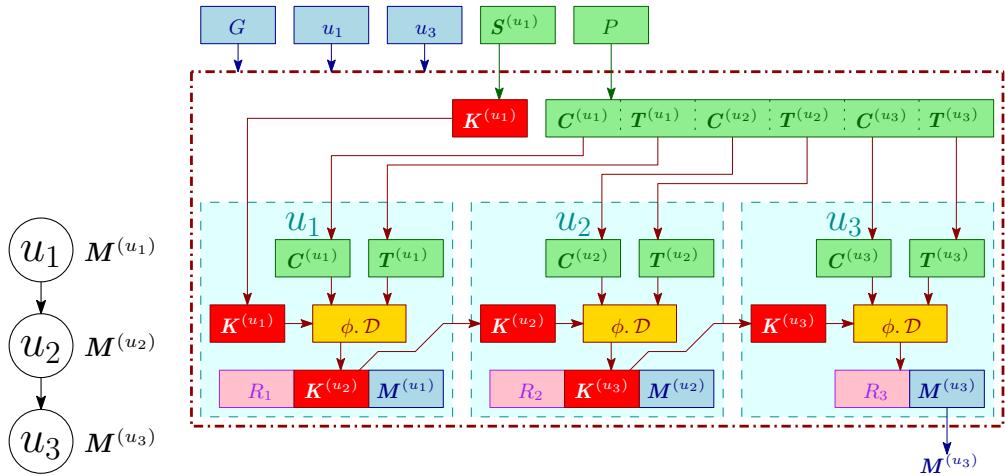


Figure 5.23: Pictorial description of the *decryption* function $\check{\psi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\check{\psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\check{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.23 and 5.24. The algorithm works as follows: first it initializes $\mathbf{K}^{(u_{i_1})}$ with $\mathbf{S}^{(u_{i_1})}$; and then it parses P into $\mathbf{C}^{(u_1)} \parallel \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \parallel \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} \parallel \mathbf{T}^{(u_m)} := P$. Now,

```

 $\check{\psi} \cdot \mathcal{D}(\text{params}^{(\check{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 
#Initialization.
 $\mathbf{K}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})};$ 
If ( $u_{i_1} < u_{i_2}$ ), then return  $\perp$ ;

# Computing file.
 $\mathbf{C}^{(u_1)} \parallel \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \parallel \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} \parallel \mathbf{T}^{(u_m)} := P;$ 
for ( $j := i_1, i_1 + 1, \dots, i_2$ )
     $\text{temp} := \phi \cdot \mathcal{D}(\text{params}^{(\phi)}, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)})$ ;
    If ( $\text{temp} = \perp$ ), then return  $\perp$ ;
    If ( $j = m$ ), then  $\text{temp} \parallel \mathbf{M}^{(u_j)} := \text{temp}$ ;
    Else  $\text{temp} \parallel \mathbf{K}^{(u_{j+1})} \parallel \mathbf{M}^{(u_j)} := \text{temp}$ ;

#Computing final output.
return  $\mathbf{M}^{(u_{i_2})}$ ;

```

Figure 5.24: Algorithmic description of the *decryption* function $\check{\psi} \cdot \mathcal{D}$.

the file is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, i_2$, in the following way: first it decrypts $\mathbf{C}^{(u_j)}$ to compute $\text{temp} := \phi \cdot \mathcal{D}(\text{params}^{(\phi)}, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)})$; after that it checks if $\text{temp} = \perp$, then it returns an *invalid* string \perp ; and finally it checks if $j = m$, then it parses the string temp into $\text{temp} \parallel \mathbf{M}^{(u_j)} := \text{temp}$, otherwise it parses temp into $\text{temp} \parallel \mathbf{K}^{(u_{j+1})} \parallel \mathbf{M}^{(u_j)} := \text{temp}$.

Now, the file $\mathbf{M}^{(u_{i_2})}$, computed above, is returned.

5.6.1.4 Security of $\check{\psi}$

Theorem 15. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\check{\psi}$ has been defined in Section 5.6.1.3. For all poly-time KASAE-KR adversaries \mathcal{A} against $\check{\psi}$, there exists a poly-time MLE-KR adversary \mathcal{B} against MLE scheme ϕ , such that:*

$$\text{Adv}_{\check{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{S}, \mathcal{B}}^{\text{MLE-KR}}(1^\lambda).$$

Here, the KASAE-KR game is defined in Figure 5.1, and the MLE-KR game is defined in Figure 3.15.

Proof. We poly-reduce any KASAE-KR adversary \mathcal{A} against $\check{\psi}$ into a MLE-KR adversary \mathcal{B} against ϕ by making use of a special *unpredictable* message-source $\mathcal{S}_{u_{i_1}, \mathbf{M}}(\cdot)$. The details of the reduction are readily observable from

the Figure 5.25, except the description of $\mathcal{S}_{u_{i_1}, M}(\cdot)$, which we discuss below. In this reduction, the **MLE-KR** adversary \mathcal{B} is constructed using **KASAE-KR** adversary \mathcal{A} .

The $\mathcal{S}_{u_{i_1}, M}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs message M and auxiliary information Z . Here, u_{i_1} is the node (or security class) that \mathcal{A} chooses to attack, M is the vector of *files* that the *KAS-AE-chain* adversary \mathcal{A} generates, and m is the number of nodes in the graph G . The pseudocode for $\mathcal{S}_{u_{i_1}, M}(1^\lambda)$ is given in Figure 5.26. The message source $\mathcal{S}_{u_{i_1}, M}(1^\lambda)$ begins by initializing $\mathbf{K}^{(u_{m+1})}$ with empty string ϵ , and then computes M and Z iteratively, as j runs through $m, m-1, \dots, 1$, in the following way: first it randomly chooses a λ -bit string R ; and then it computes $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)}) := \phi \cdot \mathcal{E}(\text{params}^{(\phi)}, R \| \mathbf{K}^{(u_{j+1})} \| M^{(u_j)})$. Now, the message $M := \phi \cdot \mathcal{D}(\text{params}^{(\phi)}, \mathbf{K}^{(u_{i_1})}, \mathbf{C}^{(u_{i_1})}, \mathbf{T}^{(u_{i_1})})$ and the auxiliary information $Z := \mathbf{C}^{(u_1)} \| \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \| \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_{i_1-1})} \| \mathbf{T}^{(u_{i_1-1})} \circ \mathbf{C}^{(u_{i_1+1})} \| \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{C}^{(u_m)} \| \mathbf{T}^{(u_m)} \circ \mathbf{K}^{(u_{i_1+1})} \circ \mathbf{K}^{(u_{i_1+2})} \circ \dots \circ \mathbf{K}^{(u_m)}$. ■

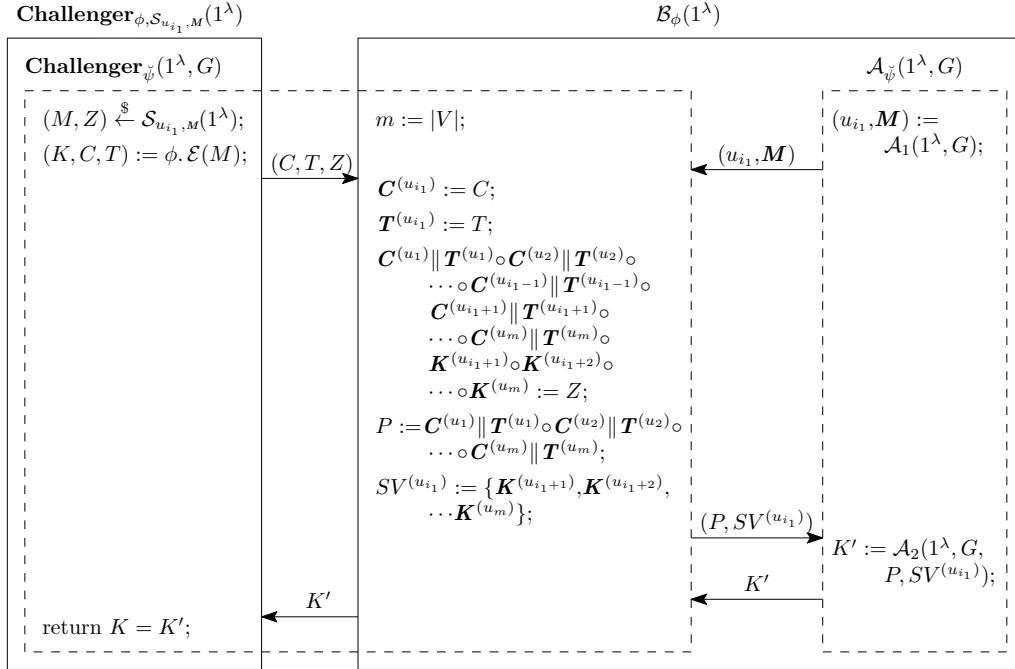


Figure 5.25: Reduction used in Theorem 15.

Theorem 16. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\check{\psi}$ has been defined in Section 5.6.1.3. For all poly-time **KASAE-IND** adversaries \mathcal{A} against $\check{\psi}$, there exists a poly-time **MLE-PRV** adversary \mathcal{B} against MLE scheme ϕ , such that:

```

 $\underline{\mathcal{S}_{u_{i_1}, M}(1^\lambda)}$ 

#Initialization.
 $\mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_m)} := \mathbf{M}; \quad \mathbf{K}^{(u_{m+1})} := \epsilon;$ 

#Computing ciphertext and private information for each user  $u_j$ .
for ( $j := m, m - 1, \dots, 1$ )
     $R \xleftarrow{\$} \{0, 1\}^\lambda;$ 
     $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, \mathbf{T}^{(u_j)}) := \phi. \mathcal{E}(\text{params}^{(\phi)}, R \| \mathbf{K}^{(u_{j+1})} \| \mathbf{M}^{(u_j)});$ 

#Computing final outputs.
 $M := \phi. \mathcal{D}(\text{params}^{(\phi)}, \mathbf{K}^{(u_{i_1})}, \mathbf{C}^{(u_{i_1})}, \mathbf{T}^{(u_{i_1})});$ 
 $Z := \mathbf{C}^{(u_1)} \| \mathbf{T}^{(u_1)} \circ \mathbf{C}^{(u_2)} \| \mathbf{T}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_{i_1-1})} \| \mathbf{T}^{(u_{i_1-1})} \circ$ 
 $\quad \mathbf{C}^{(u_{i_1+1})} \| \mathbf{T}^{(u_{i_1+1})} \circ \dots \circ \mathbf{C}^{(u_m)} \| \mathbf{T}^{(u_m)} \circ \mathbf{K}^{(u_{i_1+1})} \circ$ 
 $\quad \mathbf{K}^{(u_{i_1+2})} \circ \dots \circ \mathbf{K}^{(u_m)};$ 
return ( $M, Z$ );

```

Figure 5.26: Algorithmic description of message-source $\mathcal{S}_{u_{i_1}, M}$ used in Theorem 15.

$$\text{Adv}_{\check{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{S}, \mathcal{B}}^{\text{MLE-PRV}}(1^\lambda).$$

Here, the KASAE-IND game is defined in Figure 5.2, and the MLE-PRV game is defined in Figure 3.12.

Proof. We poly-reduce any KASAE-IND adversary \mathcal{A} against $\check{\psi}$ into a MLE-PRV adversary \mathcal{B} against ϕ by making use of a special unpredictable message-source $\mathcal{S}_{M_0, M_1}(\cdot)$. The details of the reduction are readily observable from the Figure 5.27, except the description of $\mathcal{S}_{M_0, M_1}(\cdot)$, which we discuss below. In this reduction, the MLE-PRV adversary \mathcal{B} is constructed using KASAE-IND adversary \mathcal{A} .

The $\mathcal{S}_{M_0, M_1}(\cdot)$ mimics the functioning of KAS-AE-chain scheme $\check{\psi}$, except that instead of giving $(\mathbf{S}, \mathbf{K}, P)$ as output, it outputs two vectors of files \mathbf{M}_0 and \mathbf{M}_1 , and auxiliary information Z . In \mathcal{S}_{M_0, M_1} , the \mathbf{M}_0 and \mathbf{M}_1 are the two vectors of files that the KAS-AE-chain adversary \mathcal{A} generates, and m is the number of nodes (or security classes) in the graph G .² The pseudocode for $\mathcal{S}_{M_0, M_1}(1^\lambda)$ is given in Figure 5.28. Below we

²The vectors \mathbf{M}_0 and \mathbf{M}_1 generated by the adversary \mathcal{A} are different from the vectors \mathbf{M}_0 and \mathbf{M}_1 output by \mathcal{S}_{M_0, M_1} . Here, the \mathcal{A} generates \mathbf{M}_0 and \mathbf{M}_1 , such that for $i \in [m]$, $|\mathbf{M}_0^{(i)}| = |\mathbf{M}_1^{(i)}|$ (as already defined in Section 5.3.3), which results in

give the textual description. The message source $\mathcal{S}_{M_0, M_1}(1^\lambda)$ begins by initializing both $K_0^{(u_{m+1})}$ and $K_1^{(u_{m+1})}$ with empty string ϵ , and then performs the following operations for the values of $b := \{0, 1\}$ and as j runs through $m, m-1, \dots, 1$: first it randomly chooses a λ -bit string R ; then it updates message $M_b^{(u_j)} := R \| K_b^{(u_{j+1})} \| M_b^{(u_j)}$; and finally it computes $(K_b^{(u_j)}, C_b^{(u_j)}, T_b^{(u_j)}) := \phi \cdot \mathcal{E}(\text{params}(\phi), M_b^{(u_j)})$. Now, the vectors of files $M_0 := M_0^{(1)} \circ M_0^{(2)} \circ \dots \circ M_0^{(m)}$ and $M_1 := M_1^{(1)} \circ M_1^{(2)} \circ \dots \circ M_1^{(m)}$ and the auxiliary information Z , computed above, are returned. ■

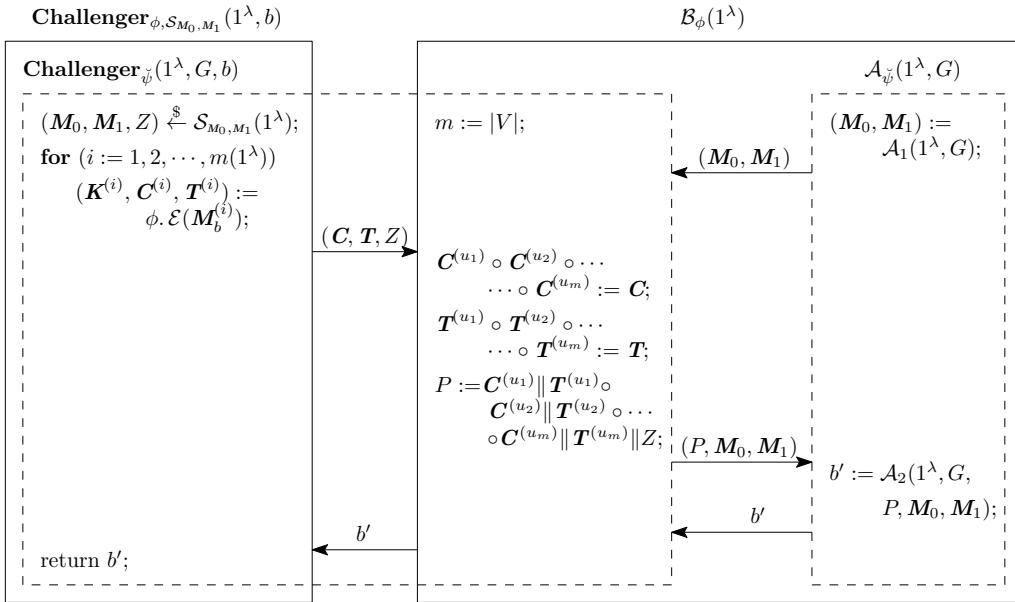


Figure 5.27: Reduction used in Theorem 16.

Theorem 17. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain construction $\check{\psi}$ has been defined in Section 5.6.1.3. For all poly-time KASAE-INT adversaries \mathcal{A} against $\check{\psi}$, there exists a poly-time MLE-TC adversary \mathcal{B} against MLE scheme ϕ , such that:

$$\text{Adv}_{\check{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{B}}^{\text{MLE-TC}}(1^\lambda).$$

Here, the KASAE-INT game is defined in Figure 5.3, and the MLE-TC game is defined in Figure 3.14.

$|M_0^{(i)}| = |M_1^{(i)}|$ when $S_{M_0, M_1}(1^\lambda)$ generates the output M_0 , M_1 and Z .

```

 $\mathcal{S}_{M_0, M_1}(1^\lambda)$ 

#Initialization.
 $M_0^{(u_1)} \circ M_0^{(u_2)} \circ \cdots \circ M_0^{(u_m)} := M_0;$ 
 $M_1^{(u_1)} \circ M_1^{(u_2)} \circ \cdots \circ M_1^{(u_m)} := M_1;$ 
 $K_0^{(u_{m+1})} := K_1^{(u_{m+1})} := \epsilon;$ 

#Computing ciphertext and private information for each user  $u_j$ .
for ( $b := 0, 1$ )
    for ( $j := m, m - 1, \dots, 1$ )
         $R \xleftarrow{\$} \{0, 1\}^\lambda$ ;  $M_b^{(u_j)} := R \| K_b^{(u_{j+1})} \| M_b^{(u_j)}$ ;
         $(K_b^{(u_j)}, C_b^{(u_j)}, T_b^{(u_j)}) := \phi. \mathcal{E}(\text{params}^{(\phi)}, M_b^{(u_j)})$ ;
         $M_b := M_b^{(u_1)} \circ M_b^{(u_2)} \circ \cdots \circ M_b^{(u_m)}$ ;

#Computing final outputs.
return ( $M_0, M_1, Z$ );

```

Figure 5.28: Algorithmic description of message-source \mathcal{S}_{M_0, M_1} used in Theorem 16.

Proof. We poly-reduce any **KASAE-INT** adversary \mathcal{A} against $\check{\psi}$ into an **MLE-TC** adversary \mathcal{B} against ϕ , as shown in Figure 5.29. In this reduction, the **MLE-TC** adversary \mathcal{B} is constructed using **KASAE-INT** adversary \mathcal{A} . The proof readily follows from it. ■

5.6.1.5 Construction $\dot{\psi}$: Based on **APE**

In this section, we describe a *KAS-AE-chain* construction $\dot{\psi}$ from two functions $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$, whose designs are inspired from the *encryption* and *decryption* functions of the **APE** authenticated encryption. The main idea is to spontaneously generate decryption key of the *immediate successor* node, while decrypting ciphertext of a user. To achieve this, we design $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$ satisfying the following conditions: $\mathcal{F}_1^\pi(\cdot)$ encrypts key of the *immediate successor* node along with the file of the node; and, given a ciphertext and corresponding key, $\mathcal{F}_2^\pi(\cdot)$ generates the key for its *immediate successor* node along with the file. For satisfying these constraints, we exploit the unique *reverse decryption* property of **APE**.

Throughout the section, we assume that the message-length in bits is a multiple of security parameter λ .

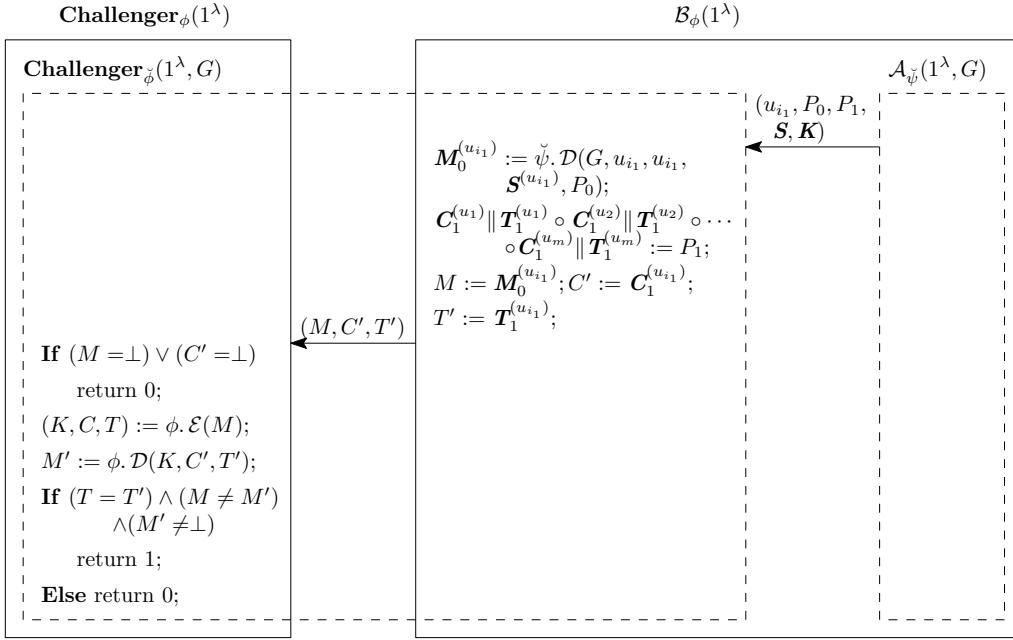


Figure 5.29: Reduction used in Theorem 17.

Functions based on *APE*

In this section, we design two functions – $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$ – that are motivated by the encryption and decryption of authenticated encryption algorithm *APE* [ABB⁺14]. At the outset, we want to remark that the *APE* variant used by us is slightly different from the original *APE* construction by Andreeva *et al.* The main difference is: in the original *APE*, the secret key is applied both in the beginning, as well as, at the end of the encryption process; however, in the *APE* variant used by us, these two operations are completely eliminated (pictorial description can be found in Figures 2.17 and 5.30). This, among others, justifies, to some extent, why our construction supports *keyless* encryption. The pictorial description and pseudocode for the functions $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$ are given in Figures 5.30, 5.31, 5.32 and 5.33. Below, we give the textual description.

- The function $\mathcal{F}_1^\pi(\cdot)$ is a randomized algorithm that takes as input security parameter $\lambda \in \mathbb{N}$, message M and two initialization values IV_1 and IV_2 , and outputs key K and ciphertext C . The pictorial description and pseudocode are given in Figures 5.30 and 5.31. The algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then it initializes the strings r and s with IV_1 and IV_2 ; and finally it randomly chooses a λ -bit string

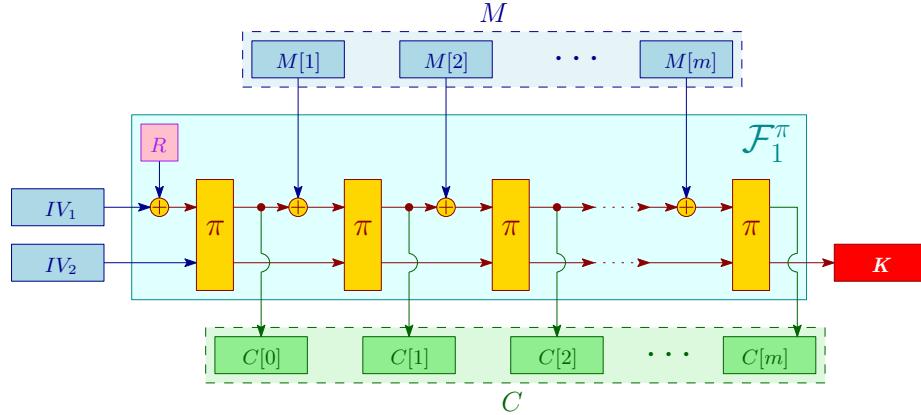


Figure 5.30: Pictorial description of the function \mathcal{F}_1^π .

$\mathcal{F}_1^\pi(1^\lambda, M, IV_1, IV_2)$

#Initialization.

$m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$
 $r := IV_1; \quad s := IV_2; \quad M[0] \xleftarrow{\$} \{0, 1\}^\lambda;$

#Processing the blocks.

for ($j := 0, 1, \dots, m$)
 $r\|s := \pi((r \oplus M[j])\|s); \quad C[j] := r;$

#Computing final output.

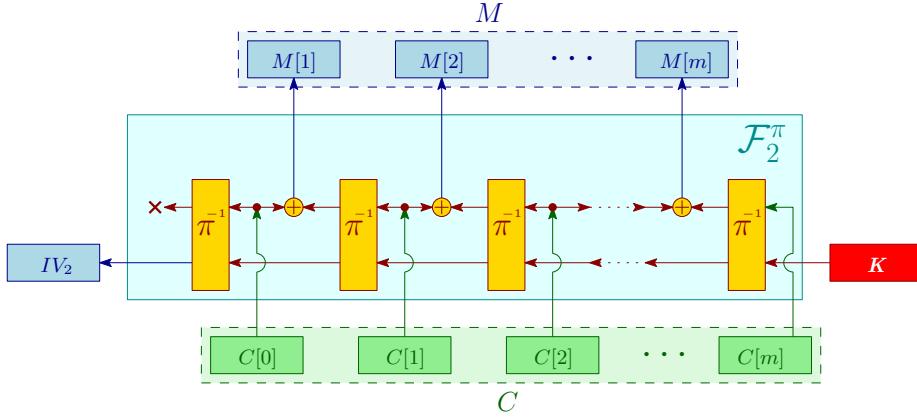
$K := s; \quad C := C[0]\|C[1]\|\cdots\|C[m];$
return (K, C);

Figure 5.31: Algorithmic description of the function \mathcal{F}_1^π .

$M[0]$. Now, the ciphertext is computed iteratively, as j runs through $0, 1, \dots, m$, in the following way: first it computes $r\|s := \pi((r \oplus M[j])\|s)$; and then it assigns $C[j] := r$.

Now, the key $K := s$ and the ciphertext $C := C[0]\|C[1]\|\cdots\|C[m]$.

- The function $\mathcal{F}_2^\pi(\cdot)$ is a deterministic algorithm that takes as input security parameter $\lambda \in \mathbb{N}$, key K and ciphertext C , and outputs message M and value IV_2 . The pictorial description and pseudocode are given in Figures 5.32 and 5.33. The algorithm works as follows: first it parses C into λ -bit blocks $C[0]\|C[1]\|\cdots\|C[m] := C$; and initializes the string s with key K . Now, the message is computed iteratively, as j runs through $m, m - 1, \dots, 0$, in the following way:

Figure 5.32: Pictorial description of the function \mathcal{F}_2^π .

```

 $\mathcal{F}_2^\pi(1^\lambda, K, C)$ 
#Initialization.
 $m := (|C|/\lambda) - 1; \quad C[0]\|C[1]\|\cdots\|C[m] := C; \quad s := K;$ 

```

#Processing the blocks.

```

for ( $j := m, m - 1, \dots, 0$ )
   $(r\|s) := \pi^{-1}(C[j]\|s);$ 
  If ( $j \neq 0$ ), then  $M[j] := r \oplus C[j - 1];$ 

```

#Computing final output.

```

 $M := M[1]\|M[2]\|\cdots\|M[m]; \quad IV_2 := s;$ 
return  $(M, IV_2);$ 

```

Figure 5.33: Algorithmic description of the function \mathcal{F}_2^π .

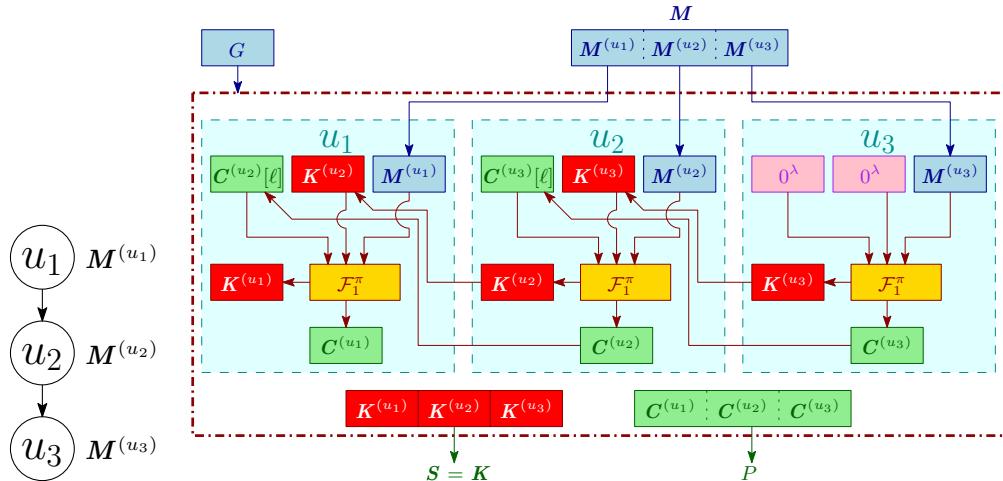
first it computes $r\|s := \pi^{-1}(C[j]\|s)$; and then it checks if $j \neq 0$, then it computes the message $M[j] := r \oplus C[j - 1]$.

Now, the message $M := M[1]\|M[2]\|\cdots\|M[m]$ and the variable $IV_2 := s$.

Description of $\dot{\psi}$

The *KAS-AE-chain* construction $\dot{\psi} = (\dot{\psi}, \mathcal{E}, \dot{\psi}, \mathcal{DER}, \dot{\psi}, \mathcal{D})$ over $\dot{\psi}$. **Setup** is built from the functions $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$. The pictorial description and pseudocode for the 3-tuple of algorithms in $\dot{\psi}$ are given in Figures 5.34, 5.35, 5.36, 5.37, 5.38 and 5.39. Below, we give the textual description.

- The *setup* algorithm $\dot{\psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\dot{\psi})}$, a set of *access graphs* $\Gamma^{(\dot{\psi})}$, *key-space* $\mathcal{K}^{(\dot{\psi})} := \{0, 1\}^\lambda$ and *message-space* $\mathcal{M}^{(\dot{\psi})} := \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$.



(a) The access graph G is shown in Figure 5.34(a).
(b) Pictorial description of $\dot{\psi}.\mathcal{E}(\text{params}^{(\dot{\psi})}, G, \mathbf{M})$, where G is shown in Figure 5.34(a) and vector of files $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \mathbf{M}^{(u_3)}$.

Figure 5.34: Pictorial description of the *encryption* function $\dot{\psi}.\mathcal{E}$.

$\dot{\psi}.\mathcal{E}(\text{params}^{(\dot{\psi})}, G, \mathbf{M})$

#Initialization.

$(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G);$
 $\mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_m)} := \mathbf{M}; \quad IV_1 := IV_2 := 0^\lambda;$

#Computing ciphertext and private information for each user u_j .

for $(j := m, m - 1, \dots, 1)$
 $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}) := \mathcal{F}_1^\pi(1^\lambda, \mathbf{M}^{(u_j)}, IV_1, IV_2);$
 $IV_1 := \mathbf{C}^{(u_j)}[\text{last_block}]; \quad IV_2 := \mathbf{K}^{(u_j)};$

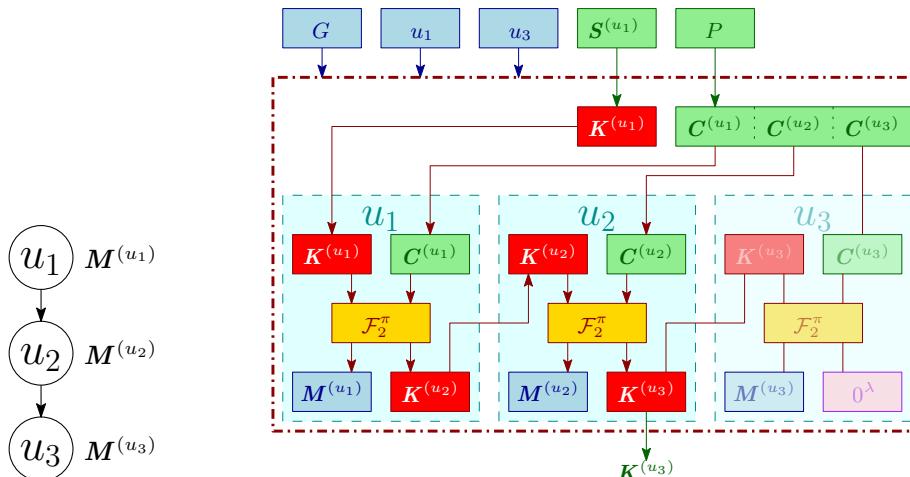
#Computing final outputs.

$\mathbf{S} := \mathbf{K} := \mathbf{K}^{(u_1)} \circ \mathbf{K}^{(u_2)} \circ \dots \circ \mathbf{K}^{(u_m)};$
 $P := \mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)};$
return $(\mathbf{S}, \mathbf{K}, P);$

Figure 5.35: Algorithmic description of the *encryption* function $\dot{\psi}.\mathcal{E}$.

- The *encryption* algorithm $\psi \cdot \mathcal{E}(\cdot)$ is randomised that takes as input parameter $params^{(\psi)}$, graph G and vector of files \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pictorial description and pseudocode are given in Figures 5.34 and 5.35. The algorithm works as follows: first it computes a sequence of vertices $(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G)$, such that $u_m \leq u_{m-1}, u_{m-1} \leq u_{m-2}, \dots, u_2 \leq u_1$; and then it initializes both the strings IV_1 and IV_2 with 0^λ . Now, the ciphertext is computed iteratively, as j runs through $m, m-1, \dots, 1$, in the following way: first, it computes a pair of key and ciphertext $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}) := \mathcal{F}_1^{\pi}(1^\lambda, \mathbf{M}^{(u_j)}, IV_1, IV_2)$; then it computes $IV_1 := \mathbf{C}^{(u_j)}[\text{last_block}]$; and finally it assigns $IV_2 := \mathbf{K}^{(u_j)}$.

Now, the vectors of *private information* and *keys* $\mathbf{S} := \mathbf{K} := \mathbf{K}^{(u_1)} \circ \mathbf{K}^{(u_2)} \circ \dots \circ \mathbf{K}^{(u_m)}$, and the *public information* $P := \mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}$.



(a) The access (b) Pictorial description of $\psi \cdot \mathcal{DERR}(params^{(\psi)}, G, u_1, u_3, \mathbf{S}^{(u_1)}, P)$ and graph G . G is shown in Figure 5.36(a).

Figure 5.36: Pictorial description of the *key derivation* function $\psi \cdot \mathcal{DERR}$.

- The *key derivation* algorithm $\psi \cdot \mathcal{DERR}(\cdot)$ is deterministic that takes as input parameter $params^{(\psi)}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.36 and 5.37. Very briefly, the algorithm works as follows: first it initializes $\mathbf{K}^{(u_{i_1})}$ with

```

 $\dot{\psi} \cdot \mathcal{DER}(\text{params}^{(\dot{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $\mathbf{K}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})};$ 
If ( $u_{i_1} < u_{i_2}$ ), then return  $\perp$ ;
If ( $u_{i_1} = u_{i_2}$ ), then return  $\mathbf{K}^{(u_{i_2})} := \mathbf{K}^{(u_{i_1})};$ 

# Computing key.
 $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P;$ 
for ( $j := i_1, i_1 + 1, \dots, i_2 - 1$ )
   $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)})$ ;

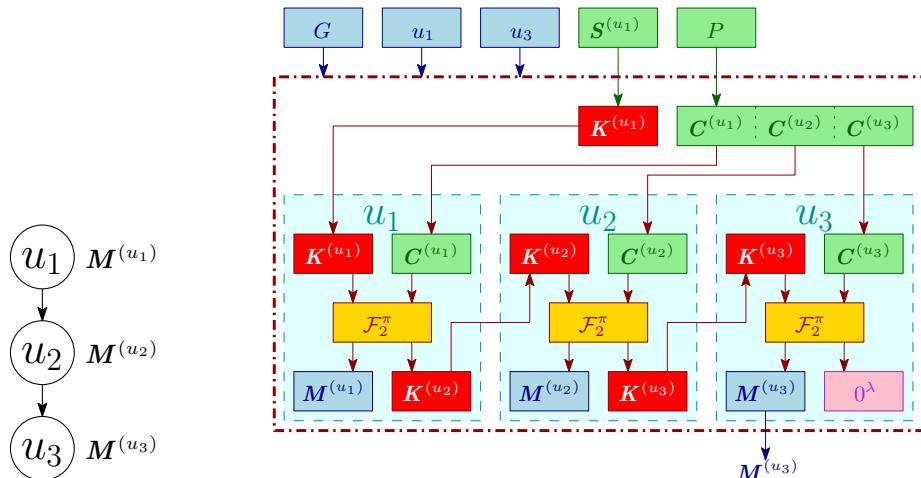
#Computing final output.
return  $\mathbf{K}^{(u_{i_2})};$ 

```

Figure 5.37: Algorithmic description of the *key derivation* function $\dot{\psi} \cdot \mathcal{DER}$.

$\mathbf{S}^{(u_{i_1})}$; then it checks if $u_{i_1} = u_{i_2}$, then returns $\mathbf{K}^{(u_{i_2})} := \mathbf{K}^{(u_{i_1})}$; and finally it parses P into $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P$. Now, the key is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, i_2 - 1$, by computing $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)})$.

The key $\mathbf{K}^{(u_{i_2})}$, computed in the last iteration, is returned.



(a) The access graph G . (b) Pictorial description of $\dot{\psi} \cdot \mathcal{D}(\text{params}^{(\dot{\psi})}, G, u_1, u_3, \mathbf{S}^{(u_1)}, P)$ and G is shown in Figure 5.38(a).

Figure 5.38: Pictorial description of the *decryption* function $\dot{\psi} \cdot \mathcal{D}$.

```

 $\dot{\psi} \cdot \mathcal{D}(params^{(\dot{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $\mathbf{K}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})};$ 
If  $(u_{i_1} < u_{i_2})$ , then return  $\perp$ ;

# Computing file.
 $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P;$ 
for  $(j := i_1, i_1 + 1, \dots, m)$ 
 $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)})$ ;

#Computing final output.
If  $(\mathbf{K}^{(u_{m+1})} = 0^\lambda)$ , then return  $\mathbf{M}^{(u_{i_2})}$ ; Else return  $\perp$ ;

```

Figure 5.39: Algorithmic description of the *decryption* function $\dot{\psi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\dot{\psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\dot{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.38 and 5.39. The algorithm works as follows: first it initializes $\mathbf{K}^{(u_{i_1})}$ with $\mathbf{S}^{(u_{i_1})}$; and then it parses P into $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P$. Now, the file is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, m$, by computing $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)})$. If $\mathbf{K}^{(u_{m+1})} = 0^\lambda$, then it returns message $\mathbf{M}^{(u_{i_2})}$; otherwise it returns *invalid* string \perp .

5.6.1.6 Security of $\dot{\psi}$

Theorem 18. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\dot{\psi}$ has been defined in Section 5.6.1.5. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\dot{\psi}, \mathcal{A}, G}^{KASAE-KR}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)+1}{2^\lambda}.$$

Here, the KASAE-KR game is defined in Figure 5.1, and the message-length in bits is $m\lambda$.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $KASAE-KR_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G)$; and our successive games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$. Then,

we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} Adv_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) &\stackrel{\text{def}}{=} \Pr[\text{KASAE-KR}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned} Adv_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

Therefore, $Adv_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4$, (5.1)

$$\begin{aligned} \text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\ \Delta_4 &\stackrel{\text{def}}{=} \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

In the following, we compute Δ_1 , Δ_2 , Δ_3 and Δ_4 (see Figure 5.40 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\dot{\psi}^{(1)}$ instead of $\dot{\psi}$. In $\dot{\psi}^{(1)}$, the function \mathcal{F}_1^π is replaced by $\mathcal{F}_1^{\text{rf}}$, that is, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_1 \leq \frac{m(m-1)}{2^{2\lambda}}. (5.2)$$

Game G_2 : This game is identical to **Game G_1** , except that it uses $\dot{\psi}^{(2)}$ instead of $\dot{\psi}^{(1)}$. In $\dot{\psi}^{(2)}$, the function \mathcal{F}_2^π is replaced by $\mathcal{F}_2^{\text{rf}'}$, that is, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_2 \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (5.3)$$

Game G_3 : This game is identical to **Game G_2** , except that it uses $\dot{\psi}^{(3)}$ instead of $\dot{\psi}^{(2)}$. The $\dot{\psi}^{(3)}$ uses $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ and $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$ instead of $\mathcal{F}_1^{\text{rf}}$ and $\mathcal{F}_2^{\text{rf}'}$ (as described in Figure 5.41), where, the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf of $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ or rf' of $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$. From Figure 5.41, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_3 \leq \frac{m(m-1)}{2^\lambda}. \quad (5.4)$$

In **Game G_3** , we observe that there is no information about the secret key in the adversary's view. Thus, the probability of correct guess of the key is $1/2^\lambda$, and we obtain:

$$\Delta_4 = \frac{1}{2^\lambda}. \quad (5.5)$$

Using (5.1) – (5.5), we get:

$$\begin{aligned} \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 \\ &\leq \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^\lambda} + \frac{1}{2^\lambda} \\ &\leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)+1}{2^\lambda}. \end{aligned}$$

■

Theorem 19. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\dot{\psi}$ has been defined in Section 5.6.1.5. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-2}} + \frac{m(m-1)}{2^{\lambda-1}}.$$

Here, the *KASAE-IND* game is defined in Figure 5.2, and the message-length in bits is $m\lambda$.

Game $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$
$(\text{params}^{(\dot{\psi})}, \Gamma^{(\dot{\psi})}, \mathcal{K}^{(\dot{\psi})}, \mathcal{M}^{(\dot{\psi})}) := \dot{\psi}.\text{Setup}(1^\lambda);$
If ($G \notin \Gamma^{(\dot{\psi})}$), then return Error ;
$(u_{i_1}, \mathbf{M}) := \mathcal{A}_1(1^\lambda, G);$
$(\mathbf{S}, \mathbf{K}, P) := \dot{\psi}.\mathcal{E}(\text{params}^{(\dot{\psi})}, G, \mathbf{M});$
$SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} u_{i_2} < u_{i_1}\};$
$K' := \mathcal{A}_2(1^\lambda, G, P, SV^{(u_{i_1})});$
return ($\mathbf{K}^{(u_{i_1})} = K'$);
Game $\mathbf{G}_i(\mathcal{A}, 1^\lambda)$
$(\text{params}^{(\dot{\psi}^{(i)})}, \Gamma^{(\dot{\psi}^{(i)})}, \mathcal{K}^{(\dot{\psi}^{(i)})}, \mathcal{M}^{(\dot{\psi}^{(i)})}) := \dot{\psi}^{(i)}.\text{Setup}(1^\lambda);$
If ($G \notin \Gamma^{(\dot{\psi}^{(i)})}$), then return Error ;
$(u_{i_1}, \mathbf{M}) := \mathcal{A}_1(1^\lambda, G);$
$(\mathbf{S}, \mathbf{K}, P) := \dot{\psi}^{(i)}.\mathcal{E}(\text{params}^{(\dot{\psi}^{(i)})}, G, \mathbf{M});$
$SV^{(u_{i_1})} := \{\mathbf{S}^{(u_{i_2})} \in \mathbf{S} u_{i_2} < u_{i_1}\};$
$K' := \mathcal{A}_2(1^\lambda, G, P, SV^{(u_{i_1})});$
return ($\mathbf{K}^{(u_{i_1})} = K'$);

Figure 5.40: Games used in the proof of Theorem 18.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{KASAE-IND}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G, b = 1)$; our last game is $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$, which is identical to $\text{KASAE-IND}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G, b = 0)$; and our intermediate game is $\mathbf{G}_R(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} Adv_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{KASAE-IND}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{KASAE-IND}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G, b = 0) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned} Adv_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

$\mathcal{F}_1^{\text{rf}}(1^\lambda, M, IV_1, IV_2)$	$\mathcal{F}_2^{\text{rf}'}(1^\lambda, K, C)$
$\boxed{\mathcal{F}_1^{\text{rf}, \text{bad}_1}(1^\lambda, M, IV_1, IV_2)}$	$\boxed{\mathcal{F}_2^{\text{rf}', \text{bad}_1}(1^\lambda, K, C)}$
#Initialization.	#Initialization.
$m := M /\lambda; \quad M[0] \xleftarrow{\$} \{0,1\}^\lambda;$ $M[1]\ M[2]\ \cdots\ M[m] := M;$ $r := IV_1; \quad s := IV_2;$ $U := \emptyset; \quad \text{bad}_1 := 0;$	$m := (C /\lambda) - 1; \quad s := K;$ $C[0]\ C[1]\ \cdots\ C[m] := C;$ $U := \emptyset; \quad \text{bad}_1 := 0;$
#Processing the blocks.	#Processing the blocks.
for ($j := 0, 1, \dots, m$) If ($s \in U$), then $\boxed{\text{bad}_1 := 1;}$ $U := U \cup \{s\};$ $r\ s := \text{rf}((r \oplus M[j])\ s);$ $C[j] := r;$	for ($j := m, m-1, \dots, 0$) If ($s \in U$), then $\boxed{\text{bad}_1 := 1;}$ $U := U \cup \{s\};$ $(r\ s) := \text{rf}'(C[j]\ s);$ If ($j \neq 0$) $M[j] := r \oplus C[j-1];$
#Computing final output.	#Computing final output.
$K := s;$ $C := C[0]\ C[1]\ \cdots\ C[m];$ return (K, C);	$M := M[1]\ M[2]\ \cdots\ M[m];$ $IV_2 := s;$ return (M, IV_2);

Figure 5.41: Algorithmic description of $\dot{\psi}^{(2)}$ and $\dot{\psi}^{(3)}$.

$$\text{Therefore, } \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \Delta_1 + \Delta_2, \quad (5.6)$$

where, $\Delta_1 \stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right|$, and
 $\Delta_2 \stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|$.

In the following, we compute Δ_1 and Δ_2 , (see Figure 5.42 for the algorithmic description of all the games).

Game \mathbf{G}_R : This game is identical to **Game \mathbf{G}_S** , except that it computes the ciphertext $\mathbf{C}^{(u_i)}$ for each node u_i as a random string of length $(|\mathbf{M}^{(u_i)}| + \lambda)$ instead of execution of $\dot{\psi} \cdot \mathcal{E}(\cdot)$.

Computation of Δ_1 :

To compute the value of Δ_1 , we design a series of games (or hybrids): our starting and last games are $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_R(\mathcal{A}, 1^\lambda)$ respectively (as described above); and our intermediate games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$

and $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the value of Δ_1 :

$$\begin{aligned}\Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right|.\end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned}\Delta_1 &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right|.\end{aligned}$$

$$\text{Therefore, } \Delta_1 \leq \Delta_a + \Delta_b + \Delta_c + \Delta_d, \tag{5.7}$$

$$\begin{aligned}\text{where, } \Delta_a &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_b &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_c &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\ \Delta_d &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_R(\mathcal{A}, 1^\lambda) = 1] \right|.\end{aligned}$$

In the following, we compute Δ_a , Δ_b , Δ_c and Δ_d (see Figure 5.42 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\dot{\psi}^{(1)}$ instead of $\dot{\psi}$. In $\dot{\psi}^{(1)}$, the function \mathcal{F}_1^π is replaced by $\mathcal{F}_1^{\text{rf}}$, that is, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_a \leq \frac{m(m-1)}{2^{2\lambda}}. \tag{5.8}$$

Game \mathbf{G}_2 : This game is identical to **Game \mathbf{G}_1** , except that it uses $\dot{\psi}^{(2)}$ instead of $\dot{\psi}^{(1)}$. In $\dot{\psi}^{(2)}$, the function \mathcal{F}_2^π is replaced by $\mathcal{F}_2^{\text{rf}'}$, that is, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore,

by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_b \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (5.9)$$

Game \mathbf{G}_3 : This game is identical to **Game \mathbf{G}_2** , except that it uses $\dot{\psi}^{(3)}$ instead of $\dot{\psi}^{(2)}$. The $\dot{\psi}^{(3)}$ uses $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ or $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$ (as described in Figure 5.41), where, the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf of $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ or rf' of $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$. From Figure 5.41, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_c \leq \frac{m(m-1)}{2^\lambda}. \quad (5.10)$$

Since, $\dot{\psi}^{(3)}. \mathcal{E}(\text{params}^{(\dot{\psi}^{(3)})}, G, \mathbf{M})$ is uniformly distributed over $(\{0, 1\}^{|\mathcal{C}_0^{u_i}|})_{u_i \in V}$, the two games \mathbf{G}_3 and \mathbf{G}_R are indistinguishable. Hence, we obtain:

$$\Delta_d = 0. \quad (5.11)$$

Using (5.7) – (5.11), we get:

$$\begin{aligned} \Delta_1 &\leq \Delta_a + \Delta_b + \Delta_c + \Delta_d \\ &\leq \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^\lambda} + 0 \\ &\leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}. \end{aligned} \quad (5.12)$$

Similarly, we compute:

$$\Delta_2 \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}. \quad (5.13)$$

Using (5.6), (5.12) and (5.13), we get:

$$\begin{aligned} \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) &\leq \Delta_1 + \Delta_2 \\ &\leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda} + \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda} \\ &\leq \frac{m(m-1)}{2^{2\lambda-2}} + \frac{m(m-1)}{2^{\lambda-1}}. \end{aligned}$$

■

Game G_S ($\mathcal{A}, 1^\lambda$)

$(params^{(\dot{\psi})}, \Gamma^{(\dot{\psi})}, \mathcal{K}^{(\dot{\psi})}, \mathcal{M}^{(\dot{\psi})}) := \dot{\psi}. \text{Setup}(1^\lambda);$
If $(G \notin \Gamma^{(\dot{\psi})})$, **then** return **Error**;
 $(\mathbf{M}_0, \mathbf{M}_1) := \mathcal{A}_1(1^\lambda, G);$
 $(\mathbf{S}, \mathbf{K}, P) := \dot{\psi}. \mathcal{E}(params^{(\dot{\psi})}, G, \mathbf{M}_1);$
 $b' := \mathcal{A}_2(1^\lambda, G, P, \mathbf{M}_0, \mathbf{M}_1);$
return b' ;

Game G_R ($\mathcal{A}, 1^\lambda$)

$(params^{(\dot{\psi})}, \Gamma^{(\dot{\psi})}, \mathcal{K}^{(\dot{\psi})}, \mathcal{M}^{(\dot{\psi})}) := \dot{\psi}. \text{Setup}(1^\lambda);$
If $(G \notin \Gamma^{(\dot{\psi})})$, **then** return **Error**;
 $(\mathbf{M}_0, \mathbf{M}_1) := \mathcal{A}_1(1^\lambda, G);$
for $(j := m, m - 1, \dots, 1)$
 If $(|\mathbf{M}_0^{(u_j)}| \neq |\mathbf{M}_1^{(u_j)}|)$, **then** return **Error**;
 $\mathbf{C}^{(u_j)} \xleftarrow{\$} \{0, 1\}^{|\mathbf{M}_0^{(u_j)}| + \lambda};$
 $P := \mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)};$
 $b' := \mathcal{A}_2(1^\lambda, G, P, \mathbf{M}_0, \mathbf{M}_1);$
return b' ;

Game G_L ($\mathcal{A}, 1^\lambda$)

$(params^{(\dot{\psi})}, \Gamma^{(\dot{\psi})}, \mathcal{K}^{(\dot{\psi})}, \mathcal{M}^{(\dot{\psi})}) := \dot{\psi}. \text{Setup}(1^\lambda);$
If $(G \notin \Gamma^{(\dot{\psi})})$, **then** return **Error**;
 $(\mathbf{M}_0, \mathbf{M}_1) := \mathcal{A}_1(1^\lambda, G);$
 $(\mathbf{S}, \mathbf{K}, P) := \dot{\psi}. \mathcal{E}(params^{(\dot{\psi})}, G, \mathbf{M}_0);$
 $b' := \mathcal{A}_2(1^\lambda, G, P, \mathbf{M}_0, \mathbf{M}_1);$
return b' ;

Game G_i ($\mathcal{A}, 1^\lambda$)

$(params^{(\dot{\psi}^{(i)})}, \Gamma^{(\dot{\psi}^{(i)})}, \mathcal{K}^{(\dot{\psi}^{(i)})}, \mathcal{M}^{(\dot{\psi}^{(i)})}) := \dot{\psi}^{(i)}. \text{Setup}(1^\lambda);$
If $(G \notin \Gamma^{(\dot{\psi}^{(i)})})$, **then** return **Error**;
 $(\mathbf{M}_0, \mathbf{M}_1) := \mathcal{A}_1(1^\lambda, G);$
 $(\mathbf{S}, \mathbf{K}, P) := \dot{\psi}^{(i)}. \mathcal{E}(params^{(\dot{\psi}^{(i)})}, G, \mathbf{M}_0);$
 $b' := \mathcal{A}_2(1^\lambda, G, P, \mathbf{M}_0, \mathbf{M}_1);$
return b' ;

Figure 5.42: Games used in the proof of Theorem 19.

Theorem 20. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\dot{\psi}$ has been defined in Section 5.6.1.5. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}.$$

Here, the KASAE-INT game is defined in Figure 5.3, and the message-length in bits is $m\lambda$.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{KASAE-INT}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G)$; and our successive games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) &\stackrel{\text{def}}{=} \Pr[\text{KASAE-INT}_{\dot{\psi}}^{\mathcal{A}}(1^\lambda, G) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

Using the triangle inequality [BR06], we get:

$$\begin{aligned} \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

$$\text{Therefore, } \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4, \quad (5.14)$$

$$\begin{aligned} \text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\ \Delta_4 &\stackrel{\text{def}}{=} \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

In the following, we compute Δ_1 , Δ_2 , Δ_3 and Δ_4 (see Figure 5.43 for the algorithmic description of all the games).

Game G_1 : This game is identical to **Game G_S** , except that it uses $\dot{\psi}^{(1)}$ instead of $\dot{\psi}$. In $\dot{\psi}^{(1)}$, the function \mathcal{F}_1^π is replaced by $\mathcal{F}_1^{\text{rf}}$, that is, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_1 \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (5.15)$$

Game G_2 : This game is identical to **Game G_1** , except that it uses $\dot{\psi}^{(2)}$ instead of $\dot{\psi}^{(1)}$. In $\dot{\psi}^{(2)}$, the function \mathcal{F}_2^π is replaced by $\mathcal{F}_2^{\text{rf}'}$, that is, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_2 \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (5.16)$$

Game G_3 : This game is identical to **Game G_2** , except that it uses $\dot{\psi}^{(3)}$ instead of $\dot{\psi}^{(2)}$. The $\dot{\psi}^{(3)}$ uses $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ and $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$ instead of $\mathcal{F}_1^{\text{rf}}$ and $\mathcal{F}_2^{\text{rf}'}$ (as described in Figure 5.41), where, the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf of $\mathcal{F}_1^{\text{rf}, \text{bad}_1}$ or rf' of $\mathcal{F}_2^{\text{rf}', \text{bad}_1}$. From Figure 5.41, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the vector of files \mathbf{M} (where the length of \mathbf{M} in bits is $m\lambda$), we obtain:

$$\Delta_3 \leq \frac{m(m-1)}{2^\lambda}. \quad (5.17)$$

Since, $\dot{\psi}^{(3)}. \mathcal{E}(\text{params}^{(\dot{\psi}^{(3)})}, G, \mathbf{M})$ is uniformly distributed over $(\{0, 1\}^{|\mathcal{C}_0^{u_i}|})_{u_i \in V}$, the two games **G_3** and **G_R** are indistinguishable. Hence, we obtain:

$$\Delta_4 = 0. \quad (5.18)$$

Using (5.14) – (5.18), we get:

$$\begin{aligned} \text{Adv}_{\dot{\psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 \\ &\leq \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^\lambda} + 0 \\ &\leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}. \end{aligned}$$

■

Game \mathbf{G}_S ($\mathcal{A}, 1^\lambda$)
$(params^{(\dot{\psi})}, \Gamma^{(\dot{\psi})}, \mathcal{K}^{(\dot{\psi})}, \mathcal{M}^{(\dot{\psi})}) := \dot{\psi}. \text{Setup}(1^\lambda);$
If ($G \notin \Gamma^{(\dot{\psi})}$), then return Error;
$(u_{i_1}, P_0, P_1, \mathbf{S}, \mathbf{K}) := \mathcal{A}(1^\lambda, G);$
$\mathbf{M}_0^{(u_{i_1})} := \dot{\psi}. \mathcal{D}(params^{(\dot{\psi})}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_0);$
$\mathbf{M}_1^{(u_{i_1})} := \dot{\psi}. \mathcal{D}(params^{(\dot{\psi})}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_1);$
If ($\mathbf{M}_0^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_1^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_0^{(u_{i_1})} \neq \mathbf{M}_1^{(u_{i_1})}$)
return 1;
Else return 0;
Game \mathbf{G}_i ($\mathcal{A}, 1^\lambda$)
$(params^{(\dot{\psi}^{(i)})}, \Gamma^{(\dot{\psi}^{(i)})}, \mathcal{K}^{(\dot{\psi}^{(i)})}, \mathcal{M}^{(\dot{\psi}^{(i)})}) := \dot{\psi}^{(i)}. \text{Setup}(1^\lambda);$
If ($G \notin \Gamma^{(\dot{\psi}^{(i)})}$), then return Error;
$(u_{i_1}, P_0, P_1, \mathbf{S}, \mathbf{K}) := \mathcal{A}(1^\lambda, G);$
$\mathbf{M}_0^{(u_{i_1})} := \dot{\psi}^{(i)}. \mathcal{D}(params^{(\dot{\psi}^{(i)})}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_0);$
$\mathbf{M}_1^{(u_{i_1})} := \dot{\psi}^{(i)}. \mathcal{D}(params^{(\dot{\psi}^{(i)})}, G, u_{i_1}, u_{i_1}, \mathbf{S}^{(u_{i_1})}, P_1);$
If ($\mathbf{M}_0^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_1^{(u_{i_1})} \neq \perp \wedge \mathbf{M}_0^{(u_{i_1})} \neq \mathbf{M}_1^{(u_{i_1})}$)
return 1;
Else return 0;

Figure 5.43: Games used in the proof of Theorem 20.

5.6.1.7 Construction $\ddot{\psi}$: Based on \mathbf{FP}

In this section, we describe a *KAS-AE-chain* construction $\ddot{\psi}$ from two functions $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$, whose designs are inspired from the \mathbf{FP} hash mode of operation. The main idea is to spontaneously generate the decryption key of the *immediate successor* node, while decrypting ciphertext of a user. To achieve this, we design $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$ satisfying the following conditions: $\mathcal{G}_1^\pi(\cdot)$ encrypts key of the *immediate successor* node along with the file of the node; also, given a ciphertext and corresponding key, $\mathcal{G}_2^\pi(\cdot)$ generates the key for its *immediate successor* node along with the file. As before, here also, we use the *reverse decryption* property of the \mathbf{FP} to achieve these goals.

Throughout the section, we assume that the message-length in bits is a multiple of security parameter λ .

Functions based on FP

In this section, we design two functions – $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$ – that are motivated by FP hash mode of operation [PHG12]. We want to emphasize that FP , being a secure hash function, necessarily needs to be *pre-image resistant* (*i.e.* “one-way”); however, the functions $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$ are cleverly designed in a way that they are inverse of each other, as well as, easy-to-compute. The pictorial description and pseudocode for the functions $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$ are given in Figures 5.44, 5.45, 5.46 and 5.47. Below, we give the textual description.

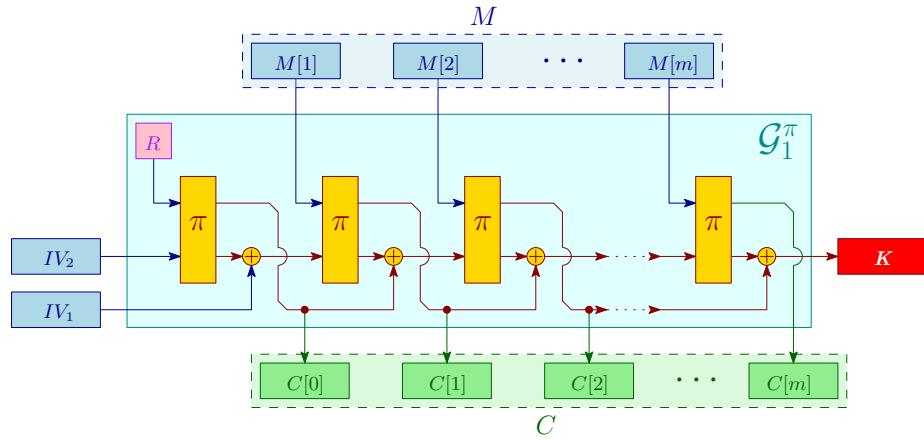


Figure 5.44: Pictorial description of the function \mathcal{G}_1^π .

$\mathcal{G}_1^\pi(1^\lambda, M, IV_1, IV_2)$

#Initialization.

$$m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M; \\ s := IV_2; \quad t := IV_1; \quad M[0] \xleftarrow{\$} \{0, 1\}^\lambda;$$

#Processing the blocks.

$$\text{for } (j := 0, 1, \dots, m) \\ r\|s := \pi(M[j]\|s); \quad s := s \oplus t; \quad C[j] := r; \quad t := r;$$

#Computing final output.

$$K := s; \quad C := C[0]\|C[1]\|\cdots\|C[m]; \\ \text{return } (K, C);$$

Figure 5.45: Algorithmic description of the function \mathcal{G}_1^π .

- The function $\mathcal{G}_1^\pi(\cdot)$ is a randomized algorithm that takes as input security parameter $\lambda \in \mathbb{N}$, message M and two initialization values IV_1 and IV_2 , and outputs key K and ciphertext C . The pictorial description and pseudocode are given in Figures 5.44 and 5.45. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1]||M[2]||\cdots||M[m] := M$; then it initializes the strings s and t with IV_2 and IV_1 ; and finally it randomly chooses a λ -bit string $M[0]$. Now, the ciphertext is computed iteratively, as j runs through $0, 1, \dots, m$, in the following way: first it computes $r||s := \pi(M[j]||s)$; then it updates $s := s \oplus t$; and finally it assigns $C[j] := r$ and $t := r$.

Now, the key $K := s$ and the ciphertext $C := C[0]||C[1]||\cdots||C[m]$.

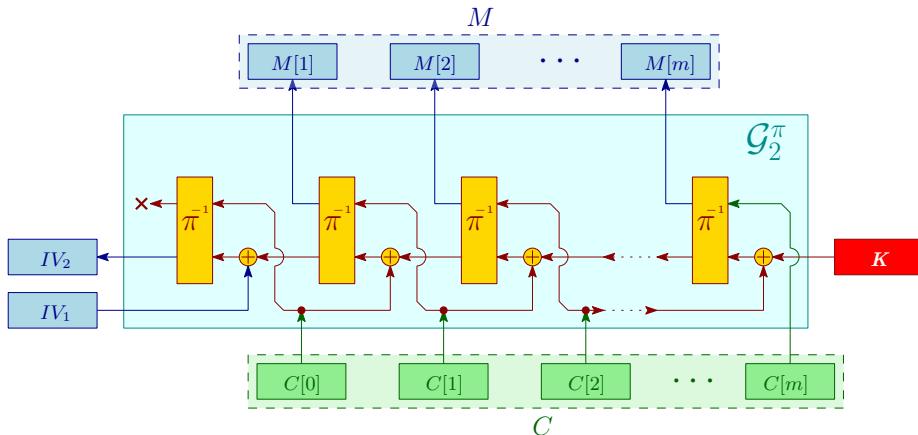


Figure 5.46: Pictorial description of the function \mathcal{G}_2^π .

- The function $\mathcal{G}_2^\pi(\cdot)$ is a deterministic algorithm that takes as input security parameter $\lambda \in \mathbb{N}$, key K , ciphertext C and initialization value IV_1 , and outputs message M and variable IV_2 . The pictorial description and pseudocode are given in Figures 5.46 and 5.47. The algorithm works as follows: first it parses C into λ -bit blocks $C[0]||C[1]||\cdots||C[m] := C$; and then initializes the strings s and t with K and $C[m]$. Now, the message is computed iteratively, as j runs through $m, m-1, \dots, 0$, in the following way: first it assigns $r := t$; then it checks if $j = 0$, then it assigns $t := IV_1$, otherwise, it assigns $t := C[j-1]$; next it updates $s := s \oplus t$; and finally it computes $M[j]||s := \pi^{-1}(r||s)$.

Now, the message $M := M[1]||M[2]||\cdots||M[m]$ and the variable $IV_2 := s$.

```

 $\mathcal{G}_2^\pi(1^\lambda, K, C, IV_1)$ 

#Initialization.
 $m := (|C|/\lambda) - 1; \quad C[0]\|C[1]\|\cdots\|C[m] := C;$ 
 $s := K; \quad t := C[m];$ 

#Processing the blocks.
for ( $j := m, m - 1, \dots, 0$ )
     $r := t;$ 
    If ( $j = 0$ ), then  $t := IV_1$ ; Else  $t := C[j - 1]$ ;
     $s := s \oplus t; \quad M[j]\|s := \pi^{-1}(r\|s);$ 

#Computing final output.
 $M := M[1]\|M[2]\|\cdots\|M[m]; \quad IV_2 := s;$ 
return ( $M, IV_2$ );

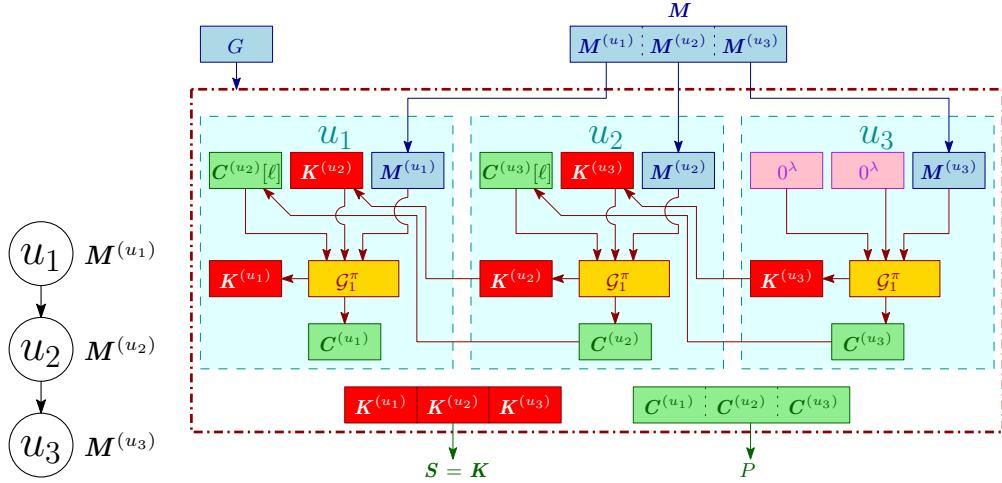
```

Figure 5.47: Algorithmic description of the function \mathcal{G}_2^π .

Description of $\ddot{\psi}$

The *KAS-AE-chain* construction $\ddot{\psi} = (\ddot{\psi}. \mathcal{E}, \ddot{\psi}. \mathcal{DER}, \ddot{\psi}. \mathcal{D})$ over $\ddot{\psi}$. **Setup** is built from the functions $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$. The pictorial description and pseudocode for the 3-tuple of algorithms in $\ddot{\psi}$ are given in Figures 5.48, 5.49, 5.50, 5.51, 5.52 and 5.53. Below, we give the textual description.

- The *setup* algorithm $\ddot{\psi}. \text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\ddot{\psi})}$, a set of *access graphs* $\Gamma^{(\ddot{\psi})}$, *key-space* $\mathcal{K}^{(\ddot{\psi})} := \{0, 1\}^\lambda$ and *message-space* $\mathcal{M}^{(\ddot{\psi})} := \bigcup^{i \geq 1} \{0, 1\}^{i\lambda}$.
- The *encryption* algorithm $\ddot{\psi}. \mathcal{E}(\cdot)$ is randomised that takes as input parameter $\text{params}^{(\ddot{\psi})}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pictorial description and pseudocode are given in Figures 5.48 and 5.49. Very briefly, the algorithm works as follows: first it computes a sequence of vertices $(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G)$, such that $u_m \lessdot u_{m-1}, u_{m-1} \lessdot u_{m-2}, \dots, u_2 \lessdot u_1$; and then it initializes both the strings IV_1 and IV_2 with 0^λ . Now, the ciphertext is computed iteratively, as j runs through $m, m - 1, \dots, 1$, in the following way: first it computes a pair of key and ciphertext $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}) := \mathcal{G}_1^\pi(1^\lambda, \mathbf{M}^{(u_j)}, IV_1, IV_2)$; then it computes $IV_1 := \mathbf{C}^{(u_j)}[\text{last_block}]$; and finally it assigns $IV_2 := \mathbf{K}^{(u_j)}$.



(a) The access (b) Pictorial description of $\ddot{\psi}.\mathcal{E}(\text{params}^{(\ddot{\psi})}, G, \mathbf{M})$, where G is shown in Figure 5.48(a) and vector of files $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \mathbf{M}^{(u_3)}$.

Figure 5.48: Pictorial description of the *encryption* function $\ddot{\psi}.\mathcal{E}$.

$\ddot{\psi}.\mathcal{E}(\text{params}^{(\ddot{\psi})}, G, \mathbf{M})$

#Initialization.

$(u_1, u_2, \dots, u_m) := \text{vertex_in_order}(G);$
 $\mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_m)} := \mathbf{M}; \quad IV_1 := IV_2 := 0^\lambda;$

#Computing ciphertext and private information for each user u_j .

for $(j := m, m - 1, \dots, 1)$
 $(\mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}) := \mathcal{G}_1^\pi(1^\lambda, \mathbf{M}^{(u_j)}, IV_1, IV_2);$
 $IV_1 := \mathbf{C}^{(u_j)}[\text{last_block}]; \quad IV_2 := \mathbf{K}^{(u_j)};$

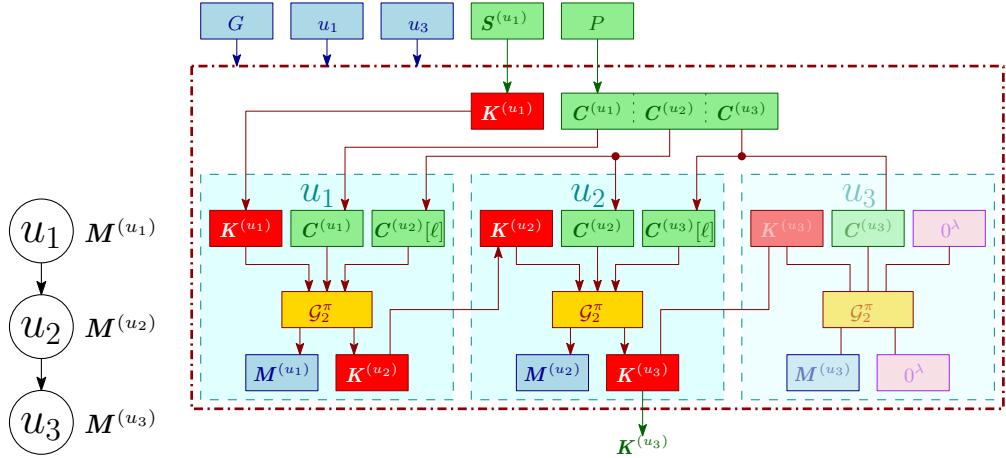
#Computing final outputs.

$\mathbf{S} := \mathbf{K} := \mathbf{K}^{(u_1)} \circ \mathbf{K}^{(u_2)} \circ \dots \circ \mathbf{K}^{(u_m)};$
 $P := \mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)};$
return $(\mathbf{S}, \mathbf{K}, P);$

Figure 5.49: Algorithmic description of the *encryption* function $\ddot{\psi}.\mathcal{E}$.

Now, the vectors of *private information* and *keys* $\mathbf{S} := \mathbf{K} := \mathbf{K}^{(u_1)} \circ \mathbf{K}^{(u_2)} \circ \dots \circ \mathbf{K}^{(u_m)}$, and the *public information* $P := \mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)}$.

- The *key derivation* algorithm $\ddot{\psi}.\mathcal{DER}(\cdot)$ is deterministic that takes



(a) The access graph G is shown in Figure 5.50(a).
(b) Pictorial description of $\vec{\psi}. D\mathcal{ER}(params^{(\vec{\psi})}, G, u_1, u_3, S^{(u_1)}, P)$ and

Figure 5.50: Pictorial description of the key derivation function $\vec{\psi}. D\mathcal{ER}$.

```

 $\vec{\psi}. D\mathcal{ER}(params^{(\vec{\psi})}, G, u_{i_1}, u_{i_2}, S^{(u_{i_1})}, P)$ 
#Initialization.
 $K^{(u_{i_1})} := S^{(u_{i_1})};$ 
If ( $u_{i_1} < u_{i_2}$ ), then return  $\perp$ ;
If ( $u_{i_1} = u_{i_2}$ ), then return  $K^{(u_{i_2})} := K^{(u_{i_1})};$ 

# Computing key.
 $C^{(u_1)} \circ C^{(u_2)} \circ \dots \circ C^{(u_m)} := P;$ 
for ( $j := i_1, i_1 + 1, \dots, i_2 - 1$ )
   $IV_1 := C^{(u_{j+1})}[last\_block];$ 
   $(M^{(u_j)}, K^{(u_{j+1})}) := \mathcal{G}_2^\pi(1^\lambda, K^{(u_j)}, C^{(u_j)}, IV_1);$ 

#Computing final output.
return  $K^{(u_{i_2})};$ 

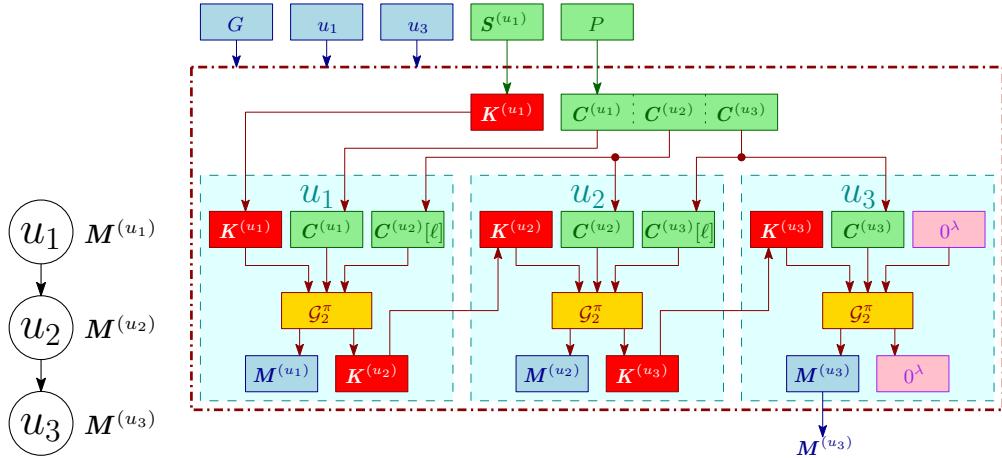
```

Figure 5.51: Algorithmic description of the key derivation function $\vec{\psi}. D\mathcal{ER}$.

as input parameter $params^{(\vec{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's private information $S^{(u_{i_1})}$ and public information P , and outputs key $K^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.50 and 5.51. Very briefly, the algorithm works as follows: first it initializes $K^{(u_{i_1})}$ with $S^{(u_{i_1})}$; next it checks if $u_{i_1} = u_{i_2}$, then it returns $K^{(u_{i_2})} := K^{(u_{i_1})}$;

and finally it parses P into $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P$. Now, the key is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, i_2 - 1$, in the following way: first it computes $IV_1 := \mathbf{C}^{(u_{j+1})}[last_block]$; and then it computes $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, IV_1)$.

The key $\mathbf{K}^{(u_{i_2})}$, computed in the last iteration, is returned.



(a) The access graph G . (b) Pictorial description of $\ddot{\psi}. \mathcal{D}(params^{(\ddot{\psi})}, G, u_1, u_3, \mathbf{S}^{(u_1)}, P)$ shown in Figure 5.52(a).

Figure 5.52: Pictorial description of the *decryption* function $\ddot{\psi}. \mathcal{D}$.

$\ddot{\psi}. \mathcal{D}(params^{(\ddot{\psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$

#Initialization.
 $\mathbf{K}^{(u_{i_1})} := \mathbf{S}^{(u_{i_1})};$
If $(u_{i_1} < u_{i_2})$, **then** return \perp ;

Computing file.
 $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P;$
for $(j := i_1, i_1 + 1, \dots, m)$
 If $(j = m)$, **then** $IV_1 := 0^\lambda$; **Else** $IV_1 := \mathbf{C}^{(u_{j+1})}[last_block];$
 $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, IV_1);$

#Computing final output.
If $(\mathbf{K}^{(u_{m+1})} = 0^\lambda)$, **then** return $\mathbf{M}^{(u_{i_2})}$; **Else** return \perp ;

Figure 5.53: Algorithmic description of the *decryption* function $\ddot{\psi}. \mathcal{D}$.

- The *decryption* algorithm $\ddot{\psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\ddot{\psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description and pseudocode are given in Figures 5.52 and 5.53. Very briefly, the algorithm works as follows: first it initializes $\mathbf{K}^{(u_{i_1})}$ with $\mathbf{S}^{(u_{i_1})}$; and then it parses P into $\mathbf{C}^{(u_1)} \circ \mathbf{C}^{(u_2)} \circ \dots \circ \mathbf{C}^{(u_m)} := P$. Now, the file is computed iteratively, as j runs through $i_1, i_1 + 1, \dots, m$, in the following way: first it checks if $j = m$, then it assigns $IV_1 := 0^\lambda$, otherwise it computes $IV_1 := \mathbf{C}^{(u_{j+1})}[last_block]$; and then it computes $(\mathbf{M}^{(u_j)}, \mathbf{K}^{(u_{j+1})}) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(u_j)}, \mathbf{C}^{(u_j)}, IV_1)$. If $\mathbf{K}^{(u_{m+1})} = 0^\lambda$, then it returns message $\mathbf{M}^{(u_{i_2})}$, otherwise it returns *invalid* string \perp .

5.6.1.8 Security of $\ddot{\psi}$

Theorem 21. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\ddot{\psi}$ has been defined in Section 5.6.1.7. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\ddot{\psi}, \mathcal{A}, G}^{KASAE-KR}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)+1}{2^\lambda}.$$

Here, the KASAE-KR game is defined in Figure 5.1, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 18. ■

Theorem 22. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\ddot{\psi}$ has been defined in Section 5.6.1.7. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\ddot{\psi}, \mathcal{A}, G}^{KASAE-IND}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-2}} + \frac{m(m-1)}{2^{\lambda-1}}.$$

Here, the KASAE-IND game is defined in Figure 5.2, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 19. ■

Theorem 23. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE-chain scheme $\ddot{\psi}$ has been defined in Section 5.6.1.7. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\ddot{\psi}, \mathcal{A}, G}^{KASAE-INT}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}.$$

Here, the *KASAE-INT* game is defined in Figure 5.3, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 20. \blacksquare

5.6.2 The Modified Chain Partition

The *modified chain partition* construction can be viewed as an adaptation of the *chain partition* construction used for designing *KAS* from *KAS-chain* constructions, as described in Section 3.6.2.

Let (V, \leq) and $G = (V, E)$ be, respectively, a poset and the access graph corresponding to it. A *chain partition* of V into w chains C_1, C_2, \dots, C_w is selected in such a way that C_i contains nodes (or classes) $u_1^i, u_2^i, \dots, u_{l_i}^i$, where $l_i = |C_i|$, $u_{j+1}^i < u_j^i$ for $1 \leq j < l_i$. We set $l_{max} = \max_{i \in [w]} l_i$. Let $\psi = (\psi, \mathcal{E}, \psi, \mathcal{DER}, \psi, \mathcal{D})$ over ψ . *Setup* be *KAS-AE-chain* scheme of length at most l_{max} .

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *modified chain partition* construction $\vec{\Psi} = (\vec{\Psi}, \mathcal{E}, \vec{\Psi}, \mathcal{DER}, \vec{\Psi}, \mathcal{D})$ over $\vec{\Psi}$. *Setup* is constructed below using *KAS-AE-chain* $\psi = (\psi, \mathcal{E}, \psi, \mathcal{DER}, \psi, \mathcal{D})$ over ψ . *Setup* (see Section 5.3 for details). The core idea behind this construction is similar to that of *chain partition* construction, and is outlined below for the sake of completeness: we first partition the access graph into *disjoint* chains, for each of which, we then design the corresponding *KAS-AE-chain*; and finally, we securely join these *KAS-AE-chains* to form the *KAS-AE* for the *full* access graph. The pseudocode for the 3-tuple of algorithms in $\vec{\Psi}$ are given in Figures 5.54, 5.55 and 5.56. The subroutines used by the algorithms are described in Section 3.2.3. Below, we give the textual description.

- The *setup* algorithm $\vec{\Psi}.Setup(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\vec{\Psi})}$, a set of *access graphs* $\Gamma^{(\vec{\Psi})}$, *key-space* $\mathcal{K}^{(\vec{\Psi})}$ and *message-space* $\mathcal{M}^{(\vec{\Psi})}$. The $\vec{\Psi}.Setup(1^\lambda)$ is designed in the following way: first it invokes $\psi.Setup(1^\lambda)$ to obtain $params^{(\psi)}$, $\Gamma^{(\psi)}$, $\mathcal{K}^{(\psi)}$ and $\mathcal{M}^{(\psi)}$; then it computes $params^{(\vec{\Psi})}$ that includes, among others, $params^{(\psi)}$; next it computes the set of *access graphs* $\Gamma^{(\vec{\Psi})}$ that contains all the graphs for chains in $\Gamma^{(\psi)}$; after that it assigns *key-space* $\mathcal{K}^{(\vec{\Psi})} := \mathcal{K}^{(\psi)}$; and finally it assigns *message-space* $\mathcal{M}^{(\vec{\Psi})} := \mathcal{M}^{(\psi)}$.
- The *encryption* algorithm $\vec{\Psi}.E(\cdot)$ is randomized that takes as input parameter $params^{(\vec{\Psi})}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pseudocode is given in Figure 5.54. Very

```

 $\vec{\Psi} \cdot \mathcal{E}(\text{params}^{(\vec{\Psi})}, G, \mathbf{M})$ 

#Initialization.
 $(w, C[ ]) := \text{partition}(G);$ 

#Generating  $(\mathbf{S}, \mathbf{K}, P)$  for each chain using KAS-AE-chain.
for  $(j := 1, 2, \dots, w)$ 
     $\mathbf{M}_j := (\mathbf{M}^{(u_i)})_{u_i \in C_j};$ 
     $(\mathbf{S}_j, \mathbf{K}_j, P_j) := \psi \cdot \mathcal{E}(\text{params}^{(\psi)}, C_j, \mathbf{M}_j);$ 

#Computing private information for each node.
for  $u_i \in V$ 
     $(\hat{u}_{i_1}, \hat{u}_{i_2}, \dots, \hat{u}_{i_w}) := \text{max\_isect\_chs}(u_i, G);$ 
     $\mathbf{S}^{(u_i)} := \mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})};$ 

#Computing final output.
 $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}; \quad \mathbf{K} := (\mathbf{K}^{(u_i)})_{u_i \in V}; \quad P := P_1 \circ P_2 \circ \dots \circ P_w;$ 
return  $(\mathbf{S}, \mathbf{K}, P);$ 

```

Figure 5.54: Algorithmic description of the *encryption* function $\vec{\Psi} \cdot \mathcal{E}$.

briefly, the algorithm works as follows: first it invokes $(w, C[]) := \text{partition}(G)$ to compute the *chain partition* C_1, C_2, \dots, C_w of graph G ; and then, for each chain C_j , where $j \in [w]$, first it computes vector of *files* as $\mathbf{M}_j := (\mathbf{M}^{(u_i)})_{u_i \in C_j}$ and then it computes $(\mathbf{S}_j, \mathbf{K}_j, P_j) := \psi \cdot \mathcal{E}(\text{params}^{(\psi)}, C_j, \mathbf{M}_j)$. Then, for every node $u_i \in V$, the following operations are performed: first it computes maximum nodes of the sets $\downarrow u_i \cap C_g$ for all chains $(\hat{u}_{i_1}, \hat{u}_{i_2}, \dots, \hat{u}_{i_w}) := \text{max_isect_chs}(u_i, G)$; and finally it computes the *private information* of u_i , $\mathbf{S}^{(u_i)} := \mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})}$.

Now, the vector of *private information* $\mathbf{S} := (\mathbf{S}^{(u_i)})_{u_i \in V}$, vector of *keys* $\mathbf{K} := (\mathbf{K}^{(u_i)})_{u_i \in V}$, and *public information* $P := P_1 \circ P_2 \circ \dots \circ P_w$.

- The *key derivation* algorithm $\vec{\Psi} \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\vec{\Psi})}$, graph G , two nodes u_j^i and u_h^g , such that $u_h^g \leq u_j^i$, node u_j^i 's *private information* $\mathbf{S}^{(u_j^i)}$ and *public information* P , and outputs key $\mathbf{K}^{(u_h^g)}$ corresponding to u_h^g . See Figure 5.55 for the pseudocode. The algorithm works as follows: first it computes maximum node of the set $\downarrow u_j^i \cap C_g$ by invoking $\hat{u}_{i_g} :=$

```

 $\vec{\Psi} \cdot \mathcal{DER}(params^{(\vec{\Psi})}, G, u_j^i, u_h^g, \mathbf{S}^{(u_j^i)}, P)$ 

#Initialization.
 $\hat{u}_{i_g} := \text{max\_isect}(u_j^i, C_g, G);$ 
 $\mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_g^{(\hat{u}_{i_g})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})} := \mathbf{S}^{(u_j^i)};$ 
 $P_1 \circ P_2 \circ \dots \circ P_w := P;$ 

# Computing key.
 $\mathbf{K}^{(u_h^g)} := \psi \cdot \mathcal{DER}(params^{(\psi)}, C_g, \hat{u}_{i_g}, u_h^g, \mathbf{S}_g^{(\hat{u}_{i_g})}, P_g);$ 
return  $\mathbf{K}^{(u_h^g)};$ 

```

Figure 5.55: Algorithmic description of the *key derivation* function $\vec{\Psi} \cdot \mathcal{DER}$.

$\text{max_isect}(u_j^i, C_g, G)$; then it parses the *private information* of u_j^i into $\mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_g^{(\hat{u}_{i_g})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})} := \mathbf{S}^{(u_j^i)}$; next it parses *public information* P into $P_1 \circ P_2 \circ \dots \circ P_w := P$; and finally it computes key $\mathbf{K}^{(u_h^g)} := \psi \cdot \mathcal{DER}(params^{(\psi)}, C_g, \hat{u}_{i_g}, u_h^g, \mathbf{S}_g^{(\hat{u}_{i_g})}, P_g)$.

Now, the key $\mathbf{K}^{(u_h^g)}$, computed above, is returned.

```

 $\vec{\Psi} \cdot \mathcal{D}(params^{(\vec{\Psi})}, G, u_j^i, u_h^g, \mathbf{S}^{(u_j^i)}, P)$ 

#Initialization.
 $\hat{u}_{i_g} := \text{max\_isect}(u_j^i, C_g, G);$ 
 $\mathbf{S}_1^{(\hat{u}_{i_1})} \cup \mathbf{S}_2^{(\hat{u}_{i_2})} \cup \dots \cup \mathbf{S}_g^{(\hat{u}_{i_g})} \cup \dots \cup \mathbf{S}_w^{(\hat{u}_{i_w})} := \mathbf{S}^{(u_j^i)};$ 
 $P_1 \circ P_2 \circ \dots \circ P_w := P;$ 

# Computing file.
 $\mathbf{M}^{(u_h^g)} := \psi \cdot \mathcal{D}(params^{(\psi)}, C_g, \hat{u}_{i_g}, u_h^g, \mathbf{S}_g^{(\hat{u}_{i_g})}, P_g);$ 
return  $\mathbf{M}^{(u_h^g)};$ 

```

Figure 5.56: Algorithmic description of the *decryption* function $\vec{\Psi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\vec{\Psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\vec{\Psi})}$, graph G , two nodes u_j^i and u_h^g , such that $u_h^g \leq u_j^i$, node u_j^i 's *private information* $\mathbf{S}^{(u_j^i)}$ and *public information* P , and outputs file $\mathbf{M}^{(u_h^g)}$ corresponding to u_h^g . See Figure 5.56 for the pseudocode. Very briefly, the algorithm works as follows: first it computes maximum node of the set $\downarrow u_j^i \cap C_g$ by invoking $\hat{u}_{i_g} :=$

$\max_isect(u_j^i, C_g, G)$; then it parses the *private information* of u_j^i into $S_1^{(\hat{u}_{i1})} \cup S_2^{(\hat{u}_{i2})} \cup \dots \cup S_g^{(\hat{u}_{ig})} \cup \dots \cup S_w^{(\hat{u}_{iw})} := S^{(u_j^i)}$; next it parses *public information* P into $P_1 \circ P_2 \circ \dots \circ P_w := P$; and finally it computes file $M^{(u_h^g)} := \psi \cdot \mathcal{D}(params^{(\psi)}, C_g, \hat{u}_{ig}, u_h^g, S_g^{(\hat{u}_{ig})}, P_g)$.

Now, the file $M^{(u_h^g)}$, computed above, is returned.

5.6.3 Security of Modified Chain Partition

Theorem 24. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\vec{\Psi}$ has been defined in Section 5.6.2. For all poly-time KASAE-KR adversaries \mathcal{A} against $\vec{\Psi}$, there exists a poly-time KASAE-KR adversary \mathcal{B} against KAS-AE-chain scheme ψ , such that:

$$Adv_{\vec{\Psi}, \mathcal{A}, G}^{KASAE-KR}(1^\lambda) \leq Adv_{\psi, \mathcal{B}, G}^{KASAE-KR}(1^\lambda).$$

Here, the KASAE-KR game is defined in Figure 5.1.

Proof. We poly-reduce any KASAE-KR adversary \mathcal{A} against $\vec{\Psi}$ into a KAS-KR adversary \mathcal{B} against ψ , as used by Freire *et al.* [FPP13]. The proof readily follows from it. ■

Theorem 25. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\vec{\Psi}$ has been defined in Section 5.6.2. For all poly-time KASAE-IND adversaries \mathcal{A} against $\vec{\Psi}$, there exists a poly-time KASAE-IND adversary \mathcal{B} against KAS-AE-chain scheme ψ , such that:

$$Adv_{\vec{\Psi}, \mathcal{A}, G}^{KASAE-IND}(1^\lambda) \leq Adv_{\psi, \mathcal{B}, G}^{KASAE-IND}(1^\lambda).$$

Here, the KASAE-IND game is defined in Figure 5.2.

Proof. We poly-reduce any KASAE-IND adversary \mathcal{A} against $\vec{\Psi}$ into a KAS-IND adversary \mathcal{B} against ψ , as used by Freire *et al.* [FPP13]. The proof readily follows from it. ■

Theorem 26. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\vec{\Psi}$ has been defined in Section 5.6.2. For all poly-time KASAE-INT adversaries \mathcal{A} against $\vec{\Psi}$, there exists a poly-time KASAE-INT adversary \mathcal{B} against KAS-AE-chain scheme ψ , such that:

$$Adv_{\vec{\Psi}, \mathcal{A}, G}^{KASAE-INT}(1^\lambda) \leq Adv_{\psi, \mathcal{B}, G}^{KASAE-INT}(1^\lambda).$$

Here, the KASAE-INT game is defined in Figure 5.3.

Proof. We poly-reduce any **KASAE-INT** adversary \mathcal{A} against $\vec{\Psi}$ into a **KAS-INT** adversary \mathcal{B} against ψ , as used by Freire *et al.* [FPP13]. The proof readily follows from it. ■

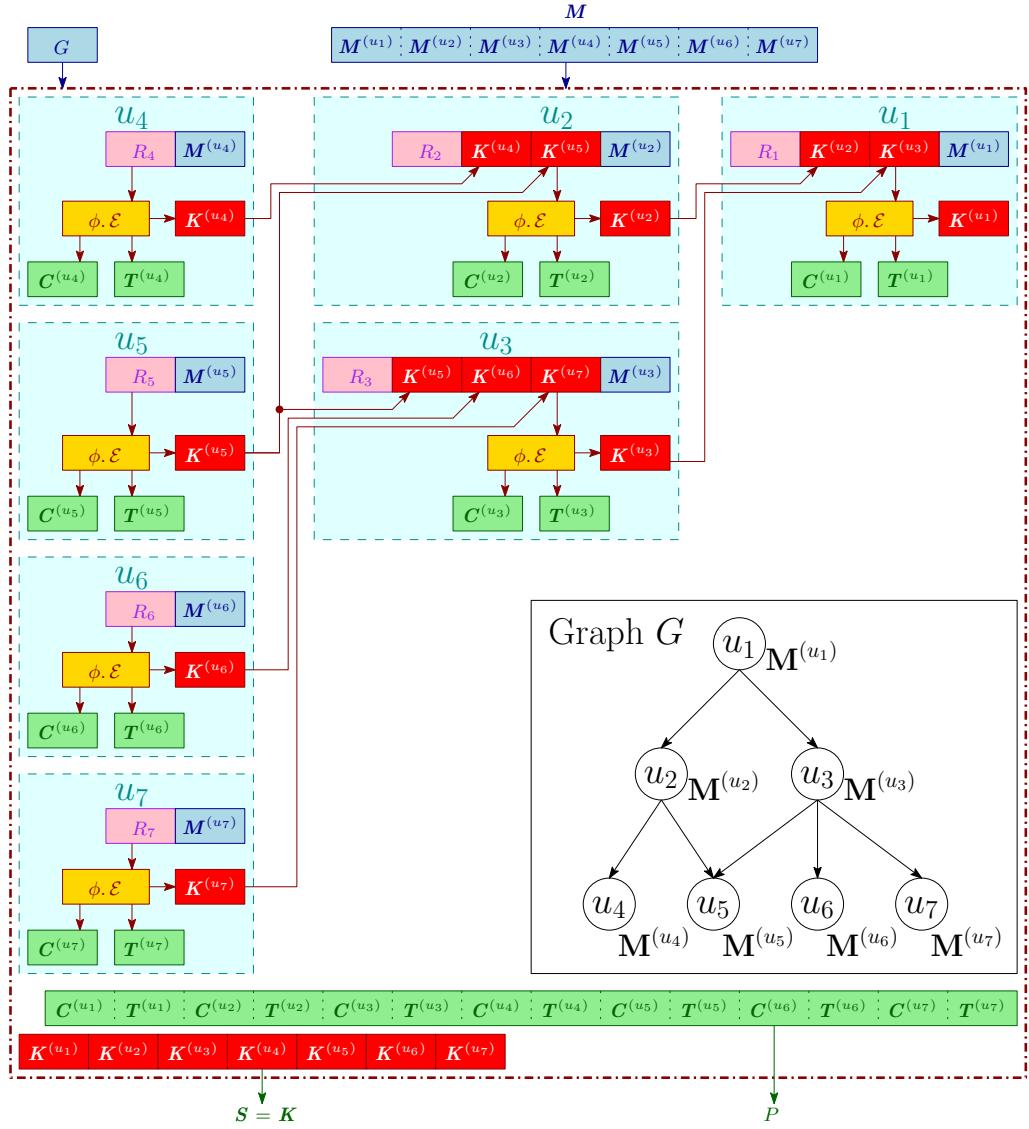
5.7 Construction $\check{\Psi}$: **KAS-AE** based on **MLE**

In this section, we describe a *KAS-AE* construction $\check{\Psi}$ being built using an *MLE* scheme $\phi = (\phi.\mathcal{E}, \phi.\mathcal{D})$ over $\phi.\text{Setup}$. Here, the main idea is to prepend the keys of all the *immediate successor* nodes to the file before encrypting it using *MLE* for generating the key, ciphertext and tag. This scheme is more efficient than all the *KAS-AE* constructions described in Sections 5.5 and 5.6. Our scheme naturally inherits the *integrity* and *confidentiality* security properties of the underlying *MLE* scheme, which results in a huge reduction in memory requirements.

5.7.1 Description of $\check{\Psi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\check{\Psi}$ are given in Figures 5.57, 5.58, 5.59, 5.60, 5.61 and 5.62. Below, we give the textual description.

- The *setup* algorithm $\check{\Psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\check{\Psi})}$, a set of *access graphs* $\Gamma^{(\check{\Psi})}$, *key-space* $\mathcal{K}^{(\check{\Psi})}$ and *message-space* $\mathcal{M}^{(\check{\Psi})}$. The $\check{\Psi}.\text{Setup}(1^\lambda)$ is designed in the following way: first it invokes $\phi.\text{Setup}(1^\lambda)$; then it computes $\text{params}^{(\check{\Psi})}$ that includes, among others, $\text{params}^{(\phi)}$; and finally it computes *key-space* $\mathcal{K}^{(\check{\Psi})} = \mathcal{K}^{(\phi)}$ and *message-space* $\mathcal{M}^{(\check{\Psi})} = \mathcal{M}^{(\phi)}$.
- The *encryption* algorithm $\check{\Psi}.\mathcal{E}(\cdot)$ is randomised that takes as input parameter $\text{params}^{(\check{\Psi})}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pictorial description of $\check{\Psi}.\mathcal{E}(\text{params}^{(\check{\Psi})}, G, \mathbf{M})$ with access graph $G = (V, E)$, where $V = \{u_1, u_2, \dots, u_7\}$ and vector of *files* $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_7)}$, is given in Figure 5.57. The pseudocode is given in Figure 5.58. This encryption function is designed in such a way that any node u_i is able to decrypt the files of its successors. In order to do that, for each node u_i , we encrypt file $\mathbf{M}^{(u_i)}$ after prepending decryption keys of all children of u_i to it. Very briefly, the algorithm works as follows: first it assigns the level $\text{level}[u_i]$ to

Figure 5.57: Pictorial description of the *encryption* function $\check{\Psi}.\mathcal{E}$.

each node $u_i \in V$ and calculates maximum depth h of graph G , by computing $(\text{level}[], h) := \text{height}(G)$; and then it encrypts files at level h , followed by encryption of files at level $h - 1$, and so on, until root node is reached.

For each node u_i , the following operations are executed: first it invokes function $\text{ch_seq}(u_i, G)$ which returns sequence of children $(u_{i_1}, u_{i_2}, \dots, u_{i_d})$ of u_i (in ascending order); then it randomly chooses a λ -bit string temp ; next it appends to temp the decryption keys $K^{(u_{i_1})}, K^{(u_{i_2})}, \dots,$

```

 $\check{\Psi} \cdot \mathcal{E}(params^{(\check{\Psi})}, G, M)$ 

#Initialization.
 $(level[], h) := \text{height}(G); \quad M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_m)} := M;$ 

#Computing values for each user level-by-level.
for ( $j := h, h - 1, \dots, 0$ )
   $V_j := \text{nodes\_at\_level}(V, level[], j);$ 
  For all  $u_i \in V_j$ 
     $\tilde{u}_i \stackrel{\text{def}}{=} (u_{i_1}, u_{i_2}, \dots, u_{i_d}) := \text{ch\_seq}(u_i, G);$ 
     $temp \stackrel{\$}{\leftarrow} \{0, 1\}^\lambda;$ 
    If ( $\tilde{u}_i = \text{NULL}$ )
       $temp := temp \| M^{(u_i)};$ 
    Else
       $temp := temp \| K^{(u_{i_1})} \| K^{(u_{i_2})} \| \dots \| K^{(u_{i_d})} \| M^{(u_i)};$ 
    ( $K^{(u_i)}, C^{(u_i)}, T^{(u_i)}$ ) :=  $\phi \cdot \mathcal{E}(params^{(\phi)}, temp);$ 

#Computing final output.
 $S := K := (K^{(u_i)})_{u_i \in V}; \quad P := (C^{(u_i)} \| T^{(u_i)})_{u_i \in V};$ 
return ( $S, K, P$ );

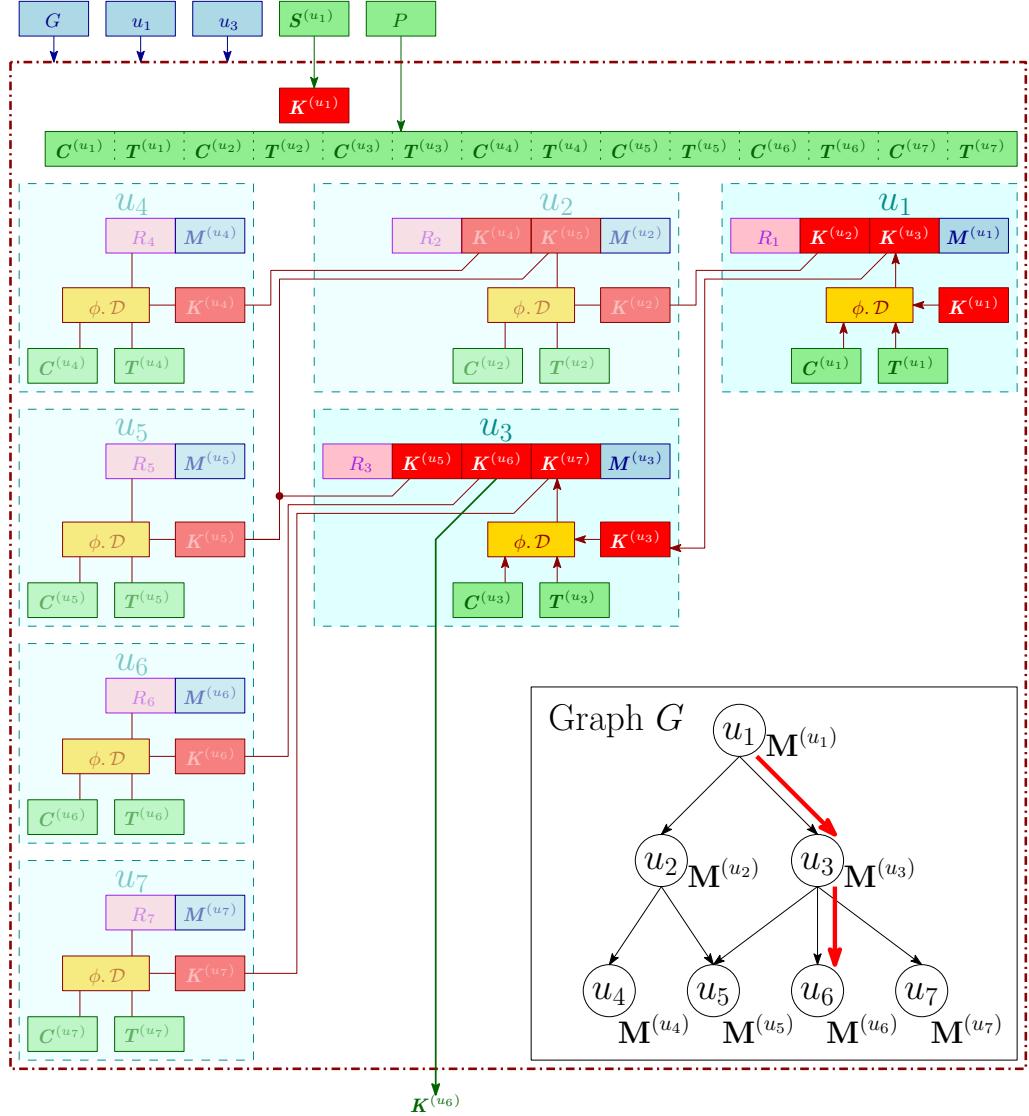
```

Figure 5.58: Algorithmic description of the *encryption* function $\check{\Psi} \cdot \mathcal{E}$.

$K^{(u_{i_d})}$ – which have been already generated in previous iterations – and file $M^{(u_i)}$ to compute $temp := temp \| K^{(u_{i_1})} \| K^{(u_{i_2})} \| \dots \| K^{(u_{i_d})} \| M^{(u_i)}$; and finally it computes a 3-tuple of decryption key, ciphertext and tag $(K^{(u_i)}, C^{(u_i)}, T^{(u_i)}) := \phi \cdot \mathcal{E}(params^{(\phi)}, temp)$.

Now, the vectors of *private information* and *keys* $S := K := (K^{(u_i)})_{u_i \in V}$, and the *public information* $P := (C^{(u_i)} \| T^{(u_i)})_{u_i \in V}$.

- The *key derivation* algorithm $\check{\Psi} \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $params^{(\check{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $S^{(u_{i_1})}$ and *public information* P , and outputs key $K^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description of $\check{\Psi} \cdot \mathcal{DER}(params^{(\check{\Psi})}, G, u_1, u_6, S^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , is given in Figure 5.59. The pseudocode is given in Figure 5.60. Very briefly, the algorithm works as follows: first it computes a path $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; then it initializes v and $K^{(v)}$ with u_{i_1} and $S^{(u_{i_1})}$; and finally it parses *public information* P into

Figure 5.59: Pictorial description of the *key derivation* function $\check{\Psi} \cdot \mathcal{DER}$.

$$(\mathbf{C}^{(u_i)} \| \mathbf{T}^{(u_i)})_{u_i \in V} := P.$$

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}$, the following operations are executed: first it computes $temp := \phi \cdot \mathcal{D}(params^{(\phi)}, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, \mathbf{T}^{(v)})$; then it invokes function $\text{ch_seq}(v, G)$ which returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); next it parses $temp$ into $R \| \mathbf{K}^{(u_{k_1})} \| \mathbf{K}^{(u_{k_2})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp$, where R is the random number that was used during encryption; and finally it searches next node in the path in sequence \tilde{v} , and extracts its

```

 $\check{\Psi} \cdot \mathcal{DER}(params^{(\check{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)} \parallel \mathbf{T}^{(u_i)})_{u_i \in V} := P;$ 

#Computing key.
for ( $p := 1, 2, \dots, \ell + 1$ )
   $temp := \phi \cdot \mathcal{D}(params^{(\phi)}, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, \mathbf{T}^{(v)});$ 
  If  $temp = \perp$ , then return  $\perp$ ;
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
   $R \parallel \mathbf{K}^{(u_{k_1})} \parallel \mathbf{K}^{(u_{k_2})} \parallel \dots \parallel \mathbf{K}^{(u_{k_d})} \parallel \mathbf{M}^{(v)} := temp;$ 
  Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p$ ;  $\mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}$ ;  $v := u_{j_p}$ ;

#Computing final output.
return  $\mathbf{K}^{(u_{i_2})};$ 

```

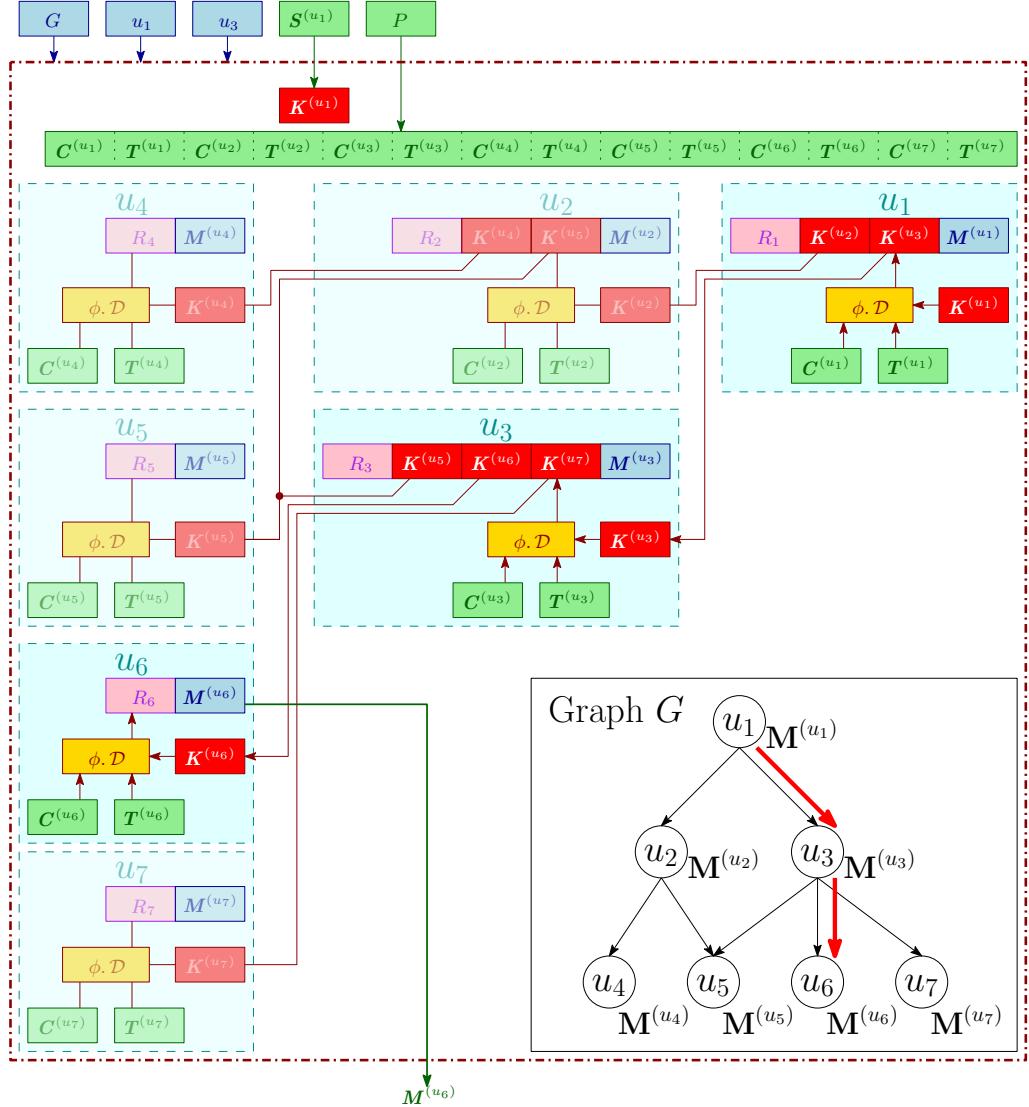
Figure 5.60: Algorithmic description of the *key derivation* function $\check{\Psi} \cdot \mathcal{DER}$.

corresponding key, before the next iteration begins.

Now, the key $\mathbf{K}^{(u_{i_2})}$, computed above, is returned.

- The *decryption* algorithm $\check{\Psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\check{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description of $\check{\Psi} \cdot \mathcal{D}(params^{(\check{\Psi})}, G, u_1, u_6, \mathbf{S}^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , is given in Figure 5.61. The pseudocode is given in Figure 5.62. The algorithm works as follows: first it computes a path $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; next it initializes v and $\mathbf{K}^{(v)}$ with u_{i_1} and $\mathbf{S}^{(u_{i_1})}$; and then it parses *public information* P into $(\mathbf{C}^{(u_i)} \parallel \mathbf{T}^{(u_i)})_{u_i \in V} := P$.

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_{\ell+1}}$, the following operations are executed: first it computes $temp := \phi \cdot \mathcal{D}(params^{(\phi)}, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, \mathbf{T}^{(v)})$; then it invokes function $\text{ch_seq}(v, G)$ which returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); next it checks if v is a leaf-node, then it parses $temp$ into $R \parallel \mathbf{M}^{(v)} := temp$, otherwise it parses $temp$ into $R \parallel \mathbf{K}^{(u_{k_1})} \parallel \mathbf{K}^{(u_{k_2})} \parallel \dots \parallel \mathbf{K}^{(u_{k_d})} \parallel \mathbf{M}^{(v)} := temp$, where R is the random number that was used during encryp-

Figure 5.61: Pictorial description of the decryption function $\check{\Psi} \cdot \mathcal{D}$.

tion; and finally it searches next node in the path in sequence \tilde{v} , and extracts its corresponding key, before the next iteration begins.

Now, the file $M^{(u_{i2})}$, computed above, is returned.

5.7.2 Security of $\check{\Psi}$

Theorem 27. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\check{\Psi}$ has been defined in Section 5.7.1. For all poly-time KASAE-KR

```

 $\check{\Psi} \cdot \mathcal{D}(params^{(\check{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)} \parallel \mathbf{T}^{(u_i)})_{u_i \in V} := P;$ 

#Computing file.
for ( $p := 1, 2, \dots, \ell + 2$ )
   $temp := \phi \cdot \mathcal{D}(params^{(\phi)}, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, \mathbf{T}^{(v)});$ 
  If  $temp = \perp$ , then return  $\perp$ ;
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
  If  $(\tilde{v} = \text{NULL})$ , then  $R \parallel \mathbf{M}^{(v)} := temp;$ 
  Else  $R \parallel \mathbf{K}^{(u_{k_1})} \parallel \mathbf{K}^{(u_{k_2})} \parallel \dots \parallel \mathbf{K}^{(u_{k_d})} \parallel \mathbf{M}^{(v)} := temp;$ 
  If  $(p \leq \ell + 1)$ 
    Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p$ ;  $\mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}$ ;  $v := u_{j_p}$ ;
```

#Computing final output.

```

  return  $\mathbf{M}^{(u_{i_2})};$ 
```

Figure 5.62: Algorithmic description of the *decryption* function $\check{\Psi} \cdot \mathcal{D}$.

adversaries \mathcal{A} against $\check{\Psi}$, there exists a poly-time MLE-KR adversary \mathcal{B} against MLE scheme ϕ , such that:

$$\text{Adv}_{\check{\Psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{S}, \mathcal{B}}^{\text{MLE-KR}}(1^\lambda).$$

Here, the KASAE-KR game is defined in Figure 5.1, and the MLE-KR game is defined in Figure 3.15.

Proof. The proof is similar to that of Theorem 15. ■

Theorem 28. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\check{\Psi}$ has been defined in Section 5.7.1. For all poly-time KASAE-IND adversaries \mathcal{A} against $\check{\Psi}$, there exists a poly-time MLE-PRV adversary \mathcal{B} against MLE scheme ϕ , such that:

$$\text{Adv}_{\check{\Psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{S}, \mathcal{B}}^{\text{MLE-PRV}}(1^\lambda).$$

Here, the KASAE-IND game is defined in Figure 5.2, and the MLE-PRV game is defined in Figure 3.12.

Proof. The proof is similar to that of Theorem 16. ■

Theorem 29. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE construction $\check{\Psi}$ has been defined in Section 5.7.1. For all poly-time KASAE-INT adversaries \mathcal{A} against $\check{\Psi}$, there exists a poly-time MLE-TC adversary \mathcal{B} against MLE scheme ϕ , such that:

$$\text{Adv}_{\check{\Psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \text{Adv}_{\phi, \mathcal{B}}^{\text{MLE-TC}}(1^\lambda).$$

Here, the KASAE-INT game is defined in Figure 5.3, and the MLE-TC game is defined in Figure 3.14.

Proof. The proof is similar to that of Theorem 17. ■

5.8 Building *KAS-AE* by Tweaking *APE* and *FP*

So far we have constructed *KAS-AE* schemes using *KAS*, *KAS-AE-chain* and *MLE* schemes used as black boxes. Here, we take a focused look on designing *KAS-AE* schemes from scratch in an attempt to extract significant efficiency. We eventually designed two *KAS-AE* schemes – denoted $\hat{\Psi}$ and $\tilde{\Psi}$ – that turned out to be the most efficient ones. This efficiency is attributed to the *reverse decryption* – of *APE* authenticated encryption and *FP* hash mode – which integrates the key and the message, to provide *authenticated encryption*. This results in significant reduction in memory requirements.

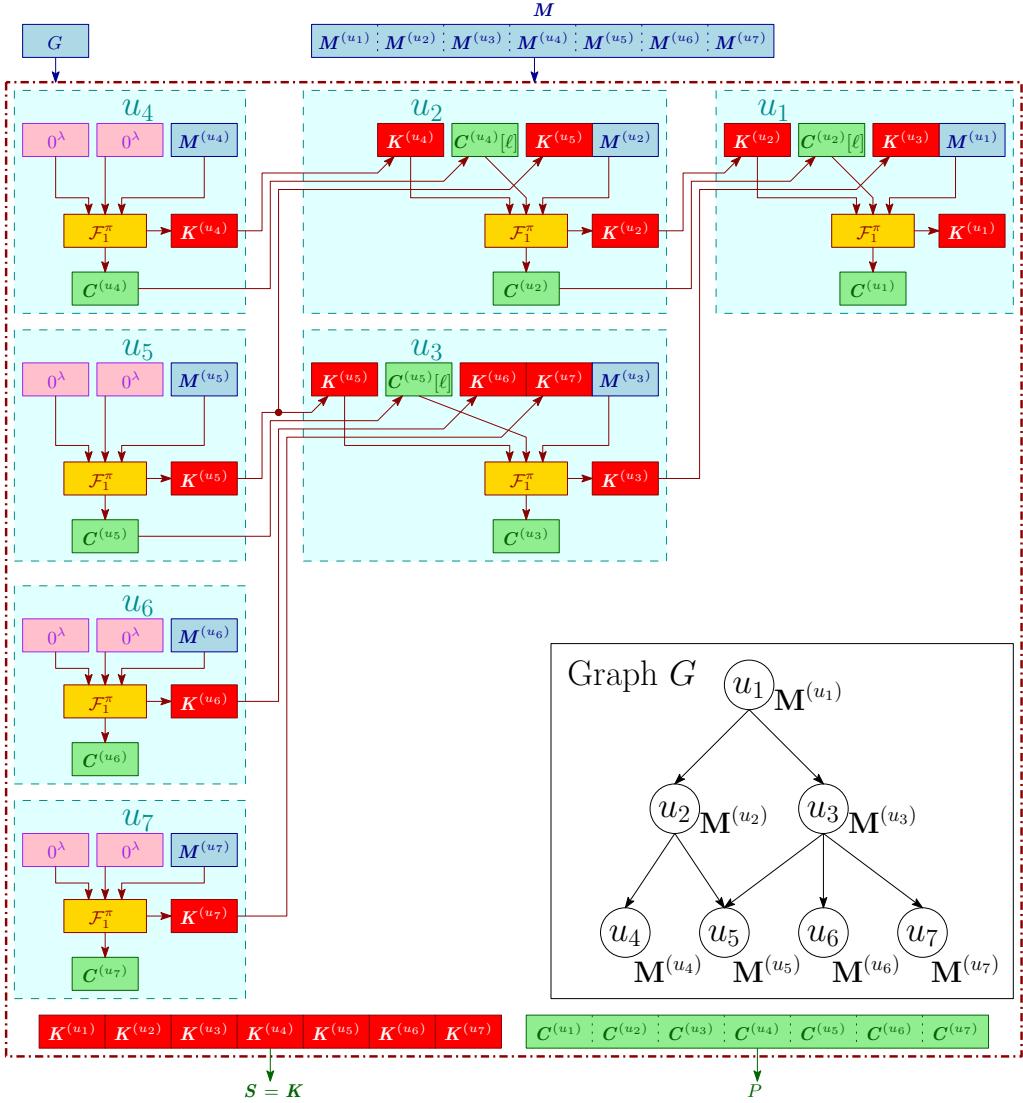
5.8.1 Construction $\hat{\Psi}$: *KAS-AE* based on *APE*

In Section 5.6.1.5, we have described two functions $\mathcal{F}_1^\pi(\cdot)$ and $\mathcal{F}_2^\pi(\cdot)$ whose designs are inspired from *APE* authenticated encryption. In this section, we utilize them again, for designing *KAS-AE* construction $\hat{\Psi}$.

5.8.1.1 Description of $\hat{\Psi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\hat{\Psi}$ are given in Figures 5.63, 5.64, 5.65, 5.66, 5.67 and 5.68. Below, we give the textual description.

- The *setup* algorithm $\hat{\Psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\hat{\Psi})}$, a set of *access graphs* $\Gamma^{(\hat{\Psi})}$, *key-space* $\mathcal{K}^{(\hat{\Psi})} := \{0, 1\}^\lambda$ and *message-space* $\mathcal{M}^{(\hat{\Psi})} := \bigcup^{i \geq 1} \{0, 1\}^{i\lambda}$.

Figure 5.63: Pictorial description of the $\hat{\Psi}.\mathcal{E}$ function.

- The $\hat{\Psi}.\mathcal{E}$ algorithm takes as input parameter $\text{params}^{(\hat{\Psi})}$, graph G and vector of files M , and outputs vector of private information S , vector of keys K and public information P . The pictorial description of $\hat{\Psi}.\mathcal{E}(\text{params}^{(\hat{\Psi})}, G, M)$ with access graph $G = (V, E)$, where $V = \{u_1, u_2, \dots, u_7\}$ and vector of files $M = M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_7)}$, is given in Figure 5.63. The pseudocode is given in Figure 5.64. This encryption function is designed in such a way that any node u_i is able to decrypt the files of its successors. In order to do that, for each node u_i , we encrypt file $M^{(u_i)}$

```

 $\hat{\Psi} \cdot \mathcal{E}(params^{(\hat{\Psi})}, G, M)$ 


---


#Initialization.
 $(level[], h) := \text{height}(G); \quad M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_m)} := M;$ 

#Computing values for each user level-by-level.
for  $(j := h, h - 1, \dots, 0)$ 
   $V_j := \text{nodes\_at\_level}(V, level[], j);$ 
  For all  $u_i \in V_j$ 
     $\tilde{u}_i \stackrel{\text{def}}{=} (u_{i_1}, u_{i_2}, \dots, u_{i_d}) := \text{ch\_seq}(u_i, G);$ 
    If  $(\tilde{u}_i = \text{NULL})$ 
       $IV_1 := IV_2 := 0^\lambda; \quad temp := M^{(u_i)};$ 
    Else
       $IV_1 := C^{(u_{i_1})}[last\_block]; \quad IV_2 := K^{(u_{i_1})};$ 
       $temp := K^{(u_{i_2})} \| K^{(u_{i_3})} \| \dots \| K^{(u_{i_d})} \| M^{(u_i)};$ 
       $(K^{(u_i)}, C^{(u_i)}) := \mathcal{F}_1^\pi(1^\lambda, temp, IV_1, IV_2);$ 

#Computing final output.
 $S := K := (K^{(u_i)})_{u_i \in V}; \quad P := (C^{(u_i)})_{u_i \in V};$ 
  return  $(S, K, P);$ 

```

Figure 5.64: Algorithmic description of the *encryption* function $\hat{\Psi} \cdot \mathcal{E}$.

as well as decryption keys of children of u_i , such that, on decrypting ciphertext corresponding to u_i , decryption keys of all its children are revealed. Very briefly, the algorithm works as follows: first it assigns the level $level[u_i]$ to each node $u_i \in V$ and calculates maximum depth h of graph G , by computing $(level[], h) := \text{height}(G)$; and then it encrypts files at level h , followed by encryption of files at level $h - 1$, and so on, until root node is reached.

For each node u_i , the following operations are executed: first it invokes function $\text{ch_seq}(u_i, G)$ that returns sequence of children $(u_{i_1}, u_{i_2}, \dots, u_{i_d})$ of u_i (in ascending order); then it assigns key of first child $K^{(u_{i_1})}$ to IV_2 and last λ -bit block of ciphertext $C^{(u_{i_1})}$ to IV_1 ; next it prepends the decryption keys $K^{(u_{i_2})}, K^{(u_{i_3})}, \dots, K^{(u_{i_d})}$ – which have been already generated in previous iterations – to file $M^{(u_i)}$ to compute $temp := K^{(u_{i_2})} \| K^{(u_{i_3})} \| \dots \| K^{(u_{i_d})} \| M^{(u_i)}$; and finally it computes a pair of decryption key and ciphertext $(K^{(u_i)}, C^{(u_i)}) := \mathcal{F}_1^\pi(1^\lambda, temp, IV_1, IV_2)$. For the leaf nodes, value of IV_1 and IV_2 are 0^λ .

Now, the vectors of *private information* and *keys* $S := K := (K^{(u_i)})_{u_i \in V}$,

and the *public information* $P := (\mathbf{C}^{(u_i)})_{u_i \in V}$.

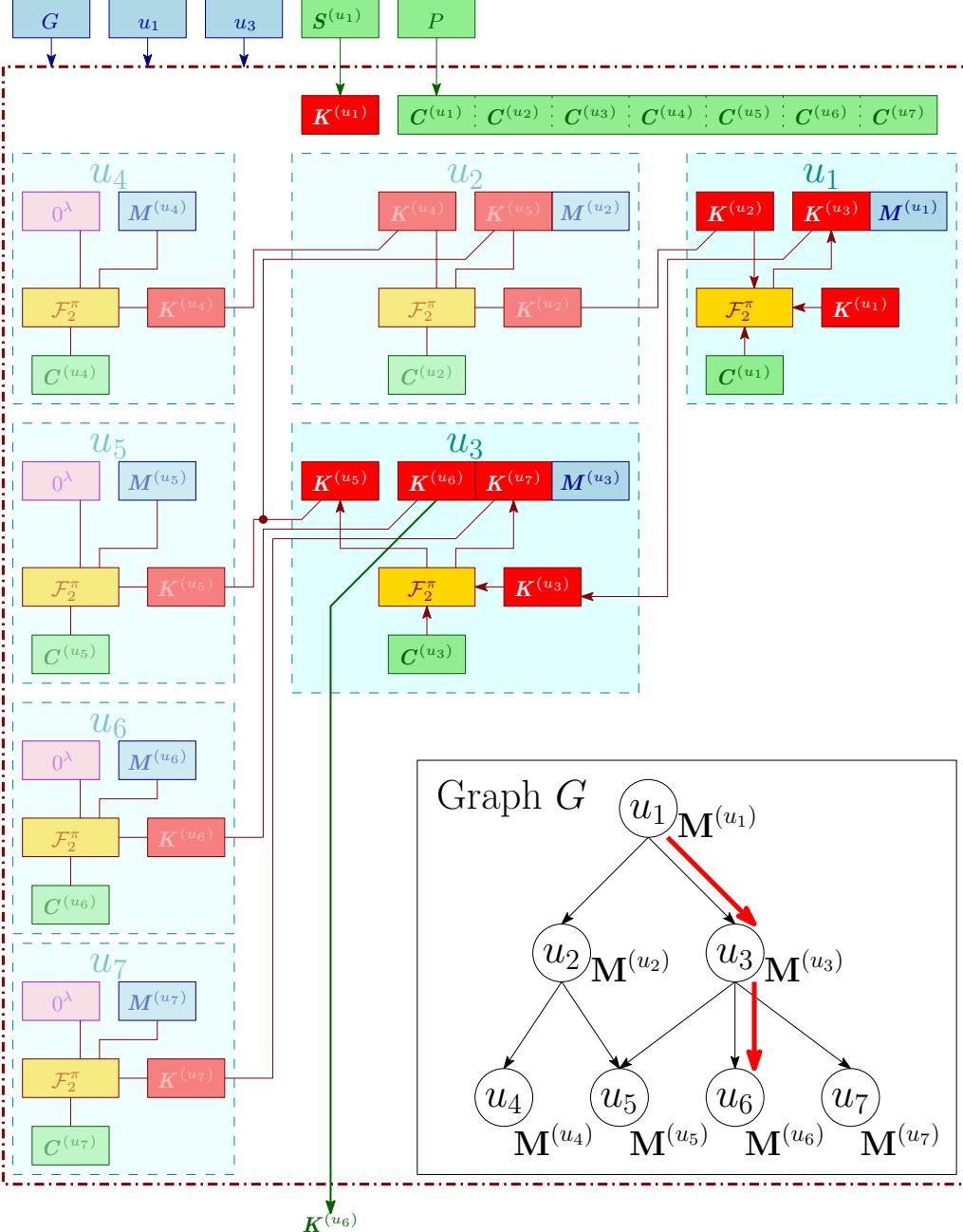


Figure 5.65: Pictorial description of the *key derivation* function $\hat{\Psi} \cdot \mathcal{DER}$.

- The *key derivation* algorithm $\hat{\Psi} \cdot \mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such

```

 $\hat{\Psi} \cdot \mathcal{D}\mathcal{E}\mathcal{R}(params^{(\hat{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 


---


#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 

#Computing key.
for ( $p := 1, 2, \dots, \ell + 1$ )
   $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)});$ 
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
   $\mathbf{K}^{(u_{k_1})} := IV_2; \quad \mathbf{K}^{(u_{k_2})} \| \mathbf{K}^{(u_{k_3})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp;$ 
  Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p$ ;  $\mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}; \quad v := u_{j_p};$ 

#Computing final output.
  return  $\mathbf{K}^{(u_{i_2})};$ 

```

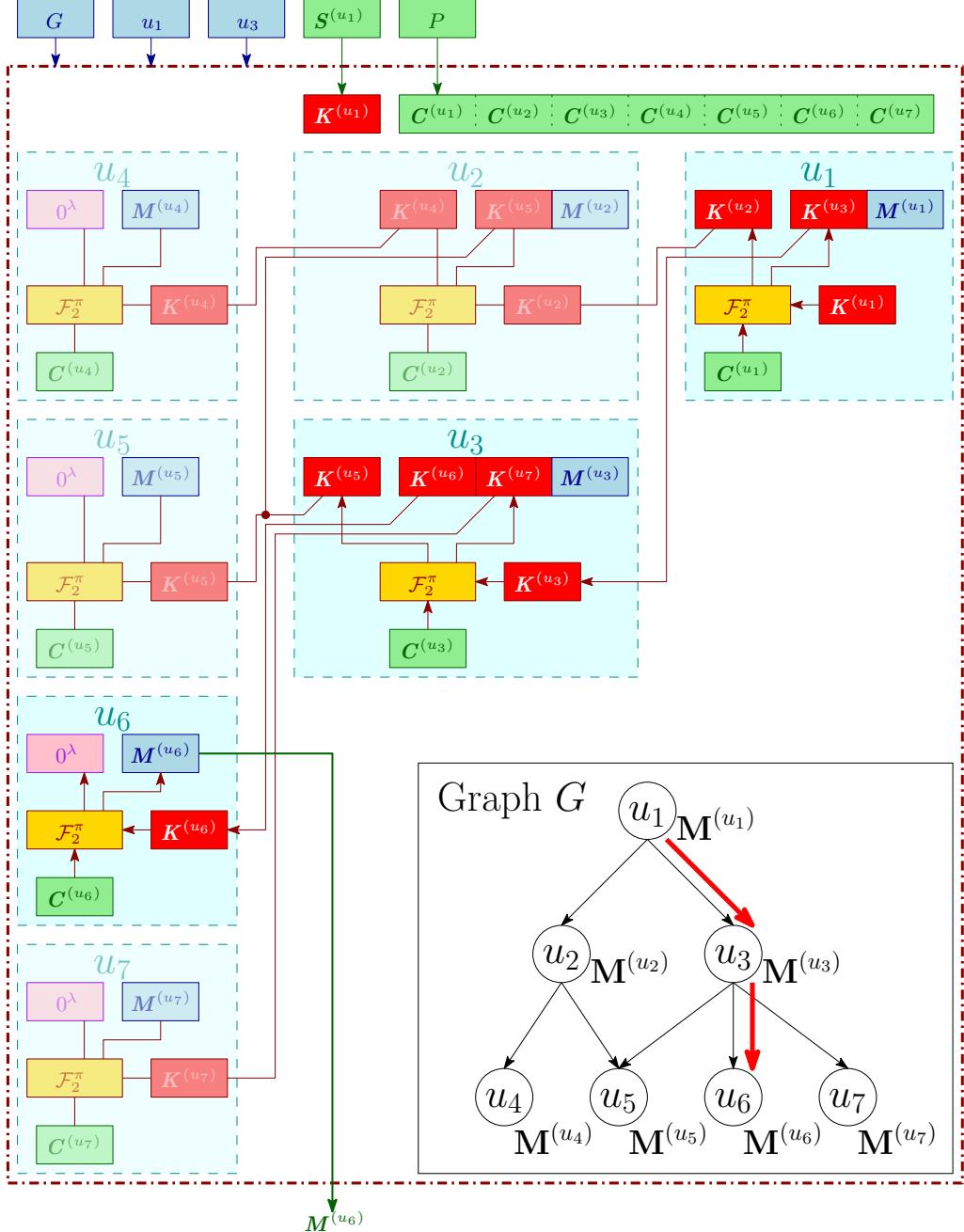
Figure 5.66: Algorithmic description of the *key derivation* function $\hat{\Psi} \cdot \mathcal{D}\mathcal{E}\mathcal{R}$.

that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description of $\hat{\Psi} \cdot \mathcal{D}\mathcal{E}\mathcal{R}(params^{(\hat{\Psi})}, G, u_1, u_6, \mathbf{S}^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , is given in Figure 5.65. The pseudocode is given in Figure 5.66. Very briefly, the algorithm works as follows: first it computes a path $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; then it initializes v and $\mathbf{K}^{(v)}$ with u_{i_1} and $\mathbf{S}^{(u_{i_1})}$; and finally it parses *public information* P into $(\mathbf{C}^{(u_i)})_{u_i \in V} := P$.

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}$, the following operations are executed: first it computes $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)})$; then it invokes function $\text{ch_seq}(v, G)$ which returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); next it assigns value of IV_2 to $\mathbf{K}^{(u_{k_1})}$; after that it parses $temp$ into $\mathbf{K}^{(u_{k_2})} \| \mathbf{K}^{(u_{k_3})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp$; and finally it searches next node in the path in sequence \tilde{v} , and extracts its corresponding key, before the next iteration begins.

Now, the key $\mathbf{K}^{(u_{i_2})}$, computed above, is returned.

- The *decryption* algorithm $\hat{\Psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $\mathbf{S}^{(u_{i_1})}$ and *public information*

Figure 5.67: Pictorial description of the *decryption* function $\hat{\Psi} \cdot \mathcal{D}$.

P , and outputs file $\mathbf{M}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial description of $\hat{\Psi} \cdot \mathcal{D}(\text{params}^{(\hat{\Psi})}, G, u_1, u_6, \mathbf{S}^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , is given in Figure 5.67. The pseu-

```

 $\hat{\Psi} \cdot \mathcal{D}(params^{(\hat{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 


---


#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 

#Computing file.
for ( $p := 1, 2, \dots, \ell + 2$ )
   $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)});$ 
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
  If ( $\tilde{v} = \text{NULL}$ )
    If ( $v = u_{i_2} \wedge IV_2 = 0^\lambda$ ), then return  $\mathbf{M}^{(v)} := temp;$ 
    Else return  $\perp$ ;
  Else  $\mathbf{K}^{(u_{k_1})} := IV_2; \quad \mathbf{K}^{(u_{k_2})} \| \mathbf{K}^{(u_{k_3})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp;$ 
  If ( $p \leq \ell + 1$ )
    Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p; \quad \mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}; \quad v := u_{j_p};$ 

#Verifying.
while ( $\tilde{v} \neq \text{NULL}$ )
   $\mathbf{K}^{(v)} := IV_2, v := u_{k_1}; \quad \tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
   $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)});$ 

#Computing final output.
If ( $IV_2 = 0^\lambda$ ), then return  $\mathbf{M}^{(u_{i_2})}$ ; Else return  $\perp$ ;

```

Figure 5.68: Algorithmic description of the *decryption* function $\hat{\Psi} \cdot \mathcal{D}$.

decode is given in Figure 5.68. The algorithm works as follows: first it computes a path $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; next it initializes v and $\mathbf{K}^{(v)}$ with u_{i_1} and $\mathbf{S}^{(u_{i_1})}$; and then it parses *public information* P into $(\mathbf{C}^{(u_i)})_{u_i \in V} := P$.

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_{\ell+1}}$, the following operations are executed: first it computes $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)})$; then it invokes function $\text{ch_seq}(v, G)$ that returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); after that it checks if v is a leaf-node, next it checks if $v = u_{i_2}$ and $IV_2 = 0^\lambda$, then returns $\mathbf{M}^{(v)} := temp$, otherwise returns an *invalid* string \perp ; then, if v is not a leaf-node, it assigns $\mathbf{K}^{(u_{k_1})} := IV_2$ and parses $temp$ into $\mathbf{K}^{(u_{k_2})} \| \mathbf{K}^{(u_{k_3})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp$; and finally it searches

next node in the path in sequence \tilde{v} , and extracts its corresponding key, before the next iteration begins.

To verify authentication of the file, the first child v of each node starting from u_{i_2} performs the following operations: first it computes key $\mathbf{K}^{(v)} := IV_2$; then it computes $(temp, IV_2) := \mathcal{F}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)})$, where IV_2 acts as key of the first child for execution of next iteration.

If the value of $IV_2 = 0^\lambda$ after the last iteration, then it returns file $\mathbf{M}^{(u_{i_2})}$ computed above, otherwise it returns an *invalid* string \perp .

5.8.1.2 Security of $\hat{\Psi}$

Theorem 30. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\hat{\Psi}$ has been defined in Section 5.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Psi}, \mathcal{A}, G}^{KASAE-KR}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)+1}{2^\lambda}.$$

Here, the KASAE-KR game is defined in Figure 5.1, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 18. ■

Theorem 31. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\hat{\Psi}$ has been defined in Section 5.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Psi}, \mathcal{A}, G}^{KASAE-IND}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-2}} + \frac{m(m-1)}{2^{\lambda-1}}.$$

Here, the KASAE-IND game is defined in Figure 5.2, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 19. ■

Theorem 32. *Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\hat{\Psi}$ has been defined in Section 5.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Psi}, \mathcal{A}, G}^{KASAE-INT}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}.$$

Here, the KASAE-INT game is defined in Figure 5.3, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 20. ■

5.8.2 Construction $\tilde{\Psi}$: *KAS-AE* based on *FP*

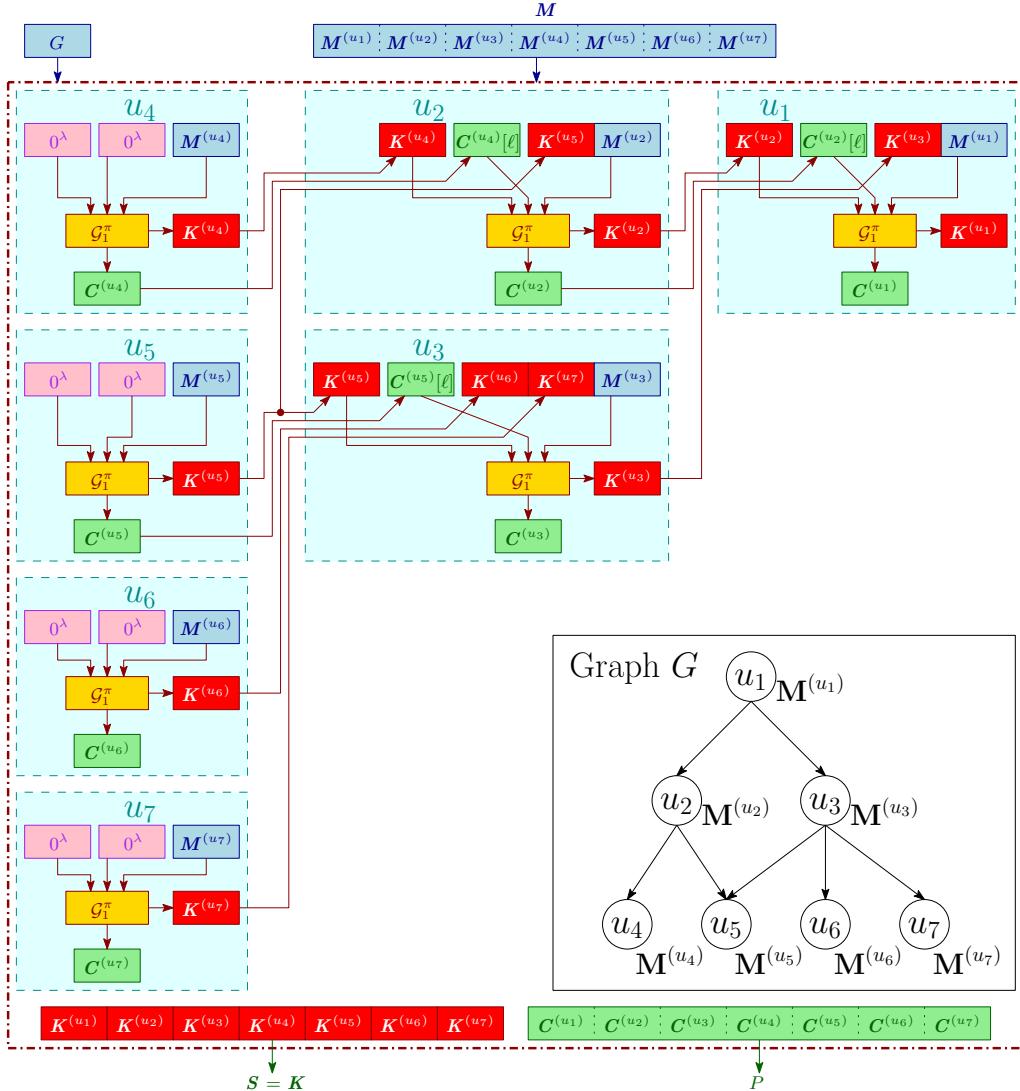
In Section 5.6.1.7, we have described two functions $\mathcal{G}_1^\pi(\cdot)$ and $\mathcal{G}_2^\pi(\cdot)$ whose designs are inspired from *FP* hash mode of operation. In this section, we utilize them again, for designing *KAS-AE* construction $\tilde{\Psi}$.

5.8.2.1 Description of $\tilde{\Psi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\tilde{\Psi}$ are given in Figures 5.69, 5.70, 5.71, 5.72, 5.73 and 5.74. Below, we give the textual description.

- The *setup* algorithm $\tilde{\Psi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\tilde{\Psi})}$, a set of *access graphs* $\Gamma^{(\tilde{\Psi})}$, *key-space* $\mathcal{K}^{(\tilde{\Psi})} := \{0, 1\}^\lambda$ and *message-space* $\mathcal{M}^{(\tilde{\Psi})} := \bigcup^{i \geq 1} \{0, 1\}^{i\lambda}$.
- The *encryption* algorithm $\tilde{\Psi}.\mathcal{E}(\cdot)$ is randomised that takes as input parameter $\text{params}^{(\tilde{\Psi})}$, graph G and vector of *files* \mathbf{M} , and outputs vector of *private information* \mathbf{S} , vector of *keys* \mathbf{K} and *public information* P . The pictorial description of $\tilde{\Psi}.\mathcal{E}(\text{params}^{(\tilde{\Psi})}, G, \mathbf{M})$ with access graph $G = (V, E)$, where $V = \{u_1, u_2, \dots, u_7\}$ and vector of *files* $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_7)}$, is given in Figure 5.69. The pseudocode is given in Figure 5.70. This encryption function is designed in such a way that any node u_i is able to decrypt the files of its successors. In order to do that, for each node u_i , we encrypt file $\mathbf{M}^{(u_i)}$ as well as decryption keys of children of u_i , such that, on decrypting ciphertext corresponding to u_i , decryption keys of all its children are revealed. Very briefly, the algorithm works as follows: first it assigns the level $\text{level}[u_i]$ to each node $u_i \in V$ and calculates maximum depth h of graph G , by computing $(\text{level}[], h) := \text{height}(G)$; and then it encrypts files at level h , followed by encryption of files at level $h - 1$, and so on, until root node is reached.

For each node u_i , the following operations are executed: first it invokes function $\text{ch_seq}(u_i, G)$ that returns sequence of children $(u_{i_1}, u_{i_2}, \dots, u_{i_d})$ of u_i (in ascending order); then it assigns key of the first child $\mathbf{K}^{(u_{i_1})}$ to IV_2 and last λ -bit block of ciphertext $\mathbf{C}^{(u_{i_1})}$ to IV_1 ; next it prepends the decryption keys $\mathbf{K}^{(u_{i_2})}, \mathbf{K}^{(u_{i_3})}, \dots, \mathbf{K}^{(u_{i_d})}$ – which have been already generated in previous iterations – to file $\mathbf{M}^{(u_i)}$ to compute $\text{temp} := \mathbf{K}^{(u_{i_2})} \parallel \mathbf{K}^{(u_{i_3})} \parallel \dots \parallel \mathbf{K}^{(u_{i_d})} \parallel \mathbf{M}^{(u_i)}$; and finally it computes a pair of decryption key and ciphertext $(\mathbf{K}^{(u_i)}, \mathbf{C}^{(u_i)}) :=$

Figure 5.69: Pictorial description of the *encryption* function $\tilde{\Psi}.\mathcal{E}$.

$\mathcal{G}_1^\pi(1^\lambda, \text{temp}, IV_1, IV_2)$. For the leaf nodes, value of IV_1 and IV_2 are 0^λ .

Now, the vectors of *private information* and *keys* $S := \mathbf{K} := (\mathbf{K}^{(u_i)})_{u_i \in V}$, and the *public information* $P := (\mathbf{C}^{(u_i)})_{u_i \in V}$.

- The *key derivation* algorithm $\tilde{\Psi}.\mathcal{DER}(\cdot)$ is deterministic that takes as input parameter $\text{params}^{(\tilde{\Psi})}$, graph G , two nodes u_{i_1} and u_{i_2} , such that $u_{i_2} \leq u_{i_1}$, node u_{i_1} 's *private information* $S^{(u_{i_1})}$ and *public information* P , and outputs key $\mathbf{K}^{(u_{i_2})}$ corresponding to u_{i_2} . The pictorial

```

 $\tilde{\Psi} \cdot \mathcal{E}(params^{(\tilde{\Psi})}, G, M)$ 


---


#Initialization.
 $(level[], h) := \text{height}(G); \quad M^{(u_1)} \circ M^{(u_2)} \circ \dots \circ M^{(u_m)} := M;$ 

#Computing values for each user level-by-level.
for  $(j := h, h - 1, \dots, 0)$ 
   $V_j := \text{nodes\_at\_level}(V, level[], j);$ 
  For all  $u_i \in V_j$ 
     $\tilde{u}_i \stackrel{\text{def}}{=} (u_{i_1}, u_{i_2}, \dots, u_{i_d}) := \text{ch\_seq}(u_i, G);$ 
    If  $(\tilde{u}_i = \text{NULL})$ 
       $IV_1 := IV_2 := 0^\lambda; \quad temp := M^{(u_i)};$ 
    Else
       $IV_1 := C^{(u_{i_1})}[last\_block]; \quad IV_2 := K^{(u_{i_1})};$ 
       $temp := K^{(u_{i_2})} \| K^{(u_{i_3})} \| \dots \| K^{(u_{i_d})} \| M^{(u_i)};$ 
       $(K^{(u_i)}, C^{(u_i)}) := \mathcal{G}_1^\pi(1^\lambda, temp, IV_1, IV_2);$ 

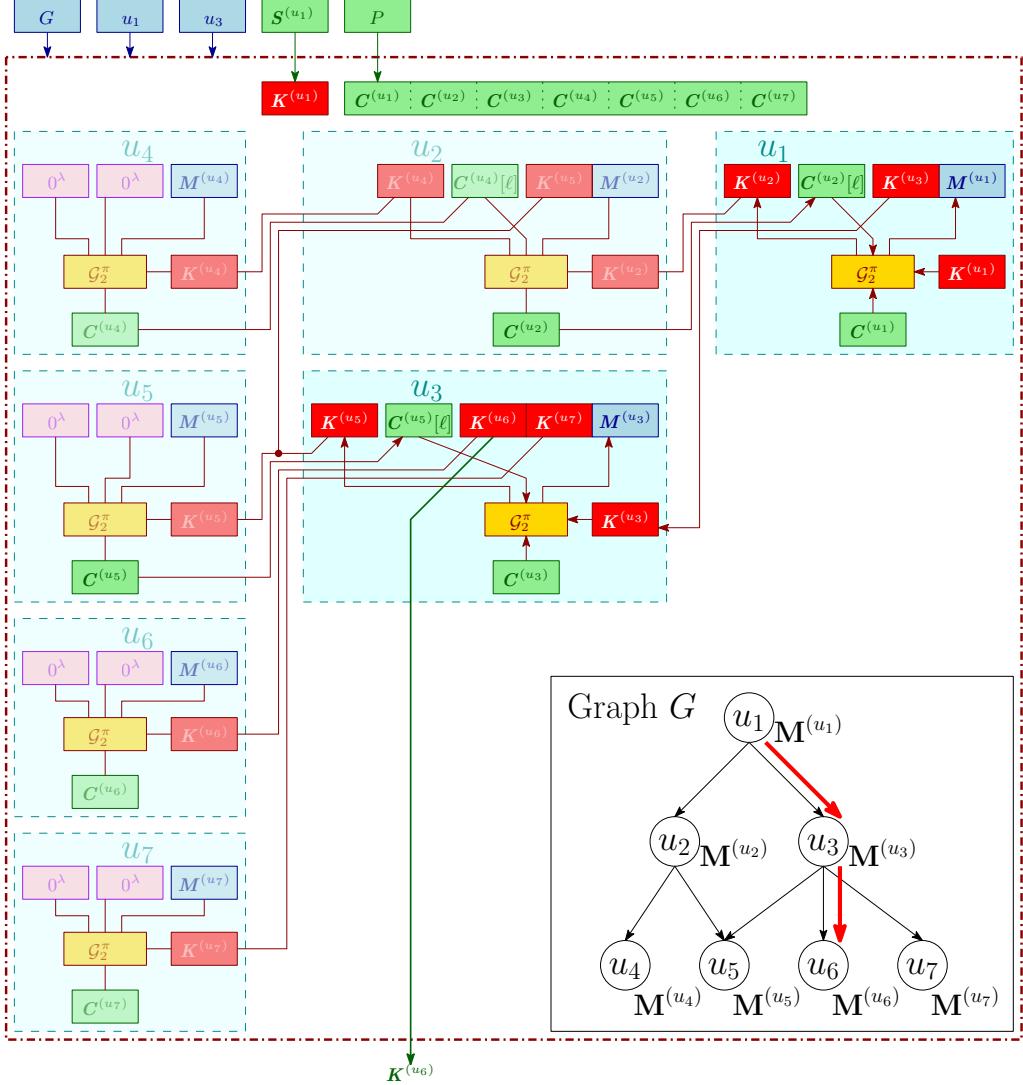
#Computing final output.
 $S := K := (K^{(u_i)})_{u_i \in V}; \quad P := (C^{(u_i)})_{u_i \in V};$ 
  return  $(S, K, P);$ 

```

Figure 5.70: Algorithmic description of the *encryption* function $\tilde{\Psi} \cdot \mathcal{E}$.

description of $\tilde{\Psi} \cdot \mathcal{DER}(params^{(\tilde{\Psi})}, G, u_1, u_6, S^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , is given in Figure 5.71. The pseudocode is given in Figure 5.72. Very briefly, the algorithm works as follows: first it computes a path $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; then it initializes v and $K^{(v)}$ with u_{i_1} and $S^{(u_{i_1})}$; and finally it parses *public information* P into $(C^{(u_i)})_{u_i \in V} := P$.

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}$, the following operations are executed: first it invokes function $\text{ch_seq}(v, G)$ that returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); after that it checks if v is a leaf node, then it initializes $IV_1 := 0^\lambda$, otherwise it computes $IV_1 := C^{(u_{k_1})}[last_block]$; next it computes $(temp, IV_2) := \mathcal{G}_2^\pi(1^\lambda, K^{(v)}, C^{(v)}, IV_1)$; after that it assigns value of IV_2 to $K^{(u_{k_1})}$; then it parses $temp$ into $K^{(u_{k_2})} \| K^{(u_{k_3})} \| \dots \| K^{(u_{k_d})} \| M^{(v)} := temp$; and finally it searches next node in the path in sequence \tilde{v} , and extracts its corresponding key, before the next iteration begins.

Figure 5.71: Pictorial description of the *key derivation function* $\tilde{\Psi} \cdot \mathcal{DER}$.

Now, the key $\mathbf{K}^{(u_{i2})}$, computed above, is returned.

- The *decryption* algorithm $\tilde{\Psi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\tilde{\Psi})}$, graph G , two nodes u_{i1} and u_{i2} , such that $u_{i2} \leq u_{i1}$, node u_{i1} 's *private information* $S^{(u_{i1})}$ and *public information* P , and outputs file $\mathbf{M}^{(u_{i2})}$ corresponding to u_{i2} . The pictorial description of $\tilde{\Psi} \cdot \mathcal{D}(params^{(\tilde{\Psi})}, G, u_1, u_6, S^{(u_1)}, P)$ using the path (u_1, u_3, u_6) which is shown in red in graph G , given in Figure 5.73. The pseudocode is given in Figure 5.74. Very briefly, the algorithm works as follows: first it computes a path $(u_{i1}, u_{j1}, u_{j2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} =$

```

 $\tilde{\Psi} \cdot \mathcal{DER}(params^{(\tilde{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 

#Computing key.
for  $(p := 1, 2, \dots, \ell + 1)$ 
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
  If  $(\tilde{v} \neq \text{NULL})$ , then  $IV_1 := \mathbf{C}^{(u_{k_1})}[\text{last\_block}]$ ;
  Else  $IV_1 := 0^\lambda$ ;
   $(temp, IV_2) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, IV_1);$ 
   $\mathbf{K}^{(u_{k_1})} := IV_2;$ 
   $\mathbf{K}^{(u_{k_2})} \parallel \mathbf{K}^{(u_{k_3})} \parallel \dots \parallel \mathbf{K}^{(u_{k_d})} \parallel \mathbf{M}^{(v)} := temp;$ 
  Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p$ ;
   $\mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}; \quad v := u_{j_p};$ 

#Computing final output.
return  $\mathbf{K}^{(u_{i_2})};$ 

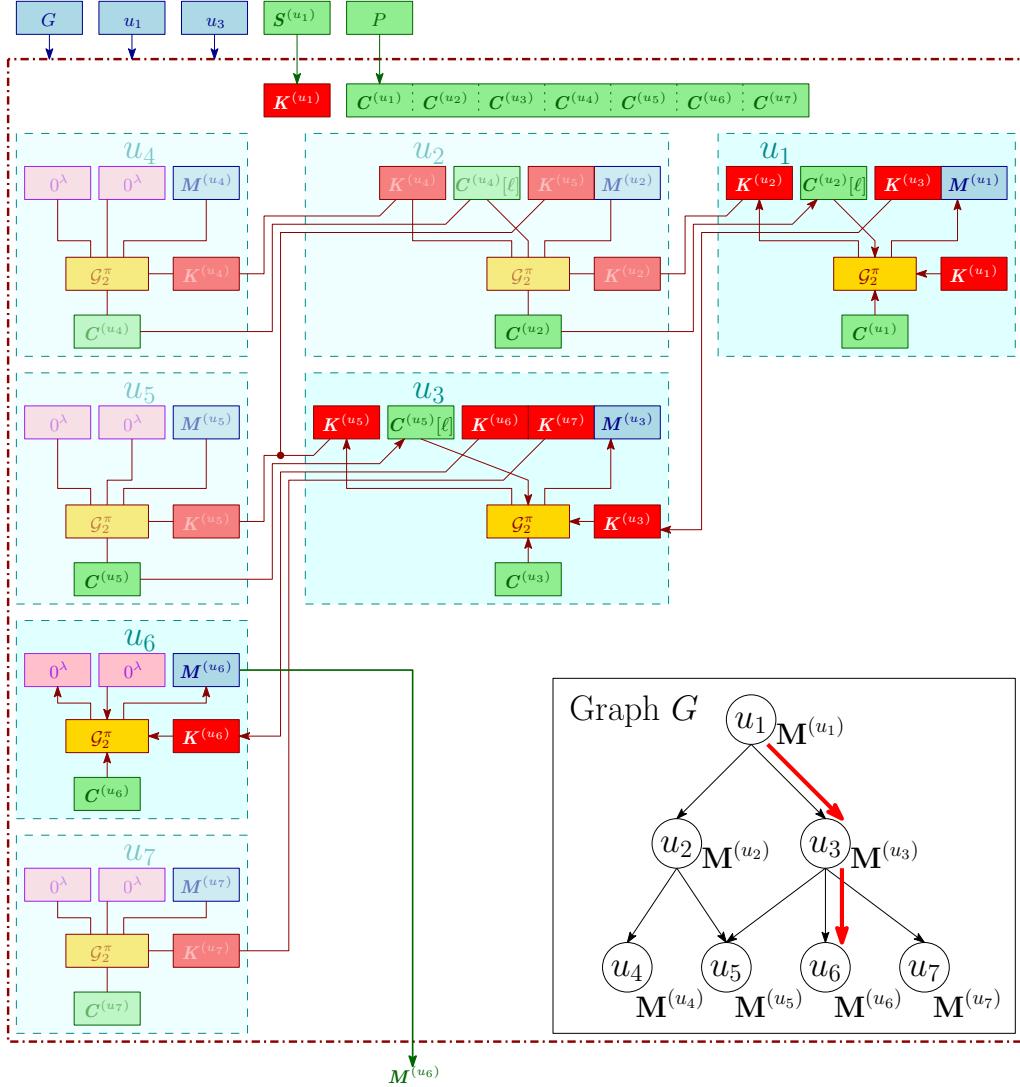
```

Figure 5.72: Algorithmic description of the *key derivation* function $\tilde{\Psi} \cdot \mathcal{DER}$.

$u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2})$ from u_{i_1} to u_{i_2} ; next it initializes v and $\mathbf{K}^{(v)}$ with u_{i_1} and $\mathbf{S}^{(u_{i_1})}$; and then it parses *public information* P into $(\mathbf{C}^{(u_i)})_{u_i \in V} := P$.

For all the successive nodes $v = u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_{\ell+1}}$, the following operations are executed: first it invokes function $\text{ch_seq}(v, G)$ that returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); after that it checks if v is a leaf node, then it initializes $IV_1 := 0^\lambda$, otherwise it computes $IV_1 := \mathbf{C}^{(u_{k_1})}[\text{last_block}]$; next it computes $(temp, IV_2) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, IV_1)$; after that it checks if v is a leaf-node, then it checks if $v = u_{i_2}$ and $IV_2 = 0^\lambda$, then it returns $\mathbf{M}^{(v)} := temp$, otherwise it returns an *invalid* string \perp ; then, if v is not a leaf-node, it assigns $\mathbf{K}^{(u_{k_1})} := IV_2$ and parses $temp$ into $\mathbf{K}^{(u_{k_2})} \parallel \mathbf{K}^{(u_{k_3})} \parallel \dots \parallel \mathbf{K}^{(u_{k_d})} \parallel \mathbf{M}^{(v)} := temp$; and finally it searches next node in the path in sequence \tilde{v} , and extracts its corresponding key, before the next iteration begins.

To verify authentication of the file, the first child v of each node starting from u_{i_2} performs the following operations: first it computes key

Figure 5.73: Pictorial description of the decryption function $\tilde{\Psi} \cdot \mathcal{D}$.

$\mathbf{K}^{(v)} := IV_2$, then it invokes function $\text{ch_seq}(v, G)$ which returns sequence of children $(u_{k_1}, u_{k_2}, \dots, u_{k_d})$ of v (in ascending order); next it checks if v is a leaf node, then it initializes $IV_1 := 0^\lambda$, otherwise it computes $IV_1 := C^{(u_{k_1})}[\text{last_block}]$; and finally it computes $(\text{temp}, IV_2) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, IV_1)$, where IV_2 acts as key of the first child for execution of next iteration.

If the value of $IV_2 = 0^\lambda$ after last iteration, then it returns file $M^{(u_{i_2})}$ computed above, otherwise it returns *invalid* string \perp .

```

 $\tilde{\Psi} \cdot \mathcal{D}(params^{(\tilde{\Psi})}, G, u_{i_1}, u_{i_2}, \mathbf{S}^{(u_{i_1})}, P)$ 

#Initialization.
 $(u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}) := \text{path}(G, u_{i_1}, u_{i_2});$ 
 $v := u_{i_1}; \quad \mathbf{K}^{(v)} := \mathbf{S}^{(u_{i_1})}; \quad (\mathbf{C}^{(u_i)})_{u_i \in V} := P;$ 

#Computing file.
for ( $p := 1, 2, \dots, \ell + 2$ )
   $\tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
  If ( $\tilde{v} \neq \text{NULL}$ ), then  $IV_1 := \mathbf{C}^{(u_{k_1})}[\text{last\_block}]$ ;
  Else  $IV_1 := 0^\lambda$ ;
   $(temp, IV_2) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, IV_1);$ 
  If ( $\tilde{v} = \text{NULL}$ )
    If ( $v = u_{i_2} \wedge IV_2 = 0^\lambda$ ), then return  $\mathbf{M}^{(v)} := temp$ ;
    Else return  $\perp$ ;
  Else  $\mathbf{K}^{(u_{k_1})} := IV_2; \quad \mathbf{K}^{(u_{k_2})} \| \mathbf{K}^{(u_{k_3})} \| \dots \| \mathbf{K}^{(u_{k_d})} \| \mathbf{M}^{(v)} := temp$ ;
  If ( $p \leq \ell + 1$ )
    Find  $u_{k_q} \in \tilde{v}$ , s.t.  $k_q = j_p$ ;  $\mathbf{K}^{(v)} := \mathbf{K}^{(u_{k_q})}; \quad v := u_{j_p}$ ;

#Verifying.
while ( $\tilde{v} \neq \text{NULL}$ )
   $\mathbf{K}^{(v)} := IV_2, v := u_{k_1}; \quad \tilde{v} \stackrel{\text{def}}{=} (u_{k_1}, u_{k_2}, \dots, u_{k_d}) := \text{ch\_seq}(v, G);$ 
  If ( $\tilde{v} \neq \text{NULL}$ ), then  $IV_1 := \mathbf{C}^{(u_{k_1})}[\text{last\_block}]$ ;
  Else  $IV_1 := 0^\lambda$ ;
   $(temp, IV_2) := \mathcal{G}_2^\pi(1^\lambda, \mathbf{K}^{(v)}, \mathbf{C}^{(v)}, IV_1);$ 

#Computing final output.
If ( $IV_2 = 0^\lambda$ ), then return  $\mathbf{M}^{(u_{i_2})}$ ;
Else return  $\perp$ ;

```

Figure 5.74: Algorithmic description of the *decryption* function $\tilde{\Psi} \cdot \mathcal{D}$.

5.8.2.2 Security of $\tilde{\Psi}$

Theorem 33. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\tilde{\Psi}$ has been defined in Section 5.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Psi}, \mathcal{A}, G}^{\text{KASAE-KR}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)+1}{2^\lambda}.$$

Here, the KASAE-KR game is defined in Figure 5.1, and the message-

length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 18. \blacksquare

Theorem 34. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\tilde{\Psi}$ has been defined in Section 5.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Psi}, \mathcal{A}, G}^{\text{KASAE-IND}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-2}} + \frac{m(m-1)}{2^{\lambda-1}}.$$

Here, the KASAE-IND game is defined in Figure 5.2, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 19. \blacksquare

Theorem 35. Let $\lambda \in \mathbb{N}$ be the security parameter. The KAS-AE scheme $\tilde{\Psi}$ has been defined in Section 5.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Psi}, \mathcal{A}, G}^{\text{KASAE-INT}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m(m-1)}{2^\lambda}.$$

Here, the KASAE-INT game is defined in Figure 5.3, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 20. \blacksquare

5.9 Comparison of KAS-AE schemes

For the access graph $G = (V, E)$ and the vector of files $\mathbf{M} = \mathbf{M}^{(u_1)} \circ \mathbf{M}^{(u_2)} \circ \dots \circ \mathbf{M}^{(u_n)}$, we use the following notation: λ is the security parameter; $n = |V|$ is the number of nodes (or security classes); w is the width of the access graph G ; $\deg(u_i)$ is the number of children of node $u_i \in V$; and $|\mathbf{M}| = \sum_{u_i \in V} |\mathbf{M}^{(u_i)}|$. Also, we consider the key and tag sizes to be λ bits each. Based on the definitions of the key derivation algorithm $\psi \cdot \mathcal{DER}$ and decryption algorithm $\psi \cdot \mathcal{D}$ of the KAS-AE-chain scheme (defined in Section 5.3), the chain C_g , and vertices u_h^g and \hat{u}_g (discussed in Sections 3.2.2 and 5.6), we define the following sets: $U_1 := \{u_i \in C_g \mid u_h^g \leq u_i \leq \hat{u}_g\}$; $U_2 := \{u_i \in C_g \mid u_i \leq \hat{u}_g\}$, so $U_1 \subseteq U_2$; $U_3 := \{u_{i_1}, u_{j_1}, u_{j_2}, \dots, u_{j_\ell}, u_{j_{\ell+1}} = u_{i_2}\}$, such that $u_{j_1} \lessdot u_{i_1}, u_{j_2} \lessdot u_{j_1}, \dots, u_{i_2} \lessdot u_{i_\ell}$; and $U_4 := U_3 \cup \{u_{i_2}, u_{k_1}, u_{k_2}, \dots, u_{k_d}\}$, such that u_{k_1} is the first child of u_{i_2} , u_{k_2} is the first child of u_{k_1} , and so on.

Here, C_g is a partition of V forming a chain that contains the nodes \hat{u}_g and u_h^g , such that $u_h^g \leq \hat{u}_g$ (see Section 3.2.2).

For the KAS construction $\Omega = (\Omega \cdot \mathcal{GEN}, \Omega \cdot \mathcal{DER})$:

Constructions → Parameters ↓	$\hat{\Psi}_{\text{TKAS}}$ (Based on TKAS [CC02, YL04])	$\hat{\Psi}_{\text{TKEKAS}}$ (Based on TKEKAS [SC02, TC95])	$\hat{\Psi}_{\text{DKEKAS}}$ (Based on DKEKAS [Gud80, ZRM01])	$\hat{\Psi}_{\text{IKEKAS}}$ (Based on IKEKAS [ABFF09, AFB05, CH05, SFM07a])	$\hat{\Psi}_{\text{NBKAS}}$ (Based on NBKAS [AT83, CHW92, HL90])
STORAGE REQUIREMENT					
• S	$2n^2\lambda$	$2n^2\lambda$	$n^2\lambda + n\lambda$	$n^2\lambda + n\lambda$	$n^2\lambda + n\lambda$
• P	$ \mathbf{M} $	$ \mathbf{M} + n\lambda$	$ \mathbf{M} + n^2\lambda$	$ \mathbf{M} + n\lambda$	$ \mathbf{M} + n\lambda$
RUNNING TIME					
• \mathcal{E}	$n \cdot c_{\mathcal{K}} + c_{\varrho, \mathcal{E}_\lambda} \cdot \frac{ \mathbf{M} }{\lambda}$	$c_{\varrho, \mathcal{E}_\lambda} \left(n + \frac{ \mathbf{M} }{\lambda} \right)$	$2n \cdot c_{\mathcal{K}} + c_{\varrho, \mathcal{E}_\lambda} \left(n^2 + \frac{ \mathbf{M} }{\lambda} \right)$	$n \cdot c_{\mathcal{K}} + c_{\varrho, \mathcal{E}_\lambda} \left(n + \frac{ \mathbf{M} }{\lambda} \right)$	$n \cdot (c_e + c_{\mathcal{K}_g}) + c_{\varrho, \mathcal{E}_\lambda} \cdot \frac{ \mathbf{M} }{\lambda}$
• \mathcal{DER}	$\mathcal{O}(1)$	$c_{\varrho, \mathcal{D}_\lambda}$	$c_{\varrho, \mathcal{D}_\lambda}$	$\mathcal{O}(n) \cdot c_{\varrho, \mathcal{D}_\lambda}$	$c_{\mathcal{K}_g}$
• \mathcal{D}	$c_{\varrho, \mathcal{D}_\lambda} + \frac{\mathcal{O}(1)}{ \mathbf{M}^{(u_{i_2})} }$	$c_{\varrho, \mathcal{D}_\lambda} + \frac{c_{\varrho, \mathcal{D}_\lambda} + c_{\varrho, \mathcal{D}_\lambda} \frac{ \mathbf{M}^{(u_{i_2})} }{\lambda}}{ \mathbf{M}^{(u_{i_2})} }$	$c_{\varrho, \mathcal{D}_\lambda} + \frac{c_{\varrho, \mathcal{D}_\lambda} + c_{\varrho, \mathcal{D}_\lambda} \frac{ \mathbf{M}^{(u_{i_2})} }{\lambda}}{ \mathbf{M}^{(u_{i_2})} }$	$\mathcal{O}(n) \cdot c_{\varrho, \mathcal{D}_\lambda} + c_{\varrho, \mathcal{D}_\lambda} + c_{\varrho, \mathcal{D}_\lambda} \frac{ \mathbf{M}^{(u_{i_2})} }{\lambda}$	

Table 5.1: Comparison table for *KAS-AE* schemes described in Section 5.5.2.

Constructions → Parameters ↓	Construction $\bar{\Psi}$ (Sections 5.6.2 & 5.6.1.1)	Construction $\check{\Psi}$ (Sections 5.6.2 & 5.6.1.3)	Construction $\dot{\Psi}$ (Sections 5.6.2 & 5.6.1.5)	Construction $\ddot{\Psi}$ (Sections 5.6.2 & 5.6.1.7)
STORAGE REQUIREMENT				
• S	$(n^2 + nw)\lambda$	$nw\lambda$	$nw\lambda$	$nw\lambda$
• P	$ M $	$ M + 2n\lambda$	$ M + n\lambda$	$ M + n\lambda$
RUNNING TIME				
• E	$c_{\Omega, gE\mathcal{N}} + c_{\varrho, \mathcal{E}_\lambda} \cdot \frac{ M }{\lambda}$	$c_{\phi, \mathcal{E}_\lambda} \cdot \left(2n + \frac{ M }{\lambda}\right)$	$c_{\mathcal{F}_1^\pi} \cdot \left(n + \frac{ M }{\lambda}\right)$	$c_{\mathcal{G}_1^\pi} \cdot \left(n + \frac{ M }{\lambda}\right)$
• $\mathcal{D}\mathcal{E}\mathcal{R}$	$c_{\Omega, \mathcal{D}\mathcal{E}\mathcal{R}}$	$c_{\phi, \mathcal{D}_\lambda} \cdot \sum_{u_i \in U_1} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$	$c_{\mathcal{F}_2^\pi} \cdot \sum_{u_i \in U_1} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$	$c_{\mathcal{G}_2^\pi} \cdot \sum_{u_i \in U_1} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$
• \mathcal{D}	$c_{\Omega, \mathcal{D}\mathcal{E}\mathcal{R}} + c_{\varrho, \mathcal{D}_\lambda} \cdot \frac{ M^{(u_h^g)} }{\lambda}$	$c_{\phi, \mathcal{D}_\lambda} \cdot \sum_{u_i \in U_1} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$	$c_{\mathcal{F}_2^\pi} \cdot \sum_{u_i \in U_2} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$	$c_{\mathcal{G}_2^\pi} \cdot \sum_{u_i \in U_2} \left(1 + \frac{ M^{(u_i)} }{\lambda}\right)$
Computation Assumption	Secure KAS & Secure AE	Secure MLE	Ideal Permutation	Ideal Permutation

Table 5.2: Comparison table for $KAS-AE$ schemes described in Section 5.6.

Constructions → Parameters ↓	Construction $\check{\Psi}$ (Section 5.7)	Construction $\hat{\Psi}$ (Section 5.8.1)	Construction $\tilde{\Psi}$ (Section 5.8.2)
STORAGE REQUIREMENT			
• S	$n\lambda$	$n\lambda$	$n\lambda$
• P	$\sum_{u_i \in V} (\deg(u_i) \cdot \lambda + \lambda + \mathbf{M}^{u_i})$	$\sum_{u_i \in V} (\deg(u_i) \cdot \lambda + \mathbf{M}^{u_i})$	$\sum_{u_i \in V} (\deg(u_i) \cdot \lambda + \mathbf{M}^{u_i})$
RUNNING TIME			
• \mathcal{E}	$c_{\phi, \mathcal{E}_\lambda} \cdot \left(\frac{ \mathbf{M} }{\lambda} + \sum_{u \in V} (\deg(u) + 1) \right)$	$c_{\mathcal{F}_1^\pi} \cdot \left(\frac{ \mathbf{M} }{\lambda} + \sum_{u \in V} \deg(u) \right)$	$c_{\mathcal{G}_1^\pi} \cdot \left(\frac{ \mathbf{M} }{\lambda} + \sum_{u \in V} \deg(u) \right)$
• \mathcal{DER}	$c_{\phi, \mathcal{D}_\lambda} \cdot \left(\sum_{u_i \in U_3} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) + 1 \right) \right)$	$c_{\mathcal{F}_2^\pi} \cdot \sum_{u_i \in U_3} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) \right)$	$c_{\mathcal{G}_2^\pi} \cdot \sum_{u_i \in U_3} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) \right)$
• \mathcal{D}	$c_{\phi, \mathcal{D}_\lambda} \cdot \sum_{u_i \in U_3} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) + 1 \right)$	$c_{\mathcal{F}_2^\pi} \cdot \sum_{u_i \in U_4} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) \right)$	$c_{\mathcal{G}_2^\pi} \cdot \sum_{u_i \in U_4} \left(\frac{ \mathbf{M}^{(u_i)} }{\lambda} + \deg(u_i) \right)$
Computation Assumption	Secure MLE	Ideal Permutation	Ideal Permutation

Table 5.3: Comparison table for *KAS-AE* schemes described in Sections Sections 5.7 and 5.8.

- $c_{\Omega.\mathcal{GEN}}$ and $c_{\Omega.\mathcal{DER}}$ denote the running time of *key generation* and *key derivation* algorithms $\Omega.\mathcal{GEN}$ and $\Omega.\mathcal{DER}$.
- c_K denotes the cost of generating single λ -bit key, for the schemes $\hat{\Psi}_\Omega$, where $\Omega \in \{\text{TKAS}, \text{TKEKAS}, \text{DKEKAS}, \text{IKEKAS}\}$.
- c_e and $c_{\mathcal{K}_g}$ denote the cost of generating one public value e and generating one λ -bit key from a given e value in $\hat{\Psi}_{(\text{NBKAS})}$.

For the *AE* scheme $\varrho = (\varrho.\mathcal{GEN}, \varrho.\mathcal{E}, \varrho.\mathcal{D})$, $c_{\varrho.\mathcal{E}_\lambda}$ and $c_{\varrho.\mathcal{D}_\lambda}$ denote the running times of algorithms $\varrho.\mathcal{E}$ and $\varrho.\mathcal{D}$ for a λ -bit input.

For the *MLE* scheme $\phi = (\phi.\mathcal{E}, \phi.\mathcal{D})$, $c_{\phi.\mathcal{E}_\lambda}$ and $c_{\phi.\mathcal{D}_\lambda}$ denote the running times of algorithms $\phi.\mathcal{E}$ and $\phi.\mathcal{D}$ for a λ -bit input.

The $c_{\mathcal{F}_1^\pi}$, $c_{\mathcal{F}_2^\pi}$, $c_{\mathcal{G}_1^\pi}$ and $c_{\mathcal{G}_2^\pi}$ denote the running times of the algorithms \mathcal{F}_1^π , \mathcal{F}_2^π , \mathcal{G}_1^π and \mathcal{G}_2^π that uses a 2λ -bit permutation.

5.10 Conclusion and Future Work

In this chapter, we presented a new cryptographic primitive, namely, *KAS-AE*, and designed three efficient constructions of it. We showed that these constructions performed better – both with respect to time and memory – than the existing mechanisms to solve the well-known *hierarchical access control* problem relevant for any multi-layered organization. The high performance of our schemes is attributed to its very unique (and rare to find) *reverse decryption* property. We leave it as an open problem to design more constructions with this property. Another future work in this line of research will be to add more functions to *KAS-AE*, such as *key revocation* and *file update*, and find efficient constructions.

CHAPTER 6

All-Or-Nothing Transform (AONT)

6.1 Introduction

ORIGIN. The notion of *all-or-nothing transform (AONT)* was first conceived by Rivest more than two decades ago (1997), to be used as a pre-processing step to enhance the security of conventional *blockcipher-based symmetric encryption (SE)* against the *key recovery* attack. In such an attack, the adversary exploits the property that, on the decryption side, a plaintext block can be viewed as a function of just *one* (or at most two) ciphertext block(s), rather than a function of *all* of them. Therefore, if the ciphertext is long, say of 2^{20} blocks, then the *key recovery* attack on a conventional mode turns out to be 2^{20} times faster than that with *AONT*. This cardinal property is called *all-or-nothing (AON)*, where the recovery of each plaintext bit requires the knowledge of *all* the ciphertext bits.

APPLICATIONS. Although invented more than two decades ago, efficient *AONT* schemes are, nowadays, finding usefulness in numerous modern applications for securing communications and database systems. Besides prevention of *key recovery* attacks as described above, *AONT* mitigates *differential side-channel* weaknesses of encryption protocols, and amplifies *memory erasability* in storage devices [CEM16, MTW⁺14].¹ It is also used

¹The *memory erasability* property ensures that it should be *hard* to recover even a single bit of a file, once it is deleted. Removing just one block of an *AONT*-transformed file, ensures memory erasability.

to construct *remotely-keyed encryption (RKE)*, *multi-stage secret sharing*, *secure regenerating codes*, and *key update mechanism* for network storage of encrypted data, among others [FEA09, KK14, SSR00, WY15].

MOTIVATION. We have already discussed at length the great variety of applications *AONT* schemes have. Interestingly, all the practical *AONT* schemes have, so far, been constructed using blockciphers [Des00, Riv97]. However, the blockcipher-based primitives have recently met with a generous dose of scepticism: it often comes with a weak *key schedule* algorithm [DSST17, KSW96, Knu95]; when implemented in hardware, it usually requires more hardware gates (this often becomes prohibitive in *lightweight* devices) [ABB⁺14, MP12]; last but not the least, from the theoretical standpoint, an ideal (block)cipher is a stronger assumption (than that of an ideal permutation) [Nat12].

All the aforementioned issues are easily avoided, if the blockciphers could be replaced by permutations. Therefore, we ask the following question:

Is it possible to design a secure and efficient *AONT* scheme based on a permutation, instead of a blockcipher?

Searching for the answer to this question is our prime focus in this chapter.

6.2 Contribution of this Chapter

So far, all practical *AONTs* are blockcipher-based. In this chapter, we explore how to design efficient *AONT* constructions based on a fixed permutation. While studying this primitive, we found that the existing definitions lack some parameters and correctness properties. Therefore, we first refine the existing definition of *AONT*, and then give two concrete constructions based on a fixed permutation; these are the first-ever *AONT* schemes based on permutations. We have given justification in favour of the high efficiency of the proposed constructions, as compared to the existing blockcipher-based ones. The high efficiency obtained here is attributed to the *reverse decryption* property, found in the *APE* authenticated encryption (DIAC 2012, FSE 2014) and the *FP* hash mode of operation (Indocrypt 2012).

6.3 Related Work

RELATED WORK. As already described, Rivest first gave the notion of cryptographic primitive *AONT* in 1997; he proposed a blockcipher-based *AONT* scheme called “Package Transform” [Riv97]. In 1999, Boyko gave a formal definition of *AONT* along with the two security notions, namely *semantic* and *indistinguishability* security, and proposed an *AONT* construction based on *optimal asymmetric encryption padding (OAEP)* [Boy99]. Desai (2000) introduced a new security notion of *non-separability of keys*; he also improved the *AONT* scheme “Package Transform” proposed by Rivest, and finally gave a formal proof of security [Des00]. In the same year, Shin, Shin and Rhee proposed an *AONT*-based *remotely-keyed encryption (RKE)* scheme [SSR00].

In 2001, Stinson gave a mathematical definition of *AONT* in terms of the *entropy function*; he also proposed an *addition*-based *AONT* construction, and analyzed some results on the existence and non-existence of general transforms for different values of the parameters [Sti01]. In 2004, Zhang, Hanaoka and Imai proposed a new definition of *AONT* that guarantees *indistinguishability* against *chosen ciphertext attack*, and proposed two constructions: the first one is based on the *full-domain cipher*, and the second one on *hash functions* [ZHI04]. Marnas, Angelis and Bleris, in 2007, proposed an *AONT* scheme based on *quasigroups* [MAB07]. In 2009, Fatemi, Eghlidos and Aref proposed a *multi-stage secret sharing* scheme based on *AONT* [FEA09].

Kuwakado and Kurihara (2014) proposed an *AONT*-based scheme to generate *secure regenerating codes* [KK14]. In 2014, McEvoy *et al.* proposed the use of *AONT* as a countermeasure against the *differential side-channel analysis* [MTW⁺14]. In 2016, Camenisch, Enderlein and Maurer proposed the use of *AONT* for memory erasability amplification [CEM16]. D’Arco, Esfahani and Stinson, in 2016, studied the *t-AONT* schemes (that is, an *AONT* scheme is *AON* secure when *t* blocks are missing), and also proposed and analysed five methods of designing an *AONT* scheme based on *invertible matrices* [DES16].

In 2018, Esfahani, Goldberg and Stinson have discussed some results on the existence of *t-AONT* over arbitrary alphabet [EGS18]. Kapusta and Memmi, in 2018, have used *AONT* to protect the outsourced data against key recovery [KM18b]. In 2018, in an another paper, Kapusta and Memmi have proposed a scheme that combines partial encryption of the segments of blocks with the *AONT* [KM18a]. In 2019, Kapusta, Memmi and Rambaud formulated a new security model, introduced a new *circular*

AONT (CAONT) algorithm that provides higher security bound, and also defined the security properties that are easy to formalise under standard cryptographic hypothesis [KMR19].

6.4 All-Or-Nothing Transform (AONT)

We first refine the existing definition of *AONT* given by Desai [Des00] by: concretely defining the scope and the usage of the *setup* algorithm; adding the definitions of *message-space* and *pseudo-message-space*; giving a quantification of the pseudo-message expansion; finally establishing the correctness properties. Below we elaborately discuss this refined definition of *AONT*.

6.4.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *AONT* scheme $\Pi = (\Pi, \mathcal{T}, \Pi, \mathcal{I})$ is a pair of algorithms over the *setup* algorithm Π .*Setup*, satisfying the following conditions.

1. The PPT *setup* algorithm Π .*Setup*(1^λ) outputs parameter $params^{(\Pi)}$, *message-space* $\mathcal{M}^{(\Pi)} \subseteq \{0, 1\}^*$ and *pseudo-message-space* $\mathcal{P}^{(\Pi)} \subseteq \{0, 1\}^*$.
2. The PPT *transformation* algorithm Π . $\mathcal{T}(\cdot)$ takes as input parameter $params^{(\Pi)}$ and message $M \in \mathcal{M}^{(\Pi)}$, and returns a pseudo-message $P := \Pi$. $\mathcal{T}(params^{(\Pi)}, M)$, where $P \in \mathcal{P}^{(\Pi)}$.
3. The *inverse transformation* algorithm Π . $\mathcal{I}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $params^{(\Pi)}$ and pseudo-message $P' \in \{0, 1\}^*$, and returns message $M := \Pi$. $\mathcal{I}(params^{(\Pi)}, P')$, where $M \in \mathcal{M}^{(\Pi)} \cup \{\perp\}$.

Note: We restrict pseudo-message expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|P| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\Pi)}$.

6.4.2 Correctness

This condition requires that if the pseudo-message is generated from the correct input, *inverse transformation* operation will produce the correct output. Mathematically, the *correctness* of an *AONT* can be defined as follows.

The correctness of Π requires that for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\Pi)}$, we have:

$$\Pi.\mathcal{I}(\textit{params}^{(\Pi)}, \Pi.\mathcal{T}(\textit{params}^{(\Pi)}, M)) = M.$$

6.4.3 Security definitions

In Figure 6.1, we define the security game **AONT-AON** for the *AONT* scheme $\Pi = (\Pi, \mathcal{T}, \Pi, \mathcal{I})$. As usual, the game is written using the challenger-adversary framework.

```

Game AONT-AON $\Pi$  $\mathcal{A}$ ( $1^\lambda, b$ )
( $\textit{params}^{(\Pi)}, \mathcal{M}^{(\Pi)}, \mathcal{P}^{(\Pi)}$ ) :=  $\Pi.\text{Setup}(1^\lambda)$ ;
( $M, S$ ) :=  $\mathcal{A}_1(1^\lambda, \Pi)$ ;
 $P_1 := \Pi.\mathcal{T}(\textit{params}^{(\Pi)}, M)$ ;  $P_0 \xleftarrow{\$} \{0, 1\}^{|P_1|}$ ;
 $P_0[1] \| P_0[2] \| \cdots \| P_0[m] := P_0$ ;
 $P_1[1] \| P_1[2] \| \cdots \| P_1[m] := P_1$ ;
 $b' := \mathcal{A}_2^\gamma(1^\lambda, S)$ ;
return  $b'$ ;

```

Figure 6.1: Security game **AONT-AON** for *AONT* scheme Π .

AONT-AON SECURITY. This security is dedicated to the most important *all-or-nothing (AON)* property. We first informally define our security goal. Suppose, $P := \Pi.\mathcal{T}(\textit{params}^{(\Pi)}, M)$, where $M \in \mathcal{M}^{(\Pi)}$. If λ (or more) bits of P are unknown to the adversary, then he/she cannot compute the message M with non-trivial success probability. The adversarial advantage is the probability of the event when the adversary is able to distinguish between the transformations of a known message and a random string without the knowledge of at least λ bits, and it is desired to be negligible.

According to the **AONT-AON** game, given access to $\Pi(\cdot)$, the adversary returns a message M and some state information S . Then the following operations are performed: first the pseudo-message $P_1 := \Pi.\mathcal{T}(\textit{params}^{(\Pi)}, M)$ is computed; after that a random string P_0 of length $|P_1|$ is generated; and finally P_0 and P_1 are parsed into λ -bit blocks $P_0[1] \| P_0[2] \| \cdots \| P_0[m] := P_0$ and $P_1[1] \| P_1[2] \| \cdots \| P_1[m] := P_1$. An oracle γ now operates the following way: it takes as input the index $j \in [m]$, and returns the block $P_b[j]$, where $b \in \{0, 1\}$. The adversary is allowed to make at most $(m - 1)$ adaptive queries to the oracle γ . Given access to γ , the adversary returns a bit b' .

The advantage of the AONT-AON adversary \mathcal{A} against Π is defined as follows:

$$\text{Adv}_{\Pi, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) \stackrel{\text{def}}{=} \left| \Pr[\text{AONT-AON}_\Pi^{\mathcal{A}}(1^\lambda, b=1) = 1] - \Pr[\text{AONT-AON}_\Pi^{\mathcal{A}}(1^\lambda, b=0) = 1] \right|.$$

The Π is AONT-AON secure, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\Pi, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda)$ is *negligible*.

6.5 New *AONT* Constructions

In this section, we present two new efficient constructions for *AONT* – denoted $\hat{\Pi}$ and $\tilde{\Pi}$ – which are based on a 2λ -bit *easy-to-invert* permutation π . We assume that the message-length in bits is a multiple of the security parameter λ .

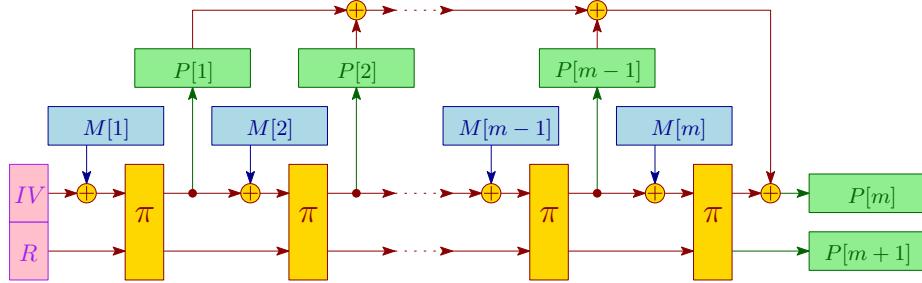
6.5.1 Construction $\hat{\Pi}$

This construction is motivated by the design of APE authenticated encryption (described in Section 2.2.5.2).

6.5.1.1 Description of $\hat{\Pi}$

The pictorial description and pseudocode for the pair of algorithms in $\hat{\Pi} = (\hat{\Pi}, \mathcal{T}, \hat{\Pi}, \mathcal{I})$ over the *setup* algorithm $\hat{\Pi}.\text{Setup}$ are given in Figures 6.2, 6.3, 6.4 and 6.5; all wires are λ -bit long. It is worth noting that the inverse transformation is executed in the *reverse* direction of transformation. Below we give the textual description.

- The *setup* algorithm $\hat{\Pi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\hat{\Pi})}$, message-space $\mathcal{M}^{(\hat{\Pi})} = \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and *pseudo-message-space* $\mathcal{P}^{(\hat{\Pi})} = \bigcup_{i \geq 2} \{0, 1\}^{i\lambda}$.
- The *transformation* algorithm $\hat{\Pi}.\mathcal{T}(\cdot)$ is randomized that takes as input parameter $\text{params}^{(\hat{\Pi})}$ and message M , and outputs pseudo-message P . The pictorial description and pseudocode are given in Figures 6.2 and 6.3. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \| M[2] \| \dots \| M[m] := M$; then it randomly chooses a λ -bit string s ; and after that it initializes the

Figure 6.2: Pictorial description of the *transformation* function $\hat{\Pi} \cdot \mathcal{T}$.

$\hat{\Pi} \cdot \mathcal{T}(params^{(\hat{\Pi})}, M)$

#Initialization.

$IV := 0^\lambda; m := |M|/\lambda; M[1]\|M[2]\|\cdots\|M[m] := M;$
 $r := IV; s \xleftarrow{\$} \{0, 1\}^\lambda; temp := 0^\lambda;$

#Processing message blocks.

```

for ( $j := 1, 2, \dots, m$ )
     $r\|s := \pi((r \oplus M[j])\|s);$ 
    If ( $j = m$ )
         $P[j] := r \oplus temp; P[j + 1] := s;$ 
    Else
         $P[j] := r; temp := temp \oplus P[j];$ 

```

#Computing final outputs.

return $P := P[1]\|P[2]\|\cdots\|P[m + 1];$

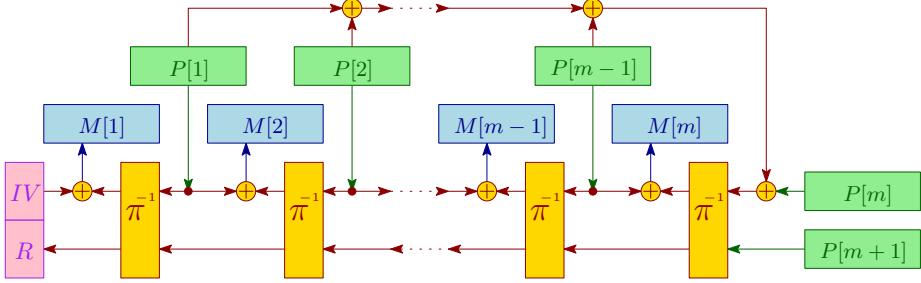
Figure 6.3: Algorithmic description of the *transformation* function $\hat{\Pi} \cdot \mathcal{T}$.

strings r and $temp$ with IV and 0^λ (here, $IV := 0^\lambda$). The transformation of M is composed of transformation of individual blocks in the same sequence.

Now, we give the details of transformation of message block $M[j]$, for all $j \neq m$. First $r\|s := \pi((r \oplus M[j])\|s)$ is computed; then $P[j] := r$ is assigned; and finally $temp := temp \oplus P[j]$ is updated.

For the transformation of last block $M[m]$, we perform the following operations: first $r\|s := \pi((r \oplus M[m])\|s)$ is computed; then $P[m] := r \oplus temp$ is computed; and finally $P[m + 1] := s$ is computed.

Now, the pseudo-message $P := P[1]\|P[2]\|\cdots\|P[m + 1]$.

Figure 6.4: Pictorial description of the *inverse transformation* function $\hat{\Pi} \cdot \mathcal{I}$.

$\hat{\Pi} \cdot \mathcal{I}(params^{(\hat{\Pi})}, P)$

#Validating pseudo-message.

If $(P \notin \mathcal{P}^{(\hat{\Pi})})$, then return \perp ;

#Initialization.

$IV := 0^\lambda; m := (|P|/\lambda) - 1; P[1] \| P[2] \| \cdots \| P[m+1] := P;$
 $r := P[1] \oplus P[2] \oplus \cdots \oplus P[m]; s := P[m+1];$

#Processing pseudo-message blocks.

for $(j := m, m - 1, \dots, 1)$
 $M[j] \| s := \pi^{-1}(r \| s);$
If $(j \neq 1)$
 $r := P[j - 1]; M[j] := r \oplus M[j];$
Else $M[j] := M[j] \oplus IV;$

#Computing final outputs.

return $M := M[1] \| M[2] \| \cdots \| M[m]$;

Figure 6.5: Algorithmic description of the *inverse transformation* function $\hat{\Pi} \cdot \mathcal{I}$.

- The *inverse transformation* algorithm $\hat{\Pi} \cdot \mathcal{I}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\Pi})}$ and pseudo-message P , and outputs message M or an *invalid* symbol \perp . See Figures 6.4 and 6.5 for the pictorial description and pseudocode. The algorithm returns \perp , if the pseudo-message $P \notin \mathcal{P}^{(\hat{\Pi})}$. Very briefly, the algorithm works as follows: first it parses P into λ -bit blocks $P[1] \| P[2] \| \cdots \| P[m+1] := P$; next it computes $r := P[1] \oplus P[2] \oplus \cdots \oplus P[m]$; and then initialises $s := P[m+1]$. The inverse transformation of P is composed of inverse transformation of individual blocks in direction *opposite* to that of the

transformation.

Now, we give the details of inverse transformation of pseudo-message block $P[j]$, for all $j \in [m]$: first $M[j]\|s := \pi^{-1}(r\|s)$ is computed; and after that if the value of $j = 1$, then $M[j] := M[j] \oplus IV$ is computed, (here, $IV := 0^\lambda$), otherwise, $r := P[j - 1]$ is assigned, and $M[j] := M[j] \oplus r$ is computed.

Now, the message $M := M[1]\|M[2]\|\cdots\|M[m]$.

6.5.1.2 Security of $\hat{\Pi}$

Theorem 36. *Let $\lambda \in \mathbb{N}$ be the security parameter. The AONT construction $\hat{\Pi}$ has been defined in Section 6.5.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Pi}, \mathcal{A}}^{AONT-AON}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m^2}{2^\lambda}.$$

Here, the AONT-AON game is defined in Figure 6.1, and the message-length in bits is $m\lambda$.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $AONT-AON_{\hat{\Pi}}^{\mathcal{A}}(1^\lambda, b = 1)$; our last game is $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$, which is identical to $AONT-AON_{\hat{\Pi}}^{\mathcal{A}}(1^\lambda, b = 0)$; and our intermediate games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_4(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games.

We now bound the adversarial advantage:

$$\begin{aligned}
Adv_{\hat{\Pi}, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{AONT-AON}_{\hat{\Pi}}^{\mathcal{A}}(1^\lambda, b=1) = 1] \right. \\
&\quad - \left. \Pr[\text{AONT-AON}_{\hat{\Pi}}^{\mathcal{A}}(1^\lambda, b=0) = 1] \right| \\
&= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right| \\
&= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right. \\
&\quad + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \\
&\quad + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \\
&\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \\
&\quad \left. + \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned}
Adv_{\hat{\Pi}, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \right| \\
&\quad + \left| \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

Therefore, $Adv_{\hat{\Pi}, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 + \Delta_5$, (6.1)

$$\begin{aligned}
\text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_4 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\
\Delta_5 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

In the following, we compute $\Delta_1, \Delta_2, \dots$ and Δ_5 (see Figure 6.6 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\hat{\Pi}^{(1)}$ instead of $\hat{\Pi}$. In $\hat{\Pi}^{(1)}$, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we

obtain:

$$\Delta_1 \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (6.2)$$

Game \mathbf{G}_2 : This game is identical to **Game \mathbf{G}_1** , except that it uses $\hat{\Pi}^{(2)}$ instead of $\hat{\Pi}^{(1)}$. In $\hat{\Pi}^{(2)}$ (as described in Figure 6.7), the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_2 \leq \frac{m(m-1)}{2^{2\lambda}}. \quad (6.3)$$

Game \mathbf{G}_3 : This game is identical to **Game \mathbf{G}_2** , except that it uses $\hat{\Pi}^{(3)}$ instead of $\hat{\Pi}^{(2)}$. In $\hat{\Pi}^{(3)}$ (as described in Figure 6.7), the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random function rf . From Figure 6.7, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the pseudo-message P (where $|P| = m\lambda$), we obtain:

$$\Delta_3 \leq \frac{m(m-1)}{2^\lambda}. \quad (6.4)$$

Game \mathbf{G}_4 : This game is identical to **Game \mathbf{G}_3** , except that it uses $\hat{\Pi}^{(4)}$ instead of $\hat{\Pi}^{(3)}$. In $\hat{\Pi}^{(4)}$ (as described in Figure 6.8), the bad_2 flag is set, when there is a collision of $P[m]$ with $P[j]$, for some $j \in [m-1]$. Since, the event that $bad_1 = 0$ makes sure that there is no collision among the lower λ -bit components of the 2λ -bit inputs to the random function rf , this implies that $P[1], P[2], \dots, P[m-1]$ are random and mutually independent (so is r). Therefore, $P[m]$ – which is $\left(\bigoplus_{i=1}^{m-1} P[i]\right) \oplus r$ – is also random and independent. Therefore, from Figure 6.8, using the *code-based game playing technique* [BR06], the probability that $P[m]$ collides with $P[j]$, for some $j \in [m-1]$, is (using the principle of inclusion-exclusion):

$$\Delta_4 \leq \frac{m}{2^\lambda}. \quad (6.5)$$

Since, $\hat{\Pi}^{(4)}. \mathcal{T}(\text{params}^{(\hat{\Pi}^{(4)})}, M)$ is uniformly distributed over $\{0, 1\}^{|M|+\lambda}$, the two games \mathbf{G}_4 and \mathbf{G}_L are indistinguishable, given all other lines of code are identical. Hence, we obtain:

$$\Delta_5 = 0. \quad (6.6)$$

Using (6.1) – (6.6), we get:

$$\begin{aligned}
 Adv_{\hat{\Pi}, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 + \Delta_5 \\
 &\leq \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^{2\lambda}} + \frac{m(m-1)}{2^\lambda} + \frac{m}{2^\lambda} + 0 \\
 &\leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m^2}{2^\lambda}.
 \end{aligned}$$

■

Game $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$
$(params^{(\hat{\Pi})}, \mathcal{M}^{(\hat{\Pi})}, \mathcal{P}^{(\hat{\Pi})}) := \hat{\Pi}.\text{Setup}(1^\lambda);$
$(M, S) := \mathcal{A}_1(1^\lambda, \hat{\Pi});$
$P[1] \ P[2] \ \cdots \ P[m] := \hat{\Pi}.\mathcal{T}(params^{(\hat{\Pi})}, M);$
$b' := \mathcal{A}_2^{\gamma_P}(1^\lambda, S);$
return b' ;
Game $\mathbf{G}_i(\mathcal{A}, 1^\lambda)$
$(params^{(\hat{\Pi}^{(i)})}, \mathcal{M}^{(\hat{\Pi}^{(i)})}, \mathcal{P}^{(\hat{\Pi}^{(i)})}) := \hat{\Pi}^{(i)}.\text{Setup}(1^\lambda);$
$(M, S) := \mathcal{A}_1(1^\lambda, \hat{\Pi}^{(i)});$
$P[1] \ P[2] \ \cdots \ P[m] := \hat{\Pi}^{(i)}.\mathcal{T}(params^{(\hat{\Pi}^{(i)})}, M);$
$b' := \mathcal{A}_2^{\gamma_P}(1^\lambda, S);$
return b' ;
Game $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$
$(params^{(\hat{\Pi})}, \mathcal{M}^{(\hat{\Pi})}, \mathcal{P}^{(\hat{\Pi})}) := \hat{\Pi}.\text{Setup}(1^\lambda);$
$(M, S) := \mathcal{A}_1(1^\lambda, \hat{\Pi});$
$P[1] \ P[2] \ \cdots \ P[m] \xleftarrow{\$} \{0, 1\}^{ M +\lambda}$
$b' := \mathcal{A}_2^{\gamma_P}(1^\lambda, S);$
return b' ;

Here, in **Game \mathbf{G}_i** , $i \in [4]$.

Figure 6.6: Games used in the proof of Theorem 36.

$\hat{\Pi}^{(2)}. \mathcal{T}(\text{params}^{(\hat{\Pi}^{(2)})}, M)$	$\hat{\Pi}^{(3)}. \mathcal{T}(\text{params}^{(\hat{\Pi}^{(3)})}, M)$
--	--

```

#Initialization.
IV := 0λ; m := |M|/λ; M[1]||M[2]||⋯||M[m] := M;
r := IV; s ← {0, 1}λ; temp := 0λ;
U := ∅; bad1 := 0;

#Processing message blocks.
for (j := 1, 2, ⋯, m)
  If (s ∈ U), then bad1 := 1;
  U := U ∪ {s}; r||s := rf((M[j] ⊕ r)||s);
  If (j = m), then P[j] := r ⊕ temp; P[j + 1] := s;
  Else P[j] := r; temp := temp ⊕ P[j];

#Computing final outputs.
return P := P[1]||P[2]||⋯||P[m + 1];

```

Figure 6.7: Algorithmic description of $\hat{\Pi}^{(2)}$ and $\hat{\Pi}^{(3)}$.

$\hat{\Pi}^{(3)}. \mathcal{T}(\text{params}^{(\hat{\Pi}^{(3)})}, M)$	$\hat{\Pi}^{(4)}. \mathcal{T}(\text{params}^{(\hat{\Pi}^{(4)})}, M)$
--	--

```

#Initialization.
IV := 0λ; m := |M|/λ; M[1]||M[2]||⋯||M[m] := M;
r := IV; s ← {0, 1}λ; temp := 0λ; U := ∅; V := ∅;
bad1 := 0; bad2 := 0;

#Processing message blocks.
for (j := 1, 2, ⋯, m)
  If (s ∈ U), then bad1 := 1;
  U := U ∪ {s}; r||s := rf((M[j] ⊕ r)||s);
  If (j = m), then P[j] := r ⊕ temp; P[j + 1] := s;
    If (P[j] ∈ V), then bad2 := 1;
  Else P[j] := r; temp := temp ⊕ P[j]; V := V ∪ {P[j]};

#Computing final outputs.
return P := P[1]||P[2]||⋯||P[m + 1];

```

Figure 6.8: Algorithmic description of $\hat{\Pi}^{(3)}$ and $\hat{\Pi}^{(4)}$.

6.5.2 Construction $\tilde{\Pi}$

This construction is motivated by the design of FP hash mode of operation (described in Section 2.2.3.2).

6.5.2.1 Description of $\tilde{\Pi}$

The pictorial description and pseudocode for the pair of algorithms in $\tilde{\Pi} = (\tilde{\Pi}, \mathcal{T}, \tilde{\Pi}, \mathcal{I})$ over the *setup* algorithm $\tilde{\Pi}.$ *Setup* are given in Figures 6.9, 6.10, 6.11 and 6.12; all wires are λ -bit long. It is worth noting that the inverse transformation is executed in the *reverse* direction of transformation. Below, we give the textual description.

- The *setup* algorithm $\tilde{\Pi}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\tilde{\Pi})}$, message-space $\mathcal{M}^{(\tilde{\Pi})} = \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and pseudo-message-space $\mathcal{P}^{(\tilde{\Pi})} = \bigcup_{i \geq 2} \{0, 1\}^{i\lambda}$.

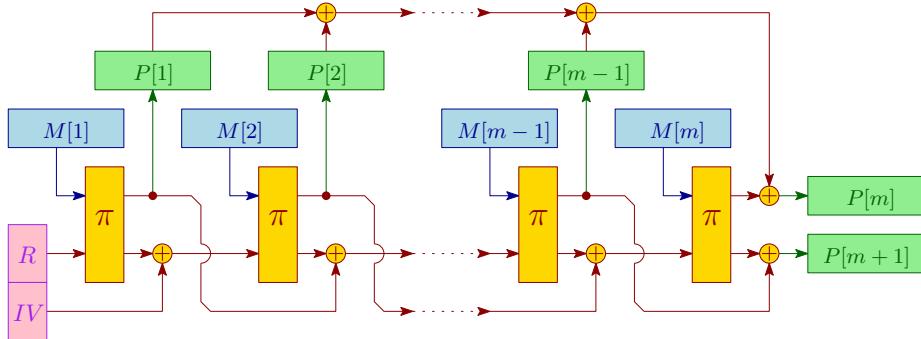


Figure 6.9: Pictorial description of the *transformation* function $\tilde{\Pi}.\mathcal{T}$.

- The *transformation* algorithm $\tilde{\Pi}.\mathcal{T}(\cdot)$, is randomized that takes as input parameter $\text{params}^{(\tilde{\Pi})}$ and message M , and outputs pseudo-message P . The pictorial description and pseudocode are given in Figures 6.9 and 6.10). Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then it randomly chooses a λ -bit string s ; and after that it initializes the strings t and temp with IV and 0^λ (here, $IV := 0^\lambda$). The transformation of M is composed of transformation of individual blocks in the same sequence.

Now, we give the details of transformation of message block $M[j]$, for all $j \neq m$. First $r \parallel s := \pi(M[j] \parallel s)$ is computed; then $P[j] := r$ is

```

 $\tilde{\Pi} \cdot \mathcal{T}(params^{(\tilde{\Pi})}, M)$ 
#Initialization.
 $IV := 0^\lambda; m := |M|/\lambda; M[1]\|M[2]\|\cdots\|M[m] := M;$ 
 $s \xleftarrow{\$} \{0, 1\}^\lambda; t := IV; temp := 0^\lambda;$ 

#Processing message blocks.
for ( $j := 1, 2, \dots, m$ )
   $r\|s := \pi(M[j]\|s);$ 
  If ( $j = m$ )
     $P[j] := r \oplus temp; P[j + 1] := s \oplus t;$ 
  Else
     $P[j] := r; temp := temp \oplus P[j]; s := s \oplus t; t := r;$ 

#Computing final output.
return  $P := P[1]\|P[2]\|\cdots\|P[m + 1];$ 

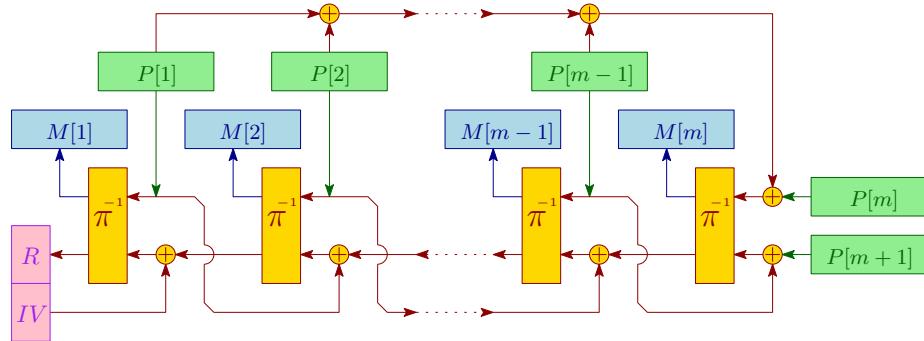
```

Figure 6.10: Algorithmic description of the transformation function $\tilde{\Pi} \cdot \mathcal{T}$.

assigned; next $temp := temp \oplus P[j]$ is computed; after that $s := s \oplus t$ is computed; and finally $t := r$ is assigned.

For the transformation of last block $M[m]$, we perform the following operations: first $r\|s := \pi(M[m]\|s)$ is computed; then $P[m] := r \oplus temp$ is computed; and finally $P[m + 1] := s \oplus t$ is computed.

Now, the pseudo-message $P := P[1]\|P[2]\|\cdots\|P[m + 1]$.

Figure 6.11: Pictorial description of the inverse transformation function $\tilde{\Pi} \cdot \mathcal{I}$.

- The inverse transformation algorithm $\tilde{\Pi} \cdot \mathcal{I}(\cdot)$, is deterministic that takes as input parameter $params^{(\tilde{\Pi})}$ and pseudo-message P , and out-

```

 $\tilde{\Pi} \cdot \mathcal{I}(params^{(\tilde{\Pi})}, P)$ 

#Validating pseudo-message.
If ( $P \notin \mathcal{P}^{(\tilde{\Pi})}$ ), then return  $\perp$ ;

#Initialization.
 $IV := 0^\lambda$ ;  $m := (|P|/\lambda) - 1$ ;  $P[1] \| P[2] \| \cdots \| P[m+1] := P$ ;
 $s := P[m+1]$ ;  $t := P[1] \oplus P[2] \oplus \cdots \oplus P[m]$ ;

#Processing pseudo-message blocks.
for ( $j := m, m-1, \dots, 1$ )
   $r := t$ ;
  If ( $j = 1$ ), then  $t := IV$ ; Else  $t := P[j-1]$ ;
   $s := s \oplus t$ ;  $M[j] \| s := \pi^{-1}(r \| s)$ ;

#Computing final outputs.
return  $M := M[1] \| M[2] \| \cdots \| M[m]$ ;

```

Figure 6.12: Algorithmic description of the *inverse transformation* function $\tilde{\Pi} \cdot \mathcal{I}$.

puts message M or an *invalid* symbol \perp . See Figures 6.11 and 6.12 for the pictorial description and pseudocode. The algorithm returns \perp , if the pseudo-message $P \notin \mathcal{P}^{(\tilde{\Pi})}$. Very briefly, the algorithm works as follows: first it parses P into λ -bit blocks $P[1] \| P[2] \| \cdots \| P[m+1] := P$; next it initializes $s := P[m+1]$; and finally it computes $t := P[1] \oplus P[2] \oplus \cdots \oplus P[m]$. The inverse transformation of P is composed of inverse transformation of individual blocks in direction *opposite* to that of the *transformation*.

Now, we give the details of inverse transformation of pseudo-message block $P[j]$, for all $j \in [m]$: first $r := t$ is assigned; after that if the value of $j = 1$, then $t := IV$ is assigned (here, $IV := 0^\lambda$), otherwise $t := P[j-1]$ is assigned; next $s := s \oplus t$ is computed; and finally $M[j] \| s := \pi^{-1}(r \| s)$ is computed.

Now, the message $M := M[1] \| M[2] \| \cdots \| M[m]$.

6.5.2.2 Security of $\tilde{\Pi}$

Theorem 37. *Let $\lambda \in \mathbb{N}$ be the security parameter. The AONT construction $\tilde{\Pi}$ has been defined in Section 6.5.2. If π is assumed to be a 2λ -bit ideal*

permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\Pi}, \mathcal{A}}^{\text{AONT-AON}}(1^\lambda) \leq \frac{m(m-1)}{2^{2\lambda-1}} + \frac{m^2}{2^\lambda}.$$

Here, the *AONT-AON* game is defined in Figure 6.1, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 36. ■

6.6 Comparison of *AONT* schemes

In this section, we compare all the existing *AONT* schemes with the new ones. Here, we assume that the message-length $|M|$ and the security parameter λ are identical for all the schemes.

Also, we consider that before an algorithm starts, entire input to the algorithm is already written onto a read-only *input tape*, and during the execution of algorithm, as soon as the output blocks are generated, they are immediately written on the write-only-once *output tape*. We do not compare the size of *input* or *output tapes*. Instead, we only compare the amount of memory required by the *processing unit* to store the intermediate variables. Also, in cases when there are two possibilities of the execution of the algorithm – parallel execution with high memory requirement, and serial execution with low memory requirement – we prefer to compare the second mode of execution over the first one.

In our comparison tables: $c_{\mathcal{E}_\lambda}$ denotes the computing cost of invocation of encryption of blockcipher for λ -bit input; $c_{\mathcal{G}_{\lambda, |M|}}$ denotes the cost of generator function which takes a λ -bit input and outputs a $|M|$ -bit string in [Boy99, ZHI04]; $c_{\mathcal{H}_{\lambda, |M|}}$ denotes the cost of hash operation on a $|M|$ -bit message, to produce a λ -bit hash; c_{+_λ} denotes the cost of adding two λ -bit inputs; $c_{\mathcal{E}'_{|M|}}$ (or $c_{\mathcal{D}'_{|M|}}$) denotes the cost of encryption (or decryption) of full-domain cipher on a message (or ciphertext) of length $|M|$; c_{\bullet_λ} denotes the cost of group operation in [MAB07] on two λ -bit inputs; and $c_{\pi_{2\lambda}}$ (or $c_{\pi_{2\lambda}^{-1}}$) denotes the cost of invocation of 2λ -bit permutation (or 2λ -bit inverse permutation).

Since the *full-domain* cipher is used to encrypt fixed-length messages, that is, it cannot be used to encrypt messages of arbitrary length, it is never preferred to the blockcipher-based or permutation-based encryptions, which are used for encrypting messages of arbitrary lengths.

The cost of invocation of encryption (or decryption) function of a blockcipher on a λ -bit input is usually greater than the cost of invocation of a

	Construction →	Rivest (1997) [Riv97]	Boyko (1999) [Boy99]*	Desai (2000) [Des00]	Stinson (2001) [Sti01]	ZHI (2004) [ZHI04]*	MAB (2007) [MM06]	Our Constructions
Property ↓	Underlying Primitive	Block Cipher	OAEP	Block Cipher	Linear Transf.	Full-Dom. Cipher	Hash Function	$\hat{\Pi}$
Memory Req. (Processing Unit):							Quasi Groups	$\tilde{\Pi}$
• \mathcal{T}							Perm.	Perm.
Pseudo-message Expansion	λ	3λ	$ M + \lambda$	3λ	λ	λ	$ M + \lambda$	$\lambda^3 + 3\lambda^2$
Running Time:							$ M + \lambda$	$\lambda^3 + 3\lambda^2$
• \mathcal{T}		$2\frac{ M }{\lambda}\mathbf{C}_{\mathcal{E}_\lambda}$	$\mathbf{C}_{\mathcal{G}_{\lambda, M }}$ $+ \mathbf{C}_{\mathcal{H}_{\lambda, M }}$	$\frac{ M }{\lambda}\mathbf{C}_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}\mathbf{C}_{+\lambda}$	$\mathbf{C}_{\mathcal{E}'_{ M }}$ $+ \mathbf{C}_{\mathcal{H}_{\lambda, M }}$	$4\frac{ M }{\lambda}\lambda^2\mathbf{C}_{\bullet_\lambda}$	$\frac{ M }{\lambda}\mathbf{C}_{\pi_{2\lambda}}$
• \mathcal{I}		$2\frac{ M }{\lambda}\mathbf{C}_{\mathcal{E}_\lambda}$	$\mathbf{C}_{\mathcal{G}_{\lambda, M }}$ $+ \mathbf{C}_{\mathcal{H}_{\lambda, M }}$	$\frac{ M }{\lambda}\mathbf{C}_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}\mathbf{C}_{+\lambda}$	$\mathbf{C}_{\mathcal{D}'_{ M }}$ $+ \mathbf{C}_{\mathcal{H}_{\lambda, M }}$	$4\frac{ M }{\lambda}\lambda^2\mathbf{C}_{\bullet_\lambda}$	$\frac{ M }{\lambda}\mathbf{C}_{\pi_{2\lambda}^{-1}}$
No. of Passes								
• ENCRYPTION	1	1	1	2	2	1	1	1
• DECRYPTION	2	2	2	2	1	2	2	2

*The proposed constructions are for fixed-length messages.

Table 6.1: Comparison table for various *AONT* schemes.

2λ -bit permutation (or an inverse permutation), that is, $c_{\mathcal{E}_\lambda} > c_{\pi_{2\lambda}}$. This is because, a blockcipher can be viewed as a family of permutations (and their inverses), indexed by the key, and it also includes a key schedule algorithm.

The cost of addition operation $c_{+\lambda}$, XOR operation c_{\oplus_λ} , etc. on two λ -bit inputs, are usually much smaller than the cost of group operation on two λ -bit inputs, c_{\bullet_λ} , as used in [MAB07].

In Table 6.1, we compare all the existing and new *AONT* schemes. Since, all the *AONT* constructions are *AONT-AON* secure, therefore, we only compare their efficiencies w.r.t. time and memory.

We observe that our proposed schemes $\hat{\Pi}$ and $\tilde{\Pi}$ outperform all the existing constructions.

6.7 Conclusion and Future Work

In this chapter, we refined the definition of an existing cryptographic primitive *AONT*, and gave two efficient design constructions for *AONT*, based on a fixed permutation. We discussed why a fixed-permutation-based approach could be more practical than the blockcipher-based ones. All our constructions make use of a property, named, *reverse decryption*, first proposed in the *APE* authenticated encryption (DIAC 2012, FSE 2014) and the *FP* hash mode of operation (Indocrypt 2012). We have also provided the efficiency comparison for all the constructions, both the existing and the new ones.

One way to extend the present work could be to add more functionalities to the *AONT* such as *update*.

7

CHAPTER

One-Time AONE: A variant of AONE

7.1 Introduction

ORIGIN. The cryptographic primitive *all-or-nothing encryption (AONE)* was, for the first time, introduced by Rivest in 1997, to enhance the security of conventional *blockcipher-based symmetric encryption (SE)* against the *key recovery* attack. In this attack, the adversary takes the advantage of the vulnerability, that springs up due to the design of blockcipher modes of operation, where, on the decryption side, the recovery of a plaintext block requires the knowledge of just *one* (or at most two) ciphertext block(s), rather than *all* of them. This drastically reduces the security of the symmetric encryption, and makes it vulnerable to several types of attacks. The *all-or-nothing (AON)*, the primary property of all *AONE* schemes, where the recovery of each plaintext bit requires the knowledge of the entire ciphertext, and is responsible for enhancing the security of various real-life applications, as described below.

APPLICATIONS. Although invented more than two decades ago, *AONE* schemes find their applications in strengthening the security of large databases used in *RFID* applications and of *searchable encryption* [MWT⁺13, PS10].

MOTIVATION. The cryptographic primitive *AONE* has diverse applications, as discussed above. We observe that the primitive *AONE* has, so far, been always designed by combining an *AONT* and an *SE*. This design paradigm disallows an *AONE* to be based on various other components, that may lead

to more efficient constructions in certain applications (if not all). Against this backdrop, our next challenge is:

Can we define (as well as design) an *AONE* without always relying on *AONT* and *SE*, and thereby, liberate this primitive from this rigid design paradigm?

We also observe that, so far, all the existing *AONE* schemes are based on a blockcipher. Blockciphers often come under *key schedule* weaknesses [ABB⁺14, DSST17, KSW96, Knu95, MP12, Nat12]. In order to mitigate these issues, fixed permutations have begun to replace the blockciphers in several primitives. Therefore, we ask the following question:

Is it possible to design a secure and efficient *AONE* scheme based on a permutation, instead of a blockcipher?

Although the aforementioned challenges are our primary focus, along the way, we have discovered various other primitives as well, than just *AONE*, and gave concrete constructions for them.

7.2 Contribution of this Chapter

In our search for efficient permutation-based *AONEs*, we have broken the conventional design paradigm of constructing it by combining *AONT* with *SE*. The security of our constructions relies neither on an *AONT* nor on an *SE*; it only depends on a “small” and fixed permutation. Our design paradigm is more flexible, and it ultimately leads to a new definition of *AONE*, that is free from any dependencies on any cryptographic primitives including *AONT* and *SE*.

Consequently, we first define the cryptographic primitive *AONE* in a framework that is more flexible than the previous definitions, in the sense that it now no longer depends on either an *AONT* or an *SE*. To justify the practical usefulness of this definition, we proposed two concrete constructions of permutation-based *AONEs*, that are based on neither an *AONT* nor an *SE*. These constructions are highly efficient.

Our final contribution is conceptualization and design of a novel cipher that requires *no encryption key*, nevertheless, generates the decryption key that is message-dependent. We call this primitive *one-time AONE* because the decryption key is valid only for a particular message. This primitive is very unique as it supports various conflicting properties: like a *public key cryptosystem (PKC)*, it is inherently an asymmetric cipher, since the

encryption and decryption keys are always different; on the other hand, unlike *PKC*, its decryption key is dependent on the message; although its security properties are *identical* to that of *AONE*, very oddly, unlike *AONE*, it does not require any (encryption) *key generation* algorithm, making this primitive a suitable candidate to be used in applications, where the leakage of secret key is an issue. We concretely define two highly-efficient constructions of *one-time AONE*.

The high efficiency of our permutation-based constructions is attributed to the very unique *reverse decryption* property, found in the *APE* authenticated encryption and the *FP* hash mode of operation, and from which we derive our inspiration; a permutation-based construction is usually advantageous because of the reason that it is not bundled with a key schedule algorithm of a blockcipher. We compare the performances of these proposed constructions against the existing ones.

7.3 Related work

The term *all-or-nothing encryption (AONE)* was coined by Rivest in 1997, with primary focus on improving the resistance against the *key recovery* attack on a conventional blockcipher mode (e.g. *CBC*, *OFB*, *CFB* and *CTR*) [Riv97]. Rivest defined *AONE* as the execution of the *transformation* function of *AONT* scheme followed by the encryption of the pseudo-message using an ordinary encryption mode.

In 2000, Desai introduced a new security notion of *non-separability of keys*, and defined *AONE* as composition of *AONT* and an ordinary encryption mode [Des00]. In 2004, Zhang, Hanaoka and Imai described *AONE* as the execution of transformation function on a message, followed by encryption of each block using a blockcipher.

In 2006, McEvoy and Murphy defined *AONE* as the mode when the pseudo-message generated by *AONT* scheme is encrypted using a symmetric blockcipher, and proposed efficient *AONE* construction based on *counter mode of operation* [MM06]. In 2010, Park and Song proposed *secret sharing* algorithm for the security of the huge databases – used in several *RFID* applications – based on *AONE* [PS10]. In 2013, Ma *et al.* proposed a scheme to enhance the security of *searchable encryption*, using *AONE* [MWT⁺13]. In 2018, Kapusta and Memmi have proposed a scheme that combines partial encryption of the segments of blocks with the *AONT* [KM18a].

7.4 All-Or-Nothing Encryption (*AONE*)

From a high level, *AONE* has three algorithms: a *key generation* algorithm, that generates a key for the encryption and decryption; an *encryption* algorithm, that takes the key and message as input, and outputs a ciphertext, which is computed in a way that enforces the *all-or-nothing* property; and a *decryption* algorithm, which decrypts the ciphertext into a message in such a way that if a few bits in the ciphertext are missing or incorrect, then the original message will *not* be recovered (with a high probability).

We now elaborately discuss the syntax, correctness and security definition of the *AONE*.

7.4.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. An *AONE* scheme $\chi = (\chi.\mathcal{GEN}, \chi.\mathcal{E}, \chi.\mathcal{D})$ is a 3-tuple of algorithms over the *setup* algorithm $\chi.\text{Setup}$, satisfying the following conditions.

1. The PPT *setup* algorithm $\chi.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\chi)}$, key-space $\mathcal{K}^{(\chi)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\chi)} \subseteq \{0, 1\}^*$ and ciphertext-space $\mathcal{C}^{(\chi)} \subseteq \{0, 1\}^*$.
2. The PPT *key generation* algorithm $\chi.\mathcal{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\chi)}$, and returns a key $K := \chi.\mathcal{GEN}(\text{params}^{(\chi)})$, where $K \in \mathcal{K}^{(\chi)}$.
3. The PPT *encryption* algorithm $\chi.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\chi)}$, key $K \in \mathcal{K}^{(\chi)}$ and message $M \in \mathcal{M}^{(\chi)}$, and returns ciphertext $C := \chi.\mathcal{E}(\text{params}^{(\chi)}, K, M)$, where $C \in \mathcal{C}^{(\chi)}$.
4. The *decryption* algorithm $\chi.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\chi)}$, key $K \in \mathcal{K}^{(\chi)}$ and ciphertext $C' \in \{0, 1\}^*$, and returns message $M := \chi.\mathcal{D}(\text{params}^{(\chi)}, K, C')$, where $M \in \mathcal{M}^{(\chi)} \cup \{\perp\}$.

Note: We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\chi)}$.

7.4.2 Correctness

This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the *correctness* of an *AONE* can be defined as follows.

Suppose $K := \chi \cdot \mathcal{GEN}(\text{params}^{(x)})$. Then, the correctness of χ requires that for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(x)}$, we have:

$$\chi \cdot \mathcal{D}(\text{params}^{(x)}, K, \chi \cdot \mathcal{E}(\text{params}^{(x)}, K, M)) = M.$$

7.4.3 Security Definitions

In Figures 7.1, 7.2 and 7.3, we define the security games AONE-KR, AONE-IND and AONE-AON, respectively, for the *AONE* scheme $\chi = (\chi \cdot \mathcal{GEN}, \chi \cdot \mathcal{E}, \chi \cdot \mathcal{D})$. As usual, the games are written using the challenger-adversary framework.

Game AONE-KR$_{\chi}^{\mathcal{A}}(1^{\lambda})$ $(\text{params}^{(x)}, \mathcal{K}^{(x)}, \mathcal{M}^{(x)}, \mathcal{C}^{(x)}) := \chi \cdot \text{Setup}(1^{\lambda});$ $K := \chi \cdot \mathcal{GEN}(\text{params}^{(x)});$ $M := \mathcal{A}_1^{\chi \cdot \mathcal{E}(\text{params}^{(x)}, K, \cdot)}(1^{\lambda});$ $C := \chi \cdot \mathcal{E}(\text{params}^{(x)}, K, M);$ $K' := \mathcal{A}_2(1^{\lambda}, C);$ If $K' = K$, then return 1; Else return 0;
--

Figure 7.1: Security game AONE-KR for *AONE* scheme χ .

AONE-KR SECURITY. This captures the intuitive notion of *key recovery* attack, as defined in Figure 7.1. The adversarial advantage is the probability of the event when the adversary is able to deduce the encryption key, and it is desired to be negligible.

According to the AONE-KR game, the adversary first returns a message M . Then the following operations are performed: first a key $K := \chi \cdot \mathcal{GEN}(\text{params}^{(x)})$ is generated; and then the message M is encrypted using K to obtain the ciphertext $C := \chi \cdot \mathcal{E}(\text{params}^{(x)}, K, M)$. Given access to C , the adversary returns a key K' . The game returns 1, if $K = K'$.

The advantage of the AONE-KR adversary \mathcal{A} against χ is defined as follows:

$$\text{Adv}_{\chi, \mathcal{A}}^{\text{AONE-KR}}(1^{\lambda}) \stackrel{\text{def}}{=} \Pr[\text{AONE-KR}_{\chi}^{\mathcal{A}}(1^{\lambda}) = 1].$$

The χ is AONE-KR secure, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\chi, \mathcal{A}}^{\text{AONE-KR}}(1^{\lambda})$ is *negligible*.

```

Game AONE-IND $_{\chi}^{\mathcal{A}}(1^{\lambda}, b)$ 


---


( $params^{(\chi)}$ ,  $\mathcal{K}^{(\chi)}$ ,  $\mathcal{M}^{(\chi)}$ ,  $\mathcal{C}^{(\chi)}$ ) :=  $\chi$ .Setup( $1^{\lambda}$ );
 $K := \chi$ . $\mathcal{GEN}(params^{(\chi)})$ ;
 $M := \mathcal{A}_1^{\chi, \mathcal{E}(params^{(\chi)}, K, \cdot)}(1^{\lambda})$ ;
 $C_1 := \chi$ . $\mathcal{E}(params^{(\chi)}, K, M)$ ;
 $C_0 \xleftarrow{\$} \{0, 1\}^{|C_1|}$ ;
 $b' := \mathcal{A}_2(1^{\lambda}, C_b)$ ;
return  $b'$ ;

```

Figure 7.2: Security game AONE-IND for AONE scheme χ .

AONE-IND SECURITY. This captures the notion of *indistinguishability privacy* attack, as defined in Figure 7.2. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a message from a random string, and it is desired to be negligible.

According to the AONE-IND game, the adversary first returns a message M . Then the following operations are performed: first a key $K := \chi$. $\mathcal{GEN}(params^{(\chi)})$ is generated; then the message M is encrypted using K to obtain the ciphertext $C_1 := \chi$. $\mathcal{E}(params^{(\chi)}, K, M)$; and finally a random string C_0 of length $|C_1|$ is computed. Given access to C_b , where $b \in \{0, 1\}$, the adversary returns a bit b' .

The advantage of the AONE-IND adversary \mathcal{A} against χ is defined as follows:

$$\text{Adv}_{\chi, \mathcal{A}}^{\text{AONE-IND}}(1^{\lambda}) \stackrel{\text{def}}{=} \left| \Pr[\text{AONE-IND}_{\chi}^{\mathcal{A}}(1^{\lambda}, b = 1) = 1] - \Pr[\text{AONE-IND}_{\chi}^{\mathcal{A}}(1^{\lambda}, b = 0) = 1] \right|.$$

The χ is AONE-IND secure, if, for all PPT adversaries \mathcal{A} , $\text{Adv}_{\chi, \mathcal{A}}^{\text{AONE-IND}}(1^{\lambda})$ is *negligible*.

AONE-AON SECURITY. This security is dedicated to the *all-or-nothing (AON)* property, as defined in Figure 7.3. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a known message from a random string without the knowledge of at least λ bits, and it is desired to be negligible.

According to the AONE-AON game, given access to $\chi(\cdot)$, the adversary returns a message M and some state information S . Then the following operations are performed: first a key $K := \chi$. $\mathcal{GEN}(params^{(\chi)})$ is generated; then the ciphertext $C_1 := \chi$. $\mathcal{E}(params^{(\chi)}, K, M)$ is computed;

```

Game AONE-AON $_{\chi}^{\mathcal{A}}(1^{\lambda}, b)$ 


---


( $params^{(\chi)}, \mathcal{K}^{(\chi)}, \mathcal{M}^{(\chi)}, \mathcal{C}^{(\chi)}$ ) :=  $\chi$ .Setup( $1^{\lambda}$ );
 $K := \chi$ .G $\mathcal{E}$ N( $params^{(\chi)}$ );
 $(M, S) := \mathcal{A}_1(1^{\lambda}, \chi)$ ;
 $C_1 := \chi$ .E( $params^{(\chi)}, K, M$ );     $C_0 \xleftarrow{\$} \{0, 1\}^{|C_1|}$ ;
 $C_0[1] \| C_0[2] \| \cdots \| C_0[m] := C_0$ ;
 $C_1[1] \| C_1[2] \| \cdots \| C_1[m] := C_1$ ;
 $b' := \mathcal{A}_2^{\beta}(1^{\lambda}, S, K)$ ;
return  $b'$ ;

```

Figure 7.3: Security game AONE-AON for *AONE* scheme χ .

after that a random string C_0 of length $|C_1|$ is generated; and finally C_0 and C_1 are parsed into λ -bit blocks $C_0[1] \| C_0[2] \| \cdots \| C_0[m] := C_0$ and $C_1[1] \| C_1[2] \| \cdots \| C_1[m] := C_1$. An oracle β now operates the following way: it takes as input the index $j \in [m]$, and returns the block $C_b[j]$, where $b \in \{0, 1\}$. The adversary is allowed to make at most $(m - 1)$ adaptive queries to the oracle β . Given access to β , the adversary returns a bit b' .

The advantage of the AONE-AON adversary \mathcal{A} against χ is defined as follows:

$$\begin{aligned} Adv_{\chi, \mathcal{A}}^{\text{AONE-AON}}(1^{\lambda}) &\stackrel{def}{=} \left| \Pr[\text{AONE-AON}_{\chi}^{\mathcal{A}}(1^{\lambda}, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{AONE-AON}_{\chi}^{\mathcal{A}}(1^{\lambda}, b = 0) = 1] \right|. \end{aligned}$$

The χ is AONE-AON secure, if, for all PPT adversaries \mathcal{A} , $Adv_{\chi, \mathcal{A}}^{\text{AONE-AON}}(1^{\lambda})$ is *negligible*.

Note: The *all-or-nothing* security of *AONE* is different from that of *AONT* in the following way: in *AONT*, the adversary is given all *pseudo-message blocks* except (at least) one, while, in *AONE*, the adversary is supplied with all *ciphertext blocks* except (at least) one, along with the decryption key.

7.5 New *AONE* Constructions

In this section, we design two concrete *AONE* schemes, denoted, $\hat{\chi}$ and $\tilde{\chi}$. The main component of these constructions is a 2λ -bit *easy-to-invert* permutation π , instead of a blockcipher. We assume that the message-length in bits is a multiple of the security parameter λ .

These schemes are suitable for most of the general applications, and helpful, especially when a key is computed once, and reused for encrypting several messages.

7.5.1 Construction $\hat{\chi}$

This construction is motivated by the design of APE authenticated encryption (described in Section 2.2.5.2).

7.5.1.1 Description of $\hat{\chi}$

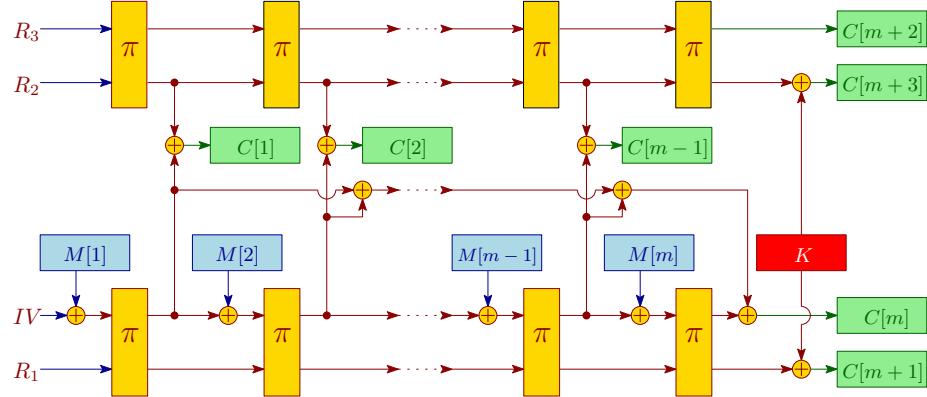
The pictorial description and pseudocode for the 3-tuple of algorithms in $\hat{\chi} = (\hat{\chi}. \mathcal{GEN}, \hat{\chi}. \mathcal{E}, \hat{\chi}. \mathcal{D})$ over the *setup* algorithm $\hat{\chi}. \text{Setup}$ are given in Figures 7.4, 7.5, 7.6, 7.7 and 7.8; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\hat{\chi}. \text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\hat{\chi})}$, *key-space* $\mathcal{K}^{(\hat{\chi})} := \{0, 1\}^\lambda$, *message-space* $\mathcal{M}^{(\hat{\chi})} := \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and *ciphertext-space* $\mathcal{C}^{(\hat{\chi})} := \bigcup_{i \geq 4} \{0, 1\}^{i\lambda}$.

```
 $\hat{\chi}. \mathcal{GEN}(\text{params}^{(\hat{\chi})})$ 
#Choosing a random  $\lambda$ -bit key.
 $K \xleftarrow{\$} \{0, 1\}^\lambda;$ 
return  $K$ ;
```

Figure 7.4: Algorithmic description of the *key generation* function $\hat{\chi}. \mathcal{GEN}$.

- The PPT *key generation* algorithm $\hat{\chi}. \mathcal{GEN}(\cdot)$ takes as input parameter $\text{params}^{(\hat{\chi})}$, and outputs key K randomly chosen from the *key-space* $\mathcal{K}^{(\hat{\chi})}$. The pseudocode is given in Figure 7.4
- The *encryption* algorithm $\hat{\chi}. \mathcal{E}(\cdot)$ is randomized that takes as input parameter $\text{params}^{(\hat{\chi})}$, key K and message M , and outputs ciphertext C . The pictorial description and pseudocode are given in Figures 7.5 and 7.6. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then it randomly chooses three λ -bit strings s, u and v ; and after that it initializes the

Figure 7.5: Pictorial description of the *encryption* function $\hat{\chi} \cdot \mathcal{E}$.

$\hat{\chi} \cdot \mathcal{E}(\text{params}^{(\hat{\chi})}, K, M)$

#Initialization.

$$\begin{aligned} IV &:= 0^\lambda; & m &:= |M|/\lambda; & M[1]\|M[2]\|\cdots\|M[m] &:= M; \\ r &:= IV; & s &\xleftarrow{\$} \{0, 1\}^\lambda; & temp &:= 0^\lambda; & u &\xleftarrow{\$} \{0, 1\}^\lambda; \\ v &\xleftarrow{\$} \{0, 1\}^\lambda; \end{aligned}$$

#Processing blocks.

```

for ( $j := 1, 2, \dots, m$ )
     $r\|s := \pi((r \oplus M[j])\|s);$      $u\|v := \pi(u\|v);$ 
    If ( $j = m$ )
         $C[m] := r \oplus temp;$      $C[m+1] := s \oplus K;$      $C[m+2] := u;$ 
         $C[m+3] := v \oplus K;$ 
    Else
         $temp := temp \oplus r;$      $C[j] := v \oplus r;$ 

```

#Computing final outputs.

```
return  $C := C[1]\|C[2]\|\cdots\|C[m+3];$ 
```

Figure 7.6: Algorithmic description of the *encryption* function $\hat{\chi} \cdot \mathcal{E}$.

strings r and $temp$ with IV and 0^λ (here, $IV := 0^\lambda$). The encryption of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$: first $r\|s := \pi((r \oplus M[j])\|s)$ is computed; then $u\|v := \pi(u\|v)$ is computed; after that $temp := temp \oplus r$ is updated; and finally

$C[j] := v \oplus r$ is computed.

For the encryption of last block $M[m]$, we perform the following operations: first $r \| s := \pi((r \oplus M[m]) \| s)$ is computed; then $u \| v := \pi(u \| v)$ is computed; after that $C[m] := r \oplus temp$ is computed; next $C[m + 1] := s \oplus K$ is computed; then $C[m + 2] := u$ is assigned; and finally $C[m + 3] := v \oplus K$ is computed.

Now, the ciphertext $C := C[1] \| C[2] \| \dots \| C[m + 3]$.

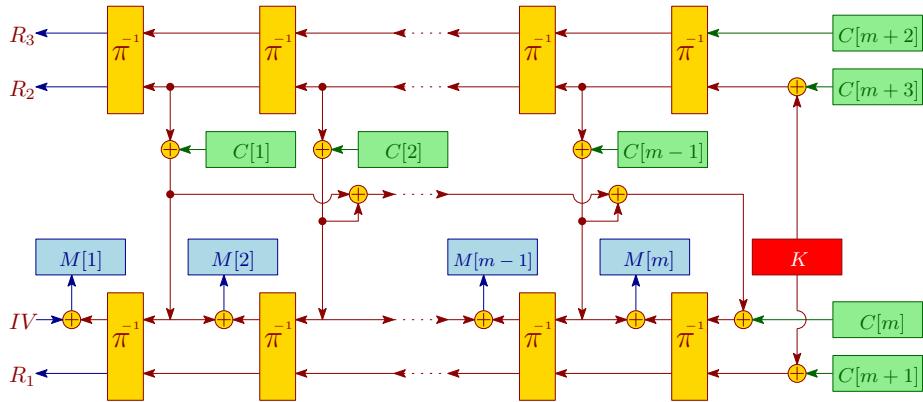


Figure 7.7: Pictorial description of the *decryption* function $\hat{\chi} \cdot \mathcal{D}$.

- The *decryption* algorithm $\hat{\chi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\chi})}$, key K and ciphertext C , and outputs message M or an *invalid* symbol \perp . See Figures 7.7 and 7.8 for the pictorial description and pseudocode. The algorithm returns \perp , if the ciphertext $C \notin \mathcal{C}^{(\hat{\chi})}$. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \| C[2] \| \dots \| C[m + 3] := C$; then it computes $s := C[m + 1] \oplus K$; next it initializes the strings $temp := 0^\lambda$ and $u := C[m + 2]$; and after that it computes $v := C[m + 3] \oplus K$. Now, the value of $temp$ is computed iteratively, as j runs through $m, m - 1, \dots, 2$, in the following way: first it computes $u \| v := \pi^{-1}(u \| v)$; and then it updates $temp := temp \oplus v \oplus C[j - 1]$. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

For the decryption of last block of ciphertext, first of all, the following operations are performed: first it computes $r := C[m] \oplus temp$; next it assigns $u := C[m + 2]$; and after that it computes $v := C[m + 3] \oplus K$.

Now, we give the details of decryption of ciphertext block $C[j]$, for all $j \in [m]$: first it computes $M[j] \| s := \pi^{-1}(r \| s)$; next it computes

```

 $\hat{\chi} \cdot \mathcal{D}(params^{(\hat{\chi})}, K, C)$ 

#Validating the length.
If ( $C \notin \mathcal{C}^{(\hat{\chi})}$ ), then return  $\perp$ ;

#Initialization.
 $IV := 0^\lambda; m := |C|/\lambda - 3; C[1]\|C[2]\|\cdots\|C[m+3] := C;$ 
 $s := C[m+1] \oplus K; temp := 0^\lambda; u := C[m+2];$ 
 $v := C[m+3] \oplus K;$ 

#Computing XOR value.
for ( $j := m, m-1, \dots, 2$ )
 $u\|v := \pi^{-1}(u\|v); temp := temp \oplus v \oplus C[j-1];$ 

#Processing blocks.
 $r := C[m] \oplus temp; u := C[m+2]; v := C[m+3] \oplus K;$ 
for ( $j := m, m-1, \dots, 1$ )
 $M[j]\|s := \pi^{-1}(r\|s); u\|v := \pi^{-1}(u\|v);$ 
If ( $j = 1$ ), then  $M[j] := M[j] \oplus IV;$ 
Else  $r := C[j-1] \oplus v; M[j] := M[j] \oplus r;$ 

#Computing final outputs.
return  $M := M[1]\|M[2]\|\cdots\|M[m];$ 

```

Figure 7.8: Algorithmic description of the *decryption* function $\hat{\chi} \cdot \mathcal{D}$.

$u\|v := \pi^{-1}(u\|v)$; and finally it checks if the value of $j = 1$, then it computes $M[j] := M[j] \oplus IV$ (here, $IV := 0^\lambda$), otherwise, it computes $r := C[j-1] \oplus v$ and $M[j] := M[j] \oplus r$.

Now, the message $M := M[1]\|M[2]\|\cdots\|M[m]$.

7.5.1.2 Security of $\hat{\chi}$

Theorem 38. Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\hat{\chi}$ has been defined in Section 7.5.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{2m(2m-1)+1}{2^\lambda}.$$

Here, the AONE-KR game is defined in Figure 7.1, and the message-length in bits is $m\lambda$.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{AONE-KR}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda)$; and our successive games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) &\stackrel{\text{def}}{=} \Pr[\text{AONE-KR}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] \\ &= \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned} \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

$$\text{Therefore, } \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4, \quad (7.1)$$

$$\begin{aligned} \text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\ \Delta_4 &\stackrel{\text{def}}{=} \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1]. \end{aligned}$$

In the following, we compute Δ_1 , Δ_2 , Δ_3 and Δ_4 (see Figure 7.9 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\hat{\chi}^{(1)}$ instead of $\hat{\chi}$. In $\hat{\chi}^{(1)}$, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_1 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.2)$$

Game G_2 : This game is identical to **Game G_1** , except that it uses $\hat{\chi}^{(2)}$ instead of $\hat{\chi}^{(1)}$. In $\hat{\chi}^{(2)}$, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_2 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.3)$$

Game G_3 : This game is identical to **Game G_2** , except that it uses $\hat{\chi}^{(3)}$ instead of $\hat{\chi}^{(2)}$. In $\hat{\chi}^{(3)}$ (as described in Figure 7.10), the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf or rf' . From Figure 7.10, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_3 \leq \frac{2m(2m-1)}{2^\lambda}. \quad (7.4)$$

In **Game G_3** , we observe that there is no information about the secret key in the adversary's view. Thus, the probability of correct guess of the key is $1/2^\lambda$, and we obtain:

$$\Delta_4 = \frac{1}{2^\lambda}. \quad (7.5)$$

Using (7.1) – (7.5), we get:

$$\begin{aligned} Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 \\ &\leq \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^\lambda} + \frac{1}{2^\lambda} \\ &\leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{2m(2m-1)+1}{2^\lambda}. \end{aligned}$$

■

Theorem 39. Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\hat{\chi}$ has been defined in Section 7.5.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(2m-1)}{2^{\lambda-1}}.$$

Here, the AONE-IND game is defined in Figure 7.2, and the message-length in bits is $m\lambda$.

Game $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$	Game $\mathbf{G}_i(\mathcal{A}, 1^\lambda)$
$(params^{(\hat{\chi})}, \mathcal{K}^{(\hat{\chi})}, \mathcal{M}^{(\hat{\chi})}, \mathcal{C}^{(\hat{\chi})})$	$(params^{(\hat{\chi}^{(i)})}, \mathcal{K}^{(\hat{\chi}^{(i)})}, \mathcal{M}^{(\hat{\chi}^{(i)})}, \mathcal{C}^{(\hat{\chi}^{(i)})})$
$:= \hat{\chi} \cdot \text{Setup}(1^\lambda);$	$:= \hat{\chi}^{(i)} \cdot \text{Setup}(1^\lambda);$
$K := \hat{\chi} \cdot \mathcal{GEN}(params^{(\hat{\chi})});$	$K := \hat{\chi}^{(i)} \cdot \mathcal{GEN}(params^{(\hat{\chi}^{(i)})});$
$M := \mathcal{A}_1^{\hat{\chi} \cdot \mathcal{E}(params^{(\hat{\chi})}, K, \cdot)}(1^\lambda);$	$M := \mathcal{A}_1^{\hat{\chi}^{(i)} \cdot \mathcal{E}(params^{(\hat{\chi}^{(i)})}, K, \cdot)}(1^\lambda);$
$C := \hat{\chi} \cdot \mathcal{E}(params^{(\hat{\chi})}, K, M);$	$C := \hat{\chi}^{(i)} \cdot \mathcal{E}(params^{(\hat{\chi}^{(i)})}, K, M);$
$K' := \mathcal{A}_2(1^\lambda, C);$	$K' := \mathcal{A}_2(1^\lambda, C);$
If $(K' = K)$, then return 1;	If $(K' = K)$, then return 1;
Else return 0;	Else return 0;

Here, in **Game \mathbf{G}_i** , $i \in [3]$.

Figure 7.9: Games used in the proof of AONE-KR security in Theorem 38.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{AONE-IND}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 1)$; our last game is $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$, which is identical to $\text{AONE-IND}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 0)$; and our intermediate games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) &\stackrel{def}{=} \left| \Pr[\text{AONE-IND}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{AONE-IND}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

Using the *triangle inequality* [BR06], we get:

$$\begin{aligned} Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

$\hat{\chi}^{(2)} \cdot \mathcal{E}(\text{params}^{(\hat{\chi}^{(2)})}, K, M)$ <div style="border: 1px solid black; padding: 5px; display: inline-block;">$\hat{\chi}^{(3)} \cdot \mathcal{E}(\text{params}^{(\hat{\chi}^{(3)})}, K, M)$</div>	$\hat{\chi}^{(2)} \cdot \mathcal{D}(\text{params}^{(\hat{\chi}^{(2)})}, K, C)$ <div style="border: 1px solid black; padding: 5px; display: inline-block;">$\hat{\chi}^{(3)} \cdot \mathcal{D}(\text{params}^{(\hat{\chi}^{(3)})}, K, C)$</div>
<i>#Initialization.</i> $IV := 0^\lambda; m := M /\lambda;$ $M[1]\ M[2]\ \cdots\ M[m] := M;$ $r := IV; s \xleftarrow{\$} \{0, 1\}^\lambda;$ $temp := 0^\lambda; u \xleftarrow{\$} \{0, 1\}^\lambda;$ $v \xleftarrow{\$} \{0, 1\}^\lambda;$ $U := \emptyset; bad_1 := 0;$	<i>#Validating the length.</i> If ($C \notin \mathcal{C}^{(\hat{\chi})}$) then return \perp ;
<i>#Initialization.</i> $IV := 0^\lambda; m := C /\lambda - 3;$ $C[1]\ C[2]\ \cdots\ C[m+3] := C;$ $s := C[m+1] \oplus K; temp := 0^\lambda;$ $u := C[m+2]; v := C[m+3] \oplus K;$ $U := \emptyset; bad_1 := 0;$	<i>#Initialization.</i>
<i>#Processing blocks.</i> for ($j := 1, 2, \dots, m$) If ($s \in U$) OR ($v \in U$) <div style="border: 1px solid black; padding: 2px; display: inline-block;">$bad_1 := 1$</div> $U := U \cup \{s, v\};$ $r\ s := \text{rf}((M[j] \oplus r)\ s);$ $u\ v := \text{rf}(u\ v);$ If ($j = m$) $C[m] := r \oplus temp;$ $C[m+1] := s \oplus K;$ $C[m+2] := u;$ $C[m+3] := v \oplus K;$ Else $temp := temp \oplus r;$ $C[j] := v \oplus r;$	<i>#Computing XOR value.</i> for ($j := m, m-1, \dots, 2$) $u\ v := \text{rf}'(u\ v);$ $temp := temp \oplus v \oplus C[j-1];$
<i>#Processing blocks.</i> $r := C[m] \oplus temp; u := C[m+2];$ $v := C[m+3] \oplus K;$ for ($j := m, m-1, \dots, 1$) If ($s \in U$) OR ($v \in U$) <div style="border: 1px solid black; padding: 2px; display: inline-block;">$bad_1 := 1$</div> $U := U \cup \{s, v\}; u\ v := \text{rf}'(u\ v);$ $M[j]\ s := \text{rf}'(r\ s);$ If ($j = 1$) $M[j] := M[j] \oplus IV;$ Else $r := C[j-1] \oplus v;$ $M[j] := M[j] \oplus r;$	<i>#Computing final outputs.</i> $M := M[1]\ M[2]\ \cdots\ M[m];$ return M ;

Figure 7.10: Algorithmic description of $\hat{\chi}^{(2)}$ and $\hat{\chi}^{(3)}$.

$$\text{Therefore, } Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4, \quad (7.6)$$

$$\begin{aligned} \text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_2 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\ \Delta_3 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\ \Delta_4 &\stackrel{\text{def}}{=} \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

In the following, we compute Δ_1 , Δ_2 , Δ_3 and Δ_4 (see Figure 7.11 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\hat{\chi}^{(1)}$ instead of $\hat{\chi}$. In $\hat{\chi}^{(1)}$, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_1 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.7)$$

Game \mathbf{G}_2 : This game is identical to **Game \mathbf{G}_1** , except that it uses $\hat{\chi}^{(2)}$ instead of $\hat{\chi}^{(1)}$. In $\hat{\chi}^{(2)}$, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get:

$$\Delta_2 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.8)$$

Game \mathbf{G}_3 : This game is identical to **Game \mathbf{G}_2** , except that it uses $\hat{\chi}^{(3)}$ instead of $\hat{\chi}^{(2)}$. In $\hat{\chi}^{(3)}$ (as described in Figure 7.10), the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf or rf' . From Figure 7.10, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_3 \leq \frac{2m(2m-1)}{2^\lambda}. \quad (7.9)$$

Since, $\hat{\chi}^{(3)}. \mathcal{E}(\text{params}^{(\hat{\chi}^{(3)})}, K, M)$ is uniformly distributed over $\{0, 1\}^{|M|+3\lambda}$, the two games \mathbf{G}_3 and \mathbf{G}_L are indistinguishable, given all other lines of code are identical. Hence, we obtain:

$$\Delta_4 = 0. \quad (7.10)$$

Using (7.6) – (7.10), we get:

$$\begin{aligned}
 Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 \\
 &\leq \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^\lambda} + 0 \\
 &\leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(2m-1)}{2^{\lambda-1}}.
 \end{aligned}$$

■

Game \mathbf{G}_S ($\mathcal{A}, 1^\lambda$)
$(params^{(\hat{\chi})}, \mathcal{K}^{(\hat{\chi})}, \mathcal{M}^{(\hat{\chi})}, \mathcal{C}^{(\hat{\chi})}) := \hat{\chi}.\text{Setup}(1^\lambda);$
$K := \hat{\chi}.\mathcal{GEN}(params^{(\hat{\chi})});$
$M := \mathcal{A}_1^{\hat{\chi}, \mathcal{E}_K(params^{(\hat{\chi})}, \cdot)}(1^\lambda);$
$C_1 := \hat{\chi}.\mathcal{E}_K(params^{(\hat{\chi})}, M);$
$b' := \mathcal{A}_2(1^\lambda, C_0);$
return b' ;
Game \mathbf{G}_i ($\mathcal{A}, 1^\lambda$)
$(params^{(\hat{\chi}^{(i)})}, \mathcal{K}^{(\hat{\chi}^{(i)})}, \mathcal{M}^{(\hat{\chi}^{(i)})}, \mathcal{C}^{(\hat{\chi}^{(i)})}) := \hat{\chi}^{(i)}.\text{Setup}(1^\lambda);$
$K := \hat{\chi}^{(i)}.\mathcal{GEN}(params^{(\hat{\chi}^{(i)})});$
$M := \mathcal{A}_1^{\hat{\chi}^{(i)}, \mathcal{E}_K(params^{(\hat{\chi}^{(i)})}, \cdot)}(1^\lambda);$
$C_1 := \hat{\chi}^{(i)}.\mathcal{E}_K(params^{(\hat{\chi}^{(i)})}, M);$
$b' := \mathcal{A}_2(1^\lambda, C_0);$
return b' ;
Game \mathbf{G}_L ($\mathcal{A}, 1^\lambda$)
$(params^{(\hat{\chi})}, \mathcal{K}^{(\hat{\chi})}, \mathcal{M}^{(\hat{\chi})}, \mathcal{C}^{(\hat{\chi})}) := \hat{\chi}.\text{Setup}(1^\lambda);$
$K := \hat{\chi}.\mathcal{GEN}(params^{(\hat{\chi})});$
$M := \mathcal{A}_1^{\hat{\chi}, \mathcal{E}_K(params^{(\hat{\chi})}, \cdot)}(1^\lambda);$
$C_1 := \hat{\chi}.\mathcal{E}_K(params^{(\hat{\chi})}, M);$
$C_0 \xleftarrow{\$} \{0, 1\}^{ C_1 };$
$b' := \mathcal{A}_2(1^\lambda, C_0);$
return b' ;

Figure 7.11: Games used in the proof of AONE-IND security in Theorem 39.

Theorem 40. Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\hat{\chi}$ has been defined in Section 7.5.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(4m-1)}{2^\lambda}.$$

Here, the AONE-AON game is defined in Figure 7.3, and the message-length in bits is $m\lambda$.

Proof. We prove the security by constructing a series of games (or hybrids): our starting game is $\mathbf{G}_S(\mathcal{A}, 1^\lambda)$, which is identical to $\text{AONE-AON}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 1)$; our last game is $\mathbf{G}_L(\mathcal{A}, 1^\lambda)$, which is identical to $\text{AONE-AON}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 0)$; and our intermediate games are $\mathbf{G}_1(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_2(\mathcal{A}, 1^\lambda)$, $\mathbf{G}_3(\mathcal{A}, 1^\lambda)$ and $\mathbf{G}_4(\mathcal{A}, 1^\lambda)$. Then, we compute the advantages between the successive games. We now bound the adversarial advantage:

$$\begin{aligned} \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{AONE-AON}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{AONE-AON}_{\hat{\chi}}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right| \\ &= \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \right. \\ &\quad \left. + \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

Using the triangle inequality [BR06], we get:

$$\begin{aligned} \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) &\leq \left| \Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \right| \\ &\quad + \left| \Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - \Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|. \end{aligned}$$

$$\text{Therefore, } \text{Adv}_{\hat{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) \leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 + \Delta_5, \quad (7.11)$$

$$\begin{aligned}
\text{where, } \Delta_1 &\stackrel{\text{def}}{=} \left| Pr[\mathbf{G}_S(\mathcal{A}, 1^\lambda) = 1] - Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_2 &\stackrel{\text{def}}{=} \left| Pr[\mathbf{G}_1(\mathcal{A}, 1^\lambda) = 1] - Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_3 &\stackrel{\text{def}}{=} \left| Pr[\mathbf{G}_2(\mathcal{A}, 1^\lambda) = 1] - Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] \right|, \\
\Delta_4 &\stackrel{\text{def}}{=} \left| Pr[\mathbf{G}_3(\mathcal{A}, 1^\lambda) = 1] - Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] \right|, \text{ and} \\
\Delta_5 &\stackrel{\text{def}}{=} \left| Pr[\mathbf{G}_4(\mathcal{A}, 1^\lambda) = 1] - Pr[\mathbf{G}_L(\mathcal{A}, 1^\lambda) = 1] \right|.
\end{aligned}$$

In the following, we compute $\Delta_1, \Delta_2, \dots$ and Δ_5 (see Figure 7.12 for the algorithmic description of all the games).

Game \mathbf{G}_1 : This game is identical to **Game \mathbf{G}_S** , except that it uses $\hat{\chi}^{(1)}$ instead of $\hat{\chi}$. In $\hat{\chi}^{(1)}$, the 2λ -bit permutation π is replaced by a 2λ -bit random function rf . Therefore, by using the *PRP/PRF switching lemma* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_1 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.12)$$

Game \mathbf{G}_2 : This game is identical to **Game \mathbf{G}_1** , except that it uses $\hat{\chi}^{(2)}$ instead of $\hat{\chi}^{(1)}$. In $\hat{\chi}^{(2)}$, the 2λ -bit permutation π^{-1} is replaced by a 2λ -bit random function rf' . Therefore, by using the *PRP/PRF switching lemma* [BR06], as before, we get::

$$\Delta_2 \leq \frac{2m(2m-1)}{2^{2\lambda}}. \quad (7.13)$$

Game \mathbf{G}_3 : This game is identical to **Game \mathbf{G}_2** , except that it uses $\hat{\chi}^{(3)}$ instead of $\hat{\chi}^{(2)}$. In $\hat{\chi}^{(3)}$ (as described in Figure 7.10), the bad_1 flag is set when there is a collision among the lower λ -bit components of the 2λ -bit inputs to the random functions rf or rf' . From Figure 7.10, using the *code-based game playing technique* [BR06], for an adversary \mathcal{A} with the message M (where $|M| = m\lambda$), we obtain:

$$\Delta_3 \leq \frac{2m(2m-1)}{2^\lambda}. \quad (7.14)$$

Game \mathbf{G}_4 : This game is identical to **Game \mathbf{G}_3** , except that it uses $\hat{\chi}^{(4)}$ instead of $\hat{\chi}^{(3)}$. In $\hat{\chi}^{(4)}$ (as described in Figure 7.13), the bad_2 flag is set when there is a collision of $C[m]$ with $C[j]$, for some $j \in [m-1]$. Since, the event that $bad_1 = 0$ makes sure that there is no collision among the

lower λ -bit components of the 2λ -bit inputs to the random function rf , this implies that $C[1], C[2], \dots, C[m - 1]$ are random and independent (so is r). Therefore, $C[m]$ – which is $\left(\bigoplus^{m-1} C[i]\right) \oplus r$ – is also random and independent. Therefore, from Figure 7.13, using the *code-based game playing technique* [BR06], the probability that $C[m]$ collides with $C[j]$, for some $j \in [m - 1]$, is (using the principle of inclusion-exclusion):

$$\Delta_4 \leq \frac{m}{2^\lambda} \quad (7.15)$$

Since, $\hat{\chi}^{(4)}. \mathcal{E}(params^{(\hat{\chi}^{(4)}, K, M)})$ is uniformly distributed over $\{0, 1\}^{|M|+3\lambda}$, the two games \mathbf{G}_4 and \mathbf{G}_L are indistinguishable, given all other lines of code are identical. Hence, we obtain:

$$\Delta_5 = 0. \quad (7.16)$$

Using (7.11) – (7.16), we get:

$$\begin{aligned} Adv_{\hat{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) &\leq \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 + \Delta_5 \\ &\leq \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^{2\lambda}} + \frac{2m(2m-1)}{2^\lambda} + \frac{m}{2^\lambda} + 0 \\ &\leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(4m-1)}{2^\lambda}. \end{aligned}$$

■

7.5.2 Construction $\tilde{\chi}$

This construction is motivated by the design of FP hash mode of operation (described in Section 2.2.3.2).

7.5.2.1 Description of $\tilde{\chi}$

The pictorial description and pseudocode for the 3-tuple of algorithms in $\tilde{\chi} = (\tilde{\chi}. \mathcal{GEN}, \tilde{\chi}. \mathcal{E}, \tilde{\chi}. \mathcal{D})$ over the *setup* algorithm $\tilde{\chi}. \mathsf{Setup}$, are given in Figures 7.14, 7.15, 7.16, 7.17 and 7.18; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\tilde{\chi}. \mathsf{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $params^{(\tilde{\chi})}$, key-space $\mathcal{K}^{(\tilde{\chi})} := \{0, 1\}^\lambda$, message-space $\mathcal{M}^{(\tilde{\chi})} := \bigcup^{i \geq 1} \{0, 1\}^{i\lambda}$ and ciphertext-space $\mathcal{C}^{(\tilde{\chi})} := \bigcup^{i \geq 4} \{0, 1\}^{i\lambda}$.

Game G_S ($\mathcal{A}, 1^\lambda$)

$$(params^{(\hat{x})}, \mathcal{K}^{(\hat{x})}, \mathcal{M}^{(\hat{x})}, \mathcal{C}^{(\hat{x})}) := \hat{x}.\text{Setup}(1^\lambda);$$

$$K := \hat{x}.\mathcal{GEN}(params^{(\hat{x})});$$

$$(M, S) := \mathcal{A}_1(1^\lambda, \hat{x});$$

$$C[1] \| C[2] \| \cdots \| C[m] := \hat{x}.\mathcal{E}(params^{(\hat{x})}, K, M);$$

$$b' := \mathcal{A}_2^{\beta_C}(1^\lambda, S, K);$$

$$\text{return } b';$$
Game G_i ($\mathcal{A}, 1^\lambda$)

$$(params^{(\hat{x}^{(i)})}, \mathcal{K}^{(\hat{x}^{(i)})}, \mathcal{M}^{(\hat{x}^{(i)})}, \mathcal{C}^{(\hat{x}^{(i)})}) := \hat{x}^{(i)}.\text{Setup}(1^\lambda);$$

$$K := \hat{x}^{(i)}.\mathcal{GEN}(params^{(\hat{x}^{(i)})});$$

$$(M, S) := \mathcal{A}_1(1^\lambda, \hat{x}^{(i)});$$

$$C[1] \| C[2] \| \cdots \| C[m] := \hat{x}^{(i)}.\mathcal{E}(params^{(\hat{x}^{(i)})}, K, M);$$

$$b' := \mathcal{A}_2^{\beta_C}(1^\lambda, S, K);$$

$$\text{return } b';$$
Game G_L ($\mathcal{A}, 1^\lambda$)

$$(params^{(\hat{x})}, \mathcal{K}^{(\hat{x})}, \mathcal{M}^{(\hat{x})}, \mathcal{C}^{(\hat{x})}) := \hat{x}.\text{Setup}(1^\lambda);$$

$$K := \hat{x}.\mathcal{GEN}(params^{(\hat{x})});$$

$$(M, S) := \mathcal{A}_1(1^\lambda, \hat{x});$$

$$C[1] \| C[2] \| \cdots \| C[m] \xleftarrow{\$} \{0, 1\}^{|M|+3\lambda};$$

$$b' := \mathcal{A}_2^{\beta_C}(1^\lambda, S, K);$$

$$\text{return } b';$$

Figure 7.12: Games used in the proof of AONE-AON security in Theorem 40.

- The PPT *key generation* algorithm $\tilde{x}.\mathcal{GEN}(\cdot)$ takes as input parameter $params^{(\hat{x})}$, and outputs key K randomly chosen from the the *key-space* $\mathcal{K}^{(\hat{x})}$. The pseudocode is given in Figure 7.14.
- The *encryption* algorithm $\tilde{x}.\mathcal{E}(\cdot)$ is randomized that takes as input parameter $params^{(\hat{x})}$, key K and message M , and outputs ciphertext C . The pictorial description and pseudocode are given in Figures 7.15 and 7.16. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \| M[2] \| \cdots \| M[m] := M$; then it randomly chooses three λ -bit strings s, u and v ; and after that it initializes the strings t and $temp$ with IV and 0^λ (here, $IV := 0^\lambda$). The encryp-

$\hat{\chi}^{(3)}. \mathcal{E}(params^{(\hat{\chi}^{(3)})}, K, M)$	$\hat{\chi}^{(4)}. \mathcal{E}(params^{(\hat{\chi}^{(4)})}, K, M)$
--	--

```

#Initialization.
 $IV := 0^\lambda; m := |M|/\lambda; M[1]\|M[2]\|\cdots\|M[m] := M;$ 
 $r := IV; s \xleftarrow{\$} \{0,1\}^\lambda; temp := 0^\lambda; u \xleftarrow{\$} \{0,1\}^\lambda;$ 
 $v \xleftarrow{\$} \{0,1\}^\lambda; U := \emptyset; V := \emptyset; bad_1 := 0; bad_2 := 0;$ 

#Processing blocks.
for ( $j := 1, 2, \dots, m$ )
  If ( $s \in U$ ) OR ( $v \in U$ ), then  $bad_1 := 1$ ;
   $U := U \cup \{s, v\}; r\|s := \text{rf}((M[j] \oplus r)\|s); u\|v := \text{rf}(u\|v);$ 
  If ( $j = m$ )
     $C[m] := r \oplus temp; \quad \text{If } C[m] \in V, \text{then } \boxed{bad_2 := 1};$ 
     $C[m+1] := s \oplus K; \quad C[m+2] := u; \quad C[m+3] := v \oplus K;$ 
  Else
     $temp := temp \oplus r; \quad C[j] := v \oplus r; \quad V := V \cup \{C[j]\};$ 

#Computing final outputs.
return  $C := C[1]\|C[2]\|\cdots\|C[m+3];$ 

```

Figure 7.13: Algorithmic description of $\hat{\chi}^{(3)}$ and $\hat{\chi}^{(4)}$.

$\tilde{\chi}. \mathcal{GEN}(params^{(\tilde{\chi})})$
--

```

#Choosing a random  $\lambda$ -bit key.
 $K \xleftarrow{\$} \{0,1\}^\lambda;$ 
return  $K;$ 

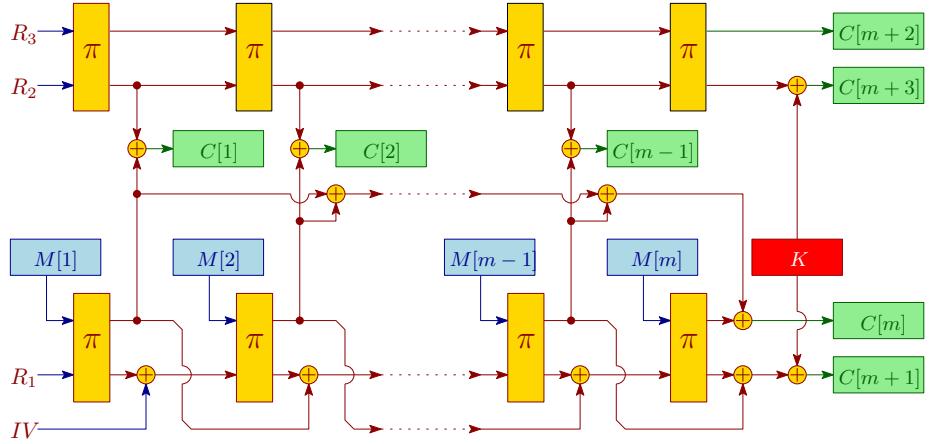
```

Figure 7.14: Algorithmic description of the *key generation* function $\tilde{\chi}. \mathcal{GEN}$.

tion of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$: first $r\|s := \pi(M[j]\|s)$ is computed; then $u\|v := \pi(u\|v)$ is computed; after that $C[j] := v \oplus r$ is computed; next $temp := temp \oplus r$ is updated; after that $s := s \oplus t$ is computed; and finally $t := r$ is assigned.

For the encryption of last block $M[m]$, we perform the following operations: first $r\|s := \pi(M[m]\|s)$ is computed; then $u\|v := \pi(u\|v)$ is

Figure 7.15: Pictorial description of the *encryption* function $\tilde{\chi} \cdot \mathcal{E}$.

$\tilde{\chi} \cdot \mathcal{E}(\text{params}^{(\tilde{\chi})}, K, M)$

#Initialization.

```

 $IV := 0^\lambda; \quad m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$ 
 $s \xleftarrow{\$} \{0, 1\}^\lambda; \quad t := IV; \quad temp := 0^\lambda; \quad u \xleftarrow{\$} \{0, 1\}^\lambda;$ 
 $v \xleftarrow{\$} \{0, 1\}^\lambda;$ 

```

#Processing blocks.

```

for ( $j := 1, 2, \dots, m$ )
     $r \| s := \pi(M[j] \| s); \quad u \| v := \pi(u \| v);$ 
    If ( $j = m$ )
         $C[m] := r \oplus temp; \quad C[m + 1] := s \oplus t \oplus K;$ 
         $C[m + 2] := u; \quad C[m + 3] := v \oplus K;$ 
    Else
         $C[j] := v \oplus r; \quad temp := temp \oplus r; \quad s := s \oplus t; \quad t := r;$ 

```

#Computing final outputs.

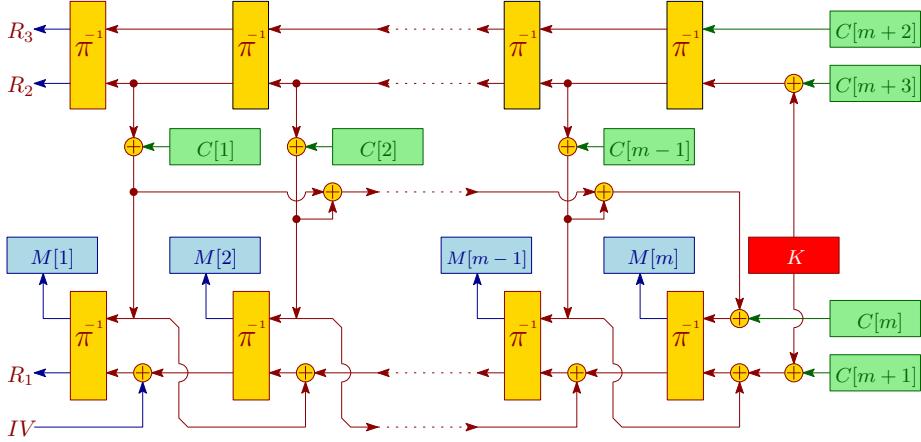
```
return  $C := C[1]\|C[2]\|\cdots\|C[m + 3];$ 
```

Figure 7.16: Algorithmic description of the *encryption* function $\tilde{\chi} \cdot \mathcal{E}$.

computed; next $C[m] := r \oplus temp$ is computed; after that $C[m + 1] := s \oplus t \oplus K$ is computed; next $C[m + 2] := u$ is assigned; and finally $C[m + 3] := v \oplus K$ is computed.

Now, the ciphertext $C := C[1]\|C[2]\|\cdots\|C[m + 3]$.

- The *decryption* algorithm $\tilde{\chi} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input

Figure 7.17: Pictorial description of the *decryption* function $\tilde{\chi} \cdot \mathcal{D}$.

```

 $\tilde{\chi} \cdot \mathcal{D}(\textit{params}^{(\tilde{\chi})}, K, C)$ 
# Validating the length.
If ( $C \notin \mathcal{C}^{(\tilde{\chi})}$ ), then return  $\perp$ ;

# Initialization.
 $IV := 0^\lambda; m := (|C|/\lambda) - 3; C[1]\|C[2]\|\cdots\|C[m+3] := C;$ 
 $s := C[m+1] \oplus K; t := 0^\lambda; u := C[m+2];$ 
 $v := C[m+3] \oplus K;$ 

# Computing XOR value.
for ( $j := m, m-1, \dots, 2$ )
   $u\|v := \pi^{-1}(u\|v); t := t \oplus v \oplus C[j-1];$ 

# Processing blocks.
 $t := C[m] \oplus t; u := C[m+2]; v := C[m+3] \oplus K;$ 
for ( $j := m, m-1, \dots, 1$ )
   $r := t; u\|v := \pi^{-1}(u\|v);$ 
  If ( $j = 1$ ), then  $t := IV$ ; Else  $t := C[j-1] \oplus v;$ 
   $s := s \oplus t; M[j]\|s := \pi^{-1}(r\|s);$ 

# Computing final outputs.
return  $M := M[1]\|M[2]\|\cdots\|M[m];$ 

```

Figure 7.18: Algorithmic description of the *decryption* function $\tilde{\chi} \cdot \mathcal{D}$.

parameter $params^{(\tilde{\chi})}$, key K and ciphertext C , and outputs message M or an *invalid* symbol \perp . See Figures 7.17 and 7.18 for the pictorial description and pseudocode. The algorithm returns \perp , if the ciphertext $C \notin \mathcal{C}^{(\tilde{\chi})}$. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \| C[2] \| \cdots \| C[m+3] := C$; then it computes $s := C[m+1] \oplus K$; next it initializes the strings t and u with 0^λ and $C[m+2]$; and finally it computes $v := C[m+3] \oplus K$. Now, the value of t is computed iteratively, as j runs through $m, m-1, \dots, 2$, in the following way: first $u \| v := \pi^{-1}(u \| v)$ is computed; and then $t := t \oplus v \oplus C[j-1]$ is updated. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

For the decryption of last block of ciphertext, first of all, the following operations are performed: first it computes $t := C[m] \oplus t$; then it assigns $u := C[m+2]$; and finally it computes $v := C[m+3] \oplus K$.

Now, we give the details of decryption of ciphertext block $C[j]$, for all $j \in [m]$: first it assigns $r := t$; then it computes $u \| v := \pi^{-1}(u \| v)$; next it checks if $j = 1$, then it assigns $t := IV$ (here, $IV := 0^\lambda$), otherwise, it computes $t := C[j-1] \oplus v$; next it computes $s := s \oplus t$; and finally it computes $M[j] \| s := \pi^{-1}(r \| s)$.

Now, the message $M := M[1] \| M[2] \| \cdots \| M[m]$.

7.5.2.2 Security of $\tilde{\chi}$

Theorem 41. *Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\tilde{\chi}$ has been defined in Section 7.5.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\tilde{\chi}, \mathcal{A}}^{\text{AONE-KR}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{2m(2m-1)+1}{2^\lambda}.$$

Here, the AONE-KR game is defined in Figure 7.1, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 38. ■

Theorem 42. *Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\tilde{\chi}$ has been defined in Section 7.5.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\tilde{\chi}, \mathcal{A}}^{\text{AONE-IND}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(2m-1)}{2^{\lambda-1}}.$$

Here, the *AONE-IND* game is defined in Figure 7.2, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 39. \blacksquare

Theorem 43. Let $\lambda \in \mathbb{N}$ be the security parameter. The AONE construction $\tilde{\chi}$ has been defined in Section 7.5.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,

$$\text{Adv}_{\tilde{\chi}, \mathcal{A}}^{\text{AONE-AON}}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(4m-1)}{2^\lambda}.$$

Here, the *AONE-AON* game is defined in Figure 7.3, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 40. \blacksquare

7.6 Comparison

In this section, we compare all the existing *AONE* schemes with the new ones. Here, we assume that the message-length $|M|$ and the security parameter λ are identical for all the schemes.

Also, we consider that before an algorithm starts, entire input to the algorithm is already written onto a read-only *input tape*, and during the execution of algorithm, as soon as the output blocks are generated, they are immediately written on the write-only-once *output tape*. We do not compare the size of *input* or *output tapes*. Instead, we only compare the amount of memory required by the *processing unit* to store the intermediate variables. Also, in cases when there are two possibilities of the execution of the algorithm – parallel execution with high memory requirement, and serial execution with low memory requirement – we prefer to compare the second mode of execution over the first one.

In our comparison tables: $c_{\mathcal{E}_\lambda}$ (or $c_{\mathcal{D}_\lambda}$) denotes the computing cost of invocation of encryption (or decryption) of blockcipher for λ -bit input; $c_{\mathcal{G}_{\lambda, |M|}}$ denotes the cost of generator function which takes a λ -bit input and outputs a $|M|$ -bit string in [Boy99, ZHI04]; $c_{\mathcal{H}_{\lambda, |M|}}$ denotes the cost of hash operation on a $|M|$ -bit message, to produce a λ -bit hash; c_{+_λ} denotes the cost of adding two λ -bit inputs; $c_{\mathcal{E}'_{|M|}}$ (or $c_{\mathcal{D}'_{|M|}}$) denotes the cost of encryption (or decryption) of full-domain cipher on a message (or ciphertext) of length $|M|$; c_{\bullet_λ} denotes the cost of group operation in [MAB07] on two λ -bit inputs; and $c_{\pi_{2\lambda}}$ (or $c_{\pi_{2\lambda}^{-1}}$) denotes the cost of invocation of 2λ -bit permutation (or 2λ -bit inverse permutation).

Cons. →	Rivest (1997) [Riv97]	Boyko (1999) [Boy99]*	Desai (2000) [Des00]	Stinson (2001) [Sti01]	ZHI (2004) [ZHI04]*	MM (2006) [MM06]	MAB (2007) [MAB07]	Our Constructions
Prop. ↓								$\hat{\chi}$
Underlying Primitive	Block Cipher	OAEP	Block Cipher	Linear Transf.	Full-Dom. Cipher	Hash Function	CTR Mode	Quasi Groups
Mem. Req. (Proc. Unit):								
• \mathcal{E}	4λ	$ M +2\lambda$	4λ	2λ	$ M +\lambda$	$ M +3\lambda$	7λ	$\lambda^3+3\lambda^2$
• \mathcal{D}	4λ	$ M +2\lambda$	3λ	2λ	$ M +\lambda$	$ M +3\lambda$	$ M +6\lambda$	$\lambda^3+3\lambda^2$
Ciphertext Expansion	λ	λ	λ	0	λ	2λ	λ	5λ
Exec. Time:								
• \mathcal{E}	$3\frac{ M }{\lambda}c_{\mathcal{E}}$	$c_{\mathcal{G}_{\lambda, M }} + c_{\mathcal{H}_{\lambda, M }} + c'_{\mathcal{E}_{ M }}$	$2\frac{ M }{\lambda}c_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}c_{\mathcal{E}_{\lambda}} + \frac{ M }{\lambda}c_{\mathcal{E}_{\lambda}}$	$2c'_{\mathcal{E}'_{ M }}$	$c_{\mathcal{G}_{\lambda, M }} + 2c_{\mathcal{H}_{\lambda, M }} + \frac{1}{2}\frac{ M }{\lambda}c_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}c_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}\lambda^2c_{\bullet_\lambda}$
• \mathcal{D}	$2\frac{ M }{\lambda}c_{\mathcal{E}}$	$c_{\mathcal{G}_{\lambda, M }} + c_{\mathcal{H}_{\lambda, M }} + c'_{\mathcal{D}'_{ M }}$	$\frac{ M }{\lambda}c_{\mathcal{E}_\lambda} + \frac{ M }{\lambda}c_{\mathcal{D}_\lambda}$	$2\frac{ M }{\lambda}c_{\mathcal{E}_\lambda} + \frac{ M }{\lambda}c_{\mathcal{D}_\lambda}$	$2c'_{\mathcal{D}'_{ M }}$	$c_{\mathcal{G}_{\lambda, M }} + 2c_{\mathcal{H}_{\lambda, M }} + \frac{1}{2}\frac{ M }{\lambda}c_{\mathcal{D}_\lambda}$	$2\frac{ M }{\lambda}c_{\mathcal{E}_\lambda}$	$2\frac{ M }{\lambda}\lambda^2c_{\bullet_\lambda}$
No. of Passes								
• ENCRYPTION	1	2	1	2	2	2.5	1	1
• DECRYPTION	2	3	2	2	2	3.5	2	2

*The proposed constructions are for fixed-length messages.

Table 7.1: Comparison table for various *AOME* schemes.

Since the *full-domain* cipher is used to encrypt fixed-length messages, that is, it cannot be used to encrypt messages of arbitrary length, it is never preferred to the blockcipher-based or permutation-based encryptions, which are used for encrypting messages of arbitrary lengths.

The cost of invocation of encryption (or decryption) function of a blockcipher on a λ -bit input is usually greater than the cost of invocation of a 2λ -bit permutation (or an inverse permutation), that is, $c_{E_\lambda} > c_{\pi_{2\lambda}}$ (or $c_{D_\lambda} > c_{\pi_{2\lambda}^{-1}}$). This is because, a blockcipher can be viewed as a family of permutations (and their inverses), indexed by the key, and it also includes a key schedule algorithm.

The cost of addition operation $c_{+\lambda}$, XOR operation c_{\oplus_λ} , etc. on two λ -bit inputs, are usually much smaller than the cost of group operation on two λ -bit inputs, c_{\bullet_λ} , as used in [MAB07].

In Table 7.1, we compare all the existing and new *AONE* schemes. Since, all the *AONE* constructions are *AONE-KR*, *AONE-IND* and *AONE-AON* secure, therefore, we only compare their efficiencies w.r.t. time and memory.

We observe that our proposed schemes $\hat{\chi}$ and $\tilde{\chi}$ outperform all the existing constructions.

7.7 *One-Time AONE*: A Novel AONE variant

The cryptographic primitive *one-time AONE* is an important variant of *AONE*, which differs from *AONE* (as discussed in Section 7.4) in the following way: in the latter, we can generate the symmetric key (i.e. it is used for both encryption and decryption) once, and then we reuse the same key for encrypting several messages; while in the former, a new decryption key is generated with every new message, and *encryption requires no key*, that is, the primitive is effectively a *keyless* encryption. Therefore, *one-time AONE* can have critical use cases such as: when a keyless encryption is required; given the fact that the primitive is basically an asymmetric encryption in a very unique way, it may find various interesting applications in the future, especially when management of secret keys is a challenging task in an application.

We now elaborately discuss the syntax, correctness and security definition of the *one-time AONE*.

7.7.1 Syntax

Suppose $\lambda \in \mathbb{N}$ is the security parameter. A *one-time AONE* scheme $\Upsilon = (\Upsilon.\mathcal{E}, \Upsilon.\mathcal{D})$ is a pair of algorithms over the *setup* algorithm $\Upsilon.\text{Setup}$, satisfying the following conditions.¹

1. The PPT *setup* algorithm $\Upsilon.\text{Setup}(1^\lambda)$ outputs parameter $\text{params}^{(\Upsilon)}$, key-space $\mathcal{K}^{(\Upsilon)} \subseteq \{0, 1\}^*$, message-space $\mathcal{M}^{(\Upsilon)} \subseteq \{0, 1\}^*$ and ciphertext-space $\mathcal{C}^{(\Upsilon)} \subseteq \{0, 1\}^*$.
2. The PPT *encryption* algorithm $\Upsilon.\mathcal{E}(\cdot)$ takes as input parameter $\text{params}^{(\Upsilon)}$ and message $M \in \mathcal{M}^{(\Upsilon)}$, and returns a pair $(K, C) := \Upsilon.\mathcal{E}(\text{params}^{(\Upsilon)}, M)$, where key $K \in \mathcal{K}^{(\Upsilon)}$ and ciphertext $C \in \mathcal{C}^{(\Upsilon)}$.
3. The *decryption* algorithm $\Upsilon.\mathcal{D}(\cdot)$ is a deterministic *poly-time* algorithm that takes as input parameter $\text{params}^{(\Upsilon)}$, key $K \in \mathcal{K}^{(\Upsilon)}$ and ciphertext $C' \in \{0, 1\}^*$, and returns message $M := \Upsilon.\mathcal{D}(\text{params}^{(\Upsilon)}, K, C')$, where $M \in \mathcal{M}^{(\Upsilon)} \cup \{\perp\}$.

Note: We restrict ciphertext expansion to $p\lambda$, where $p \in \mathbb{N}$ is fixed. In other words, $|C| - |M| \leq p\lambda$, for all $M \in \mathcal{M}^{(\Upsilon)}$.

We remark that, according to point 2, a *one-time AONE* requires no encryption key, and the decryption key generated is message-dependent.

7.7.2 Correctness

This condition requires that if the ciphertext is generated from the correct input, decryption will produce the correct output. Mathematically, the *correctness* of an *one-time AONE* can be defined as follows.

The correctness of Υ requires that for all $(\lambda, M) \in \mathbb{N} \times \mathcal{M}^{(\Upsilon)}$, we have:

$$\Upsilon.\mathcal{D}(\text{params}^{(\Upsilon)}, \Upsilon.\mathcal{E}(\text{params}^{(\Upsilon)}, M)) = M.$$

7.7.3 Security Definitions

The security properties of *one-time AONE* are identical to that of *AONE*, i.e. *key recovery resistance*, *indistinguishability privacy* and *all-or-nothing privacy*, but the security games differ, because of dissimilar signatures of the

¹Note that, strictly speaking, from the definitional standpoint, a *one-time AONE* is not an *AONE*, since *AONE* has three algorithms (see Section 7.4), while *one-time AONE* has two.

functions. Nevertheless, we define them for the sake of completeness. In Figures 7.19, 7.20 and 7.21, we define the security games OTAONE-KR, OTAONE-IND and OTAONE-AON, respectively, for the *one-time AONE* scheme $\Upsilon = (\Upsilon.\mathcal{E}, \Upsilon.\mathcal{D})$. As usual, the games are written using the challenger-adversary framework.

Game $\text{OTAONE-KR}_{\Upsilon}^{\mathcal{A}}(1^\lambda)$

$$(params^{(\Upsilon)}, \mathcal{K}^{(\Upsilon)}, \mathcal{M}^{(\Upsilon)}, \mathcal{C}^{(\Upsilon)}) := \Upsilon.\text{Setup}(1^\lambda);$$

$$M := \mathcal{A}_1^{\Upsilon.\mathcal{E}(params^{(\Upsilon)}, \cdot)}(1^\lambda);$$

$$(K, C) := \Upsilon.\mathcal{E}(params^{(\Upsilon)}, M);$$

$$K' := \mathcal{A}_2(1^\lambda, C);$$
If $(K' = K)$, **then** return 1;
Else return 0;

Figure 7.19: Security game OTAONE-KR for *one-time AONE* scheme Υ .

OTAONE-KR SECURITY. This captures the intuitive notion of *key recovery* attack, as defined in Figure 7.19. The adversarial advantage is the probability of the event when the adversary is able to deduce the key, and it is desired to be negligible.

According to the OTAONE-KR game, the adversary first returns a message M , and then the message M is encrypted to obtain $(K, C) := \Upsilon.\mathcal{E}(params^{(\Upsilon)}, M)$. Given access to C , the adversary returns a key K' . The game returns 1, if $K = K'$.

The advantage of the OTAONE-KR adversary \mathcal{A} against Υ is defined as follows:

$$Adv_{\Upsilon, \mathcal{A}}^{\text{OTAONE-KR}}(1^\lambda) \stackrel{\text{def}}{=} \Pr[\text{OTAONE-KR}_{\Upsilon}^{\mathcal{A}}(1^\lambda) = 1].$$

The Υ is OTAONE-KR secure, if, for all PPT adversaries \mathcal{A} , the value of $Adv_{\Upsilon, \mathcal{A}}^{\text{OTAONE-KR}}(1^\lambda)$ is *negligible*.

OTAONE-IND SECURITY. This captures the notion of *indistinguishability privacy* attack, as defined in Figure 7.20. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a message from a random string, and it is desired to be negligible.

According to the OTAONE-IND game, the adversary first returns a message M . Then the following operations are performed: first the message M is encrypted to obtain $(K, C_1) := \Upsilon.\mathcal{E}(params^{(\Upsilon)}, M)$; and then a random

```

Game OTAONE-IND $_{\Upsilon}^{\mathcal{A}}(1^\lambda, b)$ 
( $params^{(\Upsilon)}, \mathcal{K}^{(\Upsilon)}, \mathcal{M}^{(\Upsilon)}, \mathcal{C}^{(\Upsilon)}$ ) :=  $\Upsilon.\text{Setup}(1^\lambda)$ ;
 $M := \mathcal{A}_1^{\Upsilon, \mathcal{E}(params^{(\Upsilon)}, \cdot)}(1^\lambda)$ ;
 $(K, C_1) := \Upsilon.\mathcal{E}(params^{(\Upsilon)}, M)$ ;
 $C_0 \xleftarrow{\$} \{0, 1\}^{|C_1|}$ ;
 $b' := \mathcal{A}_2(1^\lambda, C_b)$ ;
return  $b'$ ;

```

Figure 7.20: Security game OTAONE-IND for *one-time AONE* scheme Υ .

string C_0 of length $|C_1|$ is computed. Given access to C_b , where $b \in \{0, 1\}$, the adversary returns a bit b' .

The advantage of the OTAONE-IND adversary \mathcal{A} against Υ is defined as follows:

$$Adv_{\Upsilon, \mathcal{A}}^{\text{OTAONE-IND}}(1^\lambda) \stackrel{def}{=} \left| \Pr[\text{OTAONE-IND}_{\Upsilon}^{\mathcal{A}}(1^\lambda, b = 1) = 1] - \Pr[\text{OTAONE-IND}_{\Upsilon}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|.$$

The Υ is OTAONE-IND secure, if, for all PPT adversaries \mathcal{A} , the value of $Adv_{\Upsilon, \mathcal{A}}^{\text{OTAONE-IND}}(1^\lambda)$ is *negligible*.

```

Game OTAONE-AON $_{\Upsilon}^{\mathcal{A}}(1^\lambda, b)$ 
( $params^{(\Upsilon)}, \mathcal{K}^{(\Upsilon)}, \mathcal{M}^{(\Upsilon)}, \mathcal{C}^{(\Upsilon)}$ ) :=  $\Upsilon.\text{Setup}(1^\lambda)$ ;
 $(M, S) := \mathcal{A}_1(1^\lambda, \Upsilon)$ ;
 $(K, C_1) := \Upsilon.\mathcal{E}(params^{(\Upsilon)}, M)$ ;  $C_0 \xleftarrow{\$} \{0, 1\}^{|C_1|}$ ;
 $C_0[1] \| C_0[2] \| \cdots \| C_0[m] := C_0$ ;
 $C_1[1] \| C_1[2] \| \cdots \| C_1[m] := C_1$ ;
 $b' := \mathcal{A}_2^{\beta}(1^\lambda, S, K)$ ;
return  $b'$ ;

```

Figure 7.21: Security game OTAONE-AON for *one-time AONE* scheme Υ .

OTAONE-AON SECURITY. This security is dedicated to the *all-or-nothing (AON)* property, as defined in Figure 7.21. The adversarial advantage is the probability of the event when the adversary is able to distinguish the encryption of a known message from a random string without the knowledge of at least λ bits, and it is desired to be negligible.

According to the OTAONE-AON game, given access to $\Upsilon(\cdot)$, the adversary first returns a message M and some state information S . Then the following operations are performed: first the message M is encrypted to obtain $(K, C_1) := \Upsilon.\mathcal{E}(\text{params}^{(\Upsilon)}, M)$; then a random string C_0 of length $|C_1|$ is generated; and finally C_0 and C_1 are parsed into λ -bit blocks $C_0[1]\|C_0[2]\|\cdots\|C_0[m] := C_0$ and $C_1[1]\|C_1[2]\|\cdots\|C_1[m] := C_1$. An oracle β now operates the following way: it takes as input the index $j \in [m]$, and returns the block $C_b[j]$, where $b \in \{0, 1\}$. The adversary is allowed to make at most $(m - 1)$ adaptive queries to the oracle β . Given access to β , the adversary returns a bit b' .

The advantage of the OTAONE-AON adversary \mathcal{A} against Υ is defined as follows:

$$\begin{aligned} \text{Adv}_{\Upsilon, \mathcal{A}}^{\text{OTAONE-AON}}(1^\lambda) &\stackrel{\text{def}}{=} \left| \Pr[\text{OTAONE-AON}_{\Upsilon}^{\mathcal{A}}(1^\lambda, b = 1) = 1] \right. \\ &\quad \left. - \Pr[\text{OTAONE-AON}_{\Upsilon}^{\mathcal{A}}(1^\lambda, b = 0) = 1] \right|. \end{aligned}$$

The Υ is OTAONE-AON secure, if, for all PPT adversaries \mathcal{A} , the value of $\text{Adv}_{\Upsilon, \mathcal{A}}^{\text{OTAONE-AON}}(1^\lambda)$ is *negligible*.

7.8 New *One-Time AONE* Constructions

In this section, we design two concrete *one-time AONE* schemes, denoted, $\hat{\Upsilon}$ and $\tilde{\Upsilon}$. The main component of these constructions is an *easy-to-invert* permutation, instead of a blockcipher. These schemes are exceptionally good – both in terms of computation cost and memory requirements (in terms of the ciphertext expansion). Since the key and ciphertext are generated together, we save the cost of key generation. We assume that the message-length in bits is a multiple of the security parameter λ .

7.8.1 Construction $\hat{\Upsilon}$

This construction is motivated by the design of APE authenticated encryption (described in Section 2.2.5.2).

7.8.1.1 Description of $\hat{\Upsilon}$

The pictorial description and pseudocode for the pair of algorithms in $\hat{\Upsilon} = (\hat{\Upsilon}.\mathcal{E}, \hat{\Upsilon}.\mathcal{D})$ over the *setup* algorithm $\hat{\Upsilon}.\text{Setup}$, are given in Figures 7.22, 7.23, 7.24 and 7.25; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\hat{\Upsilon}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\hat{\Upsilon})}$, key-space $\mathcal{K}^{(\hat{\Upsilon})} := \{0, 1\}^\lambda$, message-space $\mathcal{M}^{(\hat{\Upsilon})} := \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and ciphertext-space $\mathcal{C}^{(\hat{\Upsilon})} := \bigcup_{i \geq 3} \{0, 1\}^{i\lambda}$.

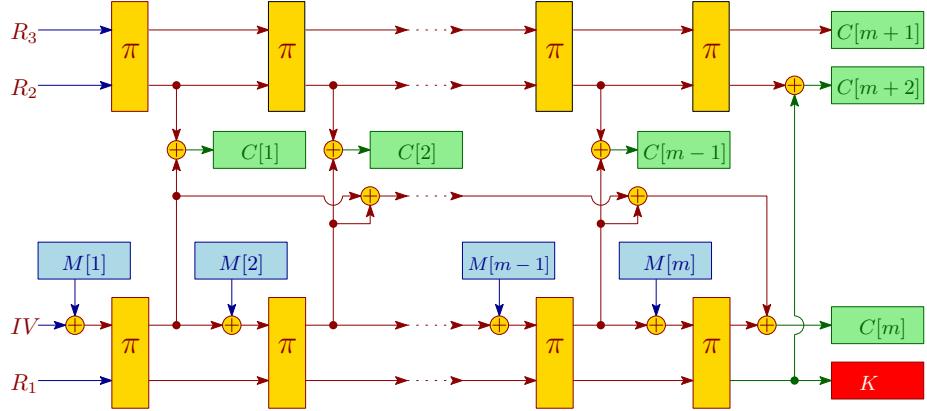


Figure 7.22: Pictorial description of the *encryption* function $\hat{\Upsilon}.\mathcal{E}$.

$\hat{\Upsilon}.\mathcal{E}(\text{params}^{(\hat{\Upsilon})}, M)$

#Initialization.

$$IV := 0^\lambda; \quad m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$$

$$r := IV; \quad s \xleftarrow{\$} \{0, 1\}^\lambda; \quad temp := 0^\lambda; \quad u \xleftarrow{\$} \{0, 1\}^\lambda;$$

$$v \xleftarrow{\$} \{0, 1\}^\lambda;$$

#Processing blocks.

for ($j := 1, 2, \dots, m$)

$$r\|s := \pi((r \oplus M[j])\|s); \quad u\|v := \pi(u\|v);$$

If ($j = m$)

$$C[m] := r \oplus temp; \quad C[m+1] := u; \quad C[m+2] := v \oplus s;$$

Else

$$temp := temp \oplus r; \quad C[j] := v \oplus r;$$

#Computing final outputs.

$$K := s; \quad C := C[1]\|C[2]\|\cdots\|C[m+2];$$

return (K, C) ;

Figure 7.23: Algorithmic description of the *encryption* function $\hat{\Upsilon}.\mathcal{E}$.

- The *encryption* algorithm $\hat{\Upsilon} \cdot \mathcal{E}(\cdot)$ is randomized that takes as input parameter $params^{(\hat{\Upsilon})}$ and message M , and outputs key K and ciphertext C . The pictorial description and pseudocode are given in Figures 7.22 and 7.23. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1] \parallel M[2] \parallel \dots \parallel M[m] := M$; then it randomly chooses three λ -bit strings s, u and v ; and after that it initializes the strings r and $temp$ with IV and 0^λ (here, $IV := 0^\lambda$). The encryption of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$: first $r \parallel s := \pi((r \oplus M[j]) \parallel s)$ is computed; then $u \parallel v := \pi(u \parallel v)$ is computed; after that $temp := temp \oplus r$ is updated; and finally $C[j] := v \oplus r$ is computed.

For the encryption of last block $M[m]$, we perform the following operations: first $r \parallel s := \pi((r \oplus M[m]) \parallel s)$ is computed; then $u \parallel v := \pi(u \parallel v)$ is computed; after that $C[m] := r \oplus temp$ is computed; next $C[m+1] := u$ is assigned; and finally $C[m+2] := v \oplus s$ is computed.

Now, the key $K := s$, and the ciphertext $C := C[1] \parallel C[2] \parallel \dots \parallel C[m+2]$.

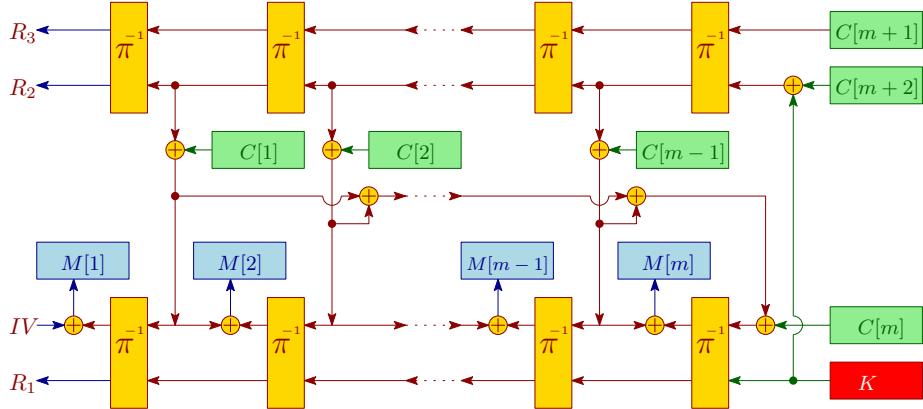


Figure 7.24: Pictorial description of the *decryption* function $\hat{\Upsilon} \cdot \mathcal{D}$.

- The *decryption* algorithm $\hat{\Upsilon} \cdot \mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\hat{\Upsilon})}$, key K and ciphertext C , and outputs message M or an *invalid* symbol \perp . See Figures 7.24 and 7.25 for the pictorial description and pseudocode. The algorithm returns \perp , if the ciphertext $C \notin \mathcal{C}^{(\hat{\Upsilon})}$. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1] \parallel C[2] \parallel \dots \parallel C[m+2] := C$; then

```

 $\hat{\Upsilon} \cdot \mathcal{D}(params^{(\hat{\Upsilon})}, K, C)$ 
# Validating the length.
If ( $C \notin \mathcal{C}^{(\hat{\Upsilon})}$ ), then return  $\perp$ ;

# Initialization.
 $IV := 0^\lambda; m := |C|/\lambda - 2; C[1]\|C[2]\|\cdots\|C[m+2] := C;$ 
 $s := K; temp := 0^\lambda; u := C[m+1]; v := C[m+2] \oplus s;$ 

# Computing XOR value.
for ( $j := m, m-1, \dots, 2$ )
   $u\|v := \pi^{-1}(u\|v); temp := temp \oplus v \oplus C[j-1];$ 

# Processing blocks.
 $r := C[m] \oplus temp; u := C[m+1]; v := C[m+2] \oplus s;$ 
for ( $j := m, m-1, \dots, 1$ )
   $M[j]\|s := \pi^{-1}(r\|s); u\|v := \pi^{-1}(u\|v);$ 
  If ( $j = 1$ ), then  $M[j] := M[j] \oplus IV;$ 
  Else
     $r := C[j-1] \oplus v; M[j] := M[j] \oplus r;$ 

# Computing final outputs.
return  $M := M[1]\|M[2]\|\cdots\|M[m];$ 

```

Figure 7.25: Algorithmic description of the *decryption* function $\hat{\Upsilon} \cdot \mathcal{D}$.

it assigns $s := K$; after that it initializes $temp := 0^\lambda$; next it assigns $u := C[m+1]$; and finally it computes $v := C[m+2] \oplus s$. Now, the value of $temp$ is computed iteratively, as j runs through $m, m-1, \dots, 2$, in the following way: first $u\|v := \pi^{-1}(u\|v)$ is computed; and then $temp := temp \oplus v \oplus C[j-1]$ is updated. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

For the decryption of last block of ciphertext, first of all, the following operations are performed: first it computes $r := C[m] \oplus temp$; then it assigns $u := C[m+1]$; and after that it computes $v := C[m+2] \oplus s$.

Now, we give the details of decryption of the ciphertext block $C[j]$, for all $j \in [m]$: first it computes $M[j]\|s := \pi^{-1}(r\|s)$; then it computes $u\|v := \pi^{-1}(u\|v)$; and finally if $j = 1$, then it computes $M[j] := M[j] \oplus IV$ (here, $IV := 0^\lambda$), otherwise it computes $r := C[j-1] \oplus v$

and $M[j] := M[j] \oplus r$.

Now, the message $M := M[1] \| M[2] \| \cdots \| M[m]$.

7.8.1.2 Security of $\hat{\Upsilon}$

Theorem 44. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\hat{\Upsilon}$ has been defined in Section 7.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Upsilon}, \mathcal{A}}^{OTAONE-KR}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{2m(2m-1)+1}{2^\lambda}.$$

Here, the OTAONE-KR game is defined in Figure 7.19, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 38. ■

Theorem 45. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\hat{\Upsilon}$ has been defined in Section 7.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Upsilon}, \mathcal{A}}^{OTAONE-IND}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(2m-1)}{2^{\lambda-1}}.$$

Here, the OTAONE-IND game is defined in Figure 7.20, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 39. ■

Theorem 46. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\hat{\Upsilon}$ has been defined in Section 7.8.1. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$Adv_{\hat{\Upsilon}, \mathcal{A}}^{OTAONE-AON}(1^\lambda) \leq \frac{m(2m-1)}{2^{2\lambda-2}} + \frac{m(4m-1)}{2^\lambda}.$$

Here, the OTAONE-AON game is defined in Figure 7.21, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 40. ■

7.8.2 Construction $\tilde{\Upsilon}$

This construction is motivated by the design of FP hash mode of operation (described in Section 2.2.3.2).

7.8.2.1 Description of $\tilde{\Upsilon}$

The pictorial description and pseudocode for the pair of algorithms in $\tilde{\Upsilon} = (\tilde{\Upsilon}.E, \tilde{\Upsilon}.D)$ over the *setup* algorithm $\tilde{\Upsilon}.\text{Setup}$, are given in Figures 7.26, 7.27, 7.28 and 7.29; all wires are λ -bit long. It is worth noting that the decryption is executed in the *reverse* direction of encryption. Below, we give the textual description.

- The *setup* algorithm $\tilde{\Upsilon}.\text{Setup}(\cdot)$ takes as input security parameter $\lambda \in \mathbb{N}$, and outputs parameter $\text{params}^{(\tilde{\Upsilon})}$, key-space $\mathcal{K}^{(\tilde{\Upsilon})} := \{0, 1\}^\lambda$, message-space $\mathcal{M}^{(\tilde{\Upsilon})} := \bigcup_{i \geq 1} \{0, 1\}^{i\lambda}$ and ciphertext-space $\mathcal{C}^{(\tilde{\Upsilon})} := \bigcup_{i \geq 3} \{0, 1\}^{i\lambda}$.

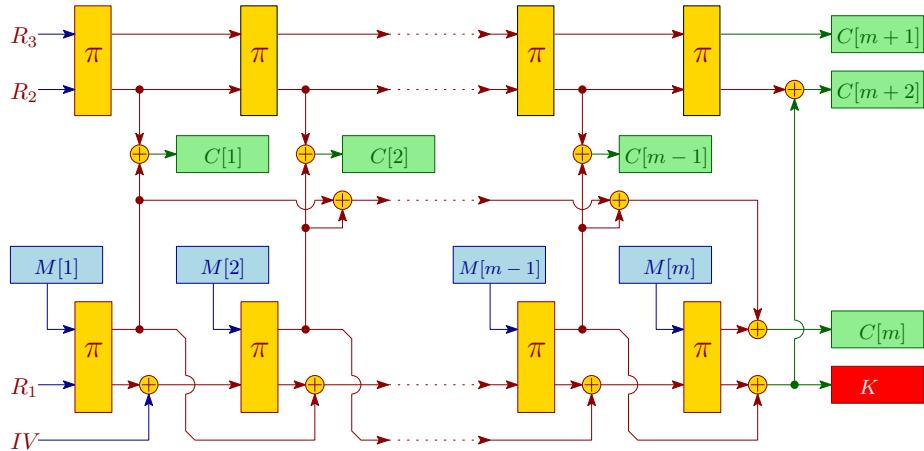


Figure 7.26: Pictorial description of the *encryption* function $\tilde{\Upsilon}.E$.

- The *encryption* algorithm $\tilde{\Upsilon}.E(\cdot)$ is randomized that takes as input parameter $\text{params}^{(\tilde{\Upsilon})}$ and message M , and outputs key K and ciphertext C . The pictorial description and pseudocode are given in Figures 7.26 and 7.27. Very briefly, the algorithm works as follows: first it parses M into λ -bit blocks $M[1]||M[2]||\dots||M[m] := M$; then it randomly chooses three λ -bit strings s, u and v ; and next it initializes the strings t and temp with IV and 0^λ (here, $IV := 0^\lambda$). The encryption of M is composed of encryption of individual blocks in the same sequence.

Now, we give the details of encryption of message block $M[j]$, for all $j \neq m$: first it computes $r||s := \pi(M[j]||s)$; next it computes $u||v := \pi(u||v)$; after that it computes $s := s \oplus t$; next it computes

```

 $\tilde{\Upsilon}.\mathcal{E}(params^{(\tilde{\Upsilon})}, M)$ 

#Initialization.
 $IV := 0^\lambda; \quad m := |M|/\lambda; \quad M[1]\|M[2]\|\cdots\|M[m] := M;$ 
 $s \xleftarrow{\$} \{0, 1\}^\lambda; \quad t := IV; \quad u \xleftarrow{\$} \{0, 1\}^\lambda; \quad v \xleftarrow{\$} \{0, 1\}^\lambda;$ 
 $temp := 0^\lambda;$ 

#Processing blocks.
for ( $j := 1, 2, \dots, m$ )
   $r\|s := \pi(M[j]\|s); \quad u\|v := \pi(u\|v); \quad s := s \oplus t;$ 
  If ( $j = m$ )
     $C[m] := r \oplus temp; \quad C[m + 1] := u; \quad C[m + 2] := v \oplus s;$ 
  Else
     $C[j] := v \oplus r; \quad temp := temp \oplus r; \quad t := r;$ 

#Computing final outputs.
 $K := s; \quad C := C[1]\|C[2]\|\cdots\|C[m + 2];$ 
return ( $K, C$ );

```

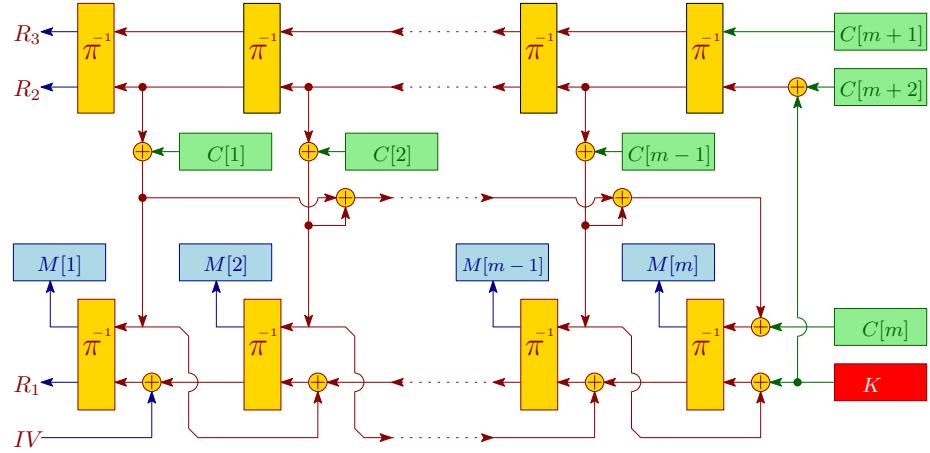
Figure 7.27: Algorithmic description of the *encryption* function $\tilde{\Upsilon}.\mathcal{E}$.

$C[j] := v \oplus r$; then it updates $temp := temp \oplus r$; and finally it assigns $t := r$.

For the encryption of last block $M[m]$, we perform the following operations: first it computes $r\|s := \pi(M[m]\|s)$; then it computes $u\|v := \pi(u\|v)$; after that it computes $s := s \oplus t$; next it computes $C[m] := r \oplus temp$; then it assigns $C[m + 1] := u$; and finally it computes $C[m + 2] := v \oplus s$.

Now, the key $K := s$, and the ciphertext $C := C[1]\|C[2]\|\cdots\|C[m + 2]$.

- The *decryption* algorithm $\tilde{\Upsilon}.\mathcal{D}(\cdot)$ is deterministic that takes as input parameter $params^{(\tilde{\Upsilon})}$, key K and ciphertext C , and outputs message M or an *invalid* symbol \perp . See Figures 7.28 and 7.29 for the pictorial description and pseudocode. The algorithm returns \perp , if the ciphertext $C \notin \mathcal{C}^{(\tilde{\Upsilon})}$. Very briefly, the algorithm works as follows: first it parses C into λ -bit blocks $C[1]\|C[2]\|\cdots\|C[m + 2] := C$; then it assigns $s := K$; next it initializes the string $t := 0^\lambda$; after that it assigns $u := C[m + 1]$; and finally it computes $v := C[m + 2] \oplus s$. Now, the value of t is computed iteratively, as j runs through $m, m - 1, \dots, 2$

Figure 7.28: Pictorial description of the *decryption* function $\tilde{\Upsilon} \cdot \mathcal{D}$.

```

 $\tilde{\Upsilon} \cdot \mathcal{D}(\text{params}^{(\tilde{\Upsilon})}, K, C)$ 
# Validating the length.
If ( $C \notin \mathcal{C}^{(\tilde{\Upsilon})}$ ), then return  $\perp$ ;

# Initialization.
 $IV := 0^\lambda; m := |C|/\lambda - 2; C[1]\|C[2]\|\dots\|C[m+2] := C;$ 
 $s := K; t := 0^\lambda; u := C[m+1]; v := C[m+2] \oplus s;$ 

# Computing XOR value.
for ( $j := m, m-1, \dots, 2$ )
   $u\|v := \pi^{-1}(u\|v); t := t \oplus v \oplus C[j-1];$ 

# Processing blocks.
 $t := C[m] \oplus t; u := C[m+1]; v := C[m+2] \oplus s;$ 
for ( $j := m, m-1, \dots, 1$ )
   $r := t; u\|v := \pi^{-1}(u\|v);$ 
  If ( $j = 1$ ), then  $t := IV$ ; Else  $t := C[j-1] \oplus v;$ 
   $s := s \oplus t; M[j]\|s := \pi^{-1}(r\|s);$ 

# Computing final outputs.
return  $M := M[1]\|M[2]\|\dots\|M[m];$ 

```

Figure 7.29: Algorithmic description of the *decryption* function $\tilde{\Upsilon} \cdot \mathcal{D}$.

in the following way: first it computes $u\|v := \pi^{-1}(u\|v)$; and then

it updates $t := t \oplus v \oplus C[j - 1]$. The decryption of C is composed of decryption of individual blocks in direction *opposite* to that of the encryption.

For the decryption of last block of ciphertext, first of all, the following operations are performed: first it computes $t := C[m] \oplus t$; then it assigns $u := C[m + 1]$; and finally it computes $v := C[m + 2] \oplus s$.

Now, we give the details of decryption of ciphertext block $C[j]$, for all $j \in [m]$: first it assigns $r := t$; then it computes $u \| v := \pi^{-1}(u \| v)$; next it checks if $j = 1$, then it assigns $t := IV$ (here, $IV := 0^\lambda$), otherwise it computes $t := C[j - 1] \oplus v$; after that it computes $s := s \oplus t$; and finally it computes $M[j] \| s := \pi^{-1}(r \| s)$.

Now, the message $M := M[1] \| M[2] \| \dots \| M[m]$.

7.8.2.2 Security of $\tilde{\Upsilon}$

Theorem 47. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\tilde{\Upsilon}$ has been defined in Section 7.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$\text{Adv}_{\tilde{\Upsilon}, \mathcal{A}}^{\text{OTAONE-KR}}(1^\lambda) \leq \frac{m(2m - 1)}{2^{2\lambda-2}} + \frac{2m(2m - 1) + 1}{2^\lambda}.$$

Here, the OTAONE-KR game is defined in Figure 7.19, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 38. ■

Theorem 48. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\tilde{\Upsilon}$ has been defined in Section 7.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$\text{Adv}_{\tilde{\Upsilon}, \mathcal{A}}^{\text{OTAONE-IND}}(1^\lambda) \leq \frac{m(2m - 1)}{2^{2\lambda-2}} + \frac{m(2m - 1)}{2^{\lambda-1}}.$$

Here, the OTAONE-IND game is defined in Figure 7.20, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 39. ■

Theorem 49. *Let $\lambda \in \mathbb{N}$ be the security parameter. The one-time AONE construction $\tilde{\Upsilon}$ has been defined in Section 7.8.2. If π is assumed to be a 2λ -bit ideal permutation, then for all adversaries \mathcal{A} ,*

$$\text{Adv}_{\tilde{\Upsilon}, \mathcal{A}}^{\text{OTAONE-AON}}(1^\lambda) \leq \frac{m(2m - 1)}{2^{2\lambda-2}} + \frac{m(4m - 1)}{2^\lambda}.$$

Here, the *OTAONE-AON* game is defined in Figure 7.21, and the message-length in bits is $m\lambda$.

Proof. The proof is similar to that of Theorem 40. ■

7.9 Conclusion and Future Work

In this chapter, we proposed a new cryptographic primitive *one-time AONE*, besides providing a flexible definition of *AONE*. We gave four efficient design constructions – two *AONES* and two *one-time AONES* – all based on a fixed permutation. All our constructions make use of the *reverse decryption* property found in the *APE* authenticated encryption and the *FP* hash mode of operation. We have also provided the efficiency comparison for all the *AONE* constructions. One future work could be to add more functions to *AONE* such as *update* and *authentication*.

CHAPTER 8

Conclusion and Future Scope

In this thesis, we have designed cryptographic primitives with potential applications in cloud storage systems and in lightweight applications of the *IoT* devices. In particular, we have designed about a dozen of cryptographic schemes that fall into the following five notions: *file-updatable message-locked encryption (FMLE)*, *key assignment scheme with authenticated encryption (KAS-AE)*, *all-or-nothing transform (AONT)*, *all-or-nothing encryption (AONE)* and *one-time AONE*; out of these, *FMLE*, *KAS-AE* and *one-time AONE* have been introduced for the first time. All our schemes have been shown to be more efficient and secure than the existing ones. At the heart of our constructions is a unique property known as the *reverse decryption* that is found only in *APE* authenticated encryption and *FP* hash mode of operation. We exploited this basic property to support in our constructions the following attributes: loss of the last ciphertext-block makes it impossible to recover any bit of the plaintext; a bit-flip in an intermediate ciphertext-block induces error in all preceding plaintext-blocks.

In all our schemes, we have quantified the parameter p that computes the ciphertext (or pseudo-message) expansion (in terms of λ -bit blocks), which is the difference between the lengths of ciphertext (or pseudo-message) and plaintext. For our *FMLE* constructions, this ciphertext expansion is just one λ -bit block, that is a constant value, while in the state-of-the-art *UMLE* scheme, it is in order of logarithmic in the size of message (for a 1024-bit message and $\lambda = 128$ bits, the *UMLE* scheme employing 256-bit block-size, the ciphertext expansion is approximately 20 λ -bit blocks). For our *KAS-AE* schemes, the ciphertext expansion for each node is in order of degree

of nodes (i.e. the number of immediate children of the node in the access graph) and for the wholesome file, it is in the order of quadratic in terms of the number of nodes in the access graph. Similarly, for our *AONTs*, the pseudo-message expansion is just one λ -bit block, and for our *AONEs* and *one-time AONEs*, the ciphertext expansion is just two λ -bit blocks and three λ -bit blocks, respectively.

We also have provided the rigorous security proofs for all our constructions, using various sophisticated techniques, such as *code-based game playing technique*, *PRP/PRF switching lemma*, *triangle inequality*, *principle of inclusion-exclusion*, and various graph-theoretic results. We have computed the bounds for various security notions on all our schemes, which turns out to be $\lambda/2$ bits each. This essentially means that if m is the number of queries/blocks that the adversary is allowed to make, then it is bounded by a *polynomial* in λ (and is certainly not *exponential* in λ). For example, if $\lambda = 80$ bits, then number of queries $m \ll 2^{80}$. If $m \geq 2^{80}$, then the security collapses.

We would like to point out that *deduplication* (or *HAC*) is an overlying multi-party protocol that critically uses the underlying cryptographic primitive *MLE* (or *KAS*) and its variants. In fact, it would not be an exaggeration to say that an *MLE* (or *KAS*) primitive is at the heart of the *deduplication* (or *HAC*) protocol. In this thesis, we only compare the performance of the cryptographic primitives *MLE* (or *KAS*) and its variants, which is given in Chapter 4 (and Chapter 5). Comparison of the performance of *deduplication* or *HAC* protocols per se at a high-level requires a separate discussion, and is outside of the scope of this thesis.

In all our primitives, the management of secret keys (decryption keys for *FMLE*, *KAS-AE* and *one-time AONE*, and encryption/decryption key for *AONE*) is identical to the existing one. We do not introduce any change in this context. Also, the theoretical/practical load on *key exchange* protocol should be identical. This is because we do not introduce any change in the size or structure of the key, or the design of the *key exchange* protocol itself.

Our work leaves open interesting research directions for future work. We outline them below.

- Constructions of STC secure and efficient *FMLEs*.
- Design of more functions, such as *key revocation* and *file update*, in *KAS-AE* and designing efficient constructions.
- Design of *update* function into *AONT*.
- Design of *update* and *authentication of data* in *AONE*.

- Discovering more cryptographic primitives to solve real-life problems exploiting the rare *reverse decryption* property.
- Analysis of our constructions in the *RUP* setting (shorthand for *release of unverified plaintext*).

Bibliography

- [AAB15] Javad Alizadeh, Mohammad Reza Aref, and Nasour Bagheri. Artemia v1, 2015. 2
- [ABB⁺14] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. APE: authenticated permutation-based encryption for lightweight cryptography. 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 168–186. Springer, 2014. 2, 28, 134, 148, 162, 218, 238
- [ABF07] Mikhail J. Atallah, Marina Blanton, and Keith B. Frikken. Incorporating temporal capabilities in existing key management schemes. In Joachim Biskup and Javier Lopez, editors, *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2007. 74
- [ABFF09] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12(3):18:1–18:43, 2009. 70, 74, 76, 213
- [ABM⁺13] Martín Abadi, Dan Boneh, Ilya Mironov, Ananth Raghunathan, and Gil Segev. Message-Locked Encryption for Lock-Dependent Messages. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 374–391. Springer, 2013. 58
- [ACS15] Megha Agrawal, Donghoon Chang, and Somitra Sanadhya. spaelm: Sponge based authenticated encryption scheme for

- memory constrained devices. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings*, volume 9144 of *Lecture Notes in Computer Science*, pages 451–468. Springer, 2015. [2](#)
- [ADMA15] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 364–384. Springer, 2015. [22](#)
- [AFB05] Mikhail J. Atallah, Keith B. Frikken, and Marina Blanton. Dynamic and efficient key management for access hierarchies. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 190–202. ACM, 2005. [70](#), [74](#), [76](#), [213](#)
- [ASFM06] Giuseppe Ateniese, Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Provably-secure time-bound hierarchical key assignment schemes. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 288–297. ACM, 2006. [70](#), [74](#)
- [ASFM12] Giuseppe Ateniese, Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Provably-secure time-bound hierarchical key assignment schemes. *J. Cryptology*, 25(2):243–270, 2012. [74](#)
- [ASFM13] Giuseppe Ateniese, Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. A note on time-bound hierarchical key assignment schemes. *Inf. Process. Lett.*, 113(5-6):151–155, 2013. [74](#)

- [AT83] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst.*, 1(3):239–248, 1983. [70](#), [74](#), [76](#), [213](#)
- [BDC05] Alex Biryukov and Christophe De Cannière. "Data encryption standard (DES)", pages 129–135. Springer US, Boston, MA, 2005. [11](#)
- [BDJR97] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *FOCS*, pages 394–403. IEEE Computer Society, 1997. [23](#)
- [BDP⁺14] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Caesar submission: Ketje v1, 2014. [22](#)
- [BDPA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge Functions. ECRYPT 2007, 2007. <http://sponge.noekeon.org/SpongeFunctions.pdf>. Accessed March 2012. [21](#), [22](#)
- [BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008. [22](#)
- [BDPA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak Hash Function. The 1st SHA-3 Candidate Conference, 2009. [22](#)
- [BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011. [22](#)
- [Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958. [37](#)

- [BK15] Mihir Bellare and Sriram Keelvedhi. Interactive message-locked encryption and secure deduplication. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 516–538. Springer, 2015. 58
- [BKR13a] Mihir Bellare, Sriram Keelvedhi, and Thomas Ristenpart. DupLESS: Server-Aided Encryption for Deduplicated Storage. In Sam King, editor, *USENIX*, pages 179–194, 2013. 58
- [BKR13b] Mihir Bellare, Sriram Keelvedhi, and Thomas Ristenpart. Message-Locked Encryption and Secure Deduplication. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 296–312. Springer, 2013. 49, 51, 53, 56, 57, 58, 87, 129, 130, 134, 148
- [BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptology*, 21(4):469–491, 2008. 137, 148
- [Boy99] Victor Boyko. On the security properties of OAEP as an all-or-nothing transform. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 503–518. Springer, 1999. 219, 233, 234, 262, 263
- [BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. 33, 115, 116, 169, 170, 171, 173, 174, 176, 177, 226, 227, 248, 249, 250, 252, 254, 255, 256
- [BRW03] Mihir Bellare, Phillip Rogaway, and David A. Wagner. EAX: A conventional authenticated-encryption mode. *IACR Cryptology ePrint Archive*, 2003:69, 2003. 137, 148

- [BSJ08] Elisa Bertino, Ning Shang, and Samuel S. Wagstaff Jr. An efficient time-bound hierarchical key management scheme for secure broadcasting. *IEEE Trans. Dependable Sec. Comput.*, 5(2):65–70, 2008. [74](#)
- [CC02] Tzer-Shyong Chen and Yu-Fang Chung. Hierarchical access control based on chinese remainder theorem and symmetric algorithm. *Computers & Security*, 21(6):565–570, 2002. [70](#), [74](#), [76](#), [213](#)
- [CCM15] Massimo Cafaro, Roberto Civino, and Barbara Masucci. On the equivalence of two security notions for hierarchical key assignment schemes in the unconditional setting. *IEEE Trans. Dependable Sec. Comput.*, 12(4):485–490, 2015. [70](#), [74](#)
- [CDM10] Jason Crampton, Rosli Daud, and Keith M. Martin. Constructing key assignment schemes from chain partitions. In Sara Foresti and Sushil Jajodia, editors, *Data and Applications Security and Privacy XXIV, 24th Annual IFIP WG 11.3 Working Conference, Rome, Italy, June 21-23, 2010. Proceedings*, volume 6166 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2010. [70](#), [74](#), [76](#), [77](#)
- [CEM16] Jan Camenisch, Robert R. Enderlein, and Ueli Maurer. Memory erasability amplification. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, volume 9841 of *Lecture Notes in Computer Science*, pages 104–125. Springer, 2016. [217](#), [219](#)
- [CH05] Tzer-Shyong Chen and Jen-Yan Huang. A novel key management scheme for dynamic access control in a user hierarchy. *Applied Mathematics and Computation*, 162(1):339–351, 2005. [76](#), [213](#)
- [Chi04] Hung-Yu Chien. Efficient time-bound hierarchical key assignment scheme. *IEEE Trans. Knowl. Data Eng.*, 16(10):1301–1304, 2004. [74](#)
- [CHW92] Chin-Chen Chang, Ren-Junn Hwang, and Tzong-Chen Wu. Cryptographic key assignment scheme for access control in a hierarchy. *Inf. Syst.*, 17(3):243–247, 1992. [76](#), [213](#)

- [CLP16] Sébastien Canard, Fabien Laguillaumie, and Marie Paindavoine. Verifiable message-locked encryption. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 299–315, 2016. [58](#)
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. [36](#), [38](#)
- [CMW06] Jason Crampton, Keith M. Martin, and Peter R. Wild. On key assignment for hierarchical access control. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 98–111. IEEE Computer Society, 2006. [70](#), [74](#)
- [CMYG15] Rongmao Chen, Yi Mu, Guomin Yang, and Fuchun Guo. BL-MLE: block-level message-locked encryption for secure large file deduplication. *IEEE Trans. Information Forensics and Security*, 10(12):2643–2652, 2015. [58](#)
- [CSM16a] Arcangelo Castiglione, Alfredo De Santis, and Barbara Masucci. Key indistinguishability versus strong key indistinguishability for hierarchical key assignment schemes. *IEEE Trans. Dependable Sec. Comput.*, 13(4):451–460, 2016. [70](#), [74](#)
- [CSM⁺16b] Arcangelo Castiglione, Alfredo De Santis, Barbara Masucci, Francesco Palmieri, and Aniello Castiglione. On the relations between security notions in hierarchical key assignment schemes for dynamic structures. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy - 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part II*, volume 9723 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2016. [70](#), [74](#)
- [DAB⁺02] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002. [56](#), [129](#), [130](#)
- [DC05] Christophe De Cannière. *Triple-DES*, pages 626–627. Springer US, Boston, MA, 2005. [11](#)

- [DEMS14] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1, 2014. [2](#)
- [Des00] Anand Desai. The security of all-or-nothing encryption: Protecting against exhaustive key search. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 359–375. Springer, 2000. [6](#), [218](#), [219](#), [220](#), [234](#), [239](#), [263](#)
- [DES16] Paolo D’Arco, Navid Nasr Esfahani, and Douglas R. Stinson. All or nothing at all. *Electr. J. Comb.*, 23(4):P4.10, 2016. [219](#)
- [Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959. [37](#)
- [Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. of Math.*, 51(2):161–166, 1950. [39](#)
- [DSFM10] Paolo D’Arco, Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Variations on a theme by akl and taylor: Security and tradeoffs. *Theor. Comput. Sci.*, 411(1):213–227, 2010. [70](#), [74](#)
- [DSST17] Yuanxi Dai, Yannick Seurin, John P. Steinberger, and Aishwarya Thiruvengadam. Indifferentiability of iterated even-mansour ciphers with non-idealized key-schedules: Five rounds are necessary and sufficient. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 524–555. Springer, 2017. [2](#), [218](#), [238](#)
- [EGS18] Navid Nasr Esfahani, Ian Goldberg, and Douglas R. Stinson. Some results on the existence of t-all-or-nothing transforms over arbitrary alphabets. *IEEE Trans. Information Theory*, 64(4):3136–3143, 2018. [219](#)
- [FEA09] Mitra Fatemi, Taraneh Eghlidos, and Mohammad Reza Aref. A multi-stage secret sharing scheme using all-or-nothing transform approach. In Sihan Qing, Chris J. Mitchell, and

- Guilin Wang, editors, *Information and Communications Security, 11th International Conference, ICICS 2009, Beijing, China, December 14-17, 2009. Proceedings*, volume 5927 of *Lecture Notes in Computer Science*, pages 449–458. Springer, 2009. [218](#), [219](#)
- [For56] Lester R. Ford. *Network Flow Theory*. Paper P. Rand Corporation, 1956. [37](#)
- [FP11] Eduarda S. V. Freire and Kenneth G. Paterson. Provably secure key assignment schemes from factoring. In Udaya Parampalli and Philip Hawkes, editors, *Information Security and Privacy - 16th Australasian Conference, ACISP 2011, Melbourne, Australia, July 11-13, 2011. Proceedings*, volume 6812 of *Lecture Notes in Computer Science*, pages 292–309. Springer, 2011. [70](#), [74](#), [76](#)
- [FPP13] Eduarda S. V. Freire, Kenneth G. Paterson, and Bertram Poettering. Simple, efficient and strongly ki-secure hierarchical key assignment schemes. In Ed Dawson, editor, *Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013, San Francisco, CA, USA, February 25-March 1, 2013. Proceedings*, volume 7779 of *Lecture Notes in Computer Science*, pages 101–114. Springer, 2013. [44](#), [70](#), [74](#), [76](#), [77](#), [137](#), [148](#), [189](#), [190](#)
- [Gud80] Ehud Gudes. The design of a cryptography based secure file system. *IEEE Trans. Software Eng.*, 6(5):411–420, 1980. [76](#), [213](#)
- [HC04] Hui-Feng Huang and Chin-Chen Chang. A new cryptographic key assignment scheme with time-constraint access control in a hierarchy. *Computer Standards & Interfaces*, 26(3):159–166, 2004. [74](#)
- [HL90] Lein Harn and Hung-Yu Lin. A cryptographic key generation scheme for multilevel data security. *Computers & Security*, 9(6):539–546, 1990. [70](#), [74](#), [76](#), [213](#)
- [HRRV15] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online authenticated-encryption and its nonce-reuse misuse-resistance. 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 I, volume 9215 of *Lecture Notes in Computer Science*, pages 493–517. Springer, 2015. 2

- [HZW17] Ke Huang, Xiao-Song Zhang, and Xiao-Fen Wang. Block-level message-locked encryption with polynomial commitment for iot data. *J. Inf. Sci. Eng.*, 33(4):891–905, 2017. 58
- [JCW⁺16] Tao Jiang, Xiaofeng Chen, Qianhong Wu, Jianfeng Ma, Willy Susilo, and Wenjing Lou. Towards efficient fully randomized message-locked encryption. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy - 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part I*, volume 9722 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2016. 58
- [KK14] Hidenori Kuwakado and Masazumi Kurihara. Secure regenerating codes using linear MBR/MSR codes and the all-or-nothing transform. In *International Symposium on Information Theory and its Applications, ISITA 2014, Melbourne, Australia, October 26-29, 2014*, pages 221–225. IEEE, 2014. 218, 219
- [KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. 10
- [KM18a] Katarzyna Kapusta and Gérard Memmi. PE-AONT: partial encryption combined with an all-or-nothing transform. *CoRR*, abs/1811.09144, 2018. 219, 239
- [KM18b] Katarzyna Kapusta and Gérard Memmi. Selective all-or-nothing transform: Protecting outsourced data against key exposure. In Arcangelo Castiglione, Florin Pop, Massimo Ficco, and Francesco Palmieri, editors, *Cyberspace Safety and Security - 10th International Symposium, CSS 2018, Amalfi, Italy, October 29-31, 2018, Proceedings*, volume 11161 of *Lecture Notes in Computer Science*, pages 181–193. Springer, 2018. 219

- [KMR19] Katarzyna Kapusta, Gérard Memmi, and Matthieu Rambaud. Circular all-or-nothing: Revisiting data protection against key exposure. *CoRR*, abs/1901.08083, 2019. [220](#)
- [Knu95] Lars R. Knudsen. A key-schedule weakness in SAFER K-64. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 274–286. Springer, 1995. [2](#), [218](#), [238](#)
- [KSW96] John Kelsey, Bruce Schneier, and David A. Wagner. Key-schedule cryptanalysis of idea, g-des, gost, safer, and triple-des. In 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*, pages 237–251. Springer, 1996. [2](#), [218](#), [238](#)
- [MAB07] Stelios I. Marnas, Lefteris Angelis, and Georgios L. Bleris. An application of quasigroups in all-or-nothing transform. *Cryptologia*, 31(2):133–142, 2007. [219](#), [233](#), [235](#), [262](#), [263](#), [264](#)
- [MM06] Robert P. McEvoy and Colin C. Murphy. Efficient all-or-nothing encryption using CTR mode. In Joaquim Filipe and Mohammad S. Obaidat, editors, *E-Business and Telecommunication Networks - Third International Conference, ICETE 2006, Setúbal, Portugal, August 7-10, 2006. Selected Papers*, volume 9 of *Communications in Computer and Information Science*, pages 92–106, 2006. [234](#), [239](#), [263](#)
- [MP12] Bart Mennink and Bart Preneel. Hash functions based on three permutations: A generic security analysis. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 330–347. Springer, 2012. [2](#), [218](#), [238](#)
- [MTMA85] Stephen J. MacKinnon, Peter D. Taylor, Henk Meijer, and Selim G. Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Trans. Computers*, 34(9):797–802, 1985. [74](#)

- [MTW⁺14] Robert P. McEvoy, Michael Tunstall, Claire Whelan, Colin C. Murphy, and William P. Marnane. All-or-nothing transforms as a countermeasure to differential side-channel analysis. *Int. J. Inf. Sec.*, 13(3):291–304, 2014. [217](#), [219](#)
- [MVM09] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009. [11](#)
- [MWT⁺13] Wenqi Ma, Qingbo Wu, Yusong Tan, Chunguang Wang, Quanyuan Wu, and Huaping Hu. Using all-or-nothing encryption to enhance the security of searchable encryption. In *16th IEEE International Conference on Computational Science and Engineering, CSE 2013, December 3-5, 2013, Sydney, Australia*, pages 104–108. IEEE Computer Society, 2013. [237](#), [239](#)
- [Nat01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). 2001. [17](#)
- [Nat12] National Institute of Standards and Technology (NIST). Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. 2012. [2](#), [218](#), [238](#)
- [Nat15] National Institute of Standards and Technology (NIST). Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance. 2015. [17](#)
- [Nat17] National Institute of Standards and Technology (NIST). Report on Lightweight Cryptography. 2017. [17](#)
- [NP10] Mridul Nandi and Souradyuti Paul. Speeding up the wide-pipe: Secure and fast hashing. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2010. [20](#)
- [PHG12] Souradyuti Paul, Ekawat Homsirikamol, and Kris Gaj. A Novel Permutation-based Hash Mode of Operation FP and the Hash Function SAMOSA. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 514 – 532, Kolkata, WB, INDIA, 2012. Springer-Verlag. [2](#), [20](#), [134](#), [148](#), [179](#)

- [PS10] Namje Park and Youjin Song. Secure RFID application data management using all-or-nothing transform encryption. In Gopal Pandurangan, V. S. Anil Kumar, Gu Ming, Yun-hao Liu, and Yingshu Li, editors, *Wireless Algorithms, Systems, and Applications, 5th International Conference, WASA 2010, Beijing, China, August 15-17, 2010. Proceedings*, volume 6221 of *Lecture Notes in Computer Science*, pages 245–252. Springer, 2010. [237](#), [239](#)
- [PWCW15a] Jeng-Shyang Pan, Tsu-Yang Wu, Chien-Ming Chen, and Eric Ke Wang. An efficient solution for time-bound hierarchical key assignment scheme. In Thi Thi Zin, Jerry Chun-Wei Lin, Jeng-Shyang Pan, Pyke Tin, and Mitsuhiro Yokota, editors, *Genetic and Evolutionary Computing - Proceedings of the Ninth International Conference on Genetic and Evolutionary Computing, ICGEC 2015, August 26-28, 2015, Yangon, Myanmar - Volume II*, volume 388 of *Advances in Intelligent Systems and Computing*, pages 3–9. Springer, 2015. [74](#)
- [PWCW15b] Jeng-Shyang Pan, Tsu-Yang Wu, Chien-Ming Chen, and Eric Ke Wang. Security analysis of a time-bound hierarchical key assignment scheme. In Jeng-Shyang Pan, Ivan Lee, Hsiang-Cheh Huang, and Ching-Yu Yang, editors, *2015 International Conference on Intelligent Information Hiding and Multimedia Signal Processing, IIH-MSP 2015, Adelaide, Australia, September 23-25, 2015*, pages 203–206. IEEE, 2015. [74](#)
- [Riv97] Ronald L. Rivest. All-or-nothing encryption and the package transform. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 210–218. Springer, 1997. [6](#), [218](#), [219](#), [234](#), [239](#), [263](#)
- [Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 98–107. ACM, 2002. [137](#), [148](#)
- [Saa14] Markku-Juhani O. Saarinen. The stribobr1 authenticated encryption algorithm, 2014. [2](#)

- [SC02] Victor R. L. Shen and Tzer-Shyong Chen. A novel key management scheme based on discrete logarithms and polynomial interpolations. *Computers & Security*, 21(2):164–171, 2002. [70](#), [74](#), [76](#), [213](#)
- [SFM07a] Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Efficient provably-secure hierarchical key assignment schemes. In Ludek Kucera and Antonín Kucera, editors, *Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*, volume 4708 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2007. [70](#), [74](#), [76](#), [213](#)
- [SFM07b] Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. In Volkmar Lotz and Bhavani M. Thuraisingham, editors, *12th ACM Symposium on Access Control Models and Technologies, SACMAT 2007, Sophia Antipolis, France, June 20-22, 2007, Proceedings*, pages 133–138. ACM, 2007. [74](#)
- [SFM08] Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. *Theor. Comput. Sci.*, 407(1-3):213–230, 2008. [74](#)
- [Sim06] Gustavus J. Simmons. Cryptology. <https://www.britannica.com/topic/cryptology>, 2006. Accessed: 2019-12-04. [9](#)
- [SSR00] Sang-Uk Shin, Weon Shin, and Kyung Hyune Rhee. All-or-nothing transform and remotely keyed encryption protocols. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography, Third International Workshop on Practice and Theory in Public Key Cryptography, PKC 2000, Melbourne, Victoria, Australia, January 18-20, 2000, Proceedings*, volume 1751 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2000. [218](#), [219](#)
- [Sti95] Douglas R. Stinson. *Cryptography - theory and practice*. Discrete mathematics and its applications series. CRC Press, 1995. [12](#), [15](#)

- [Sti01] Douglas R. Stinson. Something About All or Nothing (Transforms). *Des. Codes Cryptography*, 22(2):133–138, 2001. 219, 234, 263
- [TC95] Hui-Min Tsai and Chin-Chen Chang. A cryptographic implementation for dynamic access control in a user hierarchy. *Computers & Security*, 14(2):159–166, 1995. 76, 213
- [Til99] Henk C. A. Van Tilborg. *Fundamentals of Cryptology: A Professional Reference and Interactive Tutorial*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1999. 9, 10
- [Tze02] Wen-Guey Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *IEEE Trans. Knowl. Data Eng.*, 14(1):182–188, 2002. 74
- [Tze06] Wen-Guey Tzeng. A secure system for data access based on anonymous authentication and time-dependent hierarchical keys. In Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Paul Lin, Shiuhyung Shieh, and Sushil Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006*, pages 223–230. ACM, 2006. 74
- [WC01] Tzong-Chen Wu and Chin-Chen Chang. Cryptographic key assignment scheme for hierarchical access control. *Comput. Syst. Sci. Eng.*, 16(1):25–28, 2001. 70, 74
- [WCQ⁺17] Huige Wang, Kefei Chen, Baodong Qin, Xuejia Lai, and Yun-hua Wen. A new construction on randomized message-locked encryption in the standard model via uces. *SCIENCE CHINA Information Sciences*, 60(5):52101, 2017. 58
- [Wik19a] Wikipedia contributors. Cryptography — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Cryptography&oldid=927364827>, 2019. [Online; accessed 4-December-2019]. 9
- [Wik19b] Wikipedia contributors. Cryptosystem — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Cryptosystem&oldid=914547458>, 2019. [Online; accessed 4-December-2019]. 10

- [WL06] Shyh-Yih Wang and Chi-Sung Laih. Merging: An efficient solution for a time-bound hierarchical key assignment scheme. *IEEE Trans. Dependable Sec. Comput.*, 3(1):91–100, 2006. [74](#)
- [WY15] Dai Watanabe and Masayuki Yoshino. Key update mechanism using all-or-nothing transform for network storage of encrypted data. *IEICE Transactions*, 98-A(1):162–170, 2015. [218](#)
- [YCN03] Jyh-haw Yeh, Randy Chow, and Richard E. Newman. Key assignment for enforcing access control policy exceptions in distributed systems. *Inf. Sci.*, 152:63–88, 2003. [70](#), [74](#)
- [Yeh05] Jyh-haw Yeh. An rsa-based time-bound hierarchical key assignment scheme for electronic article subscription. In Ottinein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 285–286. ACM, 2005. [74](#)
- [Yi05] Xun Yi. Security of chien’s efficient time-bound hierarchical key assignment scheme. *IEEE Trans. Knowl. Data Eng.*, 17(9):1298–1299, 2005. [74](#)
- [YL04] Cungang Yang and Celia Li. Access control in a hierarchy using one-way hash functions. *Computers & Security*, 23(8):659–664, 2004. [76](#), [213](#)
- [YY03] Xun Yi and Yiming Ye. Security of tzeng’s time-bound key assignment scheme for access control in a hierarchy. *IEEE Trans. Knowl. Data Eng.*, 15(4):1054–1055, 2003. [74](#)
- [ZC17] Yongjun Zhao and Sherman S. M. Chow. Updatable block-level message-locked encryption. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 449–460. ACM, 2017. [5](#), [58](#), [59](#), [129](#), [130](#)
- [ZHI04] Rui Zhang, Goichiro Hanaoka, and Hideki Imai. On the security of cryptosystems with all-or-nothing transform. In

- Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Applied Cryptography and Network Security, Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004, Proceedings*, volume 3089 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2004. [219](#), [233](#), [234](#), [262](#), [263](#)
- [ZRM01] Xukai Zou, Byrav Ramamurthy, and Spyros S. Magliveras. Chinese remainder theorem based hierarchical access control for secure group communication. In Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou, editors, *Information and Communications Security, Third International Conference, ICICS 2001, Xian, China, November 13-16, 2001*, volume 2229 of *Lecture Notes in Computer Science*, pages 381–385. Springer, 2001. [76](#), [213](#)

About the Author

Suyash Kandele was born on September 11, 1992 in Bhilai, Chhattisgarh (formerly a part of Madhya Pradesh), India; the city is popular for *Bhilai Steel Plant* which is the largest iron and steel plant in Asia.

He is a recipient of the scholarship under *National Talent Search Examination (NTSE)* since 2007, awarded by *National Council of Educational Research and Training (NCERT)*. Mr. Kandele has also attained All India rank 22 (State rank 2) in the 10th *National Cyber Olympiad* (2011), conducted by *Science Olympiad Foundation (SOF)*. He finished his schooling from Krishna Public School, Bhilai in 2011.

He was the *silver medallist* in his Bachelor of Technology in Computer Science and Engineering, awarded by National Institute of Technology (NIT) Raipur (2015). He was awarded the prestigious *Start-Early PhD Fellowship* by Indian Institute of Technology (IIT) Gandhinagar in 2015, because of his exceptional academic achievements.

He was a doctoral scholar in the Discipline of Computer Science and Engineering at Indian Institute of Technology (IIT) Gandhinagar (July 2015 to December 2017). After this, he continued his doctoral study in the Discipline of Computer Science and Engineering at Indian Institute of Technology (IIT) Bhilai since December 2017.

His research interests are in various areas related to computer and information security, such as cryptographic primitives and protocols, network security, cloud security, IoT, Blockchains, etc. At leisure, he enjoys photography and gardening, as well as cooking innovative Indian dishes for friends and family.

He can be contacted at:

- suyashk@iitbhilai.ac.in
- sk.11.1992@gmail.com
- (+91) 975-465-8975