



Design and Analysis of Password-based Authentication Systems

By
Sweta Mishra

Under the Supervision of
Dr. Donghoon Chang
Dr. Somitra Kumar Sanadhya

Indraprastha Institute of Information Technology Delhi
October, 2017

©Sweta Mishra, 2017.



Design and Analysis of Password-based Authentication Systems

By
Sweta Mishra

Submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

to the

Indraprastha Institute of Information Technology Delhi
October, 2017

Certificate

This is to certify that the thesis titled - “**Design and Analysis of Password-based Authentication Systems**” being submitted by **Sweta Mishra** to Indraprastha Institute of Information Technology, Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under our supervision. In our opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

Dr. Donghoon Chang

October, 2017
Department of Computer Science
IIIT Delhi
New Delhi, 110020

Dr. Somitra Kumar Sanadhya

October, 2017
Department of Computer Science
IIT Ropar
Punjab, 140001

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Donghoon Chang, for his continuous support, for his patience, motivation, emotional support, and invaluable guidance. His enthusiasm for research and optimistic attitude have always inspired and motivated me. His extensive knowledge and experience have been the biggest help and resource for my research. I am extremely fortunate and very appreciative for all he has done for me.

I also express my sincere gratitude to my co-advisor, Dr. Somitra Kumar Sanadhya, who has helped me immensely throughout my Ph.D. life. I also appreciate very much the fact that he agreed to be my advisor when I emailed him asking about the possibility after joining the PhD programme at IIIT Delhi. His kind words at that time gave me the necessary encouragement to proceed. My research would not have been possible without his assistance.

I am grateful to Dr. Anuradha Sharma, for the inspiring and fruitful discussions I had with her while taking the important decision to pursue Ph.D. from IIIT Delhi.

I would also like to thank all my dear friends from IIIT-Delhi, especially, Mohona, Monalisa, Megha, Monika, Amani and Sumesh for their valuable friendship and providing pleasant academic atmosphere. They had major contribution to make my PhD life at IIIT-Delhi easy going and memorable.

I am grateful to all of my collaborators, especially the two undergraduate students, Akshima and Aarushi Goel for their sincerity at work.

I would also like to forward my gratitude to Tata Consultancy Services (TCS), India for awarding me the prestigious TCS fellowship for my full Ph.D. period.

I also express my regards to all of those who supported me in any respect during the completion of my research.

On a personal note, I wish to express my deepest gratitude to my parents for their immense support, care, love, encouragement and unshakable belief in me from my childhood. Special thanks to my father who is my inspiration and strength to explore the world of academic research. I am also grateful to my cousin, Ashutosh, for being my unconditional support and being so understanding. Without the support of my parents, my siblings and my cousin this dissertation would not have got completed.

List of Publications

The author names are in the alphabetical order.

1. Donghoon Chang, Arpan Jati, Sweta Mishra and Somitra Kumar Sanadhya. Rig: A Simple, Secure and Flexible Design for Password Hashing, Information Security and Cryptology - 10th International Conference, Beijing, China, December 13-15, 2014, Revised Selected Papers, volume 8957 of Lecture Notes in Computer Science, pages 361–381. Springer, 2014.
Note: Complete version of the paper is uploaded on IACR Cryptology ePrint Archive, 2015:009, 2015. <http://eprint.iacr.org/2015/009>.
2. Donghoon Chang and Arpan Jati and Sweta Mishra and Somitra Kumar Sanadhya. Cryptographic Module Based Approach for Password Hashing Schemes. Technology and Practice of Passwords - International Conference on Passwords, PASSWORDS'14, Trondheim, Norway, December 8-10, 2014, Revised Selected Papers, Volume 9393 of Lecture Notes in Computer Science, Pages 39–57. Springer, 2014.
3. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Time memory tradeoff analysis of graphs in password hashing constructions. In Preproceedings of PASSWORDS'14, pages 256–266, 2014. available at <http://passwords14.item.ntnu.no/Preproceedings.Passwords14.pdf>.

List of Journal Submissions

1. Donghoon Chang, Arpan Jati, Sweta Mishra and Somitra Kumar Sanadhya. Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes. Submitted to International Journal of Information Security, 2016. [Under submission].
2. Akshima, Donghoon Chang, Aarushi Goel, Sweta Mishra and Somitra Kumar Sanadhya. Generation of Secure and Reliable Honeywords. Submitted to Transactions on Dependable and Secure Computing, 2017. [Under submission].
3. Donghoon Chang, Sweta Mishra, Somitra Kumar Sanadhya and Ajit Pratap Singh. On Making U2F Protocol Leakage-Resilient via Re-keying. Submitted to ACM Transactions on Information and System Security, 2017. [Under submission].

List of Technical Reports

1. Donghoon Chang and Arpan Jati and Sweta Mishra and Somitra Kumar Sanadhya. Performance Analysis of Some Password Hashing Schemes. IACR Cryptology ePrint Archive, 2015:139, 2015. <http://eprint.iacr.org/2015/139>.

Abstract

Passwords are the most widely deployed means of human-computer authentication since the early 1960s. The use of passwords, which are usually low in entropy, is delicate in cryptography because of the possibility of launching an offline dictionary attack. It is ever challenging to design a password-based cryptosystem that is secure against this attack. Password-based cryptosystems broadly cover two areas - 1) Password-based authentication, e.g., *password hashing schemes* and 2) Password-based encryption specifically used in password-based authenticated key exchange (PAKE) protocols. This thesis is devoted to the secure design of password hashing algorithm and the analysis of existing password-based authentication systems.

The frequent reporting of password database leakage in real-world highlights the vulnerabilities existing in the current password based constructions. In order to alleviate these problems and to encourage strong password protection techniques, a Password Hashing Competition (PHC) was held from 2013 to 2015. Following the announced criteria, we propose a password hashing scheme *Rig* that fulfills all the required goals. We also present a cryptanalytic technique for password hashing. Further, we focus on the improvement of a password database breach detection technique and on the analysis of Universal 2nd Factor protocol. This report tries to list and summarize all the important results published in the field of password hashing in recent years and understand the extent of research over password-based authentication schemes. Our significant results are listed below.

1. Following the design requirements for a secure password hashing scheme as mentioned at the PHC [16], we present our design *Rig* which satisfies all required criteria. It is a memory hard and best performing algorithm under cache-timing attack resistant category. As part of the results, we present the construction explaining the design rationale and the proof of its collision resistance. We also provide the performance and security analysis.
2. In practice, most cryptographic designs are implemented inside a Cryptographic module, as suggested by National Institute of Standards and Technology (NIST) in a standard, FIPS 140. A cryptographic module has a limited memory and this makes it challenging to implement a password hashing scheme (PHS) inside it. We provide a cryptographic module based approach for password hashing. It helps to enhance the security of the existing password-based authentication framework. We also discuss the feasibility of the approach considering the submissions of PHC.
3. The increasing threat of password leakage from compromised password hashes demands a resource consuming algorithm to prevent the precomputation of the password hashes. A class of password hashing designs which ensure that any reduction in the memory leads to exponential increase in their runtime are called Memory hard designs. Time Memory Tradeoff (TMTO) technique is an effective cryptanalytic approach for such password hashing schemes (PHS). However, it is generally difficult to evaluate the “memory hardness” of a given PHS design. We present a simple technique to analyze TMTO for any password hashing schemes which can be represented as a directed acyclic graph.

4. Password database breach is a common practice among hackers; however, it is difficult to detect such breaches if not somehow disclosed by the attacker. A paper by Juels et al. provides a method for detecting password database breach known as ‘Honeyword’. Very less research has been reported in this direction. Realizing the importance, we analyse the limitations of existing honeyword generation techniques. We propose a new attack model and also present new and practical honeyword generation techniques.
5. A secure password hashing construction can prevent offline dictionary attack, but can not provide resistance to common online attacks. Therefore requirement of augmenting a second factor to strengthen the simple password-based authentication is a recent trend. The U2F protocol proposed by Fast IDentity Online (FIDO) alliance in 2014 has been introduced as a strong augmentation that can prevent online attacks currently faced in practice. A thorough third-party analysis is required to verify the claim of U2F developers. Therefore we work on the analysis of U2F protocol and show that the protocol is not secure against side channel attacks. We then present a new variant of the U2F protocol that has improved security guarantees.

In terms of memory hardness and performance, the design *Rig* presented in the thesis is among the best-known algorithms for password hashing [15, 85, 140]. The other results presented are significant contributions to the previously published results.

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Thesis Organization	5
1.3	Contributions	6
1.4	Statement on Research Ethics	9
2	Background	10
2.1	Preliminaries	10
2.1.1	Symmetric Cryptosystem	10
2.1.2	Password-based Authentication System	13
2.2	History of Computer Passwords	17
2.3	Password-Based Authentication	18
2.3.1	History of Password Hashing	19
2.4	Password Cracking	21
2.4.1	Online Attack	21
2.4.2	Offline Attack	22
3	Secure Design of Password Hashing Scheme	28
3.1	Password Hashing Competition	29
3.2	Significant Constructions before PHC	32
3.3	Our Password Hashing Design, Rig	37
3.3.1	Preliminaries	38
3.3.2	Specification of Rig	38
3.3.3	Design Rationale	42
3.3.4	Implementation aspects	45
3.3.5	Design of Rig [Blake2b, BlakeCompress, Blake2b]	46
3.3.6	Design of Rig [BlakeExpand, BlakePerm, Blake2b]	47
3.3.7	Parallelization	50
3.3.8	GPU Resistance	50
3.3.9	ASIC/FPGA Resistance	50
3.3.10	Performance Analysis	51
3.3.11	Security Analysis	52
3.3.12	Resistance Against Low Memory Attack	54

3.3.13	Resistance Against Collision Attack	58
3.3.14	Resistance Against Length Extension Attack	63
3.3.15	Resistance Against Cache-Timing Attack	63
3.3.16	Resistance Against Denial-of-Service Attack	63
3.4	Argon2 [45], PHC Finalist	64
3.4.1	Security analysis of Argon2	64
3.5	Evaluation Criteria of PHC	67
3.5.1	Selection Process	68
3.5.2	Selection Conclusions	70
3.6	Summary	71
4	Cryptographic Module Based Approach for Password Hashing Schemes	72
4.1	Password hashing and Cryptographic Module	73
4.2	General Approach for Cryptographic Module Based Password Hashing Scheme	75
4.3	Security Analysis	75
4.4	Analysis of Submitted PHC designs with respect to Cryptographic Module- Based Approach	76
4.4.1	Categorization of the Schemes	91
4.5	Summary	92
5	Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes	94
5.1	Existing Results on Cryptanalytic Time-Memory Tradeoff	95
5.2	On the Analysis of Memory-Hardness using Graph Pebbling	96
5.2.1	Pebble Game	97
5.2.2	Important Results	98
5.3	Overview of Three Cache-timing Attack Resistant Password Hashing Algorithms	100
5.3.1	Argon2i [45]	101
5.3.2	Catena [88]	102
5.3.3	Rig [59]	103
5.4	Preliminaries	104
5.4.1	Description of Some Directed Acyclic Graphs	105
5.5	Traversing a Dependency Graph to Analyse Tradeoff Penalties	109
5.5.1	Description of the Proposed Technique	110
5.6	Results	114
5.6.1	Re-computation Penalty for Argon2i	117
5.7	Performance Analysis	120
5.8	Summary	122
6	On the Improvement of Password Database Breach Detection Technique	123
6.1	Introduction	124
6.2	Overview of the Honeywords Technique [103]	125
6.2.1	Honeyword Generation	126
6.3	Attacks	127

6.3.1	Denial-of-Service (DoS) Attack	127
6.3.2	Multiple System Intersection considering Output (MSIO) Attack . . .	127
6.3.3	Multiple System Intersection considering Input (MSII) Attack	129
6.4	Existing Honeyword Generation Techniques and Their Limitations	129
6.5	The Proposed Techniques	133
6.5.1	Password Policy	133
6.5.2	Typo-Safety	134
6.5.3	Proposed Honeyword Generation Techniques	134
6.5.4	Alarm Policy	138
6.5.5	Managing Old Passwords	139
6.6	Security Analysis	140
6.6.1	Brute-Force Attack	140
6.6.2	Targeted Password Guessing	140
6.6.3	Denial-of-Service Attack	141
6.6.4	Multiple System Intersection considering Input/Output Attack	142
6.7	Experimental Data	142
6.8	Comparison with Existing Work	145
6.9	Summary	146
7	The Analysis of Universal 2nd Factor (U2F) Protocol	148
7.1	Introduction	149
7.2	Overview of the U2F Protocol	151
7.2.1	U2F Manufacturing Phase	153
7.2.2	U2F Registration Phase	154
7.2.3	U2F Authentication Phase	154
7.3	Existing Security Analysis of U2F Protocol	157
7.3.1	Denial of Service (DoS) Attack	157
7.3.2	Parallel Signing Request attack	159
7.3.3	Phishing attack	160
7.3.4	Man-In-The-Middle (MITM) attack	160
7.4	Possible Side Channel Attacks on U2F Protocol	162
7.4.1	Side Channel Key Recovery attack on Device Secret Key DS_K	162
7.4.2	Side Channel Key Recovery Attack On Attestation Private Key SK_M	164
7.5	Proposed Countermeasures	165
7.5.1	Countermeasure to protect Device Secret Key	165
7.5.2	Countermeasure to minimize the effect of Attestation Private Key Recovery	170
7.5.3	Overview of the U2F protocol with Our Proposed Modifications . . .	172
7.6	Design Rationale	175
7.6.1	Role of <i>Counter</i>	175
7.6.2	Role of ‘ctr’	177
7.6.3	Role of U2F_db	177
7.6.4	Role of S_db	178

7.7	Evaluation	178
7.7.1	Modified Registration Phase	178
7.7.2	Modified Authentication Phase	178
7.8	Limitations of U2F	181
7.9	Possible Side Channel Attacks on Universal Authentication Framework Pro- tocol and Mitigation	181
7.9.1	UAF Manufacturing Phase	182
7.9.2	UAF Registration Phase	182
7.9.3	UAF Authentication Phase	184
7.9.4	Attack on UAF	186
7.10	Summary	186
8	Conclusion	189
8.1	Summary	189
8.2	Future Work	190

List of Figures

2.1	Dictionary Attack	23
2.2	Construction of the function f	25
2.3	Matrix of images under function f	25
3.1	Client independent update	30
3.2	Server relief technique	30
3.3	Blowfish Feistel network with F being the Feistel function.	33
3.4	Overview of PBKDF2	35
3.5	Random access in Scrypt.	36
3.6	Graphical representation of Rig	40
3.7	Function H_2 implemented as function BlakeCompress.	46
3.8	Blake2b compression function	47
3.9	Function BlakeExpand	48
3.10	Function H_2 implemented as function BlakePerm.	49
3.11	Performance of Rig (at different value of n) and Scrypt.	52
3.12	Graphical representation of the calculation for the low memory complexity of Rig	56
3.13	$Rig[H_1, H_2, H_3](x, m_c, n, s)$, where H_2 signifies repetitive use of H_2	59
3.14	Five cases showing collisions for Rig $[H_1, H_2, H_3]$	60
3.15	Setup phase uses m inputs generated from α	61
3.16	H_2 implemented with RF_1 at iteration $i = n$	62
4.1	Cryptographic module used in password hashing	75
4.2	(a) Overview of AntCrypt (b) The cryptographic module implementation . .	77
4.3	(a) Overview Argon (b) The cryptographic module implementation	78
4.4	(a) Overview of battcrypt (b) The cryptographic module implementation . .	79
4.5	(a) Overview of Caten (b) The cryptographic module implementation	80
4.6	(a) Overview of CENTRIFUGE (b) The cryptographic module implementation	80
4.7	(a) Overview of EARWORM (b) The cryptographic module implementation	81
4.8	(a) Overview of Gambit (b) The cryptographic module implementation . . .	81
4.9	(a) Overview of Lanarea DF (b) The cryptographic module implementation .	82
4.10	(a) Overview of Lyra2 (b) The cryptographic module implementation	83
4.11	(a) Overview of Makwa (b) The cryptographic module implementation . . .	84
4.12	(a) Overview of MCS_PHS (b) The cryptographic module implementation . .	85

4.13	(a) Overview of ocrypt (b) The cryptographic module implementation	85
4.14	(a) Overview of Parallel (b) The cryptographic module implementation	86
4.15	(a) Overview of POMELO (b) The cryptographic module implementation	87
4.16	(a) Overview of Pufferfish (b) The cryptographic module implementation	87
4.17	(a) Overview of Rig (b) The cryptographic module implementation	88
4.18	(a) Overview of Schvrch (b) The cryptographic module implementation	89
4.19	(a) Overview of Tortuga (b) The cryptographic module implementation	90
4.20	(a) Overview of TwoCats (b) The cryptographic module implementation	90
4.21	(a) Overview of Yarn (b) The cryptographic module implementation	91
4.22	(a) Overview of yescrypt (b) The cryptographic module implementation	91
5.1	A (8, 2)-Bit-Reversal Permutation Graph	98
5.2	General overview of the design Single-pass Argon2 with p lanes and 4 slices . .	102
5.3	General overview of the design Catena applying the function <i>flap</i>	103
5.4	Overview of the design Rig applying the function G	104
5.5	Example: Bit-Reversal Permutation	105
5.6	Graphs based on different types of edges	106
5.7	(8,2)-Straight Graph	106
5.8	(8,2)-Bit-Reversal Graph	107
5.9	(8,2)-Bit-Reversal-Straight Graph	107
5.10	(8, 1)-Double-Butterfly Graph	108
5.11	(4, 2)-Bit-Reversal Graph	112
5.12	(4, 2)-Bit-Reversal Graph with only memory in the first column.	114
5.13	Re-computation Penalties for Graphs	116
5.14	Argon2i at $t=1$, $m= 8$ blocks	119
5.15	Argon2i at $t=2$, $m= 8$ blocks	119
5.16	Argon2i at $t=3$, $m= 16$ blocks	119
5.17	Execution time vs. memory, faster feeling of memory is better.	121
6.1	Overview of the MSIO attack	128
6.2	Overview of the MSII attack	130
7.1	Overview of the U2F Manufacturing Phase	153
7.2	Overview of the U2F Registration Phase	155
7.3	Instantiation of $f_K(\cdot)$ and $f_K^{-1}(\cdot)$ using AES.	156
7.4	Instantiation of $f_K(\cdot)$ using HMAC.	156
7.5	Overview of the U2F Authentication Phase	158
7.6	Side channel attack points at Registration Phase	163
7.7	The function $g(K_{i-1}, i) = HMAC_{K_{i-1}}(i) = K_i$ where $K_0 = DS_K$	167
7.8	Overview of Kocher's session key generation method with function $g(K_{i-1}, i) = K_i$ and index depth $D = 5$	170
7.9	Overview of the proposed Modified U2F Manufacturing Phase	173
7.10	Overview of the proposed Modified U2F Registration Phase	174

7.11 Overview of the proposed Modified U2F Authentication Phase	176
7.12 Overview of the entities of the UAF protocol	183
7.13 Overview of the UAF Manufacturing Phase	184
7.14 Overview of the UAF Registration Phase	185
7.15 Overview of the UAF Authentication Phase	187

List of Tables

2.1	Significant Password hashing schemes before Password Hashing Competition	20
2.2	Key search Complexity on varied length of passwords	24
2.3	Password cracking speed through the years	24
3.1	Computation/read penalties for the ranking tradeoff attack	67
4.1	Categorization of the algorithms based on the approaches taken for cryptographic module implementation.	92
5.1	Bounds on the amortized energy complexity	100
5.2	Traversal steps for (4,2)-Bit-Reversal Graph	113
5.3	Re-computation Penalties for Graphs with 4 rows ($\lambda = 3$)	116
5.4	Re-computation Penalties for Graphs with 5 rows ($\lambda = 4$)	117
5.5	Re-computation Penalties for Graphs with 6 rows ($\lambda = 5$)	117
5.6	Re-computation Penalties for Double-Butterfly Graph ($\lambda = 1$)	117
5.7	Re-computation Penalties for Argon2i (32KiB) at varying λ (t-cost)	119
6.1	Comparison of Modified-UI honeyword generation techniques	146
6.2	Comparison of honeyword-generation methods	147
7.1	Notations	152

Glossary

ASIC	:	Application Specific Integrated Circuit
AES	:	Advanced Encryption Standard
CAPTCHA	:	Completely Automated Public Turing test to tell Computers and Humans Apart
DAG	:	Directed Acyclic Graph
DES	:	Data Encryption Standard
FIPS	:	Federal Information Processing Standard
FPGA	:	Field Programmable Gate Array
GPU	:	Graphics Processing Unit
IV	:	Initialization Vector
KDF	:	Key Derivation Function
PHC	:	Password Hashing Competition
MITM	:	Meet-in-the-Middle
NIST	:	National Institute of Standards and Technology
OTP	:	One-time Password
PAKE	:	Password Authenticated Key Exchange
PBKDF2	:	Password-Based Key Derivation Function 2
PKI	:	Public Key Infrastructure
$\oplus$	:	XOR operation
PHC	:	Password Hashing Competition
PRNG	:	Pseudorandom Number Generator
PHS	:	Password Hashing Scheme
TMTO	:	Time Memory TradeOff
TLS	:	Transport Layer Security

Chapter 1

Introduction

The word *cryptography* is derived from the Greek words *kryptos*, which means “hidden”, and *graphos*, which means “writing”. It is the practice and study of techniques for secure communication in the presence of third parties called adversaries. The security of a cryptographic construction relies on the secrecy of its key. Therefore any cryptographic solution uses high-entropy secrets (keys) such that it is hard to try all possible keys from the particular key space in reasonable time. When it is possible to try all possible keys in real time, then it is feasible to decrypt any encrypted data. This is called a brute-force attack. In terms of computational security, a cryptosystem is called secure if no attacks on it exist that have time complexities lower than brute force. Today, cryptography is used in securing access to internet banking, secure login to websites, secure payment on shops, protecting the integrity of data online, secure computations on clouds etc. Usernames and passwords are commonly used by people during a log-in process that controls access to protected computer operating systems, mobile phones, automated teller machines (ATMs), etc., where cryptography provides the security to protect the passwords.

The use of passwords has a longer history than the digital era. In ancient Rome, watchwords or passwords were distributed to sentries on a wax tablet [156]. The sentries had to verify the watchwords to prevent infiltration to a protected area by allowing a person or group to pass only if they knew the watchword. However, we consider the time when digital computers introduced passwords to prevent unauthorized access to remote computers. It was the early 1960s when passwords were first used for authentication for MIT’s Compatible Time-Sharing System [126]. Typically, a password is a secret word or string of characters which is used by a principal to prove her identity as an authentic user to gain access to a resource. Being secret, passwords cannot be revealed to other users of the same system. In order to ensure the confidentiality of the password, the use of cryptographic technique was realized in late 1970’s by deploying a DES-based one-way encryption function, called *crypt* [1] as a *password hashing* algorithm.

Online authentication to access resources are predominated by the use of passwords. There exist several standardised authentication methods. The three major categories of authentication based on the user input are as follows.

- Something the user knows (e.g., password, Personal Identification Number (PIN)).
- Something the user has (e.g., mobile phone, cryptographic key).
- Something the user is (e.g., fingerprint).

The predominant use of passwords is due to the ease of their deployment as no additional devices or tools are required.

In general, passwords are human generated secrets and required to be a memorized entity. This leads to the limitation of passwords being a low entropy [13] secret as humans usually prefer short and easy to guess values [141]. Therefore, password-based constructions are designed assuming low-entropy secrets that are drawn from a space which is not exponential in input length¹ (dictionary). Hence, it is feasible for an algorithm to try all possible secrets (passwords) in a reasonable time. This is called dictionary attack [126] and it is inherent to any password-based algorithm. A dictionary attack is, therefore, one of the main challenges in password-based cryptosystems compared to general cryptographic constructions that are based on high-entropy secrets. It is ever challenging to design a password-based cryptosystem secure against this attack. Password-based cryptosystems broadly cover two areas - 1) Password-based authentication, e.g., password hashing schemes and 2) Password-based encryption specifically used in password based authenticated key exchange (PAKE) protocols. The PAKE protocols allow two or more interacting parties to establish cryptographic keys based on their knowledge of some password. PAKE protocols commonly use password based encryption where key is derived from the password and are therefore susceptible to dictionary attack.

There exists extensive research on replacing passwords [99]. However, most longer-run proposals still rely on secret knowledge at some level [49]. Despite increasing efforts to move to supposedly more secure alternatives, password-based authentication is likely to stay for the foreseeable future [131, 49] due to user experience and convenience. Even the security expert Bruce Schneier states “As insecure as passwords generally are, they’re not going away anytime soon” in an article on “Choosing a Secure Password” which appeared in *Crypto-Gram* in 2014 [148].

In view of the advancements in technology and persistent use of password-based authentication, it is of utmost importance to understand and analyze the existing designs and protocols with rigour so that security in terms of access control and privacy is maintained. Hence, the need to study, design and improve the existing password-based solutions for securing online authentication. In this report, we concentrate on the design and analysis of such password-based authentication systems.

¹It means an algorithm can efficiently list all possible passwords in time polynomial to the length of the input

1.1 Motivation

The use of passwords in cryptography, usually being low in entropy, is delicate because of the possibility of launching offline dictionary attack as mentioned above. Even insecure is the storage of plain-text passwords. However, it was first realized in 1966 when access to the password storage of Multics system was accidentally granted to all its users [126]. Leakage of a password means one can use the compromised password to pretend the identity of an authentic user. However, to protect the password, the use of cryptography was realized even later, in late 1970's. Since the first reported use of cryptographic algorithm crypt [1], it has become the minimum standard to store the hash of a password to maintain secure password-based user authentication. The leakage of plain text password database also introduces the general tendency among attackers to try to guess passwords.

The efficiency of password cracking depends on the existing computing power and the ability of the attacker to efficiently guess passwords. The real distribution of passwords is difficult to model. However, it is easy to list down all possible passwords with high accuracy by analysing human tendency while choosing passwords. Hence, it is easy to list most likely passwords in a reasonable time. The entropy of the password is typically low. To increase the entropy additional use of salt came into existence in late 1970's [126]. A salt is a random input which is stored together with the password hash. It enables a password hashing algorithm to derive different password hashes from a single password.

According to Moore's Law [144], the number of transistors on integrated circuits doubles approximately every two years. This has indeed been the case over the history of computing hardware. Following this law, the hardware is becoming more and more powerful with time. Using such efficient hardware, an adversary with multiple computing units can easily try multiple different passwords in parallel. Therefore, it happens to be the most prominent threat for existing password hashing schemes. Consequently, there is a need to raise the cost of brute force attack by controlling the performance of the massively parallel hardware available. One possible approach is to slow down password hash computation with the scope of very little parallelism in the design. One way to achieve this is by use of a 'Sequential memory-hard' algorithm, a term first introduced with the design of 'Scrypt' [132], a password hashing scheme. The main design principle of Scrypt is that it asymptotically uses almost as many memory locations as it uses operations to slow down the password-hash computation. Memory is *relatively* expensive, so, a typical Graphics Processing Unit (GPU) or other cheap massively-parallel hardware with lots of cores can only have a limited amount of memory for each single core. Hence an attacker with access to such hardware will still not be able to utilize all the available processing cores due to the lack of sufficient memory and will be forced to have an (almost) sequential implementation of the password hashing scheme.

Despite the dominance of password-based authentication system in the digital world, the study of password-based authentication could not attract the significant attention of cryptographers before the Password Hashing Competition (PHC) [16]. The Password Hashing Competition, PHC, was an effort by the independent panel of cryptographers and was held from 2013 to 2015. The main objective of the competition was to make the password hashing algorithm resource consuming which could stop the adversary from pre-computing the

dictionary and hence alleviate the existing problems of the offline dictionary attack.

The significant constructions before PHC for password hashing are PBKDF2 [160], Bcrypt [137] and Scrypt [132]. All of these do not satisfy most of the necessary requirements mentioned at the competition page [16]. PBKDF2 (NIST standard) consumes very less memory as it was mainly designed to derive keys from a seed (password). Bcrypt uses fixed memory (4KB) for its implementation. Scrypt is not simple (different internal modules) and not flexible (time and memory parameters are dependent) and susceptible to cache timing attack [87]. The frequent reporting of password database leakage in real-world and the realization of the lack of dedicated designs for password hashing motivates us to work on the secure design of password hashing scheme. Therefore, we present a password hashing scheme *Rig* which satisfies all existing requirements mentioned in [16] and our design is recognized as one of the fastest and secure designs among the submissions of PHC [15].

Most of the submissions of Password Hashing Competition [16] claim memory-hardness. However, there exists no easy way to verify the memory hardness claim of the designs following their algorithmic description. We present an easy technique to analyze the memory hardness of those algorithms whose execution can be expressed as a Directed Acyclic Graph (DAG). Most of the algorithms submitted to the PHC have multiple variants and each can be represented as a different DAG. We give a generic algorithm to traverse the DAG which allows varying the memory storage and computes the increased algorithmic runtime for different trade-offs (varying memory and time) options.

As already discussed, password database breach is a common practice amongst hackers. However, only a few of these breaches are detected and even fewer are reported. Therefore, a strong breach detection mechanism is another significant requirement towards the protection of password-based authentication system. In 2013, Juels et al. [103] described a method, which they termed as “honeyword technique”, to detect password database breach. The main idea behind the honeyword proposal is to introduce confusion inside the password database by adding a list of fake passwords along with the correct password. This approach of using fake passwords can help in detecting password database breaches by identifying any login attempt with one of the fake passwords. This would discourage the adversary to mount dictionary attack even after compromising the database. A detailed analysis of this ‘Honeyword’ proposal is required to justify its practicality and to make it more acceptable. This motivates our work on the limitations of existing honeyword generation techniques. We present a new attack model and propose new and more practical honeyword generation techniques.

Attackers are becoming more and more powerful. Therefore, a password-based authentication system is not only suffering from the offline attack but also from powerful online attacks that are becoming more severe day by day. The above mentioned techniques or approaches are not sufficient to prevent online attacks, such as phishing [139], Man-In-The-Middle (MITM) [65], etc. Therefore the requirement of augmenting a second factor to strengthen the simple password-based authentication is a recent trend. Few significant second factor solutions such as Time-based One-Time Password (TOTP) [10] and Short Message System (SMS) [21] are easy targets for phishing attack or MITM attack which in

turn affect user privacy severely and introduce major loss [122]. Frequent cases of phishing and monetary theft due to security flaws in password based solution have been reported. Therefore it is of utmost importance to prevent/mitigate the attacks on passwords to provide secure password-based authentication and to analyse if second factor augmentations increase the entropy of the overall process. The second factor solution Universal 2nd Factor (U2F) protocol is proposed by Fast IDentity Online (FIDO) alliance in 2014. It has been introduced as a strong augmentation that can overcome all known attacks currently faced in practice. Considering the claim of U2F developers that U2F is a significant contribution towards strengthening simple password-based authentication, a thorough third-party analysis of U2F solution is required to verify their assertion. This motivates our work on the analysis of U2F protocol. In this work, we present the possible side channel attacks on U2F and to the best of our knowledge we are the first to present a clear and comprehensive literature that describes the U2F protocol including detailed analysis and description of all cryptographic operations.

Through our work, we attempt to solve the problems with existing password-based authentication system with the aim to understand the extent of research over password-based authentication schemes and identify research directions for future work.

1.2 Thesis Organization

In this thesis, we present improvements on the security of the existing password-based authentication systems. We provide a secure password hashing design and present the result of cryptanalysis on some password hashing algorithms. We analyse the security of the best known second factor augmentation called the U2F protocol. Our main results are presented in Chapters 3, 4, 5, 6 and 7. Each chapter provides literature review, followed by our results and conclusions. The thesis outline is as follows:

- In Chapter 2, we first present an overview of the history of passwords and password hashing algorithm. We review a number of currently known online and offline attacks on password-based authentication system that will facilitate the understanding of the work discussed in subsequent chapters.
- In Chapter 3, we present our password hashing design *Rig*. It is a simple, secure and flexible design which provides all requirements mentioned at PHC [16]. We compare the performance of *Rig* with other significant password hashing constructions and discuss the security analysis of *Rig* in details.
- In Chapter 4, we provide a cryptographic module based approach for password hashing. It helps to enhance the security of the existing password-based authentication framework. We also discuss the feasibility of the approach for other submissions of PHC. First, we describe each of the algorithms in brief. Then we show how the algorithm supports the use of the cryptographic module.

- In Chapter 5, we present a cryptanalytic technique for password hashing. We present a simple technique to analyze Time-Memory TradeOff (TMTO) for any password hashing schemes which can be represented as a directed acyclic graph (DAG). The nodes of the DAG correspond to the storage required by the algorithm and the edges correspond to the flow of the execution. Our proposed technique provides expected run-times at varied levels of available storage for the DAG. Our analysis shows that Argon2i-Version 1.2.1 (PHC winner) is not providing expected memory hardness which is also highlighted in a recent work by Corrigan-Gibbs et al..
- In Chapter 6, we present our work on the improvement of existing honeyword technique, a method for detecting password database breach. In this work, we analyse the limitations of existing honeyword generation techniques. We propose a new attack model and call it Multiple System Intersection attack considering Input. We also propose new and practical honeyword generation techniques.
- In Chapter 7, we analyse the U2F protocol which is a significant contribution towards strengthening simple password-based authentication. In this work, we show why the U2F protocol is not secure against side channel attacks. We then present a new variant of the U2F protocol that has improved security guarantees. We also explain the protocol Universal Authentication Framework (UAF) and show how the side channel attacks of U2F protocol and the corresponding proposed countermeasures are similarly applicable to UAF.
- Finally in Chapter 8, we conclude the thesis by discussing the results and giving some future directions of research.

1.3 Contributions

The list of publications and submissions presented on page IV, [59, 58, 60, 26, 61] form the basis of this thesis. In the joint works, the author of the thesis has played a leading role in obtaining the results reported in this thesis. The main results are as follows:

- Chapter 3 presents our proposed password hashing design *Rig* which is the best-known algorithm in terms of speed in the cache-timing attack resistant category. The results include:
 1. Detailed description of the construction *Rig* along with the design rationale.
 2. Performance analysis comparing with the best-known password hashing algorithms which show *Rig* as the fastest algorithm.
 3. The security analysis considering the memory hardness and the proof of collision resistance of *Rig*.
- Chapter 4 investigates the efficient use of the cryptographic module for password hashing schemes. Our results include:

1. The efficient use of the cryptographic module to enhance the security of the existing password-based authentication framework.
 2. The feasibility of the proposed approach considering the submissions of PHC [16]. It first includes the brief description of the algorithms. Then shows how the algorithms support the use of cryptographic module for enhancing the existing security of password hashing.
 3. The security analysis of password hashing algorithms incorporating the cryptographic module.
- Chapter 5 presents a cryptanalytic technique for password hashing. Our results include:
 1. A simple technique to analyze Time-Memory TradeOff (TMTO) for any password hashing schemes which can be represented as a directed acyclic graph (DAG). The nodes of the DAG correspond to the storage required by the algorithm and the edges correspond to the flow of the execution. Our proposed technique provides expected run-times at varied levels of available storage for the DAG.
 2. Most of the algorithms submitted to the PHC have multiple variants and each can be represented as a different DAG. We experimentally show the TMTO of the algorithms representing the DAG with specific parameters. Specifically, we give a generic algorithm to traverse the DAG which allows to vary the memory storage and computes the increased algorithmic runtime for different tradeoffs (varying memory and time) options.
 3. We apply the proposed technique on three cache-timing attack resistant algorithms, namely, Argon2-Version 1.2.1 (the PHC winner), Catena-Version 3.2 and Rig-Version 2 to obtain TMTO values for various combinations of options in these algorithms.
 4. We analyze these PHS for performance under various settings of time and memory complexities. Our experimental results show (i) simple DAGs for PHS are efficient but not memory hard, (ii) complex DAGs for PHS are memory hard but less efficient, and (iii) combination of two simple graphs in the representation of a DAG for PHS achieves both memory hardness and efficiency.
 5. Our analysis also shows that Argon2i-Version 1.2.1 is not memory hard.
 - Chapter 6 investigates the effectiveness of honeyword technique and presents three new and more practical honeyword generation techniques. The results include:
 1. A new attack model called ‘Multiple System Intersection attack considering Input’ and we show how the ‘Paired Distance Protocol’ defined in [57] is not secure against this attack model.
 2. We propose efficient and practical honeyword generation techniques that can generate honeywords (fake passwords) indistinguishable from real passwords.

3. We also suggest a typo-safety measure, to avoid false detection when a legitimate user enters a honeyword because of a typing error. We use ‘Levenshtein distance’ [93] to ensure that a minimum safe distance is maintained between the password and honeywords.
 4. In our experiment, we use publicly available password databases leaked by hackers. Using experiments, we show that our proposed honeyword generation techniques are perfectly flat.
 5. To validate our claim, we compare our proposed techniques with the existing honeyword generation techniques on various parameters and provide the comparisons.
- Chapter 7 presents security analysis of U2F protocol. The results include:
 1. The U2F protocol is not secure against side channel attacks. We propose a countermeasure to fix this issue. Our proposed solution is based on the rekeying technique that derives session keys from the device secret key. Use of different session keys instead of the fixed device secret key helps in keeping the device secret key confidential.
 2. The signature function that uses the attestation private key also leaks information that can be captured through side channel analysis. Usually, in public key infrastructure (PKI) [154], the verification of the digital signature is satisfied by the use of the certificate. Intuitively, it may seem that the use of session key will prevent the side channel attack on attestation private key as well. However, getting a valid certificate for each session key generated from attestation private key is not practical in the asymmetric cryptosystem. Hence, it is difficult to prevent side channel attacks using existing solutions for signature. Therefore, we recommend a minor modification to the protocol to mitigate the attack.
 3. Incorporating our proposed countermeasure and recommended modification, we then present a new variant of the existing U2F protocol that provides strong security guarantees in the current existing scheme.
 4. There is a lack of clear and comprehensive literature that describes the U2F protocol with detailed security analysis. We present a detailed description of the U2F protocol along with its security analysis. We also try to fill the gap in the description of the protocol by providing a clear explanation of all the cryptographic operations.
 5. We also explain in brief how the side channel attacks on U2F protocol and the corresponding proposed countermeasures are similarly applicable to Universal Authentication Framework (UAF) protocol.

1.4 Statement on Research Ethics

This dissertation uses multiple databases containing real user passwords which were compromised by hackers and were made publicly available. Passwords are considered as private data and hence we keep the databases used in this study confidential. Respecting ethical and legal issues in the distribution of user private information, we can provide the list to any legitimate researcher who agrees to follow the moral and legal standards of handling private information. The databases can be obtained by contacting the author of the thesis, after agreeing to standard ethical practices in handling the same.

Chapter 2

Background

A cryptosystem is a general term referring to a set of cryptographic primitives used to provide information security services. In symmetric cryptography, each pair of communicating parties requires the knowledge of one common key which is kept secret from other unauthorized parties. Similarly, a password based cryptosystem requires the knowledge of a single key (password) and hence it can also be treated under the symmetric cryptosystem. Password-based cryptography encompasses the design and analysis of password-based cryptosystems which include password-based authentication system and password-based authenticated key exchange protocols. This thesis primarily focuses on the design and analysis of password-based authentication systems. This chapter briefly covers some background that is essential to get a full understanding of this work.

2.1 Preliminaries

The important definitions to better understand the area of password-based authentication are defined below.

2.1.1 Symmetric Cryptosystem

Following are the definitions of few key terms of symmetric cryptosystem that are useful in the understanding of this thesis contribution.

- **Block Cipher:** A block cipher is a transformation

$$E : \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n.$$

The function E takes two inputs - a k -bit string termed as the key and an n -bit string termed as the input message or the plaintext and returns an n -bit output which is called the ciphertext. The key size k and the block size n are the parameters associated with a block cipher. The general representation of a block cipher is $E_K: \{0, 1\}^n \rightarrow \{0, 1\}^n$ for each key $K \in \{0, 1\}^k$ or simply $E_K(\cdot)$ represents the encryption. The inverse of

$E_K(\cdot)$ is represented by $E_K^{-1}(\cdot)$ which denotes decryption. For any block cipher and key K , it is required that the function E_K (as well as E_K^{-1}) be a *permutation* on $\{0, 1\}^n$. Currently, the widely used block cipher is Advanced Encryption Standard (AES) [72] which supports block size 128-bits and the key lengths 128, 192 or 256-bits.

Shannon, in his seminal work [149] defines the concept of *perfect secrecy* which says that a cipher is called perfectly secure if the ciphertext does not reveal the adversary any additional information about the plaintext. In this work, Shannon also proves that to achieve perfect secrecy, the ciphers must have a keyspace that is at least as large as the message space. Achieving perfect secrecy is impractical in practice as large amounts of data need to be encrypted. Therefore, the security of block ciphers is defined under a weaker notion of security termed as *computational security* [107]. It needs to give a precise mathematical formulation to show that the given problem cannot be solved in reasonable time.

- **Hash Function:** In general, hash functions are the functions which take arbitrary-length strings and compress them into shorter strings. A cryptographic hash function is a function H that maps an input of arbitrary bit-length M , called message, to an output of fixed bit-length m , called image or hash, i.e., $H : \{0, 1\}^* \rightarrow \{0, 1\}^m$. However, not every such function H is a good cryptographic hash function. There are three main properties which a good cryptographic hash function has to preserve.

1. Collision resistance: It should be computationally infeasible to find any two different messages M, M^* such that $H(M) = H(M^*)$. The brute-force complexity of collision attack is $2^{m/2}$.
2. Preimage resistance: Given $y \in \{0, 1\}^m$, it is computationally infeasible to find a message M such that $H(M) = y$. The brute-force complexity of preimage attack is 2^m .
3. Second preimage resistance: Given a message M , it should be computationally infeasible to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$. The brute-force complexity of second preimage attack is 2^m .

Any hash function that is collision resistant is second preimage resistant. Likewise, any hash function that is second preimage resistant is also preimage resistant. The “Birthday” attack shows a minimal output length necessary for a hash function to potentially be secure against adversaries of a certain time complexity. It shows that if the output length of a hash function is m bits then one can find a collision with high probability in $O(2^{m/2})$ time. Therefore, for the hash function to resist collision-finding attacks that run in time T , the output length of the hash function needs to be at least $2 \log T$ bits.

- **Message Authentication Code (MAC):** The cryptographic construction message authentication code (MAC) aims to prevent an adversary from modifying a message sent by one party to another, without the parties detecting that a modification has

been made [107]. In this setting, the communicating parties share the same secret key K . A MAC consists of three algorithms (Gen, Mac, Vrfy). The algorithm Gen generates a secret key. The algorithm Mac generates MAC tags; i.e., it maps a key K and a message m to a tag t which is represented as $\text{Mac}_K(m)$. Finally, the algorithm Vrfy receives a key K , a message m , and a tag t which is represented as $\text{Vrfy}_K(m, t)$, and outputs either 1 (meaning valid) or 0 (meaning invalid). The formal definition from [107] is as follows.

Definition: A message authentication code or MAC is a tuple of probabilistic polynomial-time algorithms (Gen, Mac, Vrfy) fulfilling the following:

1. The algorithm Gen outputs a uniformly distributed key K of length k ; $K \xleftarrow{\$} \text{Gen}(1^k)$.
2. The algorithm Mac receives for input some $K \in \{0, 1\}^k$ and $m \in \{0, 1\}^*$ and outputs some $t \in \{0, 1\}^*$. The value t is called the MAC tag.
3. The algorithm Vrfy receives for input some $K \in \{0, 1\}^k$ and $m \in \{0, 1\}^*$ and $t \in \{0, 1\}^*$, and outputs a bit $b \in \{0, 1\}$.
4. For every k , every $K \in \{0, 1\}^k$ and every $m \in \{0, 1\}^*$ it holds that $\text{Vrfy}_K(m, \text{Mac}_K(m)) = 1$.

A MAC scheme is considered as broken if any adversary can generate any new message together with a valid MAC tag. This level of security is called existential unforgeability against a chosen message attack. The “existential unforgeability” refers to the fact that the adversary should not be able to generate a valid MAC tag on any message, and the “chosen message attack” refers to the fact that the adversary is able to obtain MAC tags on any messages it wishes during its attack.

- **Hash-based Message Authentication Code (HMAC):** In cryptography, a keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. It may be used to simultaneously verify both the data integrity and the authentication of a message. The HMAC construction consisting three algorithms (Gen, Mac, Vrfy) can be defined as follows [107].

1. $\text{Gen}(1^k)$: The algorithm Gen outputs a uniformly distributed key K of length k ; $K \xleftarrow{\$} \text{Gen}(1^k)$.
2. $\text{Mac}_K(m)$: on input $K \in \{0, 1\}^k$ and $m \in \{0, 1\}^*$, compute

$$\text{HMAC}_K(m) = H_{IV}(K \oplus \text{opad} \parallel H_{IV}(K \oplus \text{ipad} \parallel m))$$

and output the result.

3. $\text{Vrfy}_K(m, t)$: output 1 if $t = \text{Mac}_K(m)$.

The HMAC construction defined above uses two fixed constants *opad* and *ipad*. These are two strings of length n , the length of a single block of the input to hash function H . The initialization vector (IV) is an arbitrary fixed value. The security of HMAC depends on the size of the secret key that is used. It is less affected by collisions of the underlying hashing algorithms. Therefore, HMAC-MD5 does not suffer from the same weaknesses that have been found in MD5 [114]. HMAC is an industry standard and is widely used in practice. It is highly efficient and easy to implement, and is supported by a proof of security (based on assumptions that are believed to hold for all hash functions in practice that are considered collision resistant) [107].

2.1.2 Password-based Authentication System

Following are the definitions of few key terms that help to understand the extent of research over password-based authentication systems.

- **Salt:** A salt for password hashing refers to an additional random non-secret input for the algorithm along with the secret password and is stored together with the password hash. The use of salt enables to derive different password hashes from a single password. Ideally, it should be different for each user. Use of salts prevents the usage of rainbow tables (discussed later) because an attacker does not know in advance the salt used. However brute-force attacks remain as fast as with unsalted hash functions [34].
- **Key Derivation Function:** In cryptography, a key derivation function (or KDF) derives one or more secret keys from a secret value such as a master key, a password, or a passphrase using a pseudo-random function.

The key derivation methods are provided by National Institute of Standards and Technology in the Special Publication 800-57 [36]. It defines that a key-derivation function takes a secret value, together with other information, as the input and outputs the required key(s). The derivation method shall be non-reversible (i.e., a one-way function) so that the secret value cannot be determined from the derived keys. In addition, it shall not be possible to determine a derived key from other derived keys. The strength of a derived key is no greater than the strength of the derivation algorithm and the secret value from which the key is derived.

The case of key derivation when input secret is password is discussed below [36].

- Keys derived from a password: A user-generated password, by its very nature, is less random (i.e., has lower entropy) than is required for a cryptographic key; that is, the number of passwords that are likely to be used to derive a key is significantly smaller than the number of keys that are possible for a given key size. In order to increase the difficulty of exhaustively searching the likely passwords, a key-derivation function is iterated a large number of times. The key is derived using a password, entity ID, and other known information as input to the key-derivation function. The security of the derived key depends upon the security of

the password and the key-derivation process. If the password is known or can be guessed, then the corresponding derived key can be generated. Therefore, keys derived in this manner are likely to be less secure than randomly generated keys or keys derived from a shared secret or key-derivation key.

- **Key Stretching [109]:** A formal definition of key stretching is presented by Bruce Schneier, et.al in 1997. It is a mechanism to convert short s -bit keys into longer keys, such that the complexity required to brute-force search a $(s + t)$ -bit keyspace is the same as the time required to brute-force search a s -bit key stretched by t bits. It answers how hard is the brute-force search after incorporating computationally expensive calculation. The general algorithm of key stretching as documented in [109] is described below.

- Take a low entropy key, K_{short} with entropy s .
- Take a salt.
- Let $F()$ is the function that approximates a random function.
- Required $F()$ should guarantee to take the same amount of work as searching 2^t keys in a brute-force exhaustive keysearch.

$$K_{long} = F^{2^t}(K_{short}, \text{salt}).$$

The above algorithm shows a method of increasing the difficulty of trying all possible values for some low-entropy variable (e.g., password) by a fixed, well-understood amount. The authors also instantiate the algorithm with two different key-stretching schemes, one based on a block cipher and the other on a hash function. The hash-based construction was shown to have the property that a method for speeding up the exhaustive search of the low-entropy variable translates into a significant weakness in the underlying hash function.

Key stretching is no substitute for long keys, but it reduces a short keys resistance to certain attacks.

- **Password Entropy:** In information theory, entropy is a measure of unpredictability i.e., how much information is produced by a process [151]. As explained in [13] on the context of password, entropy is the measure of how hard it is to guess the password, a number which shows the strength of the password. However, there are two fundamentally different things that are measured:

1. How hard it is for someone to guess the password with essentially no knowledge of how the password is created.
2. How hard it is for someone to guess the password if he knew the process used to generate it.

Therefore the term ‘entropy’ refers to both kinds of computations, but both are entirely different. The first process takes a string as input, the latter takes a random

process as input. This random process is determined by the set of all possible outputs and the probability associated with each output. The entropy is computed by the formula provided by Shannon [151].

$$H = - \sum_i P_i \log P_i$$

where P_i is the probability of character number i showing up in a stream of characters of the given ‘script’.

It is difficult to model the process of password generation. Therefore, the measure of the strength of a password through its structure is in common trend. However, the entropy of the password is discussed in the NIST Special Publication (SP) 800-63-2 on Electronic Authentication Guideline [53]. It discusses two scenarios. 1) When passwords are selected from a random process. In that scenario, the Shannon formula directly provides the entropy considering password of certain length characters is chosen at random from an alphabet of fixed characters. 2) When passwords are user chosen, they do not follow a uniform random distribution. They roughly reflect the patterns and character frequency distributions of ordinary English text. Therefore, the use of Shannon’s estimate of the entropy in ordinary English text [150] as the starting point to estimate the entropy of user-selected passwords is suggested in [53] which is followed in many studies.

In cryptography there exist other alternative forms or definitions of entropy, including “guessing entropy” and “min-entropy”. A guessing entropy can be defined as an estimate of the average amount of work required to guess the password of a selected user whereas the min-entropy is a measure of the difficulty of guessing the easiest single password to guess in the population [53].

- **Password Policy** To encourage the use of a strong password and maintaining the security of password-based authentication system, certain rules are imposed on users. Changing the password after every certain time period and use of digits, characters and special characters are among such policies. A study on password policy as explained in [84] shows that the most restrictive password policies do not provide greater security. Sometimes such rules make the password easy to guess. As documented in [147] “The UNC researchers said if people have to change their passwords every 90 days, they tend to use a pattern and they do what we call a transformation”, Cranor explained. “They take their old passwords, they change it in some small way, and they come up with a new password”.

Even it is difficult to impose strict rules to enhance the entropy of user-chosen passwords. However, the user should avoid using dictionary words as passwords, selecting the same password on multiple websites, using commonly used passwords, and passwords correlated to the username. A nice advice to choose a password is provided by the security expert Bruce Schneier in [148]. It says to take a sentence and turn it into a password. For example, “This little piggy went to market” might become

“tlpWENT2m”. Such words are difficult to find in dictionaries and hence it is a safe approach.

- **Significant Hardware** According to Moore’s Law [144], the number of transistors on integrated circuits doubles approximately every two years. This has been the case over the history of computing hardware. Following this law, the hardware is becoming more and more powerful with time. Currently available specialized hardware are important considering the attacks on password-based authentication as the hardware-based attacks introduce a big difference in attack speed. For example, Graphics Processing Unit (GPU) attacks can be at least 100 times faster than Central Processing Unit (CPU) attacks [106].

The important electronic circuit, **Graphics Processing Units** (GPUs), and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. Initially, GPUs were designed for efficient manipulation of computer graphics and image processing with thousands of smaller, more efficient cores for handling multiple tasks simultaneously. These days, GPUs are the major threat for password hashing as it can hash multiple passwords at a time utilizing its highly parallel structure. Hence, it is efficient to mount the offline dictionary attack.

An **Application-Specific Integrated Circuit** (ASIC), is an integrated circuit (IC) which can be customized with memory chips to implement a dedicated design. An ASIC cannot be altered after final design hence the designers need to be certain of their design when it is implemented in ASIC. On the other hand, **Field Programmable Gate Arrays** (FPGAs) are programmable integrated circuits and consist of an array of logic elements together with an interconnected network and memory chips, providing high-performance. A designer can test her design on a FPGA before implementing it on an ASIC.

Both ASIC and FPGAs can be configured to perform password hashing with highly optimized performance. The cost of implementation on FPGA is cheaper than ASICs if the number of units of the hardware required is small. Therefore, one can easily use parallel FPGAs to increase the rate of password guessing. RIVYERA FPGA cluster is an example of a very powerful and cost optimized hardware. It can hash 3,56,352 passwords per second by using PBKDF2 (NIST standard, Password Based Key Derivation Function 2) with SHA-512 and 512-bit derived key length [76]. This high performance is possible on the FPGA because PBKDF2 does not consume high memory for password hashing. Comparing FPGAs with GPUs (Graphics processing units), the authors of [76] provide results of the same implementation on 4 Tesla C2070 GPUs as 1,05,351 passwords per second. ASIC is better than FPGA purely on performance in terms of the number of hashes per second. However, FPGA is preferable when cost is considered with the speed of hashing.

- **Memory-Hard Algorithm:** The term ‘Sequential memory-hard’ for an algorithm was introduced by Percival with the design of ‘Scrypt’ [132], a password hashing

scheme. An algorithm is called memory hard when it uses asymptotically almost as many memory locations as it uses operations. In other words, it requires the construction to consume a fixed amount of memory (RAM) during execution. The operation should take significantly longer time (preferably exponential) if less than this pre-specified and fixed amount of memory is available.

Definition 1 A memory-hard algorithm on a Random Access Machine is an algorithm which uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\epsilon})$ [132].

Following Moore’s law, the speed of hardware is likely to increase by almost a factor of two in less than two years. However, as processor speeds continue to outpace memory speeds [96], the gap between processor and memory performance increases by about 50 % per year [55]. Thus, there is a need to minimize the effects of such high performance hardware. Hence, we need a password hashing algorithm which consumes comparatively large memory to prevent parallel implementation. A typical GPU or other cheap massively-parallel hardware with lots of cores can only have a limited amount of memory for each single core. Hence an attacker with access to such hardware will still not be able to utilize all the available processing cores due to the lack of sufficient memory and will be forced to have an (almost) sequential implementation of the password hashing scheme. The following example from [14] nicely explains the idea. The NVidia Tesla K20X GPU has 2688 cores operating at 732 MHz, 6 GB of shared DRAM and a bandwidth of 250 GB/s. Consider a password hashing scheme (PHS) which takes 2 ms to run, consuming 0.5 MB of memory. An adversary with NVidia Tesla K20X GPU can test 2688 passwords every two ms. Resulting in 1,344,000 passwords tested/second, or 4,838,400,000 $\equiv 2^{32.17}$ per hour. If the PHS requires 20 MB of the DRAM, the maximum number of cores used simultaneously becomes 300, 11% of the total available and hence significantly decreases the performance of the GPU.

2.2 History of Computer Passwords

The use of secret words or passwords to authenticate humans has a longer history than the digital era. As documented in [156], at the ancient Roman time, the watchwords or passwords were distributed to sentries on a wax tablet. The sentries had to verify the watchwords to prevent infiltration to a protected area by allowing a person or group to pass if they knew the watchword. However, we consider the time when digital computers introduced passwords to prevent unauthorized access to remote computers.

The history of password-based authentication is nicely incorporated in [126]. The paper describes the origin of password security on a remotely accessed time-sharing system. It states that in early 1960’s when Unix system became vulnerable to remote login by unauthorized users, password-based authentication system was introduced in Massachusetts Institute of Technology (MIT) for the Compatible Time-Sharing System (CTSS). The turned off printing mechanism after typing the password was also adapted but passwords were stored in their

actual plaintext form at the `/etc/passwd` file. The need to secure passwords was realised when in 1966, a system administrator on the CTSS system at MIT was editing the password file and due to some software design error, the password file was printed on every terminal when it was logged in. Leakage of the password means one can use the compromised password to pretend the authentic user identity. After the incident, it was clear that password should not be stored as its' clear text form. Then the system administrator adapted some mathematical approach which was not cryptographic but tried to make it one-way encrypted by taking the square of each password and ANDed with a mask to discard some bits [126]. This technique was then easily broken by Air Force Tiger team. Since then, password cracking becomes a common practice among attackers and use of cryptography for the password protection was also realized. Based on the Multics experience, Morris and Thompson realized that the input passwords are easy to compromise and hence introduced the use of *salt* for UNIX system [126]. In 1979, they show how easy to crack over 80% of accounts in the first published *dictionary attack* by exploiting the weakness in user-chosen passwords. With the emerging password-cracking techniques and launch of the Morris worm [108] in 1988, the need of research into replacing or improving passwords for human-computer authentication was motivated. Since then, the cryptographic efforts are focused on the secure design of password hashing schemes.

2.3 Password-Based Authentication

A password is a secret word or string of characters which is used by a principal to prove her identity as an authentic user. For our convenience, in this thesis, we consider the password as a user-chosen secret which includes character strings chosen from a dictionary consisting of alpha-numeric characters a-z, A-Z, 0-9 and special characters like \$,%, etc. Therefore, passphrase or PIN can also be considered as a password. A Passphrase is a character string, usually longer than passwords, generate their strength (entropy) from the length and are generally easier to memories. PINs are the short passwords, chosen from the numbers 0-9. They are mainly used to secure access to the actual authentication token like a credit card or other smart cards. There exist several other forms of password as well such as Graphical Passwords like the use of CAPTCHA, Completely Automated Public Turing test to tell Computers and Humans Apart.

A typical password-based authentication follows the two generic steps of web authentication: 1) Registration, where the client registers a password corresponding to her username to the server database and 2) Authentication, where the user proves her knowledge of the pre-registered password to the server. Therefore, password-based authentication is the process of recognizing a user's identity based on the proof of knowledge of the pre-shared password. It has been widely deployed in practice due to its simplicity and efficiency. However, this method of authentication has an inherent weakness as the entropy of user-chosen passwords are low in general which are particularly subject to dictionary attacks. To protect the password and to prevent the dictionary attack, passwords are stored in their one-way transformed form. The security is then defined based on the cryptographic algorithm called

password hashing algorithm which is used for the one-way transformation of the passwords. We explain password hashing ahead.

One-Time password (OTP) A one-time password (OTP) is a password that is valid for only one login session or transaction, on a computer system or other digital device [128]. As OTPs are used only once, it does not require to be memorised like passwords. They are mainly chosen from the numeric dictionary 0-9. OTPs are popularly used as a second factor¹ authentication in addition to the password. Transaction authentication numbers (TANs) are special OTPs mainly used to authorise single financial transactions in online banking applications.

2.3.1 History of Password Hashing

In the late 1970s it was realized that password should be stored in one-way encrypted form by applying some cryptographic algorithm. The block cipher Data Encryption Standard (DES) [74] was the most popular algorithm of that time. Therefore, Morris and Thompson development the UNIX crypt() function based on DES cipher and used 12-bit random salt for each user, with passwords limited to 8 ASCII characters. It appeared in Version 3 Unix; however, Crypt was used to store passwords from 6th Edition (1974). Specifically, the encryption scheme is referred to as Crypt(1) and the hash function as Crypt(3), which sometimes confuses the reader. The Crypt(3) uses 56-bit DES to encode the password. The algorithm can be described as: Take the first 8 characters of the password (7 bits each), use this as the DES key to encrypt an all bits zero block 25 times using a 12-bit salt. The result is then base 64 encoded for storage along with the salt and stored in the password file. After that, Crypt(3) started supporting many other algorithms. However, this scheme is not secure in today's scenario as the keyspace is small (56-bits) and modern computing devices make the computation highly efficient which is not required for password hashing schemes.

It is important to note, the use the password as the key instead of encrypting the password with a given key which is known to the system. Here the problem is, if the system encrypts the password with a key, then it needs to consider the secure storage of the key. Storing the key on the system means attacker needs to target to compromise the key to decrypt the password. Basically, the property of one-wayness was required for password hashing. As time progressed, more research was done on one-way hash functions. The first designs of cryptographic hash functions were reported in late 1970s [124] and more proposals emerged in the 1980s. The hash functions MD5 and SHA-1 were deployed in an ever increasing number of applications, resulting in the name "Swiss army knives" of cryptography [136]. Hash functions were also used to hash the passwords. Typically, the repeated invocations of a cryptographic hash function for password hashing becomes the common trend which helps to slow down each password hash computation. Technically, it is called key stretching.

¹Two-factor authentication is a method of confirming a user's claimed identity by utilizing a combination of two different components while in single-factor authentication, the user provides only one factor - typically a password [70].

Table 2.1: Significant Password hashing schemes before Password Hashing Competition

PHS	Iteration Count	Memory	Issues
Crypt, late 70's	25	small	too fast
MD5Crypt, 1994	1,000	small	too fast
Sha512Crypt	1000 - 999,999	small	small memory
NTLMv1, 1990s	1	small	too fast
bcrypt, 1999	$2^4 - 2^{99}$	4,168 bytes	constant memory
PBKDF2, 2000	1 - ∞	small	small memory
scrypt, 2009	1 - ∞	flexible, large	cache-timing attack

The significant hash based password hashing constructions which follow the concept of key stretching are MD5Crypt and Sha512Crypt as listed in Table 2.1. The common structure followed by the constructions are given below.

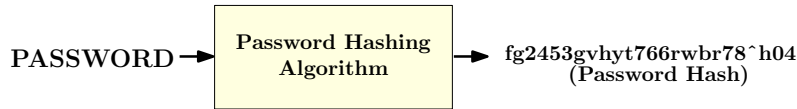
$$f(\text{Password}, \text{Salt}, \text{IterationCount}) \rightarrow \text{PasswordHash}$$

where function f represents the password hashing algorithm.

As already explained, the most common threat for password hashing algorithms is dictionary attack. With time, hardware becomes more efficient and cheaper, hence key-stretching is not enough to prevent the dictionary attack. However, there exists a gap between the performance of the processor and memory and it increases by about 50 % per year [55]. Therefore, memory is relatively expensive component. Thus, it is an effective measure to control the use of high performing hardware in the pre-computation of dictionary. To claim computationally expensive algorithms, memory hard designs for password hashing emerge. However, there is very less technique exists to verify the memory-hardness property. With advanced computational efficiency new attack emerges like GPU attack, cache-timing attack etc and presented in the literature as well as in practice. These attacks, in turn, led to the development and evolution of new criteria for secure password hashing design, e.g., required the size of the salt and the output of the password hash, computationally hard functions, memory hard designs, input independent memory access etc. These new techniques open new attacks, new design criteria and are the reason behind the Password Hashing Competition(PHC).

2.3.1.1 Password Hashing

Password hashing is the technique to one-way transform the password to make it implausible for an attacker to determine the original password even if the attacker gets the access to the password database. The one-way transformed string is called the ‘hashed’ password or password hash.



When hashing passwords, the two most important considerations are the computational expense with a large number of resource consumption, e.g., a memory unit, and the use of unique salt per user. The more resource consuming and computationally expensive the hashing algorithm, the longer it will take to brute force its output. Mainly the goal is to prevent the attempt of pre-computation of the dictionary of common password hashes by the attacker. It should also have the following properties:

- **Pre-image resistance:** Given a password hash, it should be hard to find the input password from it.
- **Second pre-image resistance:** With any given password, it should be hard to find a different password that hashes to the same value.
- **Collision resistance:** It should be hard to find two different passwords that hash to the same password hash.

2.4 Password Cracking

There are mainly two categories of password cracking techniques: online attacks and offline attacks.

2.4.1 Online Attack

In this category of attack an active adversary tries to capture the user password or guess the password or somehow compromise the password-based authentication system in real time. Following are some significant real-world online attacks.

Online Dictionary Attack In this attack we assume that the attacker has a database of possible passwords, called a dictionary. The attacker then tries the password from the dictionary one by one to guess the correct password by interacting with the login server and getting the real time response from the server. It is difficult to launch this online attack due to the following preventive measures commonly deployed by the authentication server.

Account locking: To prevent the attacker from checking sufficiently many passwords in a reasonable time, the user accounts are locked after some fixed number of unsuccessful login attempts. Such countermeasure discourages the attacker to mount the attack. However, it can enable denial of service attacks against users accounts.

Delayed response: With the intention to discourage the attacker to check sufficiently many passwords in a reasonable time, the server delays its response of acceptance or rejection for a

provided username-password pair. However, an attacker can have multiple parallel sessions to successfully launch the attack.

Phishing Attack The concept of the phishing attack was first introduced in 1987 at Interex conference by Jerry Felix and Chris Hauck [82, 139]. NIST in [115] defines phishing as ‘a technique which refers to use of deceptive computer-based means to trick individuals into disclosing sensitive personal information’. Commonly, it is a form of social engineering that uses email or malicious websites to solicit personal information from an individual or company by posing as a trustworthy organization or entity. For password-based authentication system, the attack usually compromises the user password of a legitimate account by directing the user to a corresponding fake account.

Man-in-the-Middle (MITM) Attack In the MITM attack, the common scenario involves two legitimate endpoints (victims), and a third party (attacker) [65]. The attacker has access to the communication channel between two endpoints and can eavesdrop, insert, modify or simply replay their messages.

A recent work shows an attack on password-based authentication system called “The Password Reset MitM Attack (PRMitM)”, proposed by Gelerntor et.al. [92]. As explained in [92], the basic PRMitM attack works as follows. An attacker website attracts users by providing e.g., some free software. When a legitimate user accesses the website of the attacker, it may ask the user to provide her email address. The attacker immediately accesses the email service provider website of the provided email and initiates a password reset process. The attacker forwards every challenge that he gets from the email service provider to its victim user and forwards the response from the user to the email service provider. That way, the cross-site attacker is actually a man in the middle of a password reset process.

2.4.2 Offline Attack

In this category of attack an adversary tries to capture the user password or guess the password or somehow compromise the password-based authentication system without the need to interact with the system of the user or the server. Offline attacks are difficult to detect by the server or the user. Most of the reported attacks on passwords are the offline attacks [146, 66, 67] and the work of this thesis mainly focuses on this offline attack. Following are some significant real-world offline attacks for password-based authentication systems.

Offline Dictionary Attack It uses a file (dictionary) containing words from the English and other languages, phrases, known popular passwords, sentences from books and other strings that are likely to be used as a password. This dictionary should be well optimized and should not include every string combinations to lose its efficiency. Each word in the file is then hashed using the common password hashing algorithm. With access to the database of password hashes from any compromised server, the pre-computed hash of dictionary is

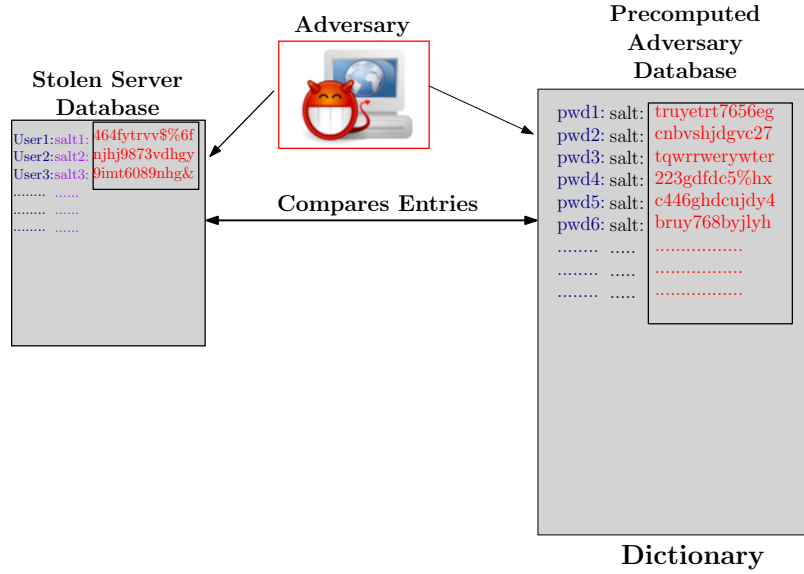


Figure 2.1: Dictionary Attack

compared. When the entries from both the hashes match, the corresponding word are the candidate password.

The first results of dictionary attack was published in 1979 [126] trying the 250,000 words from a dictionary including some common tricks to guess the user password like dictionary with the words spelled backwards, list of first names, last names, street names, city names and the above with initial upper-case letters etc. An experiment with the above collection of passwords was able to crack 86 percent of the available password file. This cryptanalytic approach was called key search as the term ‘dictionary attack’ was not in practice by that time. The key search complexity is shown by tabulating the results in an experiment with M-209 cipher of running time 1.25 milliseconds on a PDP-11/70. The results are listed in Table 2.2 where n is the length of passwords. It shows the feasibility of the dictionary attack with different length of password drawn from different character set. Since late 1970s, dictionary attack is the major threat for password based authentication as user chosen passwords are easy to guess. Therefore, Morris and Thompson introduced the use of salt as a preventive countermeasure afterward [126]. The use of specialized hardwares like GPU, ASIC, FPGA for password cracking is another challenge to prevent dictionary attack. Table 2.3 shows the password cracking speed through the years as documented in [101].

Brute Force Attack For password-based authentication system, a brute-force attack performs the exhaustive search on the password hashes by computing the hash of each and every string combination for a chosen character set and string length. The calculated hashes are compared with the hashes to be recovered until a match is found or the attack is finished. In general, the secure cryptographic systems consider large key spaces to make it impossible to find the key in a reasonable amount of time. As human-chosen passwords follow easy to

Table 2.2: Key search Complexity, taken from [126] where n is the length of passwords

n	26 lower case letters	36 lower case letters & digits	62 alphaneumeric characters	95 printable characters	all 128 ASCII characters
1	30msec	40msec	80msec	120msec	160msec
2	800msec	2sec	5sec	11sec	20sec
3	22sec	58sec	5min	17min	44min
4	10min	35min	5hrs	28hrs	93hrs
5	4hrs	21hrs	318hrs	112days	500days
6	107hrs	760hrs	2.2yrs	29yrs	174yrs

Table 2.3: Password cracking speed through the years. Speed results are reported in millions of base hashes per second (MHz). Cost figures are converted from authors' [101] contemporary estimates into 2012 US dollars using US Bureau of Labor Statistics inflation figures [11] and the XE Universal Currency Converter [12]

year	study	hash	platform	speed (MHz)	cost $\frac{US\$}{MHz}$
1991	Leong & Tham [119]	DES	PC	0.02	-
1991	Leong & Tham [119]	DES	hardware	4.2	\$786.42
1998	Gilmore [94]	DES	hardware	92,666	\$3.75
2006	Mentens et al. [123]	DES	FPGA	10,000	\$0.84
2011	Sprengers [153]	MD5	GPU	880	\$0.39

guess patterns, an exhaustive search over the all allowed character set becomes inefficient. Brute force or exhaustive search only guarantees to find the password not the feasibility in terms of time required. Therefore, a dictionary attack is the attack which is considered while designing of a password hashing scheme.

Time-Memory Tradeoff (TMTO) Attack Cryptanalysis is a searching problem that allows two extremes of exhaustive search and table lookup. Suppose there are N possible solutions to search over, and T , M denote the computation time and the required words of memory respectively. Then exhaustive search needs $T = N$, $M = 1$ i.e., a lot of computing power, and table lookup requires $T = 1$, $M = N$, i.e., a lot of memory words. A good cryptanalytic approach tries to minimize the effort of either of these extremes such as time-memory tradeoff cryptanalytic technique.

Hellman in [97] introduce the idea of Time-Memory Tradeoff to optimize the cryptanalytic effort for a search which includes N possible solutions. Specifically for TMTO, the cryptanalyst tries to optimize the product $T \cdot M$ for a task of N possible solutions where T is the number of operations performed (time), M is the number of words of memory and $T \cdot M = N$. The relative cost of CPU cycles (T) is much lesser than RAM space (M), as a result, most attacks attempt to reduce memory at the cost of increased algorithmic runtime.

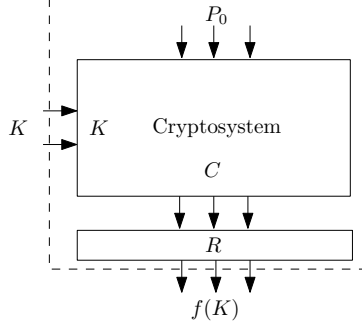


Figure 2.2: Construction of the function f , the image is taken from [97]

The introduced method of [97] considers a cryptosystem having N keys and the method can recover a key in $N^{2/3}$ operations using $N^{2/3}$ words of memory. The typical application of this method is the recovery of a key when the plaintext and the ciphertext are known.

The method: Given a fixed plaintext P_0 and the corresponding ciphertext C_0 , the method tries to find the key $k \in N$ which was used to encipher the P_0 using the cipher S where

$$C_0 = S_k(P_0)$$

As part of the pre-computation, the cryptanalyst chooses m starting points, $SP_1, SP_2, \dots, SP_m$, each an independent random variable drawn uniformly from the key space $K \in \{1, 2, \dots, N\}$. For $1 \leq i \leq m$, let $X_0^i = SP_i$ and the computation $X_i^j = f(X_i^{j-1})$, $1 \leq j \leq t$, where the function f is shown in Fig. 2.2. The ciphertexts are organised in chains whereby only the first (SP_1) and the last element (EP_1) of a chain is stored in memory to reduce the memory requirements. It is shown in Fig. 2.3.

$$\begin{array}{lll}
 SP_1 = X_0^0 \xrightarrow{f} X_1^0 \xrightarrow{f} X_2^0 \xrightarrow{f} \dots \xrightarrow{f} X_t^0 = EP_1 & & SP_1 = EP_1 \\
 SP_2 = X_0^1 \xrightarrow{f} X_1^1 \xrightarrow{f} X_2^1 \xrightarrow{f} \dots \xrightarrow{f} X_t^1 = EP_2 & \Longrightarrow & SP_2 = EP_2 \\
 \vdots & & \vdots \\
 SP_m = X_0^m \xrightarrow{f} X_1^m \xrightarrow{f} X_2^m \xrightarrow{f} \dots \xrightarrow{f} X_t^m = EP_m & & SP_m = EP_m
 \end{array}$$

Figure 2.3: Matrix of images under function f , the image is taken from [97]

Basically, it provides the tradeoff by saving memory at the cost of cryptanalysis time. Overall it creates m chains of length t where only first and last elements are stored in a table. Given a ciphertext C one can try to find out if the key used to generate C is among the ones used to generate the table. If C was indeed obtained with a key used while creating the table then one will eventually generate the key that matches the last key of the corresponding chain. That last key has been stored in memory together with the first key of the chain. Using the first key of the chain the whole chain can be regenerated and in particular the key that comes just before the endpoint. This is the key that was used to generate C .

This time-memory tradeoff is an effective cryptanalytic approach for password hashing schemes which are mostly based on the iterative method of hash computation. The increasing threat of password leakage from compromised files of password hashes demands a memory hard design for password hashing. Therefore, one of the most important attacks for a password-hashing scheme is the TMTO and any such scheme needs to have a strong TMTO defense. A strong TMTO defense would mean that the attacker would not be able to reduce memory consumption at the cost of increased runtime. GPUs, FPGAs and custom ASICs provide tremendous amounts of computational power. A good password-hashing scheme must not allow the attacker to compute a password hash faster than possible on general purpose CPUs.

Rainbow Table Attack Utilizing the concept of Hellman [97], Philippe Oechslin introduced a cryptanalytic time-memory trade-off, named rainbow table [130] in 2003. This technique allows attacking a password hashing scheme by reducing the cryptanalysis time with the help of precomputed data stored in memory. It creates chains of password hashes choosing m uniformly random passwords from provided keyspace. It generates m chains of length t using t different reduction functions. The main advantage of a rainbow table over the technique of Hellman is that the chains can collide within the same table without merging (to merge, collision at the same position is required for two different chains). This was possible as for chains of length t the rainbow table uses t different reduction functions to compute the chain. However, this technique was practically applied to password hashing schemes that do not support the use of salt.

The method: Let $c_i = H(p_i)$, where H is the hash function and c_i is the hash of the i th password p_i . A reduction function $R(\cdot)$ is applied to c_1 and yields a string that looks like a plaintext (password). Specifically, given the functions $H(\cdot)$ and $R(\cdot)$, starting from some randomly chosen plaintext p_1 from the space of all passwords, it constructs a hash chain in the following manner.

$$p_1 \longrightarrow c_1=H(p_1) \longrightarrow p_2=R(c_1) \longrightarrow c_2=H(p_2) \longrightarrow p_3=R(c_2) \longrightarrow c_3=H(p_3) \longrightarrow p_4=R(c_3) \longrightarrow \dots$$

It creates the m chains of length t but uses a successive reduction function for each point in the chain. Suppose the chain starts with reduction function 1 then ends with reduction function $t-1$. It saves only the starting point plaintext and endpoint plaintext as below.

$$\begin{array}{c} (P_0^0, P_t^0) \\ (P_0^1, P_t^1) \\ \dots\dots\dots \\ \dots\dots\dots \\ \dots\dots\dots \end{array}$$

$$(P_0^{m-1}, P_t^{m-1})$$

Thus if two chains collide, they merge only if the collision appears at the same position in both chains. If the collision does not appear at the same position, both chains will continue with a different reduction function and will thus not merge. For chains of length t , if a collision occurs, the chance of it being a merge is thus only $1/t$. The probability of success within a single table of size $m \times t$ is given by:

$$P_{table} = 1 - \prod_{i=1}^t (1 - \frac{m_i}{N})$$

where $m_1 = m$ and $m_{n+1} = N(1 - e^{-\frac{m_n}{N}})$

To lookup, a key in a rainbow table first apply R_{n-1} to the ciphertext and look up the result in the endpoints of the table. If the match is found, rebuild the chain using the corresponding starting point. If not then apply R_{n-2} , H_{n-1} , to see if the key was in the second last column of the table. Like this try to apply R_{n-3} , H_{n-2} , H_{n-1} , and so forth. The regenerated chain will look as below.

$$p_1^i \rightarrow c_1^i = H(p_1^i) \rightarrow p_2^i = R(c_1^i) \rightarrow c_2^i = H(p_2^i) \rightarrow \dots \rightarrow c_{k-1}^i = H(p_{k-1}^i) \rightarrow p_k^i = R(c_{k-1}^i)$$

The total number of calculations to make is thus $t(t-1)/2$.

Ophcrack is a free open source tool. Oechslin designed this tool to show the faster time-memory trade-off techniques in [130] against LM and NT hashes. LM hash or NTLM hash are the hash functions used for Microsoft LAN Manager and Microsoft Windows versions prior to Windows NT. Since LM and NTLM hashes contain no salts, all possible hashes for a certain character set can be pre-generated, and the Rainbow table stores them in a more efficient manner. It cracks hashes using a precomputed rainbow table such as LM Rainbow Tables, NTLM Rainbow Tables etc. It is specifically used to crack the password hashing algorithms that do not use salt for password hash computation.

Chapter 3

Secure Design of Password Hashing Scheme

In this chapter, we describe our new password hashing design *Rig*, which was submitted to Password Hashing Competition (PHC) in March 2014. The design gets several reviews from different cryptographers which show its efficiency as a good design for password hashing. The first analysis by Marsh Ray from Microsoft was presented with a title ‘What Microsoft Would like from the PHC’ at Passwords-2014 conference which considered *Rig* among the top four designs from the total 24- PHC submissions [140]. The comment of the Google programmer, Bill Cox from PHC mailing list says, “This is a **very** high performance hashing function which makes RIG the clear speed leader in the cache-timing resistant category” and “they seem like a strong entry to me [15]”. During the PHC competition, several analysis on PHS came into existence. One of such significant analysis is ‘garbage collection attack’ proposed by Stefan Lucks et. al. [85]. In the work on the overview of the candidates for the PHC and their resistance against garbage-collector attacks, Lucks et. al. show that *Rig* is resistant to all existing attacks. In performance and memory-hardness *Rig* overperforms the PHC winner Argon2i [60].

This chapter is organized as follows: In Section 3.1, we introduce the Password Hashing Competition giving a high-level overview of how the initiative to encourage the improvement over current state-of-the-art designs for PHS was taken by the independent panel of cryptographers. Section 3.2 describes the significant constructions before PHC. In Section 3.3, we describe our design *Rig* which fulfills all the requirements mentioned at the PHC competition page. The specification and design rationale of *Rig* are presented in Subsections 3.3.2 and 3.3.3 respectively. Subsequently, the implementation aspects, performance analysis and security analysis are presented in Subsections 3.3.4, 3.3.10 and 3.3.11 respectively. This is followed by the description of the PHC winner Argon2i in Section 3.4 and the selection criteria of PHC in Section 3.5. In the concluding section, we summarize the entire chapter and give some important conclusions. The original contribution of this thesis is covered in Section 3.3.

3.1 Password Hashing Competition

The password-based authentication system is widely used for websites, mobile devices, operating systems, full-disk encryption, SSH keys, etc. However, the frequent reporting of password database leakage in real-world highlights the vulnerabilities existing in the current password-based constructions. In spite of the reported vulnerabilities in the use of passwords for human-computer authentication, very little formal work has been done addressing the design and analysis of password hashing schemes. This area of cryptography has been suffered from long ignorance by cryptographers and hence lacks formal definitions, security notions, and rigorous analysis. In order to alleviate these problems and to encourage strong password protection techniques, a Password Hashing Competition (PHC) [16] was held from 2013 to 2015. It was an effort by a group of independent cryptographers to identify new password hashing schemes in order to improve on the state-of-the-art algorithms. Currently, password hashing is an active topic of interest in cryptography community. Before PHC, less number of constructions were proposed (PBKDF2, bcrypt, and scrypt are significant) based on some ad-hoc approach.

The rate at which an attacker can guess passwords is a key factor in determining the strength of the password hashing scheme. Current requirements [16] for a secure password hashing scheme are the following:

- The construction should be slow to resist password guessing attack but should have a fast response time to prove the authenticity of the user.
- It should have a simple design and should be easy to implement (coding, testing, debugging, integration), i.e., the algorithm should be simple in the sense of clarity and concise with less number of internal components and primitives.
- It should be flexible and scalable, i.e., if memory parameter is not dependent on time parameter then one would be able to scale any of the parameters to get required performance.
- Cryptographic Security [16]: The construction should behave as a random function (random-looking output, one-way, collision resistant, immune to length extension, etc.).
- Resistant to GPU Attack: A typical GPU has lots of processing cores but has a limited amount of memory for each single core. It is quite efficient for an attacker to utilize all the available processing cores with limited memory to run brute-force attack over the password choices. Use of comparatively huge memory per password hash by the password hashing construction can restrict the use of GPU. Therefore, the design should have large memory consumption to force comparatively slow and costly hardware implementation that can resist the GPU attack.
- Leakage Resilience: The construction should protect against information extraction from physical implementation, i.e., the scheme should not leak information about the

password due to cache timing or memory leakage, while supporting any length of the password.

- **Client Independent Update:** The construction should have the ability to transform an existing hash to a different cost setting without knowledge of the password.

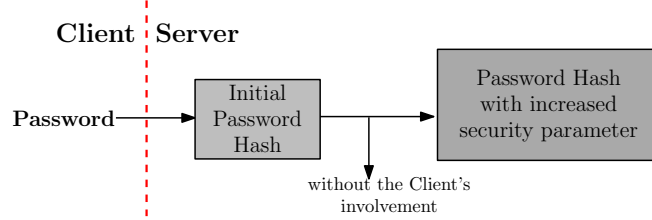


Figure 3.1: Client independent update

- **Server Relief Technique:** It is good if the construction provides server relief technique where the client performs most of the computations for password hashing and the server puts minimal effort with the minimal use of resources, to reduce the load on the server. This property needs a secure protocol to maintain the security of the hash computation. Fig. 3.2 shows an example where the password hashing process is split in two parts: (1) a slow (and possibly memory-demanding) part (S_p) and (2) a fast one-way transformation part (F_p).

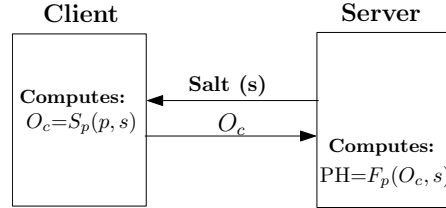


Figure 3.2: Server relief technique, an example

Other additional requirement introduced in [88] are the resistance to Garbage and Weak garbage collection attacks which are defined below.

1. **Garbage Collection Attack:** The memory-demanding password hashing may also provide completely new attack opportunities for the adversary. If we allocate a huge block of memory for password hashing, this memory becomes “garbage” after the password hashing algorithm has terminated, and will be collected for reuse, eventually. One usually assumes that the adversary learns the hash of the secret. The garbage collector attack assumes that the adversary additionally learns the memory content, after the termination of the password hashing algorithm and can easily get the password by using the collected memory content.

Let $PS_G(\cdot)$ be a memory-consuming password hashing algorithm that depends on a memory-cost parameter G and let Q be a positive constant. Furthermore, let v denote the internal state of $PS_G(\cdot)$ after its termination. Let A be a computationally unbounded but always halting adversary conducting a garbage-collector attack. A is successful if some knowledge about v reduces the runtime of A for testing a password candidate x from $O(PS_G(x))$ to $O(f(x))$, $O(f(x)) \lll O(PS_G(x))/Q$, $\forall x \in \{0, 1\}^*$.

2. **Weak Garbage-Collector (WGC) Attack:** Let $PS_G(\cdot)$ be a password hashing algorithm that depends on a memory-cost parameter G , and let $F(\cdot)$ be an underlying function of $PS_G(\cdot)$ that can be efficiently computed. An adversary A is successful in terms of a weak garbage-collector attack if a value $y = F(pwd)$ remains in memory during (almost) the entire runtime of $PS_G(pwd)$, pwd the password.

The most challenging threat faced by any password hashing scheme is the existence of cheap, massively parallel hardware such as Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). Using such efficient hardware, an adversary with multiple computing units can easily try multiple different passwords in parallel. To prevent such attempts we need to slow down password hash computation and ensure that there is little parallelism in the design. One way to achieve this is to use a ‘Sequential memory-hard’ algorithm, a term first introduced with the design of ‘Scrypt’ [132], a password hashing scheme. The main design principle of Scrypt is that it asymptotically uses almost as many memory locations as it uses operations to slow down the password-hash computation. Memory is *relatively* expensive, so, a typical GPU or other cheap massively-parallel hardware with lots of cores can only have a limited amount of memory for each single core. Hence an attacker with access to such hardware will still not be able to utilize all the available processing cores due to the lack of sufficient memory and will be forced to have an (almost) sequential implementation of the password hashing scheme.

PHC [16]: 24 initial submissions were accepted in April 2014 [16] and 2 designs were withdrawn later. Following are the list of first round candidates.

1. **AntCrypt:** Markus Duermuth, Ralf Zimmerman.
2. **Argon** and Argon2: Alex Biryukov, Dmitry Khovratovich.
3. **battcrypt:** Steve Thomas
4. **Catena:** Christian Forler, Stefan Lucks, Jakob Wenzel.
5. **Centrifuge:** Rafael Alvarez.
6. **EARWORM:** Daniel Franke
7. **Gambit:** Krisztián Pintér.
8. **Lanarea:** Haneef Mubarak.

9. **Lyra2**: Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, Paulo S. L. M. Barreto.
10. **Makwa**: Thomas Pornin.
11. **MCS_PHS**: Mikhail Maslennikov.
12. **Omega Crypt**: Brandon Enright.
13. **Parallel**: Steve Thomas.
14. **PolyPassHash**: Justin Cappos, Santiago Torres Arias.
15. **POMELO**: Hongjun Wu.
16. **Pufferfish**: Jeremi Gosney.
17. **RIG**: Donghoon Chang, Arpan Jati, Sweta Mishra, Somitra Kumar Sanadhya.
18. **Schvrch**: Rade Vuckovac.
19. **Tortuga**: Teath Sch.
20. **TwoCats**: Bill Cox.
21. **Yarn**: Evgeny Kapun.
22. **yescrypt**: Alexander Peslyak.

In July 2015, PHC selects the winner, Argon2, an algorithm designed by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich from University of Luxembourg and gives special recognition to four of the finalists: Catena, Lyra2, Makwa, yescrypt.

3.2 Significant Constructions before PHC

The significant constructions before PHC for password hashing are PBKDF2 [160], Bcrypt [137] and Scrypt [132]. All of these do not satisfy most of the necessary requirements mentioned at the competition page [16]. PBKDF2 (NIST standard) consumes very less memory as it was mainly designed to derive keys from a seed (password). Bcrypt uses fixed memory (4KB) for its implementation. Scrypt is not simple (different internal modules) and not flexible (time and memory parameters are dependent) and susceptible to cache timing attack. The details are provided below.

1. **Bcrypt- 1999 [137]**: The algorithm bcrypt is based on the block cipher Eksblowfish which is derived from the block cipher Blowfish [145]. Eksblowfish is described as a cost parameterizable, salted and an expensive key schedule Blowfish. Eksblowfish is designed to take user-chosen passwords as keys.

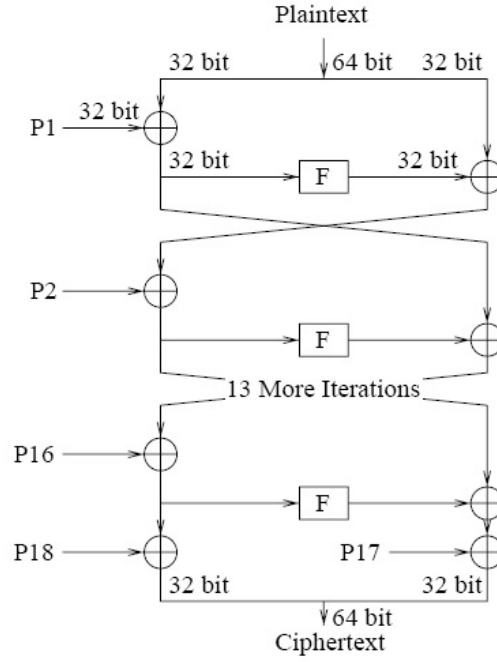


Figure 3.3: Blowfish Feistel network with F being the Feistel function. The image is taken from [137]

Blowfish is a 64-bit block cipher, structured as a 16-round Feistel network as shown in Fig 3.3. It uses 18 32-bit subkeys, $P_1, \dots, P_{18}$, which it derives from the encryption key. Internally, Blowfish uses a slow key scheduler to generate an internal state of 4,168 bytes for the key-dependent S-boxes (4×1024 bytes) and the round keys (72 bytes). The function F uses the S-boxes. Eksblowfish encrypts identically to Blowfish. It differs in the function it uses to transform encryption keys into subkeys and S-boxes. It takes three input parameters: a cost, a salt, and the encryption key. It returns a set of subkeys and S-boxes, also known as a key schedule. It is described in Algorithm 3.1.

Algorithm 3.1: Eksblowfish

Input: cost, salt, key
Output: state

1. state $\leftarrow$ InitState ()
2. state $\leftarrow$ ExpandKey (state, salt, key)
3. repeat(2^{cost})
4. state $\leftarrow$ ExpandKey (state, 0, salt)
5. state $\leftarrow$ ExpandKey (state, 0, key)
6. return state

Bcrypt uses a 128-bit salt and encrypts a 192-bit value, “OrpheanBeholderScryDoubt”. The design Bcrypt is explained in Algorithm 3.2. It takes advantage of the expensive key setup in Eksblowfish. Thus, while Bcrypt has not been designed with the intention to thwart parallelized attackers by exhaustive memory usage, the state is sufficiently large to slow down Bcrypt significantly on current GPUs, e.g., it can only be computed about 4,000 times per second on an AMD HD 7970 graphic card [155]. However, the state size is fixed so if future GPUs have a larger cache, it may actually run much faster. There is no tunable parameter to increase the memory requirement. It is the default PHS of the BSD operating system. The password is up to 56-bytes and the produced hash value is 24-bytes. The iteration count is a power of 2, denoted as 2^c . It is tunable to increase the computation power based on implementation requirement. The use of 16-byte salt defends against rainbow table attacks.

Algorithm 3.2: Bcrypt

Input: cost, salt, pwd
Output: Concatenate(cost, salt, ctext)

1. state $\leftarrow$ EksBlowfishSetup(cost,salt, key)
2. ctext $\leftarrow$ “OrpheanBeholderScryDoubt”
3. repeat(64)
4. ctext $\leftarrow$ EncryptECB(state, ctext)
5. return Concatenate(cost, salt, ctext)

2. **PBKDF2- 2000 [160]:** The Password-Based Key Derivation Function 2 (PBKDF2), is a NIST standard and is part of RSA Laboratories’ Public-Key Cryptography Standards (PKCS) series, specifically PKCS #5 v2.0, also published as Internet Engineering Task Force’s RFC 2898. PBKDF2 uses a pseudorandom function, HMAC to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key. The salt hardens dictionary and rainbow table attacks [130] (ability to use tables of precomputed hashes) and it must be up to 8 bytes. The standard was established in 2000 and recommended a minimum of 1000 iterations, but in today’s scenario, these parameter choice is not adequate. It consumes very less memory for its implementation and therefore enables cheap brute-force attacks on ASICs and

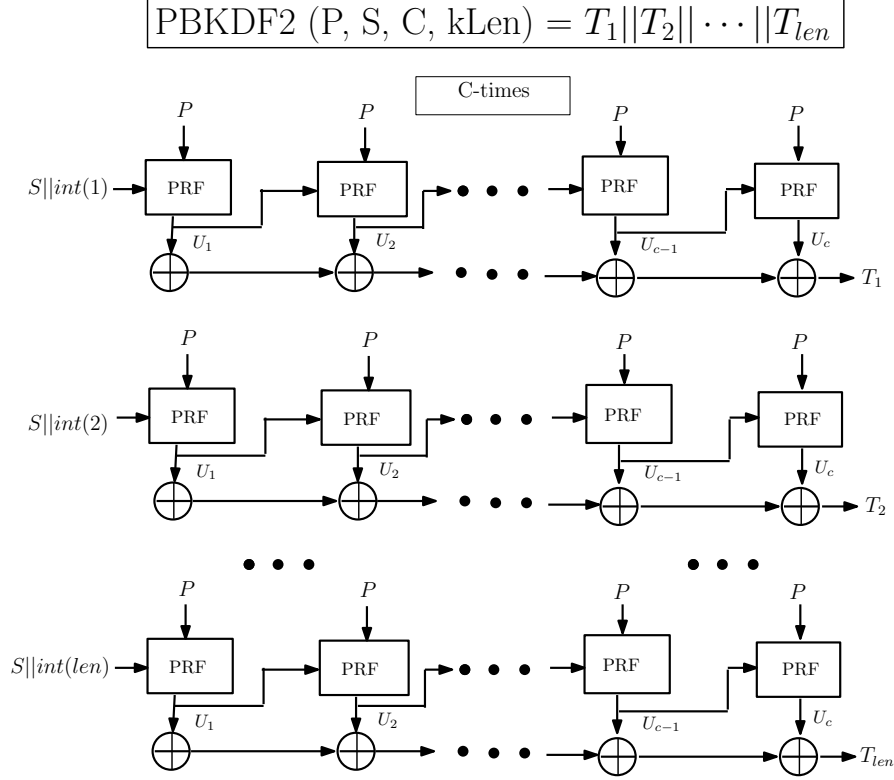


Figure 3.4: Overview of PBKDF2, the PRF is a pseudorandom function of two parameters (e.g. HMAC). The image is taken from [160]

GPUs. The design is described in Fig. 3.4, where the PRF is the pseudorandom function HMAC and kLen is the desired output length. The output is the concatenation of the T_i 's in sequence where $1 \leq i \leq len$ and $len = \text{kLen}/\text{length of HMAC-output}$.

3. **Scrypt- 2009 [132]** It is a password-based key derivation function created by Colin Percival, originally for the Tarsnap online backup service. The algorithm was specifically designed to make it costly to perform large-scale custom hardware attacks by requiring large amounts of memory. It is a sequential memory-hard algorithm a term used when the algorithm asymptotically uses almost as many memory locations as it uses operations to make the password-hash computation process slow. It uses PBKDF2 to convert a password into a bitstream (HMAC-SHA256) and feed this bitstream to ROMix (Salsa20/8 core) and then feed the output of ROMix back to PBKDF2 to generate the derived key. The algorithm ROMix is described in Algorithm 3.4. The Occupying a lot of memory hinders attacks using special-purpose hardware and GPUs. The sequentially memory-hard function ROMix, takes G units of memory and performs $2G$ operations. With only G/K units of memory, the number of operations goes up to $2G - K$. In [132], Percival recommends $G = 214$ and $G = 220$ for password hashing and key derivation, respectively. The design Scrypt is shown in Algorithm 3.3.

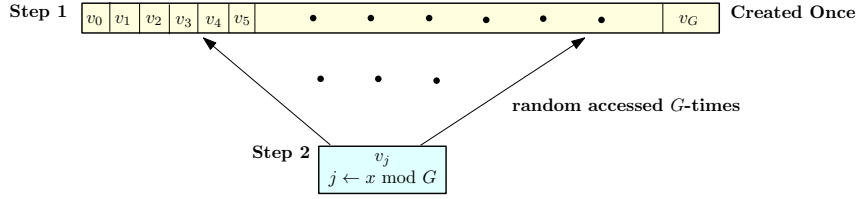


Figure 3.5: Random access in Script.

Algorithm 3.3: Script

Input: pwd, s, G {Cost parameter}
Output: x {Password hash}

1. $x \leftarrow \text{PBKDF2}(\text{pwd}, s, 1, 1)$
2. $x \leftarrow \mathbf{ROMix}(x, G)$
3. $x \leftarrow \text{PBKDF2}(\text{pwd}, x, 1, 1)$
4. return x

Algorithm 3.4: ROMix

Input: x, G {Cost parameter}
Output: x {Hash value}

1. for $i = 0$ to $G - 1$
2. $v_i = x$
3. $x = H(x)$
4. for $i = 0$ to $G - 1$
5. $j \leftarrow x \bmod G$
6. $x \leftarrow H(x \oplus v_j)$
7. return x

- **Cache-timing attack** [88] : The algorithm ROMix of Script takes two inputs: an initial state x which depends on both salt and password, and the array size G that defines the required storage. In the first phase, ROMix initializes an array v , i.e., the array variables $v_0, \dots, v_{G-1}$ are set to $x, H(x), H(H(x)), \dots, H(\dots(H(x)))$, respectively. In the second phase, ROMix updates x depending on the random access of the value v_j which is derived from x . This random access is shown in Fig. 3.5. The sequential-memory hardness comes from the way how the index j is computed, depending on the current value of x , i.e., $j \leftarrow x \bmod G$. After G updates, the final value of x is returned and undergoes the post-processing.
- **The Spy Process:** The idea to compute an unpredictable index j and then ask for the value v_j , which is useful for sequential memory-hardness, is also an issue.

Consider a spy process running on the same machine as Scrypt. This spy process cannot read the internal memory of Scrypt. However, as it is running on the same machine, it shares its cache memory with ROMix. The spy process interrupts the execution of ROMix twice:

1. When ROMix enters the second phase, the spy process reads from a bunch of addresses, to force out all the v_i that are still in the cache.
2. Now, the spy process interrupts ROMix again. By measuring access times when reading from different addresses, the spy process can figure out which of the v_i have been read by ROMix, in between.

So, the spy process can tell the indices j for which v_j has been read, and with this information, an attacker can mount the following cache-timing attack. Let pwd' denote the current password candidate. Suppose x is the output of $\text{PBKDF2}(\text{pwd}', \text{salt}, 1, 1)$. Then, the following can be applied for password candidate sieve:

- (a) Execute the first phase of ROMix, without storing the v_i .
- (b) Compute the index $j \leftarrow x \bmod G$.
- (c) If v_j is one of the values that have been read by ROMix, then store pwd' in a list.
- (d) Else, conclude that pwd' is a wrong password.

This helps to filter out the candidate password as the index j is password dependent. Therefore password independent memory access is recommended for password hashing to prevent such cache-timing attack.

3.3 Our Password Hashing Design, Rig

In this section, we introduce our password hashing framework *Rig* which is based on secure cryptographic hash functions. *Rig* is very simple to implement in software. It is flexible as the memory parameter is independent of time parameter (no actual time and memory tradeoff) and is strictly sequential (difficult to parallelize) with comparatively huge memory consumption that provides strong resistance against attackers using multiple processing units. It supports client-independent password hash up-gradation without the need of the actual password. This feature helps the server to increase the security parameters to calculate the password hash to reduce the constant threats of technological improvements, specifically in the field of hardware. *Rig* provides protection against the extraction of information from cache-timing attack and prevents the denial-of-service attack if implemented to provide server-relief technique. We analyze *Rig* and show that our proposal provides an exponential time complexity against memory-free attack. It gives the flexibility to choose different functions for different phases of the construction and we denote the general construction of *Rig* as $\text{Rig}[H_1, H_2, H_3]$. We provide two variants of $\text{Rig}[H_1, H_2, H_3]$. A strictly sequential variant, $\text{Rig}[\text{Blake2b}, \text{BlakeCompress}, \text{Blake2b}]$ and the other variant, $\text{Rig}[\text{BlakeExpand},$

BlakePerm, Blake2b] which improves the performance by performing memory operations in larger chunks.

3.3.1 Preliminaries

The techniques used in our construction are discussed below.

- **Binary 64-bit mapping:** It is a 64-bit binary representation of the decimal value. The binary number

$$a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_0$$

is denoted as $a_{n-1}a_{n-2}\dots a_0$ where $a_i \in \{0, 1\}$ and n is the number of digits to the left of the binary (radix) point. In our construction we use $n = 64$ and we denote $\text{binary}_{64}(x)$ for 64-bit binary representation of the value x .

- **Bit reversal permutation [86, 117] (br):** It is implemented to permute the indices of an array of $n = 2^k$ elements where $k \in \mathbb{N}$. We explain the steps of the permutation through Algorithm 3.5 below.

The example of a bit reversal permutation applied on an array of $m = 2^3$ elements where $k = 3$ and indices are $0, 1, \dots, 7$ is given below.

$$\begin{aligned} \text{br}[000, 001, 010, 011, 100, 101, 110, 111] &= [000, 100, 010, 110, 001, 101, 011, 111] \\ &= \text{br}[0], \text{br}[1], \text{br}[2], \text{br}[3], \dots, \text{br}[7]. \end{aligned}$$

Algorithm 3.5: Bit reversal permutation (br)

- Input:** Indices of an array A of $n = 2^k$ elements where $k \in \mathbb{N}$ and indices are: $0, 1, 2, \dots, n - 1$.
- Output:** Permuted indices of array A as: $\text{br}[0], \text{br}[1], \text{br}[2], \dots, \text{br}[n - 1]$
1. for $i = 0$ to $n - 1$
 2. $(i)_{\text{bin}_k} = i_{k-1}i_{k-2}\dots i_1i_0 = \sum_{j=0}^{k-1} 2^j i_j$
 3. $\triangleright (i)_{\text{bin}_k} = k\text{-bit binary representation of value } i$
 4. $\text{br}[i] = \sum_{j=0}^{k-1} 2^j i_{k-1-j}$
 5. return $\text{br}[0], \text{br}[1], \text{br}[2], \dots, \text{br}[n - 1]$
-

3.3.2 Specification of Rig

Our construction is described in Fig. 3.6. Following is the step-by-step description of Algorithm 3.6 which explains our construction *Rig*.

1. First we need to fix the following parameters:
 - pwd = The user password of any length.
 - s = The salt value of any length.
 - n = The number of iterations required to perform iterative transformation phase.

- m_c = The memory count from which the memory-cost is defined as: $m = 2^{m_c}$, i.e., m denotes the number of items to be stored in the memory. The value of m is updated as: $m_{i+1} = 2 \times m_i$ at each round.
- r = The number of rounds for the setup phase followed by iterative transformation phase and output generation phase.
- l = The output length of the password hash.
- t = The number of bits retained from hash output after truncation. Used with a function $\text{trunc}_t(x) = x \gg (|x| - t)$, where x is the hash output.

2. **Initialization phase:** We map the parameters, namely the values: password length pwd_l , salt length s_l , n and the output length, l to a 64-bit binary value using binary_{64} mapping. We create the value x as the concatenation ($\parallel$) of the above mentioned parameters as

$$x = \text{pwd} \parallel \text{binary}_{64}(\text{pwd}_l) \parallel s \parallel \text{binary}_{64}(s_l) \parallel \text{binary}_{64}(n) \parallel \text{binary}_{64}(l)$$

and compute $H_1(x) = \alpha$ where H_1 is the underlying hash function. We use α for further calculations in the setup phase.

3. **Setup phase** We initialize h_0 with the value of π after the decimal point. We take as many digits of π as desired to ensure that $|h_0| = |\alpha|$. The values h_0 and α are used to initialize two arrays k and a and further $m - 1$ values of the arrays are iteratively calculated as shown in the Fig. 3.6. First t -bits of each hash output is stored in the array k .

The large number of calls to the underlying hash function are guaranteed to have different inputs by the use of different counter values. H_2 denotes the underlying hash function.

4. **Iterative transformation phase** This phase is designed to make constant use of the stored array values and to update them. Here we modify each element of the arrays a and k , n -times where n is the number of iterations.

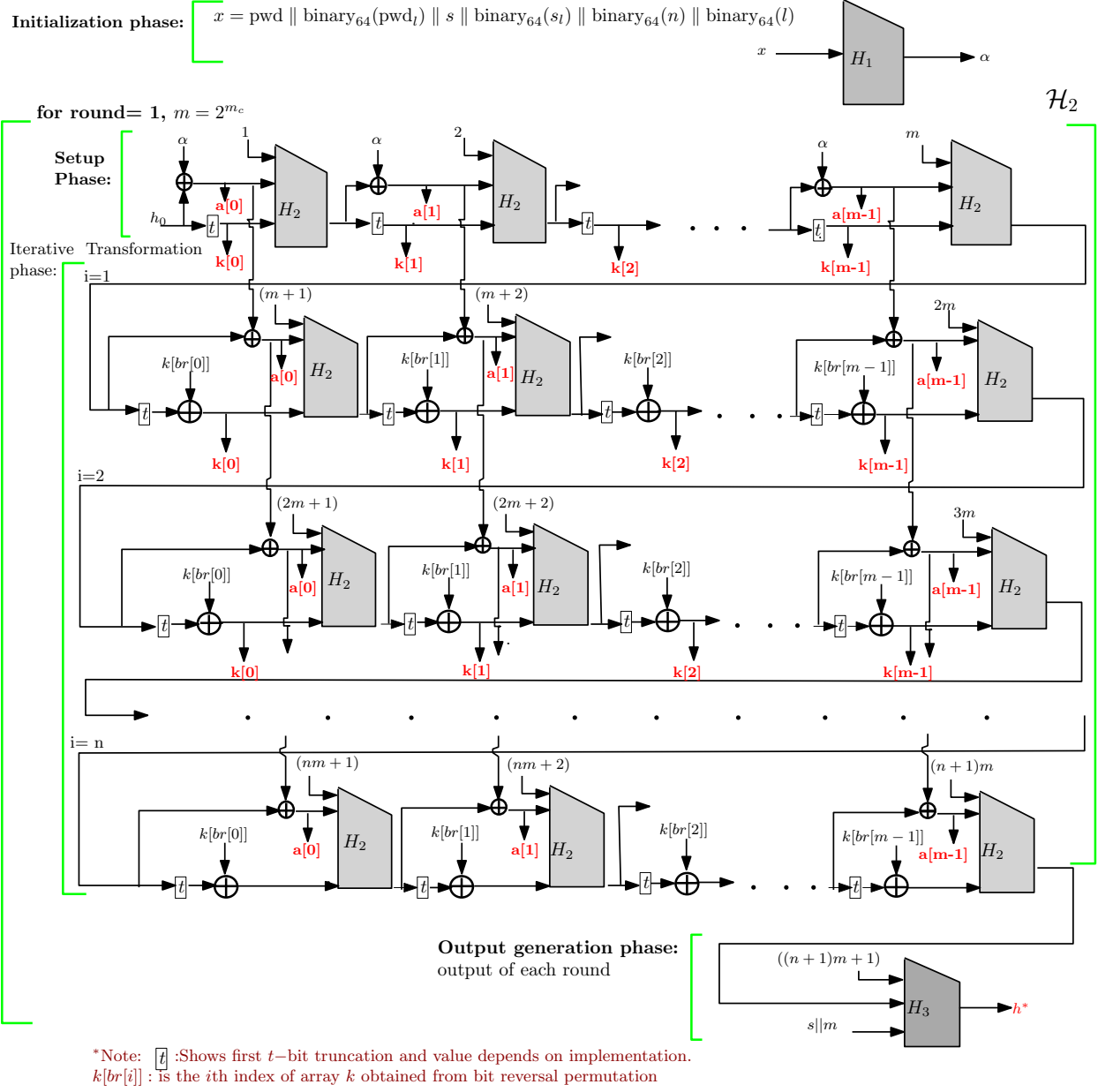


Figure 3.6: Graphical representation of the proposed construction *Rlg*.

Algorithm 3.6: Rig $[H_1, H_2, H_3]$ Construction	
Input:	Password (pwd), Password length (pwd_l), Salt (s), Salt length (s_l), No. of iterations (n), Memory count (m_c), No. of bits to be retained from hash output of the setup phase (t), Output length (l), No. of rounds (r).
Output:	l -bit hash value h_r^* obtained after r rounds
1.	▷ Initialization phase: generates α from password
2.	Initialize: a random salt (s) of atleast 16-bytes, number of iterations (n), value of memory count m_c where $m = 2^{m_c}$, value t
3.	$x = \text{pwd} \parallel \text{binary}_{64}(\text{pwd}_l) \parallel s \parallel \text{binary}_{64}(s_l) \parallel \text{binary}_{64}(n) \parallel \text{binary}_{64}(l)$ ▷ concatenation: $\parallel$
4.	$\alpha = H_1(x)$ ▷ H_1 : underlying hash function
5.	for round 1 to r
6.	▷ Initialization of Setup phase: Creates two arrays k and a where $ k = a = m$ where $m = 2^{(\text{round}-1)} \times 2^{m_c}$
7.	h_0 = initialized with the value of π after decimal, and $ h_0 = \alpha $.
8.	$a[0] = \alpha \oplus h_0$, $k[0] = \text{trunc}_t(h_0)$
9.	for $i = 1$ to m
10.	$h_i = H_2(i \parallel a[i-1] \parallel k[i-1])$ ▷ H_2 : underlying hash function
11.	if $i \neq m$
12.	$a[i] = \alpha \oplus h_i$
13.	$k[i] = \text{trunc}_t(h_i)$ ▷ retains the first t -bits of the hash output
14.	▷ Initialization of Iterative Transformation phase
15.	for $i = 1$ to n
16.	for $j = 1$ to m
17.	$a[j-1] = a[j-1] \oplus h_{\{im+j-1\}}$
18.	$br[j-1] = \text{index value of array } k \text{ obtained using bit reversal permutation}$
19.	▷ initialize a temporary array $ k_{\text{temp}} = k $
20.	$k_{\text{temp}}[j-1] = k[br[j-1]] \oplus \text{trunc}_t(h_{im+j-1})$
21.	$h_{im+j} = H_2((im+j) \parallel a[j-1] \parallel k_{\text{temp}}[j-1])$
22.	$k = k_{\text{temp}}$
23.	▷ Output generation phase
24.	$h_{\text{round}}^* = (H_3((n+1)m+1) \parallel h_{(n+1)m} \parallel s \parallel \text{binary}_{64}(m))$
25.	if round $< r$
26.	$\alpha = h_{\text{round}}^*$

and compute $H_1(x) = \alpha$ where H_1 is the underlying hash function. We use α for further calculations in the setup phase.

5. **Setup phase** We initialize h_0 with the value of π after the decimal point. We take as many digits of π as desired to ensure that $|h_0| = |\alpha|$. The values h_0 and α are used to initialize two arrays k and a and further $m - 1$ values of the arrays are iteratively calculated as shown in the Fig. 3.6. First t -bits of each hash output are stored in the array k .

A large number of calls to the underlying hash function is guaranteed to have different

inputs by the use of different counter values. H_2 denotes the underlying hash function.

6. **Iterative transformation phase** This phase is designed to make constant use of the stored array values and to update them. Here we modify each element of the arrays a and k , n -times where n is the number of iterations. Array a is accessed sequentially where values of the array k are accessed using bit-reversal permutation explained in Algorithm 1. We denote the index of array k obtained applying bit-reversal permutation as: $br[j]$, $0 \leq j \leq m - 1$.
7. **Output generation phase** After execution of the setup phase and iterative transformation phase sequentially, we calculate one more hash, denoted by H_3 to get the output of each round. If round=1, this output is considered as the password hash.
Note: The output is an l -bit value. The algorithm stores the output as the hashed password. Our construction allows for storing a truncated portion of the hash output as well. If this is desired we can take one of the following two approaches.
 - (a) The user may run the complete algorithm as described above and truncate the final output after r rounds to the desired length. This approach does not support client-independent update.
 - (b) To support client-independent updates the user can choose a length for truncation which is sufficient to claim brute-force security. Then append some constant value, we suggest the hexadecimal value of π after first 64-bytes of decimal point. Take as many digits as desired to make the output length of each round equal to the length of α of the setup phase. So this way one can reduce the storage requirement for password hashes at the server.

3.3.3 Design Rationale

Existing password hashing schemes are not simple and do not fulfill the necessary requirements as discussed in section 3.2. We have tried to design a solution which overcomes the known disadvantages of existing schemes (PBKDF2 [160], Bcrypt [137] and Scrypt [132]). The primary concerns against existing proposals are their complex design and their inefficiency to prevent hardware threats. We have tried to strengthen our design by considering the necessary requirements as mentioned in section 3.1.

1. **Initialization phase** We take concatenation of password, salt, 64-bit value of pwd_l , s_l , n and l as input to increase the size of input. This resists brute force dictionary attack.
2. **Setup phase** In this phase we initialize h_0 with π , as we want to have a random sequence and π is not known to have any pattern in the sequence of digits after the decimal point. We generate the values that are required to be stored and repeatedly accessed throughout the remaining phases. This ensures that a large memory requirement criterion for a password hashing scheme is satisfied which neutralizes the

threat of using recent technological trends, such as GPUs, ASICs etc. We use different counter values for each hash computation to make all hash inputs different. This reduces collisions and hence makes the output different.

For array k there is a flexibility to vary the bit storage by taking first t -bits of the hash output where t is taken to be close to the hash-length but not equal to the hash-length. This fulfills the demand of huge memory while at the same time ensures sequential hash calculation and forces an attacker to compute the hash at run-time thus slowing him down. Further, it also allows to extend the scope of implementation in that a low memory device may keep very few bits of the hash values stored but may increase the number of iterations. This ensures that *Rig* can be implemented in resource-constrained devices.

3. **Iterative transformation phase** To make the storage requirement compulsory, this phase progresses sequentially, accessing and updating all stored values at each iteration. Here again, we use different counters for hash input for the same reason as mentioned above. In this phase, the memory access pattern is made password independent to reduce the chance of cache timing attack which we have explained later in this section.
4. **Output generation phase** This is the last phase of each round. We reuse the salt value as input to make the collision attack difficult.

Apart from that, the output of each round can be truncated to the desired length. This is optionally mentioned to handle the situations when it is required to reduce the server storage per password.

The other important criteria taken into account in the design of the scheme are the following:

5. **Simplicity and flexibility** Symmetry (as setup phase and iterative transformation phase follows similar structure) in the design of *Rig* enhances the overall clarity of the scheme. An earlier password hashing scheme Script [132] uses PBKDF2 (internally calls HMAC-SHA256) and ROMix (internally calls BlockMix and uses Salsa20/8). Unlike Script, *Rig* uses only a single primitive (a cryptographically secure hash function). This makes our scheme easier to understand and easy to implement (coding, testing, debugging and integration).

In our scheme, the memory parameter is independent of the time parameter. This flexibility in design choice allows a user to scale any of these parameters to get the required performance. On the other hand, we have the flexibility for the choice of the functions H_1 , H_2 and H_3 (see Fig. 3.6), but proper selection of the primitives are required to maintain the overall design properties and security. Therefore there can be multiple variants of *Rig* aimed at different implementations or scenarios.

6. **Random output** Our scheme calls a hash function repeatedly. We use different counters for each of these hash calls to ensure that no input to the hash function is repeated. The security of *Rig* relies on the prevention of preimage and collision

attacks against the underlying hash function. Use of any state-of-the-art hash function (e.g. any finalist of SHA-3 competition) ensures the security of our scheme. We use Blake2b [35] in demonstrating the performance of our scheme later in this paper, although any other hash function could easily be used instead.

With the property of different input, different output and same input, same output, our scheme mimics the Random Oracle Model. This provides the theoretical justification of the security of *Rig*.

7. **Client-independent update [86]** Our design supports client independent update, i.e., a server can increase the security parameter without knowing the password. This is possible if we fix the value of n (number of iterations) and increase the number of rounds r . Each round of the algorithm doubles the memory consumption m from the previous round and hence increases the security parameter. This is possible because the output of each round can be treated as the value α at the next round and then can easily follow the Algorithm 3.6 to produce the output of the next round. The idea of client independent update of the security parameter m is fulfilled by the following way. The value of m is updated at each next round $i + 1$ (say) from its previous round i as: $m_{i+1} = 2 \times m_i$.

The overall procedure is: output of each round is the input to the next. Each round gives full hash maintaining all requirements of a good password hashing technique. By increasing the number of rounds, the scheme increases the required memory and time hence increases the security parameter without the interference of the client.

8. **Resistance against cache-timing attack** In our construction, to access the memory which is stored in an array k , we use the bit-reversal permutation, which is independent of the password used. If a password dependent permutation is used and if the array can be stored in the cache while accessing the values, an attacker can trace the access pattern observing the time difference in each access of the array index. This helps the adversary to filter the passwords that follow similar memory access pattern and to make a list of feasible passwords. Therefore, a password hashing scheme should have password-independent memory access patterns and to follow this requirement we use bit-reversal permutation as in [86].
9. **Server-relief hashing** Current requirement of a password hashing technique is that it should be slow and should demand comparatively large memory to implement. However, this requirement may put extra load on the server. Therefore we need a protocol to divide the load between the client and the server. The idea is provided in [86] and our construction supports this property following the protocol as mentioned below: First the authentication server provides the salt to the client. The client performs the initialization phase, setup phase and iterative transformation phase (see Algorithm 3.6), and sends the end result to the server. The server computes the output generation phase and produces the final hash. This way we can easily reduce the load on the server.

Note: In this case, an attacker acting as a client, can repeatedly send some random data without following the computations of the proposed algorithm to the server. Here, the attacker can easily get the access with a correct guess. But, the complexity of the random guess will be equivalent to the brute-force complexity i.e. 2^n , where n is the output length of the underlying hash function. Therefore this can not be a feasible attack.

3.3.4 Implementation aspects

This proposed construction for password hashing can be implemented efficiently on a wide range of processors. However, the same implementation will require huge number of computations if dedicated hardware such as ASIC or FPGA is used with limited memory.

Our design allows the flexibility to utilize less storage with an increased number of calculations if we retain few bits of the intermediate hash computation after truncation and increase the number of iterations n . This way, *Rig* can be efficient on low memory devices.

We designed *Rig* to have a highly flexible structure. By changing the functions H_1 , H_2 and H_3 (see Figure 3.6) we can completely change the overall design properties. From side channel resistance to GPU or ASIC/FPGA resistance, any property can be achieved by the proper selection of the above primitives. Therefore there can be multiple variants aimed at different implementations or scenarios. As mentioned before, we describe the general construction of *Rig* as *Rig* [H_1, H_2, H_3], where we can design/choose the functions H_1 , H_2 , H_3 for implementing different variants of *Rig*.

We have designed and implemented two versions of *Rig* as follows:

1. **Rig [Blake2b, BlakeCompress, Blake2b]** This variant is strictly sequential. Full Blake2b is used for H_1 and H_3 while the first round of the compression function of Blake2b is used for H_2 (and we call it *BlakeCompress*). We remove the constants in the ‘G’ function of Blake2b as suggested by the Blake authors in [35] to improve the overall performance. This version does a large number of random reads and writes and as a result, it is strictly bounded by memory latency.
2. **Rig [BlakeExpand, BlakePerm, Blake2b]** This variant is designed to improve the performance by performing memory operations in larger chunks. The functions H_1 and H_2 are parallelized internally and the idea of handling larger chunk size improves the performance significantly without changing the overall sequential nature and memory-hardness of *Rig*. It also makes this variant of *Rig* much more difficult to execute efficiently in GPUs and FPGA/ASIC (explained in sections 3.3.8 and 3.3.9). We implement the functions H_1 as ‘BlakeExpand’ and H_2 as ‘BlakePerm’. These functions are explained later. The function H_3 uses full Blake2b.

3.3.5 Design of Rig [Blake2b, BlakeCompress, Blake2b]

This strictly sequential variant follows the general construction of *Rig* as explained in section 3.3.2. The functions H_1 and H_3 implements Blake2b (full hash). The function H_2 is implemented using the first round of Blake2b compression function.

3.3.5.1 Design of the BlakeCompress function

In Figure 3.7 we graphically show the implementation of H_2 as BlakeCompress. H_2 takes 1024 bits input and initializes the initial state of Blake2b with this input. Then the eight G -functions as defined in Blake2b are applied with an all zero input message of length 1024 bits. RF_i denotes the i th call to these eight G -functions. As we use the first round only, we show the round as RF_1 in Figure 3.7. The modified state after RF_1 is then split into two equal halves and xor'ed together to produce 512-bits of output. This choice of implementation is different from the actual Blake2b in many ways. The actual Blake2b construction, shown in Figure 3.8, has an initialization phase which initializes the starting state. A permutation of the message m^i is supplied as the input to each RF_i for round $i = 1$ to 12. After 12 rounds, the finalization phase performs feedforward xor'ing with the output of RF_{12} . This preserves the onewayness of the Blake2b function. This feedforward xor'ing is omitted in our BlakeCompress implementation. This choice of implementation reduces the time of hash computation and improves the performance. The overall security is not compromised by this implementation (see section 3.3.13.1).

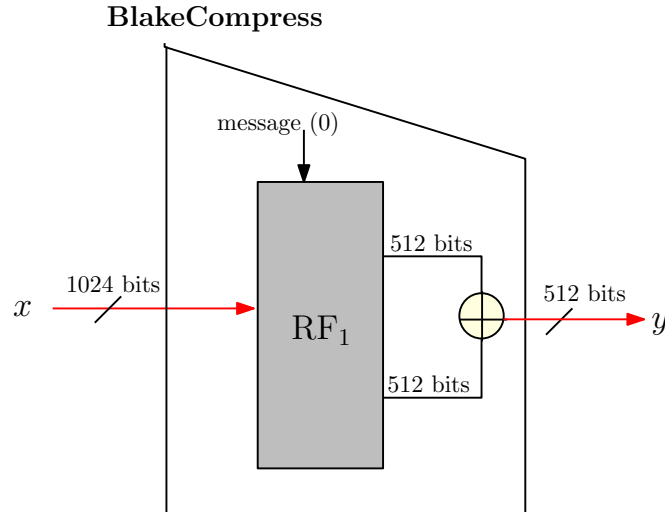
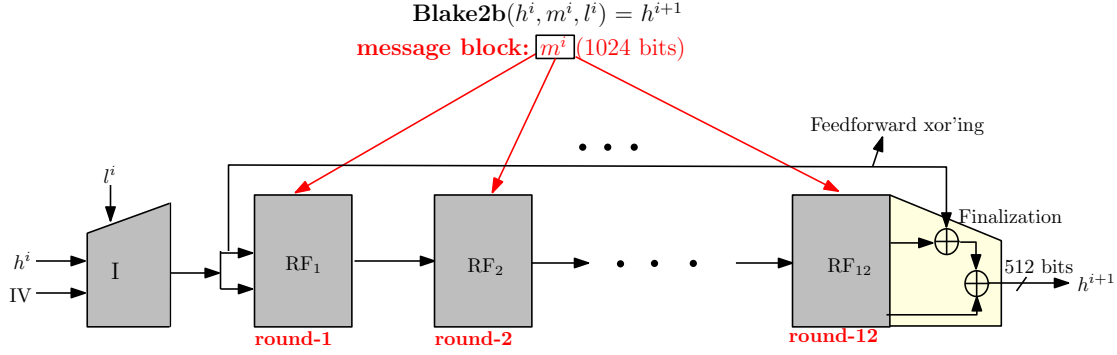


Figure 3.7: Function H_2 implemented as function $\text{BlakeCompress}[x] = y$, where input length of $x = 1024$ bits and output length of $y = 512$ bits. RF_1 is the first round of Blake2b compression function. Input size of $RF_1 = 1024$ bits.



Blake2b compression function defined as $\text{Blake2b}(h^i, m^i, l^i) = h^{i+1}$ where

- h^i is the 512 bits chaining value,
- m^i is the i th message block and each m^i is of length 1024 bits
- l^i denotes the number of data bytes in $m^0, m^1, \dots, m^i$
- I denotes the initialization phase of Blake2b
- RF_i denotes the i th call to the eight G -functions

Figure 3.8: Blake2b compression function

3.3.6 Design of Rig [BlakeExpand, BlakePerm, Blake2b]

The optimized variant of Rig uses an expansion function BlakeExpand to expand the state and a compression function BlakePerm to compress the state. Full Blake2b is used to hash the output state after the iterative-transformation phase to obtain the final hash. The design aspects are described below.

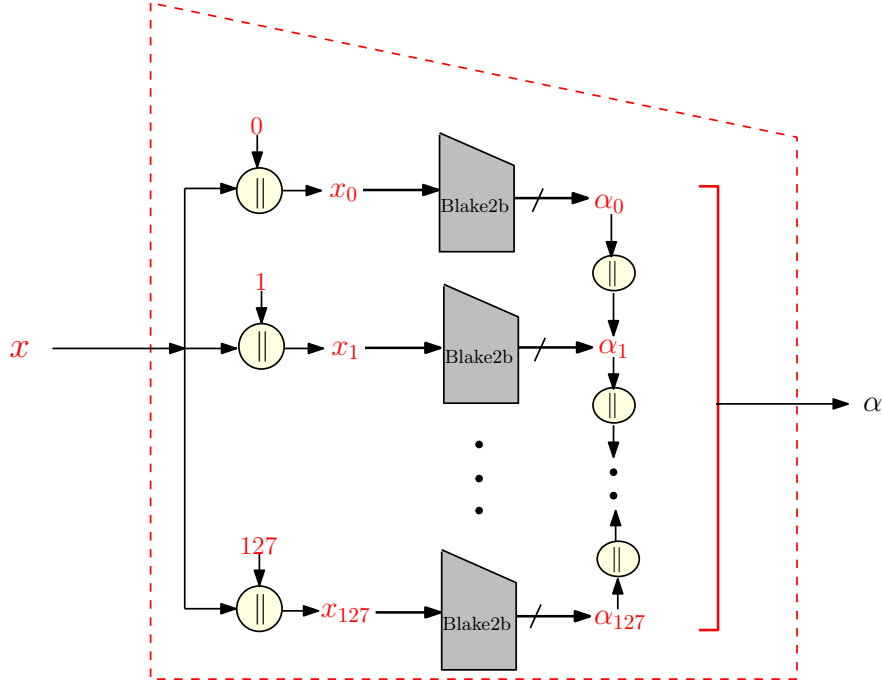
3.3.6.1 Design of the BlakeExpand Expansion Function

The BlakeExpand function is a very simple function which expands the input x to a fixed size of 8KiB. The function BlakeExpand is an instantiation of H_1 . The input x passes through 128 individual instances of Blake2b (full hash) each appended by a counter as $x_i = x \parallel i$, for $0 \leq i \leq 127$ and produces the output $\alpha = \alpha_0 \parallel \alpha_1 \parallel \dots \parallel \alpha_{127}$ where each α_i is of length 512 bits, i.e., 64 bytes. This construction ensures that the output of the function is random and the randomness depends solely on the cryptographic strength of Blake2b. Since this function needs to be executed only once, it has a negligible impact on the performance of the overall *Rig* construction.

3.3.6.2 Design of BlakePerm Function

We provide the design considerations for the function BlakePerm before the description of the design.

Design considerations for BlakePerm function The DRAM memory latency is the limiting factor for the entire design of *Rig*. Initialization and copying data takes



BlakeExpand $[x] = \alpha$ where $\alpha = \alpha_0 \parallel \alpha_1 \parallel \dots \parallel \alpha_{127}$, $x_i = (x \parallel i)$ for $0 \leq i \leq 127$
and $x = \text{pwd} \parallel \text{binary}_{64}(\text{pwd}_l) \parallel s \parallel \text{binary}_{64}(s_l) \parallel \text{binary}_{64}(n) \parallel \text{binary}_{64}(l)$

Figure 3.9: Function BlakeExpand

over 70 percent of the total run-time. In order to improve the overall performance, one trivial optimization would be to increase the size of chunks in which the read and write operations are performed. The latest high-performance processor offerings from Intel and AMD influenced many of the design decisions as they would be the most common target platform. There are several design considerations like:

- **L1 cache size** This is one of the major factors because the L1 cache has the lowest latency of around 1-1.5ns (as few as 3 clocks). Therefore it is important that the work piece (chunk) fits within this size for high performance in case of a compute-intensive task.
- **L1/L2/L3 cache line size** A typical modern processor has cache line size of 64 bytes. Therefore working in multiple of 64 bytes with preferably aligned memory access is the best strategy. The problem with working with other non-multiple sizes is that there would be a lot of extra accesses and split loads and stores, which will dramatically reduce performance in computation intensive tasks. The Blake2b compression function nicely fits this requirement as it compresses 128 bytes to 64 bytes. The only other requirement is an implementation detail for setting the proper aligned memory access while memory allocation. As a result, in our implementation, we have zero split load/stores.

- **DRAM Latency** The memory latency is the primary limiting factor in algorithms having random reads and writes. DRAM generally has latency values of 250+ clock cycles. One strategy to get around this problem would be to perform reads and writes in larger chunks. If the chunk size is large enough, the performance hit due to latency can become significantly small. We tested against various sizes from 2 KiB to 64 KiB and observed that 16 KiB is a good size, and it also fits the L1 cache. We have, as a result, chosen chunk size of 16 KiB for the H_2 function.

Design of the BlakePerm function The BlakePerm function is a compression function which compresses 16 KiB of data to 8 KiB. It is a two step function as described below.

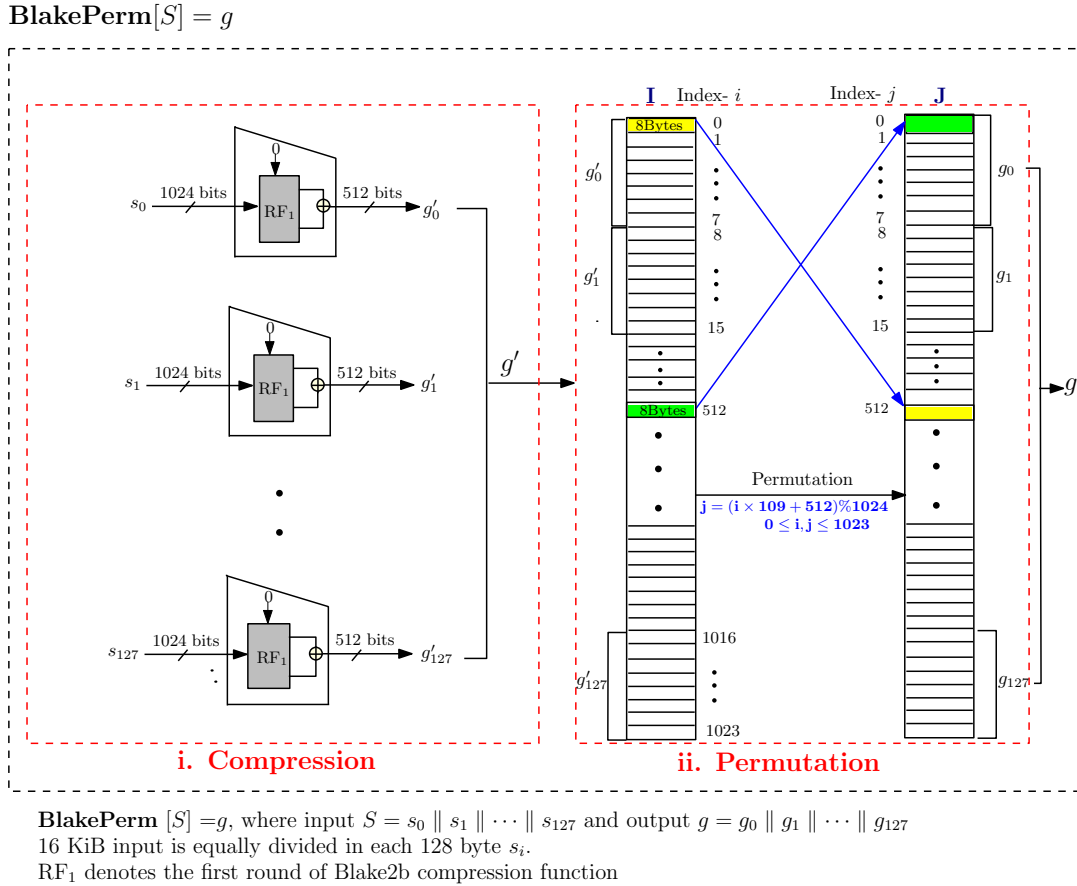


Figure 3.10: Function H_2 implemented as function BlakePerm.

1. **Compression** It compresses the data using a single round Blake2b compression function. A single such compression function compresses 128 bytes to 64 bytes, as a result we need 128 such functions to compress 16 KiB to 8 KiB.
2. **Permutation** A permutation layer is needed to mix the compressed data so that the bit-relations are spread evenly among the output bits over several rounds. Even

though, any random permutation can be chosen in such a scenario, a permutation of the form: output $O_i = (I_i \times A + B) \bmod C$ was chosen, where $0 \leq I_i \leq 1023$. The values $A = 109$, $B = 512$ and $C = 1024$ were chosen carefully after a series of experiments and diffusion tests. The permutation works on words of 8 bytes at a time, as a result, the total addresses would be $8192/8 = 1024$. The value $B = 512$ is the number by which the overall permutation is cyclically rotated, it is half of the total size. The value of A is the most critical, even though the only requirement for a permutation is that A should be co-prime with C . The value of A strongly affects the overall characteristics and it needs to be carefully selected. It takes around 5 rounds for all the output bits to be fully affected by a single change in the input data. Since H_2 function is sequentially applied hundreds of times, the function BlakePerm produces complete avalanche and is cryptographically strong.

3.3.7 Parallelization

The design of *Rig* is sequential and therefore it is impossible to parallelize the overall implementation. As a result we chose to parallelize the H_1 and H_2 functions. The most critical function which affects the performance of the overall design is the H_2 function. The optimized variant *Rig* [BlakeExpand, BlakePerm, Blake2b] takes an input of 16 KiB and compresses it using 128 Blake2b compression functions. These 128 operations can be done in parallel to improve the performance without affecting the overall sequential nature of *Rig*. H_1 can similarly be parallelized but it has a negligible effect on the overall performance.

3.3.8 GPU Resistance

We designed *Rig* to have side-channel resistance, in pursuit of which we had to choose password-independent memory access patterns. Such memory-access patterns are harder to protect against GPU attacks. Modern GPUs have very strict requirements for memory accesses and very small cache sizes per core, as a result small random reads and writes dramatically reduce performance.

While designing H_2 of *Rig* [BlakeExpand, BlakePerm, Blake2b], we chose a permutation which causes reads and writes at significantly varying distances. Combined with the bit-reversal permutation used in *Rig* at the iterative transformation phase, the overall design is hard to parallelize efficiently.

As the H_2 function is pluggable, a new function can be easily added which performs small password-dependent memory accesses and make the design significantly GPU resistant. But, any such function would break the strict side-channel resistance.

3.3.9 ASIC/FPGA Resistance

The *Rig* construction is strictly sequential and is therefore non-parallelizable. The compression function H_2 (BlakePerm) as explained above, can be parallelized. But, the size of the inputs and outputs (16 KiB to 8 KiB) which needs 128 parallel instances of Blake2b

compression function is too large for implementations with a large number of simultaneous *Rig* instances.

Even though there can be a lot of possibilities of implementations with varying numbers of compression functions, the overall space requirement still remains high.

The biggest problem in case of ASIC resistance would, however, come from the memory latency and bandwidth of the DRAM needed for storage of the extremely large state (several hundred megabytes to a few gigabytes). Even though the compression functions consume less power because of their simplicity, the latency and very high memory bandwidth requirements would make parallel implementations on ASIC prohibitively expensive. For example, for a single instance of *Rig* [BlakeExpand, BlakePerm, Blake2b] having $n = 4$ (5 memory passes), and 1 GiB of state, the bandwidth on a standard PC exceeds 7.37 GiB/s as shown in Table 2.

3.3.10 Performance Analysis

The reference implementation of *Rig* has been done in C language on an Intel Core i7-4770 CPU with 16GB RAM at 2400 MHz. For the implementation of the first round of the compression function of Blake2b¹ for the function H_2 , we use AVX2 instructions. Specifically, these AVX2 instructions are used to parallelize the implementation of first-round G-function of Blake2b. The following tables (Table 1 and Table 2) show the performance figure in terms of ‘memory hashing speed’ and ‘DRAM bandwidth’ for different values of parameter n (number of iterations).

The reference implementation is available at: <https://github.com/arpanj/Rig>.

Suggested parameters The performance figures provided in Table 1 and Table 2 show that, as expected, the memory hashing speed for *Rig* [BlakeExpand, BlakePerm, Blake2b] is significantly higher than that of the strictly sequential variant.

Due to the wide spectrum of possible uses, it is very difficult to suggest optimal values for parameters which suit every possible implementation scenario. However, we can suggest values for common applications. For the parameter ‘ n ’ (number of iterations) we suggest values higher than 3. This means that one should have at least four passes over memory (including setup phase). For some scenarios, this may be increased to make low memory attacks prohibitively expensive.

The memory count value (m_c) would depend strongly on the requirement and the actual use-case. For a server client architecture where the clients are expected to have enough free RAM, the value can be set to use few tens of megabytes to a few hundred megabytes. In a mobile environment, this can be further reduced to allow for clients with smaller memories. In the case the algorithm is to be used as a proof-of-work test, large memory requirements of a few gigabytes combined with a large ‘ n ’ value can be set. It is important to keep ‘ n ’ high (as high as 6-10) in case the overall memory cost is very small.

The performance of Scrypt (with suggested parameters [132]) and the results from Table 2 are depicted in Figure 3.11. The graph shows the memory processing rate when consumable

¹The idea of using reduced-round Blake2b is inspired from [73, 27].

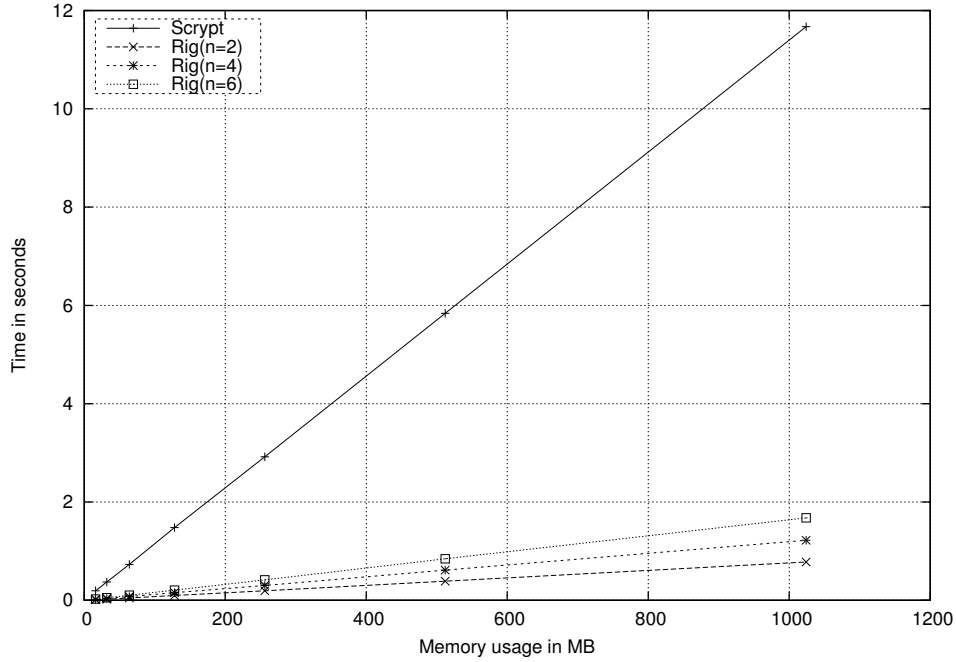


Figure 3.11: Performance of Rig (at different value of n) and Scrypt.

memory to compute the password hash is fixed. The comparison shows that the memory consumption of Scrypt is comparatively small with time. Scrypt takes approximately 6 seconds for 512 MB memory while *Rig* at $n = 2$ and $n = 4$ takes approximately 0.389 seconds and 0.613 seconds respectively.

3.3.11 Security Analysis

Rig satisfies the basic requirement of a non-invertible design for password hashing because of the following reasons: (i) the iterative use of underlying primitive, the (secure) cryptographic hash function and (ii) the initial hashing of password with random salt and other parameters and the final use of salt with chaining data. This makes recovering password from the hashed output quite challenging.

Table 1: Performance of RIG [Blake2b, BlakeCompress, Blake2b]

1)RIG[Blake2b, BlakeCompress, Blake2b]:Memory Hashing Speed(MiB/s)								
M =>	15 M	30 M	60 M	120 M	240 M	480 M	960 M	
n = 2	681	684	663	675	681	674	659	
n = 4	383	383	381	388	388	391	392	
n = 6	268	270	267	275	272	258	270	
n = 8	211	210	210	212	210	201	209	
n = 10	174	173	170	173	170	174	170	
Memory Bandwidth (GiB/s)								
n = 2	3.552	3.566	3.458	3.517	3.551	3.515	3.436	
n = 4	3.596	3.595	3.575	3.642	3.640	3.666	3.677	
n = 6	3.638	3.665	3.624	3.737	3.693	3.500	3.666	
n = 8	3.740	3.731	3.736	3.758	3.728	3.562	3.707	
n = 10	3.826	3.789	3.721	3.796	3.737	3.811	3.736	

Table 2: Performance of RIG [BlakeExpand, BlakePerm, Blake2b]

2)RIG[BlakeExpand, BlakePerm, Blake2b]:Memory Hashing Speed(MiB/s)								
M =>	64 M	128 M	256 M	512 M	1 GiB	2 GiB	4 GiB	
n = 2	1377	1345	1307	1326	1315	1312	1318	
n = 4	858	846	829	845	835	838	833	
n = 6	621	621	617	618	606	610	619	
n = 8	500	485	481	490	485	489	489	
n = 10	403	398	399	402	404	404	402	
DRAM Memory Bandwidth (GiB/s)								
n = 2	6.728	6.575	6.389	6.478	6.429	6.412	6.439	
n = 4	7.547	7.441	7.295	7.431	7.347	7.374	7.329	
n = 6	7.888	7.898	7.847	7.858	7.709	7.750	7.873	
n = 8	8.309	8.070	8.003	8.152	8.072	8.123	8.126	
n = 10	8.282	8.179	8.205	8.251	8.291	8.293	8.265	

n = no. of iterations, performance figures averaged over 20-iterations.

Another important point is the simple, sequential and symmetric design of the scheme. The simplicity makes it easy to understand and sequential design makes the parallel implementation hard and prevents significant speed up by the use of multiple processing units.

The flexibility of the design and the independence of the selection of memory parameter from time parameter makes it unique from existing constructions.

3.3.12 Resistance Against Low Memory Attack

Attacker's approach An attacker running multiple instances of *Rig* may try to do the calculations using a smaller part of the memory (low memory) or almost no-memory (memory-free) to reduce the memory cost per password guess. This approach may allow parallel implementations of independent password guesses, utilizing almost all the available processing cores. This may not give an advantage over single password hash computation but may increase the overall throughput of password guessing as compared to the legitimate implementation of the algorithm. Next, we explain how feasible the low-memory or memory-free attack approach is, from the attacker's point of view.

Attack scenario Varying the required storage values We emphasize that the goal of analyzing the complexity of low memory attack is to show the approximate impact on the overall processing cost to implement the algorithm *Rig*. Our construction needs to store two arrays a and k as shown in Figure 3.6. Therefore we try to compute the time complexity when most of these array values are not stored. The cost of calculation for the values of the array a is dominated by the cost of the array k . Therefore, for the simplicity of the evaluation, we consider the calculation cost for array k .

To vary the required storage at each iteration, we assume that **we store t consecutive values**, $0 \leq t \leq m - 1$, of both the arrays at iterative transformation phase. This assumption is without loss of generality as we can easily calculate the index value of array k from the bit-reversal permutation explained in section 3.3.1. We also store the hash chaining values after each iteration.

Effect of bit-reversal permutation on low memory scenario We use the bit-reversal permutation to shuffle the access of the array k . The effect of this yields exponential complexity for the low memory scenario. This is because at every step we update the values of the array k and each updated value depends on all previous values. Let at iteration i , $1 \leq i \leq n$, $k[j]$, $0 \leq j \leq m - 1$, is the required value that is not stored. Then we need to compute the value $k[j]$ at all previous $i - 1$ iterations and as the access was not sequential, it is difficult to calculate the exact complexity. Hence, we compute the expected time complexity of a password hashing for memory constrained scenario.

Low memory attack complexity: The algorithm *Rig* can be computed with time complexity $O((n + 1)mr)$ and space complexity $O(m)$ where $2m$ is the required number of stored values, n is the number of iterations used and r is the number of rounds. An attacker using reduced memory storage (i.e., 0 to $m - 2$ stored values) will require a time complexity

of $O(r \times m^{n+1})$ for a single password computation.

Analysis of low memory attack complexity For the legitimate implementation of the algorithm, *Rig*, we need to store $m = 2^{m_c}$ values of arrays a and k which are created at the setup phase and repeatedly accessed and updated at each iteration i of the iterative transformation phase (see Figure 3.6). Our goal is to analyse the extra cost incurred when we do not store the array values, and calculate them on the fly at each iteration i where $1 \leq i \leq n$. Specifically, we analyze the time complexity by varying the possible storage (0 to m) of the array values. At each iteration i of the iterative transformation phase, we apply bit-reversal permutation (Algorithm 3.5) on m indices of the array k which we denote by $(1, 2, \dots, m) \rightarrow (br[0], br[1], \dots, br[m-1])$ and access the output of the permutation sequentially. It is easy to calculate this permutation for all n -iterations in advance. We calculate the overall cost of password hashing depending on whether a value of array k , let, $k[br[j]]$, $0 \leq j \leq m-1$ is stored or not. As mentioned before, to compute the complexity we store first t consecutive values of array k (where $0 \leq t \leq m-1$) that are required for corresponding hash calculations at iterative transformation phase. We also store the hash chaining values after each iteration except the last one, i.e., if the implementation uses n -iterations then we store $n-1$ such values. Figure 3.12 shows the graphical view of the low memory scenario with an example where we store two consecutive values of the arrays required for corresponding hash calculations, shown in red. Other $m-2$ values that are calculated on the fly and the corresponding hash calculations are shown in green. The general approach is explained below.

We apply the **law of total expectation** to estimate the expected running time for a password hashing, conditioning on the indices of the array k . This is because the calculation cost is very high when we do not store the values of required indices of k and is the most influential parameter in the overall attack complexity. Therefore, we calculate the probability of a value at a particular index of the array k of being stored when we assume t consecutive values are stored. We also calculate the probability of a value at an index not being stored when $(m-t)$ values are not stored. Further, in case a specific index of the array k is not stored, then we estimate the expected cost to evaluate this element at each iteration i .

We know that the total required indices are m for the array k . Out of these m values, we store t values while the remaining $m-t$ values are not stored. Therefore, the probability of a stored and not-stored index is given by

$$\begin{aligned} \Pr[\text{a value of an index is stored}] &= \frac{t}{m} \\ \Pr[\text{a value of an index is not stored}] &= \frac{m-t}{m} \end{aligned}$$

For $t = 2$ we store only 2 values and $m-2$ values are remained un-stored. Then, the above probabilities are $\frac{2}{m}$ and $\frac{m-2}{m}$ respectively as shown in Figure 3.12.

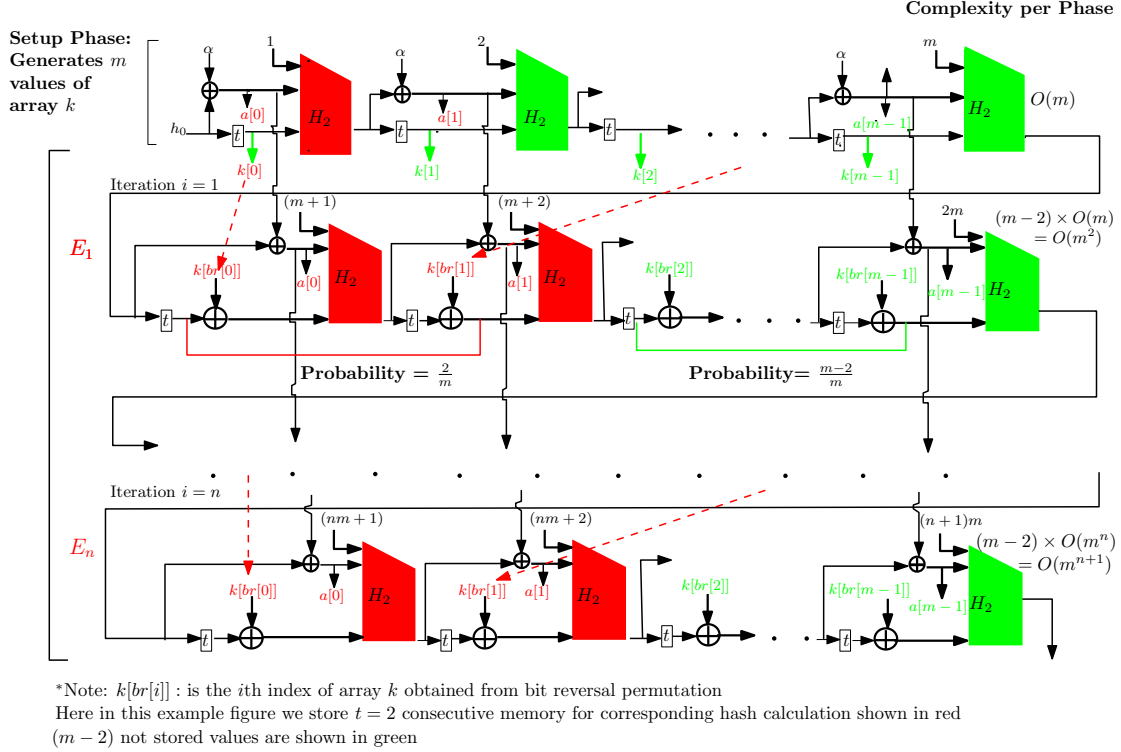


Figure 3.12: Graphical representation of the calculation for the low memory complexity of *Rig*.

To apply the law of total expectation we apply the following concept.

At the setup phase we perform m hash calculations to store m values of array k . Therefore, for these m calculations the complexity of the setup phase is $O(m)$. In our construction, at each iteration i , $1 \leq i \leq n$, we access the values that are calculated at the previous iteration $i - 1$, i.e., every next value is dependent on its previous value. Therefore at the first iteration of iterative transformation phase, if we need to access an element of array k that is not stored, we need to perform maximum m calculations of setup phase for that value. Therefore, at iteration $i = 1$ the total cost of calculation is $(m - t) \times O(m) = O(m^2)$ (as maximum m calculations are required for all $m - t$ not stored values). Similarly, at iteration $i = 2$, for all $m - t$ not stored values of array k it is required to calculate maximum m hashes of setup phase to generate the initial values. Then it is required to check when those values were updated at $i = 1$. At $i = 1$, the maximum calculations are: $(m - t) \times O(m) = O(m^2)$, and similarly at $i = 2$ the total calculations are: $(m - t) \times O(m^2) = O(m^3)$ and so on. This way we estimate the cost of calculation after each iteration of iterative transformation phase shown in Figure 3.12 as ‘Complexity per phase’. Now at each iteration we estimate the expected cost of calculation for each value $k[br[j]]$ of array k where $0 \leq j \leq m - 1$. We

denote the expected cost at iteration i as E_i . If a value is already stored then the expected cost of calculation, i.e., $E(\text{cost when value is stored})=1$ but if the value is not stored then the expected cost will be the complexity till the previous iteration as the required value is dependent on all its previous values. So the expectation at each iteration is as follows:

$$\begin{aligned}
E_1 &= \sum_{j=0}^{m-1} Pr[\text{value at } k[br[j]] \text{ is stored}] \times [\text{cost of calculation when } k[br[j]] \text{ is stored}] \\
&\quad + Pr[\text{value at } k[br[j]] \text{ is not stored}] \times E[\text{cost of calculation } k[br[j]] \text{ is not stored}] \\
&= \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m) \right) \\
E_2 &= \sum_{j=0}^{m-1} Pr[\text{value at } k[br[j]] \text{ is stored}] \times [\text{cost of calculation when } k[br[j]] \text{ is stored}] \\
&\quad + Pr[\text{value at } k[br[j]] \text{ is not stored}] \times E[\text{cost of calculation } k[br[j]] \text{ is not stored}] \\
&= \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m^2) \right) \\
&\quad \dots \dots \dots \dots \dots \dots \dots \\
E_n &= \sum_{j=0}^{m-1} Pr[\text{value at } k[br[j]] \text{ is stored}] \times [\text{cost of calculation when } k[br[j]] \text{ is stored}] \\
&\quad + Pr[\text{value at } k[br[j]] \text{ is not stored}] \times E[\text{cost of calculation } k[br[j]] \text{ is not stored}] \\
&= \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m^n) \right)
\end{aligned}$$

The total cost E after n -iterations:

$$\begin{aligned}
E &= m + E_1 + E_2 + \dots + E_n \\
&= m + \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m) \right) + \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m^2) \right) + \dots \\
&\quad \dots + \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m^k) \right) + \dots + \sum_{j=0}^{m-1} \left(\frac{t}{m} + \left(\frac{m-t}{m} \right) \times O(m^n) \right) \\
&= m + nt + \left(\frac{m-t}{m} \right) \sum_{t=0}^{m-1} [O(m) + O(m^2) + \dots + O(m^n)] \\
&= m + nt + \left(\frac{m-t}{m} \right) O(m^{n+1})
\end{aligned}$$

Conditioning on the values of t we get the following complexities.

Case (i) : $t = 0$ implies the case of memory-free attack where the attacker does not use any memory.

$$\text{The expected cost is} = m + O(m^{n+1}) \equiv O(m^{n+1})$$

Repeating for r -rounds, the complexity of the attack is $O(r \times m^{n+1})$.

Case (ii) : When $1 \leq t \leq m - 1$, i.e. the case when the attacker stores some of the memories.

$$\text{The expected cost is} = \left(\frac{m-t}{m}\right)O(m^{n+1}) \equiv O(m^{n+1})$$

Repeating for r -rounds, the complexity of the attack is $O(r \times m^{n+1})$

Case (iii) : $t = m$ implies the legitimate implementation of the algorithm.

$$\text{The complexity is} = m + nm \equiv O(n+1)m$$

Repeating for r -rounds, the complexity is $O(n+1)mr$.

Therefore, the memory-free attack complexity of *Rig* is $O(r \times m^{n+1})$ where r is the number of rounds.

3.3.13 Resistance Against Collision Attack

In the design of *Rig* (see Figure 3.13) we define three different functions, H_1 , H_2 and H_3 . The input of H_1 is x where x is the concatenation of password, 64-bit value of password length, salt, 64-bit value of salt length, 64-bit value of n (number of iterations) and 64-bit value of the output length of password hash. The output of H_1 is α . The function $\mathcal{H}_2$ signifies the repetitive computation of function H_2 at setup phase and iterative transformation phase (see Figure 3.6) and generates the output c which is the output of iterative transformation phase. Therefore the inputs of $\mathcal{H}_2$ are α , m_c and n , where m_c is the number of memory count and n , the number of iterations used. Finally H_3 takes the concatenation of a 64-bit value which is the function of m_c and n , output of $\mathcal{H}_2$, the value salt and 64-bit value of 2^{m_c} and produces the output of password hash. Here we are considering round $r = 1$ (w.l.o.g). This is because, different values of round, say r and r' implies collision of H_3 .

Theorem 3.3.1 *Collision for Rig means*

- i. collision for H_1 , or*
- ii. collision for $\mathcal{H}_2$ when $\alpha \neq \alpha'$ and $(m_c, n) = (m'_c, n')$ for two different password hash computations, where $m_c = m'_c$ = the memory count and $n = n'$ = number of iterations, or*
- iii. collision for H_3 .*

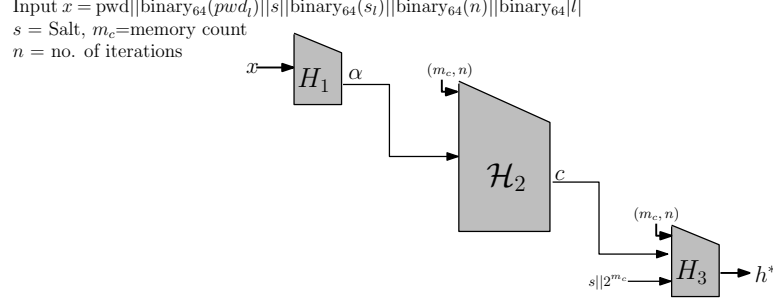


Figure 3.13: $\text{Rig}[H_1, H_2, H_3](x, m_c, n, s)$, where $\mathcal{H}_2$ signifies repetitive use of H_2 .

Proof We analyse the collision of *Rig* with five possible cases as shown in Figure 3.14. Specifically, we include all possible conditions that results in collision of H_1 or collision of $\mathcal{H}_2$ or collision of H_3 which implies the overall collision of *Rig*.

- CaseI. Collision $\text{Rig}[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n') \implies$ Collision H_1 : The construction of *Rig* is such that if we get collision at H_1 for two different inputs say, x and x' and if $(s, m_c, n) = (s', m'_c, n')$ then it implies collision of $\mathcal{H}_2$ which implies collision at H_3 , i.e., collision of $\text{Rig}[H_1, H_2, H_3]$.
- CaseII. Collision $\text{Rig}[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n')$ and $\alpha \neq \alpha' \implies$ Collision $\mathcal{H}_2$: For two different inputs x, x' if $\alpha \neq \alpha'$ then collision of $\mathcal{H}_2 \implies$ collision of H_3 when $(s, m_c, n) = (s', m'_c, n')$ for respective inputs.
- CaseIII. Collision $\text{Rig}[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n')$, $\alpha \neq \alpha'$ and $c \neq c' \implies$ Collision H_3 : For two different inputs x, x' if $\alpha \neq \alpha'$ and $c \neq c'$ and if $(s, m_c, n) = (s', m'_c, n')$ then collision $\text{Rig}[H_1, H_2, H_3] \implies$ collision H_3 , i.e., collision is due to H_3 .
- CaseIV. Collision $\text{Rig}[H_1, H_2, H_3]$ if $(s, m_c, n) \neq (s', m'_c, n') \implies$ Collision H_3 : For two different inputs x, x' if $\alpha \neq \alpha'$ and $c \neq c'$ and if $(s, m_c, n) \neq (s', m'_c, n')$ then collision $\text{Rig}[H_1, H_2, H_3] \implies$ collision H_3 .
- CaseV. Collision $\text{Rig}[H_1, H_2, H_3]$ if $x = x'$ and $m_c \neq m'_c \implies$ Collision H_3 : For two different password hash calculations if $m_c \neq m'_c$ for the same input x then collision of $\text{Rig}[H_1, H_2, H_3]$ is for collision at H_3 .

Therefore these five cases describe how collisions of *Rig* implies collision of H_1 or $\mathcal{H}_2$ or H_3 . Hence *Rig* is collision resistant if H_1 or $\mathcal{H}_2$ or H_3 are collision resistant. $\square$

3.3.13.1 Instantiation of $\text{Rig}[H_1, H_2, H_3]$ for Reference Implementations

Collision resistance of Blake2b: For our reference implementation we use Blake2b. This choice is motivated for several reasons. Blake2b is an improved version of Blake which is one of the finalists of SHA-3 competition. Further, in terms of speed of hashing, it among the

most efficient hash functions in the SHA-3 competition. The significant differences between Blake and Blake2b are that Blake2b uses fewer rounds (12 instead of 16), provides optimized (in terms of speed) rotations in G-function, uses fewer constants (8-word constants instead of 24). It is mentioned in [35] that, “Based on the security analysis performed so far, and on reasonable assumptions, it is unlikely that 16 rounds are meaningfully more secure than 12 rounds”. Despite several years of cryptanalysis, there is no significant attack on Blake. A recent paper on the cryptanalysis of Blake2b [33] shows that no collision or preimage attacks exist against Blake2b. In fact, the best differential against Blake2b covers 3.5 rounds and has a complexity of 2^{480} . Therefore Blake2b is quite secure against collision and preimage attacks, properties which we need in our design. For further details on attacks on compression function and permutation of Blake2b, we refer the reader to [33]. Hence, the use of full round Blake2b is enough to claim security of our construction against attacks utilizing collisions or preimages of the underlying hash function.

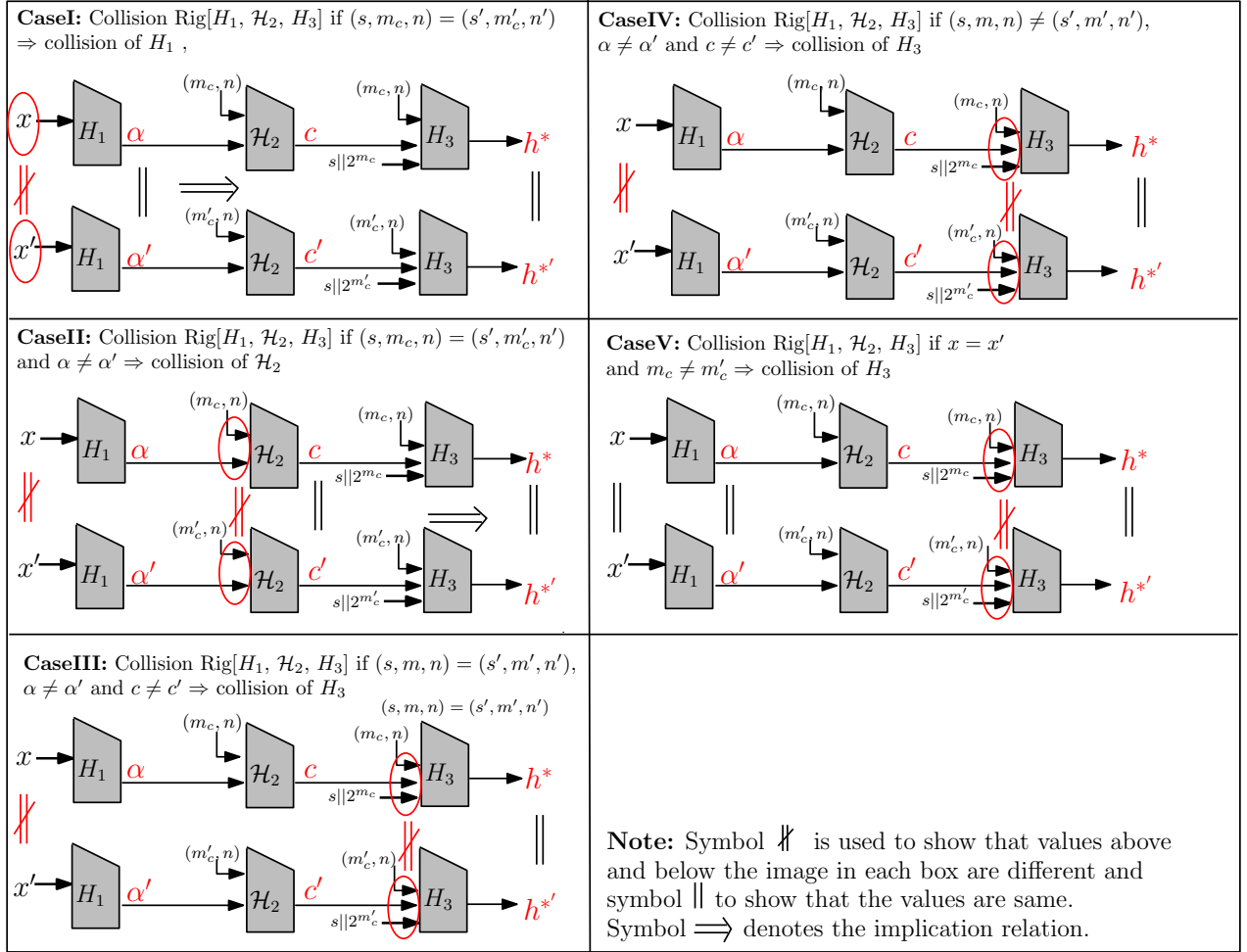


Figure 3.14: Five cases showing collisions for $\text{Rig}[H_1, H_2, H_3]$.

- **Rig [Blake2b, BlakeCompress, Blake2b]**

Instantiation of H_1 and H_3 with Blake2b: We use full round Blake2b for the functions H_1 and H_3 . We need collision resistance of both the functions for the security of *Rig*. Use of Blake2b allows us to claim collision resistance of H_1 and H_3 [33].

Instantiation of H_2 with BlakeCompress: The function $\mathcal{H}_2$ as shown in Figure 3.13 signifies the repetitive computations of the function H_2 and inputs of $\mathcal{H}_2$ are α, m_c, n . Theorem 3.3.1 shows that collision of $\mathcal{H}_2$ implies collision of *Rig* only if H_1 gives two distinct outputs say, α, α' for two different inputs $x \neq x'$ while corresponding values $(m_c, n) = (m'_c, n')$. We use first round of Blake2b compression function for the implementation of H_2 as shown in Figure 3.7.

For finalization phase, we omit the feed-forward xor'ing as explained in Blake2b (see Figure 3.8). This choice of implementation is to improve the performance, but it does not preserve the onewayness as there is no feed-forward xor'ing. In this case, it would appear that backward calculation is easy as H_2 computation is then reversible. But due to the design of *Rig* it is very difficult to perform these backward computations as input values of H_2 are arrays a and k and depends on their values at previous iteration (see Figure 3.6). Therefore, we only have the freedom to guess the values of the array a and k to move backward with a reverse calculation for the last iteration, say $i = n$. Last iteration n will fix the values of iteration $(n - 1)$ as values are dependent. So we lose the freedom of guessing the values from iteration $(n - 1)$ and onwards and the backward computations become hard. Hence omitting the feed forwardness can compromise the onewayness but the design of the scheme provides no security loss. Now consider two distinct inputs of $\mathcal{H}_2$ as $\alpha = H_1(x)$ and $\alpha' = H_1(x')$ for $x \neq x'$.

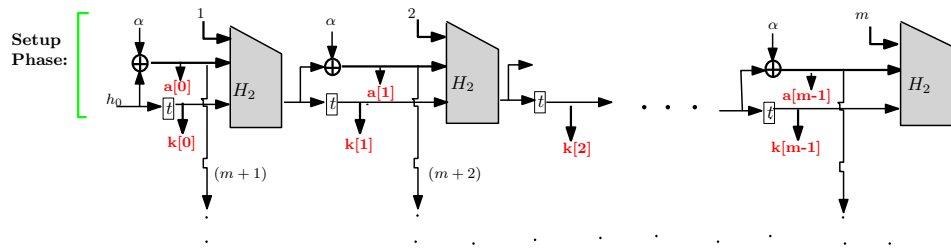


Figure 3.15: Setup phase uses m inputs generated from α .

Figure 3.15 shows that $m = 2^{m_c}$ number of computations of H_2 of the setup phase are generated from these different α or α' and it will be difficult to find collision everywhere as values generated from two different inputs are usually expected to be different. The values of array a and k are generated from α or α' and influence further calculations.

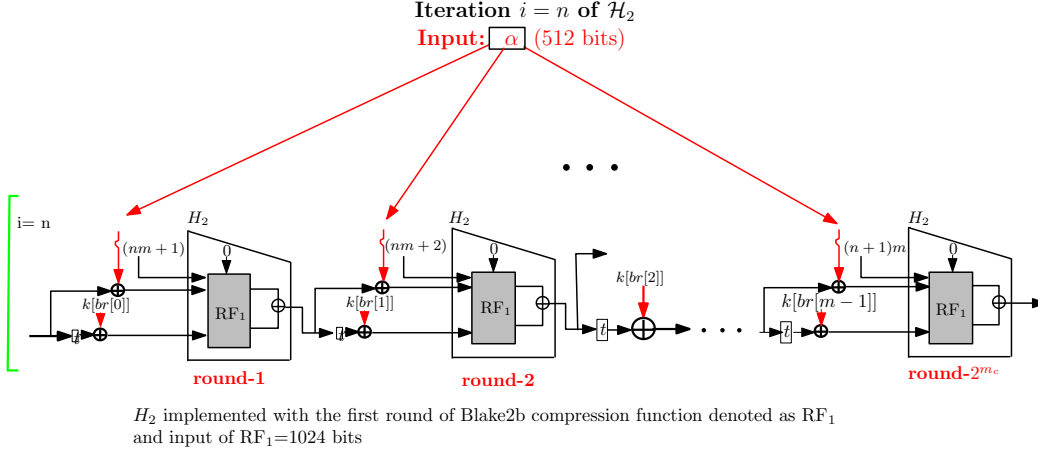


Figure 3.16: H_2 implemented with RF_1 at iteration $i = n$.

Now, consider the last iteration of iterative transformation phase, i.e., the last layer of 2^{m_c} computations of H_2 as shown in Figure 3.16. All these H_2 calls use single round Blake2b and are generated from the same value α or α' . The input values influenced by α or α' are shown in red color and most of them will have nonzero difference. We can visualize this scenario as similar to the Blake2b construction, instead of 12 rounds using 2^{m_c} rounds. Use of a comparatively large number of rounds provides enough security. Since we expect the parameter $m_c \geq 10$, we will have 1024 rounds of Blake2b compression function. Hence, we can expect $\mathcal{H}_2$ to be collision resistant.

Therefore the choice of reduced round Blake2b does not affect the collision resistance of the design.

- **Rig [BlakeExpand, BlakePerm, Blake2b]**

Instantiation of H_1 with BlakeExpand and H_3 with Blake2b: As explained in section 3.3.6 we use 128 individual instances of Blake2b (full hash) appended by a counter for the function H_1 . Therefore a collision for H_1 means collisions for each of the 128 calls to Blake2b. Therefore, collision resistance of H_1 is obtained from the collision resistance of Blake2b.

For the function H_3 we implement Blake2b (full round), therefore H_3 is collision resistant if Blake2b is so.

Instantiation of H_2 with BlakePerm: We implement each H_2 with two step functions. The first function, ‘Compression’ executes 128 instances of first round Blake2b compression function and compresses total 16 KiB input to 8 KiB. The second function, ‘Permutation’ is a permutation that mixes the compressed 8 KiB output to spread the

bit-relations evenly among the output bits (see Figure 3.10).

We can provide similar arguments for the security of $\mathcal{H}_2$ as explained for the sequential variant, Rig [Blake2b, BlakeCompress, Blake2b] above.

Hence, the above choice of implementation is expected to be secure against collision attack.

3.3.14 Resistance Against Length Extension Attack

The length extension attack on hash function works as follows. Hash functions work on blocks of data iteratively. Let the initial value of a hash be $IV=h_0$. A long message $m = m_0\|m_1\| \dots \|m_l$ is processed as follows. First $h_1 = H(h_0, m_0)$ is computed and then the recursion $h_{i+1} = H(h_i, m_i)$ is used for $1 \leq i \leq l$. The final output is h_{l+1} . An attacker, without knowing the intermediate message blocks, may simply append a new block m^* and compute the hash value of the message $m' = m_0\|m_1\| \dots \|m_l\|m^*$, by using one call to the hash function as $H(h_{l+1}, m^*)$. In our construction, the password is not the only input to the hash function. If an attacker attempts to append the password by any text, the value of α will change and it will affect all subsequent blocks. Thus the length extension attack is not feasible in our design.

3.3.15 Resistance Against Cache-Timing Attack

As discussed in section 3.3.2, our construction accesses array k using a bit-reversal permutation which is password independent. Hence cache timing attack is not possible on our construction. Further, since the only primitive used in our scheme is a secure hash function, the security of our scheme can be formulated in terms of the security of the underlying hash function. With the current state-of-the-art, we have the possibility of using SHA-3 implementation, or even any of the other finalists of SHA-3 competition, which are resilient to side-channel attacks. Thus our scheme resists cache timing attacks.

3.3.16 Resistance Against Denial-of-Service Attack

In computing, a denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users. This is possible by making the server busy injecting lots of request for some resource consuming calculation. It is quite easy if the server uses some slow password hashing technique for authentication. To handle such situations, the server-relief technique can provide some relief to the server from heavy calculations as the client will do the heavy part of the algorithm. This way we can reduce the chances of DoS attacks with slow password hashing schemes.

3.4 Argon2 [45], PHC Finalist

The term ‘memory-hard’ refers to the designs which consume lots of memory while execution. All memory-hard designs of PHC mainly follow two different approaches of memory access but it is difficult to conclude which one is better in terms of security. This is because when the memory access is input (password) dependent, the design is vulnerable to cache-timing attack while the designs with input independent access are vulnerable to time-memory tradeoff attack. There also exists scheme with a hybrid approach which initially follows data-independent memory access and then data-dependent access pattern.

Argon2 specifies two variants, Argon2d which follows input dependent memory access pattern and Argon2i which follows input independent memory access pattern. Both are effective for different use cases. Argon2d for computing cryptocurrencies and Argon2i for password hashing, key derivation for hard-drive encryption. Both the variants are different only at the point of index computation. Therefore following analysis are applicable to both the variants unless it is specified as Argon2d or Argon2i.

3.4.1 Security analysis of Argon2

The design of Argon2 can be visualized with p lanes (rows) and T blocks (columns) computed as below.

$$\begin{aligned}
 H_0 &= \mathcal{H}(d, \tau, m, t, v, \langle P \rangle, P, \langle S \rangle, S, \langle K \rangle, K, \langle X \rangle, X) \\
 M_{i,1} &\leftarrow G(H_0, i \parallel 0), 1 \leq i \leq p \\
 M_{i,2} &\leftarrow G(H_0, i \parallel 1), 1 \leq i \leq p \\
 M_{i,j} &= G(M_{\phi(i,j)}, M_{i,j-1}), 1 \leq i \leq p, 1 < j \leq T \\
 \text{Output} &\leftarrow \mathcal{H}(M_{1,T} \parallel M_{2,T} \parallel \cdots \parallel M_{p,T}) \quad \dots(1)
 \end{aligned}$$

where $M_{i,j}, 1 \leq i \leq p, 1 \leq j \leq T$ denotes each block of 8192 bits, H_0 is the value mainly derived from password P and salt S with other parameters, $\mathcal{H}$ is the Blake2b hash function and G is the compression function based on the Blake2b permutation. The function $\phi(i, j)$ computes the index and its computation is either input-dependent (Argon2d) or input-independent (Argon2i).

Parallelize implementation preventing tradeoff attack: To utilize p-cores of a system the design may implement p-parallel computations still providing resistance to tradeoff attack as follows. Let suppose the memory is organized as $p \times T$ matrix of blocks where each element (block) is represented as $M_{i,j}$ where $1 \leq i \leq p$ and $1 \leq j \leq T$ and the values are computed as explained above. Now divide each lane (row) in l segments $I_1, I_2, \dots, I_l$ and the index, $\phi(i, j)$ is defined in a way which ensures the dependency between lanes. Specifically $\phi(i, j)$ uses the following rules for the computation of each $M_{i,j}$:

1. $\phi(i, j)$ does not refer to $M_{i,j-1}$.

2. For each lane of first segment I_1 , $\phi(i, j)$ refers to the same segment (i.e., I_1).
3. For segments I_k , $k > 1$, $\phi(i, j)$ may refer to I_k of the same lane and all lanes of segments I_j , $j < k$.
4. First block of I_k can not refer to the last block of I_{k-1} in any lane.

If the above idea is implemented with p threads with all available memories and l synchronization points, it takes time l and space pl as each thread computes all l segments. An adversary can use less memory than required and for example, suppose the adversary uses less memory when computing last segment I_l sequentially from lane 1 to lane p . Then required time is $(l-1)+p$ and the time penalty is $\frac{(l+p-1)}{l} = (1 + \frac{p-1}{l})$. The memory penalty is $(1 - \frac{p-1}{pl})$ where memory used $p(l-1) + 1$ by p threads. This provides the time-memory product as:

$$TM_{new} = TM \left(1 - \frac{p-1}{pl}\right) \left(1 + \frac{p-1}{l}\right).$$

For $2 \leq p, l \leq 10$ the value is always between 1.05 and 3.

Collision attack: One of the important properties for password hashing designs is collision resistance. Argon2 uses hash function Blake2b for initial H_0 and output computations as shown in equ (1). Specifically, for the computation of H_0 the variable length P and S are prepended with their length $\langle P \rangle$, $\langle S \rangle$ respectively which ensures the absence of equal input strings and processed through collision resistant Blake2b, so no collision should occur at this stage. For the remaining computations compression function G is used which should be collision resistant to prevent internal block collisions. The function G is defined as:

$$G(X, Y) = P(Z) \oplus Z, \quad Z = X \oplus Y.$$

where P is a permutation based on the 2-round Blake2b permutation. The function G is not proved to be collision resistant or preimage-resistant. An attacker cannot control the inputs to G and the starting two blocks are different for each row as shown in the specification provided as equ (1). While computation of the blocks it is defined that first block of a segment I_k can not refer to the last block of segment I_{k-1} in any row to avoid block collisions. It is proved that if G is collision resistant and 4-generalized-birthday-resistant then all the blocks $M_{i,j}$ are different. Therefore overall scheme is assumed to be collision resistant when G is collision resistant.

Cache-timing attack Argon2i implements data-independent memory access implementing the function $\phi(i, j)$ which provides the index of a memory to be accessed. The detailed procedure is as follows.

The indexing function $\phi(i, j)$: For each reference block $M_{i,j}$ a set of indices $\mathcal{R}$ is computed. It first partitions the memory matrix into l segments and provides the matrix Q of size $p \times l$. To compute the index for a block in a segment $Q_{r,s}$ (say) the set $\mathcal{R}$ includes blocks according to the following rules:

1. All blocks of segments $Q_{r',*}$, where $r' < r$ and $*$ takes all possible values from 0 to $p - 1$.
2. All blocks of segments $Q_{r',*}$, where $r' > r$ and $*$ takes all possible values from 0 to $p - 1$ if it is the second or later pass over the memory.
3. All blocks of the segment $Q_{r,s}$ (current segment) except for the previous block;
4. For the first block of the segment $\mathcal{R}$ does not include the previous block of any lane.

Let $R = |\mathcal{R}|$. The blocks in $\mathcal{R}$ are numbered from 0 to $R - 1$ according to the following rules:

1. Blocks outside of the current segment are enumerated first;
2. Segments are enumerated from top to down, and then from left to right: $Q_{0,0}$, then $Q_{1,0}$, up to $Q_{p-1,0}$, then $Q_{0,1}$, then $Q_{1,1}$, etc.
3. Blocks within a segment are enumerated from the oldest to the newest.

Argon2i runs G^2 — the 2-round compression function G in the counter mode, where the first input is all-zero block, and the second input is constructed as:

$$r(4 - \text{byte}) \parallel l(4 - \text{byte}) \parallel s(4 - \text{byte}) \parallel i(4 - \text{bytes}) \parallel 0(1008 - \text{byte})$$

where r is the pass number, l is the lane, s is the slice, and i is the counter starting in each segment from 0. Then we increase the counter so that each application of G^2 gives 256 values of J . Then the value $J \pmod R$ determines the block number (index) from $\mathcal{R}$.

As the value, J , is derived from the predefined parameter values, Argon2i provides input independent memory access. This method ensures cache-timing attack resistance and it is easy to verify by following the definition of cache-timing attack for password hashing provided by Catena designers. It says that if considered an implementation of a password hashing algorithm, where data is read from or written to a password-dependent address $a = f(pwd)$, pwd = password. If for another password pwd' , one gets $f(pwd') \neq a$ and the adversary could observe whether the implementation accesses the data at address a or not, then it could use this information to filter out passwords candidates. Under certain circumstances, timing information related to a given machine's cache behavior may enable the adversary to observe which addresses have been accessed. Therefore input independent memory access does not provide such relation and hence prevents the cache-timing attack.

Tradeoff attack: Argon2i is vulnerable to tradeoff attack due to data-independent addressing. The tradeoff attack tries to figure out the cost of implementation especially on dedicated hardwares e.g., ASIC. The first generic tradeoff attack for password hashing schemes is provided by Dmitry et.al. They termed it as ‘generic ranking tradeoff’ method. The attack follows the idea below.

Table 3.1: Computation/read penalties for the ranking tradeoff attack, taken from [45]

Memory fraction ($1/q$)	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
1 pass	1.7	3	6.3	16.6	55
2 passes	15	410	19300	2^{20}	2^{25}
3 passes	3423	2^{22}	2^{32}		

- Split the memory into segments of length q .
- Store $M[kq]$ for all k .
- Store all $\phi(i)$ and the access complexity
- Store the T/q highest access complexities. If M_i refers to a vertex from this top, store M_i

The efficiency of tradeoff depends on the properties of function ϕ and the segment partition. Applying ranking algorithm on recommended 3-pass of Argon2i provides the result listed in Table 3.1.

3.5 Evaluation Criteria of PHC

The status report for first round selection of nine password hashing schemes is provided by the PHC panel. According to the report the selection criteria includes the following while evaluating the candidates without following a specific order.

- Defense against GPU/FPGA/ASIC attackers
- Defense against time-memory tradeoffs
- Defense against side-channel leaks
- Defense against cryptanalytic attacks
- Elegance and simplicity of design
- Quality of the documentation
- Quality of the reference implementation
- General soundness and simplicity of the algorithm
- Originality and innovation

The other objective behind the selection are the relevance of the algorithm for most applications of a password hashing scheme, from web service authentication to client login or key derivation and diversity in types and applications.

On July 20th, 2015 PHC panel announced Argon2 as a basis for the final PHC winner. Moreover, they gave special recognition to

- Catena, for its agile framework approach and side-channel resistance.
- Lyra2, for its elegant sponge-based design, and alternative approach to side-channel resistance.
- Makwa, for its unique delegation feature and its factoring-based security.
- yescrypt, for its rich feature set and easy upgrade path from scrypt.

3.5.1 Selection Process

The selection of Argon2 algorithm must not be an easy decision by the PHC panel as before the winner announcement there were open discussions in the PHC mailing list about the decision making when the panel members announced Argon2 and Catena as their top choices. Argon2 has benefited from its high performance and better tradeoff defense.

Security The PHC panel mainly checked the security properties of the finalists, based on their published design documents and cryptanalysis done during the competition, and on the security proofs provided by both the designers and other researchers. Not all designs are cryptanalysed in a same depth and it is not even possible as each algorithm has its own design logic and requires different approaches of analysis and sometimes less analyzed algorithms may less understood. However, following points can be summarized from available data and analysis.

- No finalist has a published attack that, in any real sense, threatens its practical security, and most of the finalists (Argon2, Catena, Lyra2, Makwa, Pomelo) has provided detailed security analysis while others provided initial analysis which shows that the security of the whole design is based on the security of the underlying cryptographic primitive.
- One of the important security requirements is resistance to time-memory tradeoff, i.e., the algorithm should impose significant time penalty (preferably exponential) if less memory is used for implementation than intended. This helps to measure the memory hardness of the design. The first time-memory tradeoff attack for password hashing designs was proposed by Dmitry et.al., in 2014 but the attack was applied and analysed only on two of the finalists (on initial versions) Catena and Lyra2 not on others. From the available analysis, Argon2i provides the best tradeoff resistance with recommended parameters. It is also shown that Lyra2 is more tradeoff resilient to Catena v.1 (i.e., Catena-BRG, recommended for memory constraint devices).

- There are basically two approaches of the design for memory-hard password hashing algorithms. One approach which follows input (password) independent memory access and Argon2i, Catena comes under that. Another category where memory access is input (password) dependent. Hybrid approach (ignoring the order) accommodates both these approaches and Lyra2, Pomelo comes under this category. battcrypt (uses password-dependent memory access), Makwa and Parallel are low memory designs which can be a good choice on resource constraint implementations. Password independent memory access prevents cache-timing attack and therefore preferred for password hashing. This is because cache-timing attack helps to filter out candidate password providing the timing information of cache access. However, this approach opens the path for tradeoff attack as pre-computation is possible. Therefore it is still a matter of research whether hybrid approach is better than input-independent accessing. The designs under input-independent category proving tradeoff resistance which very much depends on the deterministic function defined for memory access. Catena-BRG uses bit-reversal permutation which provides less memory hardness than Catena-DRG (performance-wise not efficient) while Argon2i generates the values as the output from Blake2b compression function which is supposed to work as PRNG. Therefore Argon2i is preferred in cache-timing attack resistant category.
- The cryptanalysis performed on Argon2, Catena, and Lyra2 appears to have a great deal of depth, while the cryptanalysis on other design are mainly based on the analysis provided by the corresponding authors. The cryptanalysis performed throughout the competition on the designs applies (not with significant difference) to their final versions. Update on Catena-v1 was tweaked more significantly to address the time-memory tradeoff attack. Even though there was no attack reported on Argon, the design is totally changed from initial Argon to final Argon2 which accommodates most of the requirements in an efficient way and tradeoff resistance is among the significant contribution while being fastest design among its category.

Argon2i provides security against side channel (cache-timing) attack and significant resistance to tradeoff attack. Both are important requirements for a good password hashing design as main objective of all such designs is to prevent the optimization over brute force attack. The report is clearly documented with security and performance analysis. The design Argon2 is based on single primitive cryptographic hash function (Blake2b). According to the PHC panelist, considering the existing analysis and the hardware threats for password hashing based on the existing trend in hardware efficiency Argon2 will remain secure in future for significant time period (if memory constraint follows the existing trend for hardware) [the attack [98] on Argon2 came into existence after the selection].

Performance All design provides their performance analysis and mostly implemented with AVX2 or SSE instructions. This is because one of the basic requirements for password hashing algorithms implemented for user authentication is to be slow to resist password guessing attack, at the same time to be having fast response time to avoid user perception of the delay.

Therefore resource consuming (time and memory) algorithms should accommodate fastest existing instructions to balance both these requirements and the measure of the bandwidth of the implementation of a particular design provides the good measure of performance. All includes software implementation of their respective design. It is difficult to provide the hardware implementations in both Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) for memory hard designs even though most of the designs provide justification if requires high cost for hardware implementations.

- Not all nine finalists are memory hard. battcrypt, Makwa and Parallel consume comparatively less memory, therefore not suitable for resource full environment and specifically preferable for light weight systems or memory constraint environments. Catena-BRG is also good for such use cases.
- Most of the designs (Argon2, Catena, Lyra2) use Blake2b as their underlying primitive. Blake2b is an improved version of Blake which is one of the finalists of SHA-3 competition. Further, in terms of speed of hashing, it is among the most efficient hash functions in the SHA-3 competition. The significant differences between Blake and Blake2b are that Blake2b uses fewer rounds (12 instead of 16), provides optimized (in terms of speed) rotations in G-function, uses fewer constants (8 word constants instead of 24). Recent cryptanalysis of Blake2b shows that no collision or preimage attacks exist against it.
- Without using parallelized implementation Argon2i provides the bandwidth of 2.6 GB/sec which best among all the finalists. Argon2 parallelize its computation in tradeoff-resilient way, thus increases time-memory and time-area product if the memory is reduced.
- One-pass Argon2 is 50% faster than Scrypt and two-pass, two-thread Argon2 is as fast as Scrypt.

3.5.2 Selection Conclusions

There are overall three categories of algorithms as finalists. The designs for password hashing with low memory includes Makwa and Parallel. The memory-hard algorithm preventing side-channel attack includes Argon2i, Catena. The third category is the memory-hard designs that follow either hybrid or data dependent memory access and includes battcrypt(not memory hard in true sense), Lyra2, Pomelo, Pufferfish and yescrypt. Among the designs, battcrypt, parallel, Pufferfish do not provide any specific advantages over other candidates. Makwa is based on an unique idea of squaring modulo a composite (Blum) integer n and security depends on factoring n but provides limited diversity. Lyra2 is vulnerable to cache-timing attacks and not better than Argon2i performance-wise. POMELO defines a design based on three different functions which are the underlying primitives, but not include depth in security analysis. It also provides partial resistance to cache timing attack. The design yescrypt does not follow simplicity.

Argon2i is faster than (about three times) Catena-Dragonfly (as reported) and Catena-Dragonfly-Full is 5-6 times slower. Argon2i supports multi-threading and best tradeoff attack security (later analysis on Argon2i shows that it provides weak memory hardness [98]) among other analysed algorithms. The computational penalty is around 2^{12} while using 1/2 of the memory and 2^{22} for 1/3 of memory and so on. As provides tradeoff resilient parallelism, an adversary with a number of parallel cores would spend much more area for the cores than for the required area if the memory is reduced by the factor of 3. Therefore, the time-area product for Argon2i can be reduced by 3 at best. The design is also simple and properly analysed. It is better than the existing memory hard design, Scrypt, in terms of security (cache-timing attack and garbage collection attack resistant), in terms of design simplicity and high performance.

3.6 Summary

In this chapter, we first discuss the Password Hashing Competition and then our password hashing scheme *Rig* which is based on secure cryptographic hash functions. We show how our design *Rig* supports all necessary and commonly used features of a password hashing including client-independent updates and server relief technique. The flexibility in the choice of the two important parameters (memory count and number of iterations) enhances the scope of its implementation. *Rig* can be implemented in various software and hardware platforms including resource constraint devices. The detailed description of *Rig* shows how the PHC requirements can easily be accommodated in a password hashing scheme. We also discuss the PHC winner Argon2 [45] with selection criteria of PHC to provide a better understanding of the current-state-of-the-art of password hashing.

Chapter 4

Cryptographic Module Based Approach for Password Hashing Schemes

Password Hashing, as discussed in the previous chapter, is the technique of performing one-way transformation of the password. One of the requirements of password hashing algorithms is to be memory demanding to provide defense against hardware attacks. In practice, most Cryptographic designs are implemented inside a Cryptographic module, as suggested by National Institute of Standards and Technology (NIST) in a set of standards (FIPS 140). A cryptographic module has a limited memory and this makes it challenging to implement a password hashing scheme (PHS) inside it.

In this chapter, we propose a novel approach to allow a limited memory cryptographic module to be used in the implementation of a memory demanding (high) password hashing algorithm. We also evaluate the suitability of all the first round entries of the Password Hashing Competition (PHC) to be implemented with a cryptographic module following the proposed approach. We graphically show that the submissions to the PHC can be securely implemented with a cryptographic module following our suggestion. To the best of our knowledge, this is the first attempt in the direction of secure implementation of password hashing algorithms.

The roadmap for this chapter is as follows: We first review cryptographic module and level of security provided by it as documented by NIST in FIPS 140 series in Section 4.1. We then present the generic approach to use a cryptographic module in securing password hashing in Section 4.2. The security analysis of the proposed approach is presented in Section 4.3. Subsequently, the analysis of the submitted PHC designs with respect to the proposed approach is included in Section 4.4. In the concluding section, we summarize the entire chapter. The original contribution of this thesis is from Section 4.2 to Section 4.4.

4.1 Password hashing and Cryptographic Module

Since the user chosen passwords usually have low entropy, it is quite feasible for the adversary to guess the passwords and to compute the corresponding password hashes. Hence, easy to implement “offline attack” against the PHS which is already explained in Chapter 2. There can also be “online attacks” where the attacker actively measures information leakage at the time of password hash computations. While evaluating the security, we can not even trust the authentication server. The natural question in this case is if it is feasible to provide security even when the server is compromised. It is challenging to provide the same level of security as could be achieved when the server was secure, however, it is indeed possible to increase the level of difficulty of the adversary to reveal the sensitive data. With the goal of restricting the active adversary as well as the off-line attacker (who compromises password hash file), the approach taken in this chapter is the use of a cryptographic module for the password hash computation. In a series of standards (FIPS 140), NIST already recommends the use of a cryptographic module for stream ciphers, block ciphers, authenticated encryptions etc. Password hashing algorithms are a natural candidate for the use of a cryptographic module to implement them and we feel that this module will be required, at least for some use cases, in the future.

A cryptographic module is a device (plus the requisite software to make it work) with a secret key stored inside it and it is assumed that the device itself is secure against all types of adversaries. Cryptographic modules are usually secured physically and the attacker is not allowed access to the input of the device. The only part where the attacker can mount an attack is the output of the module. In general, it is hard to keep a big and bulky device from attackers and hence such a device usually contains a small amount of memory. This memory could range from few bytes for tiny devices to few kilobytes for larger ones. Therefore, a cryptographic module is commonly known as a hardware security module. In practice, it is expected to be a secure hardware that can generate, store and protect cryptographic keys. This specialized hardware is needed in some applications since the expected level of security is higher than what could be expected from its software implementation counterpart. The common form in practice are smartcards or security tokens etc [32].

As mentioned above, a major limitation of a cryptographic module is its limited memory. On the other hand, all password hashing schemes require significantly large memory, by design to be GPU-resistant as explained in Chapter 3. Therefore it is not always feasible to implement and execute an entire password hash computation on a cryptographic module. Although some schemes can be implemented into a small memory device, most password hashing schemes do not. In order to fit these schemes in a small memory module, the schemes require tweaks in them. However, a security scheme cannot be tweaked easily and a careful analysis of the tweaked scheme needs to be undertaken to ensure that the security proof of the original scheme does not fail in this case. We consider possible approaches to provide security of PHS using cryptographic module and we assume the cryptographic module as a secure hardware/software that protects the key used in the proposed protocol. We also analyze first-round submissions of PHC [16] with respect to their suitability to be implemented inside this small memory crypto-device.

The Federal Information Processing Standards (FIPS) specify requirements for cryptographic modules for various security algorithms in FIPS 140 series. The standard FIPS 140-2 [8] defines a cryptographic module as: “the set of hardware, software, firmware, or some combination thereof that implements cryptographic logic or processes, including cryptographic algorithms, and is contained within the cryptographic boundary of the module”.

The standard [8] specifies the security requirements satisfied by a cryptographic module when implemented within a security system protecting sensitive but unclassified information. It provides four increasing, qualitative levels of security that are intended to cover the wide range of potential applications and environments in which cryptographic modules may be employed. The available cryptographic services for a cryptographic module are based on many factors that are specific to the application and the environment. The four security levels as defined in [8] are as follows.

1. **Security Level 1:** It provides the lowest level of security by supporting basic security requirements specified for a cryptographic module (e.g., use of at least one approved algorithm or approved security function). This level does not require any specific physical security mechanisms.
2. **Security Level 2:** This level enhances the physical security mechanisms of Level 1 by adding the requirement for tamper-evidence, which includes the use of tamper-evident coatings or seals or for pick-resistant locks on removable covers or doors of the module.
3. **Security Level 3:** This level enhances the physical security mechanisms required at Level 2. It attempts to prevent the intruder from gaining access to critical security parameters held within the cryptographic module. This level is intended to provide physical security requirements with a high probability of detection and response to attempts at physical access, use or modification of the cryptographic module.
4. **Security Level 4:** This level provides the highest level of security defined in the standard. The physical security mechanisms provide a complete envelope of protection around the cryptographic module. The implemented mechanisms intent of detecting and responding to all unauthorized attempts at physical access. There is a very high probability of detection of penetration of the cryptographic module from any direction and the detection results in the immediate ‘zeroization’ [8] of all plaintext critical security parameters.

The Cryptographic Module Validation Program of the NIST actively checks the accuracy of the applied algorithms, therefore we get the assurance that the claimed security is indeed provided and that the cryptographic module can protect the sensitive information and provide the required security.

4.2 General Approach for Cryptographic Module Based Password Hashing Scheme

From the design restriction of the cryptographic module, it is difficult to implement an algorithm which requires comparatively huge memory for its implementation. Therefore it is ever challenging to provide a complete cryptographic module based approach for password hashing scheme (PHS). Considering this challenge we come to the conclusion that the only possibility is to protect the password. The entropy of the password is commonly weak as we can not force the user to choose a strong one. Therefore another requirement is to increase the entropy. This is possible if we use a large secret key (at least 128 bit) with the password. The responsibility of the cryptographic module is to secure the key and to provide (typically) the server, the one-way transformation of the password operated with key (it can take other parameters as well depending on the algorithm). Then the server can follow further computations of the password hashing algorithm that can be suitably applied with this approach to compute the password hash. The overview of the above procedure is shown in Fig. 4.1.

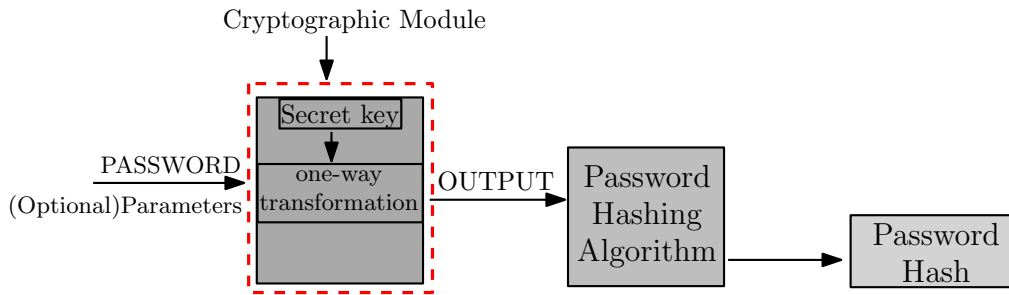


Figure 4.1: Cryptographic module used in password hashing

The Protocol for the general cryptographic module based PHS

1. Password is the input to the cryptographic module (it can take any number of other parameters depending on the password hashing algorithm).
2. Cryptographic module performs a one-way transformation over the password and the secret key (and other parameters if applicable); this output is then provided to the server.
3. Server performs the remaining steps of PHS and produces the password hash.

4.3 Security Analysis

The security of the overall design depends on the security provided by the cryptographic module to protect the password and the security of the password hashing algorithm. For

the resistance from the vulnerability of the adversary, we can assume that guessing a large random key (at least 128-bit) is quite challenging. Therefore predicting the output of the cryptographic module will be difficult.

By using the random secret key of the cryptographic module we increase the entropy of the password and also restrict the adversary if it tries to compromise the data during execution of the password hashing algorithm. Therefore this design approach for password hashing provides enhanced security if the underlying password hashing design is secure. We summarize the overall requirement for the secure implementation of the proposed method as follows:

- **The input password of the cryptographic module** should be secure. One way to achieve this security is by encrypting the password which is protected even if intercepted by the adversary before the computation. In this case we assume that a standard encryption algorithm is used.
- **The internal operation of the cryptographic module** i.e., a secure one-way transformation of the secret key with the password (and with optional parameters if applicable). The security provided by the cryptographic module as explained in [8] ensures the security of the internal operation.
- **The input to the password hashing algorithm** should be the output of the cryptographic module. The level of difficulty to guess this output depends on the algorithm implemented by the cryptographic module and the strength of the secret key. Therefore we can claim that it will be difficult to guess. Hence use of a low entropy password with a secret key to perform one-way transformation will definitely provide better security.
- **The password hashing algorithm** should fulfill the requirements as mentioned in [16] and based on these criteria we can ensure the level of security provided by the scheme.

From the above explanation, we can state that if a password hashing technique follows the above mentioned approach then the proposed system will always provide better security than the usual implementation of a PHS.

4.4 Analysis of Submitted PHC designs with respect to Cryptographic Module-Based Approach

In this section, we analyze the first round PHC submissions [16] to verify whether the use of a cryptographic module is possible without significant modification of the designs. We try to follow a similar approach to provide solutions for the designs, even though other approaches can also be suitable. We provide the graphical view of the designs and highlight (in red) the area that can be implemented in a cryptographic module. We include the security analysis of cryptographic module implementation for few of the designs (at least one from each category

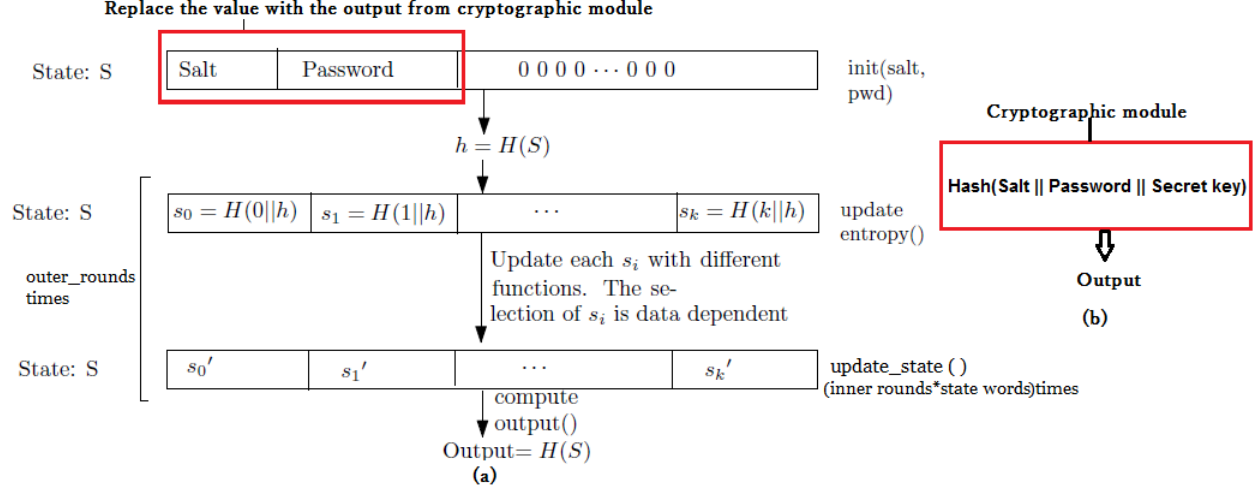


Figure 4.2: (a) Overview of the design AntCrypt (b) The cryptographic module implementation

as discussed in section 4.4.1). It is easy to extend this approach to the remaining designs for which the proof of security is not provided/ commented upon.

- **AntCrypt [77]** The State array of $2^{\text{m-cost}+8}$ bytes of AntCrypt is initialized with salt and password and updated throughout the execution of the algorithm as shown in Fig. 4.2(a). To implement the proposed approach, we can replace the values (salt and password) by the output of the cryptographic module which appends the secret key and performs the one-way transformation as shown in Fig. 4.2(b) highlighted in red. This little modification does not affect the remaining design of AntCrypt as explained in [77].

Security analysis: The security of the design as claimed by the author in [77] depends on the underlying hash function. The design writes the password to the state in the beginning and immediately overwrites it by the output of the hash function. Following our proposal as shown in Fig. 4.2(b), the initial salt and password are overwritten with a pseudo-random output of the cryptographic module. Remaining design uses this pseudo-random value in place of the (usual low entropy) password. Hence suggested modification does not affect the security of the overall design, even replacing the plain-text password with pseudo-random value enhances the security from information leakage.

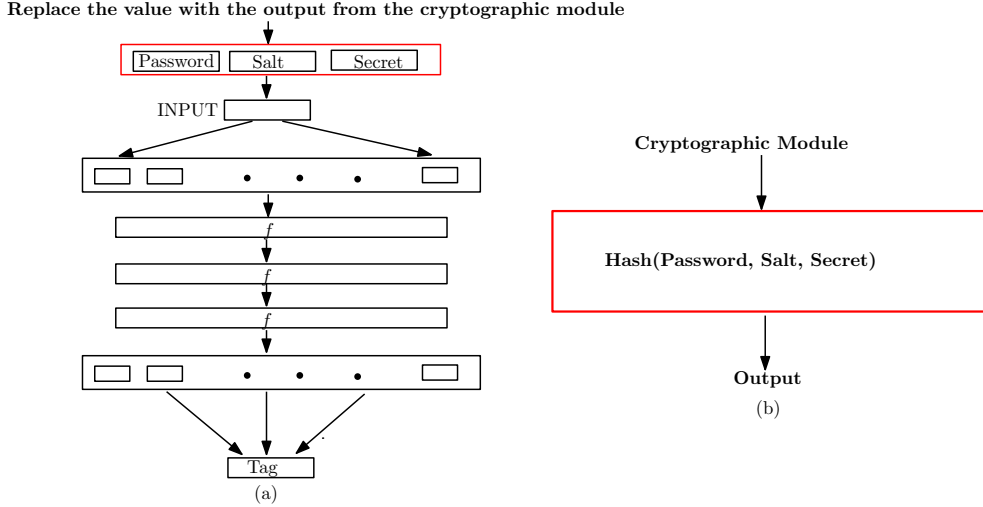


Figure 4.3: (a) Overview of the design Argon taken from [44] , (b) The cryptographic module implementation

- **ARGON** [44] The algorithm implements four phases. The first ‘Padding phase’ takes input password, salt, secret value, other parameters and use zero padding to obtain specified length (32(t-4)bytes, see [44] for more detail) and creates block of data. ‘Initial round’ applies AES to each block. In the next ‘Main round’ the transformations ShuffleSlices and SubGroups are applied alternatively L times. At the end ‘Finalization round’ generates the output tag. We suggest to replace the secret key of padding phase with the secret key of the cryptographic module and to apply the one-way transformation on the values highlighted in red to create the ‘INPUT’ as shown in Fig. 4.3(a). Argon uses the only cryptographic primitive AES. Therefore the use of other primitive will modify the design requirement. In that case, we suggest to use HMAC where the key is the secret key of the module and hash function can be any block cipher (AES) based hash function. This modification does not affect the remaining design of ARGON as explained in [44].

Security analysis: As per the suggestion, the use of HMAC with AES based hash function and the secret key inside the cryptographic module provide better security from information leakage than using the parameters in the plain-text form. This modification does not affect the remaining implementation of the design and the overall claimed security of Argon.

- **battcrypt** [158] The algorithm initializes the value key by taking a hash of salt concatenated with the password. We can use the cryptographic module to initialize this key by performing one-way transformation over the value appended with the secret key as shown in Fig. 4.4, highlighted in red. This modification does not affect the remaining design of battcrypt as explained in [158].

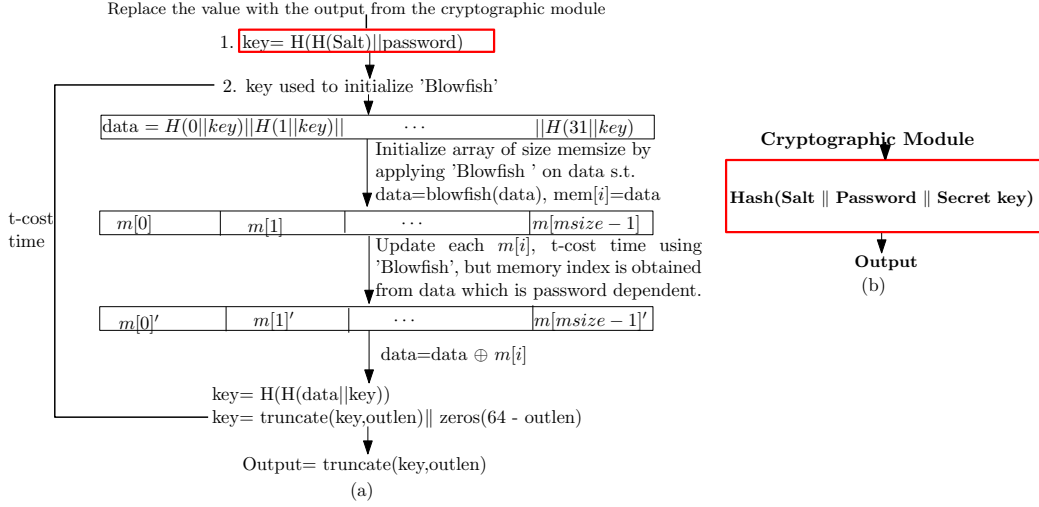


Figure 4.4: (a) Overview of the design battcrypt (b) The cryptographic module implementation

Security analysis: The first step of the algorithm computes hash of the salt and then hash of the resulting value concatenated with password. We suggest to replace the computation with a single hash as shown in Fig. 4.4(b). The one-wayness is preserved by this and memory-hardness and other properties of the whole design remains the same.

- **Catena** [87] The algorithm Catena initializes the value x by performing hash on the concatenation of values tweak t , salt and password respectively. This value x is used to initialize the array of 2^{g_0} elements and g_0 increases by 1 at each following iterations as shown in Fig. 5.3(a). We can use cryptographic module to initialize this x as shown in Fig. 5.3(b). This modification does not affect the remaining design of Catena as explained in [87].

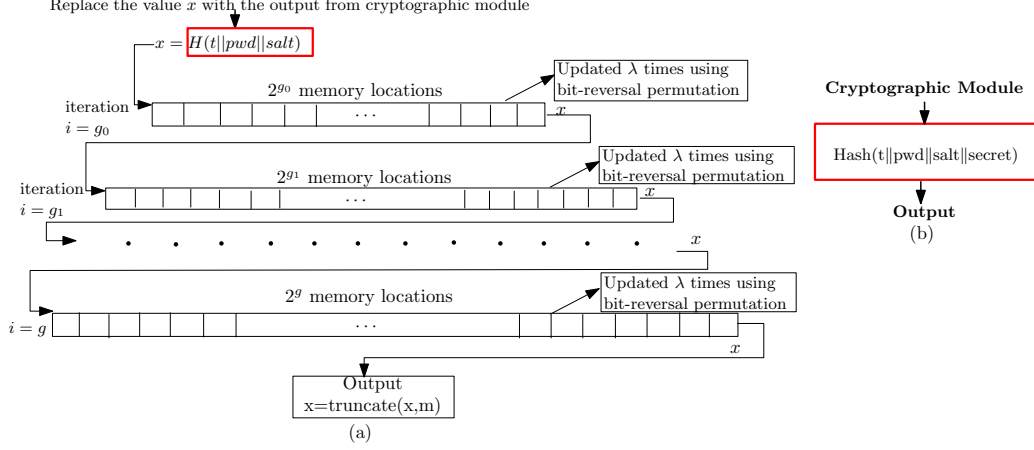


Figure 4.5: (a) Overview of the design Catena (b) The cryptographic module implementation

- **CENTRIFUGE** [28] The algorithm initializes the value seed by computing hash of salt and password and the length parameters as shown in Fig. 4.6(a). We can replace this seed value with the output of the cryptographic module which highlighted in red in Fig. 4.6(b). This modification does not affect the remaining design of CENTRIFUGE as explained in [28].

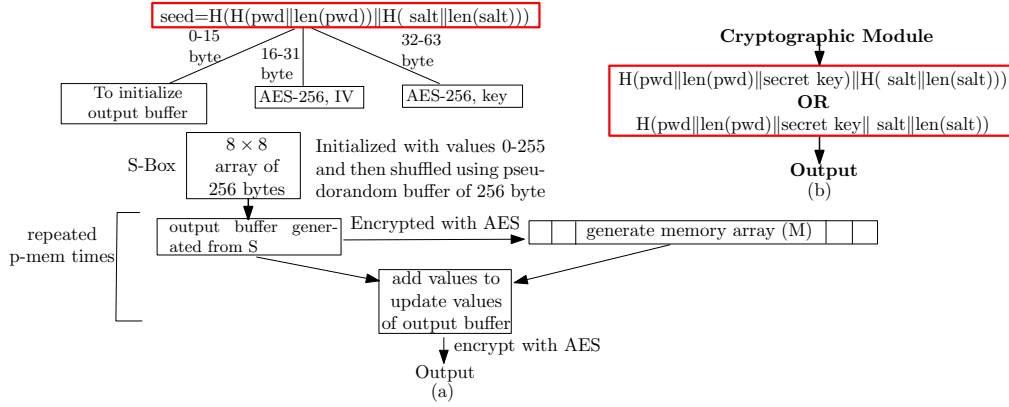


Figure 4.6: (a) Overview of the design CENTRIFUGE (b) The cryptographic module implementation

- **EARWORM** [90] The password and salt is required several times during the whole computation of EARWORM as shown in Fig 4.7(a). Apart from that the algorithm needs a secret value other than the password. This secret value is used to generate a huge array ARENA. To provide the security based on the cryptographic module we can compute the hash of the password with the secret value inside the module and whenever the password is required we can use this hash value. This will be a good

alternative from storing the password and reusing it, but it needs little modification of the design [90].

- **Gambit [134]** The algorithm is based on duplex-sponge construction and initial input of the sponge is the concatenated value of salt and password. This initial computation of password can be implemented inside the module as shown in Fig. 4.8(b). This we suggest due to preserve the sponge based approach of Gambit. In this case, we assume that the state size of the sponge is small enough to fit all required computations inside the low memory cryptographic module. This modification does not affect the remaining design.

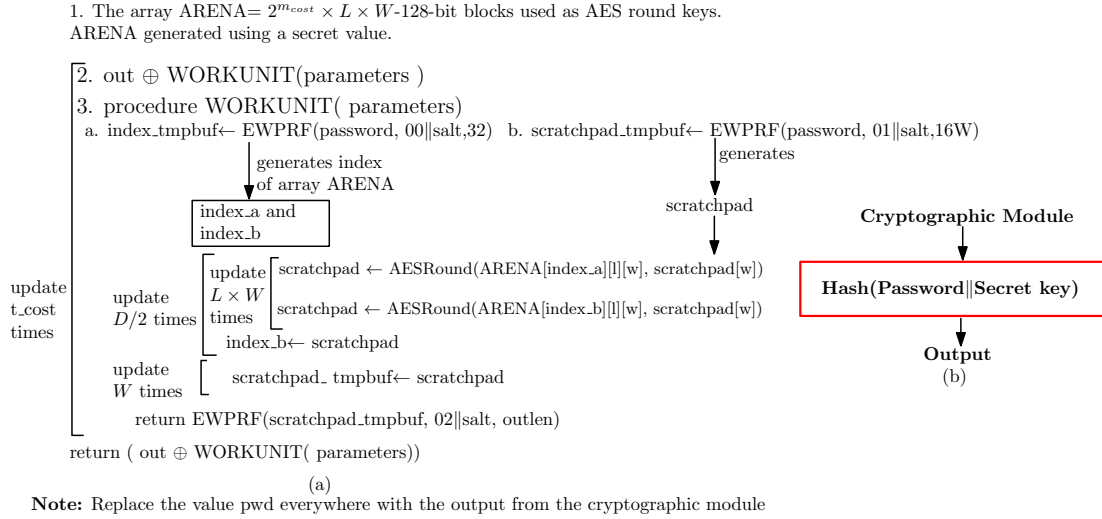


Figure 4.7: (a) Overview of the design EARWORM (b) The cryptographic module implementation

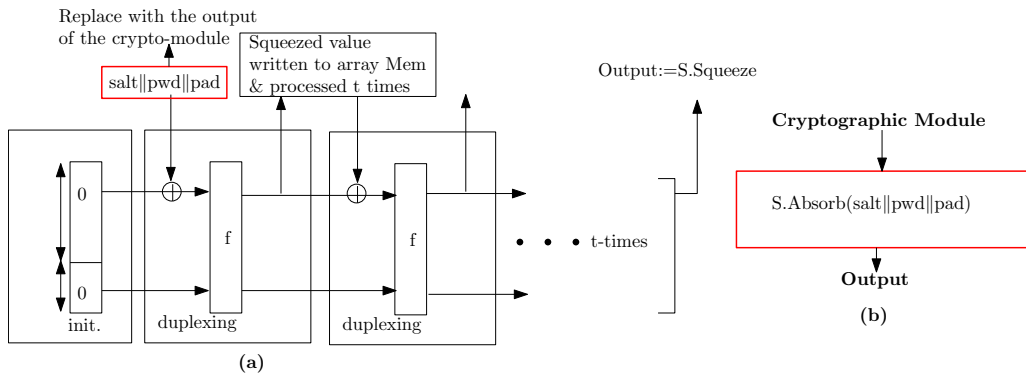


Figure 4.8: (a) Overview of the design Gambit taken from [39], (b) The cryptographic module implementation

- **Lanarea DF [127]** The algorithm has three phases, the setup phase followed by the core phase and the key extraction phase. The setup phase initializes a matrix of $m_cost \times 16 \times 16$ byte which is updated $t_cost \times m_cost \times 16$ times at the core phase. The core phase processes the password and salt through a pseudorandom function and generates an array $(h_0, \dots, h_{31})$ which determines the access pattern of the matrix. This array generation from password can be replaced with the output of the cryptographic module as shown in the Fig. 4.9(b) where F is the pseudorandom function used in the algorithm. As suggested before if the function F is a block cipher then HMAC can be used inside the module to generate the hash, where the key will be the secret key and hash will be the block cipher based hash. Hence the suggested approach does not need modification in the remaining design as explained in [127].
- **Lyra2 [102]** This is a duplex sponge based algorithm which initializes a matrix at the setup phase. The initial computation of this phase performs the sponge absorb with input salt and password. Like the Gambit design, we suggest to implement this initial computation with password to be implemented inside the cryptographic module as shown in Fig. 4.10(b). This modification does not affect the remaining design of Lyra2 [102].

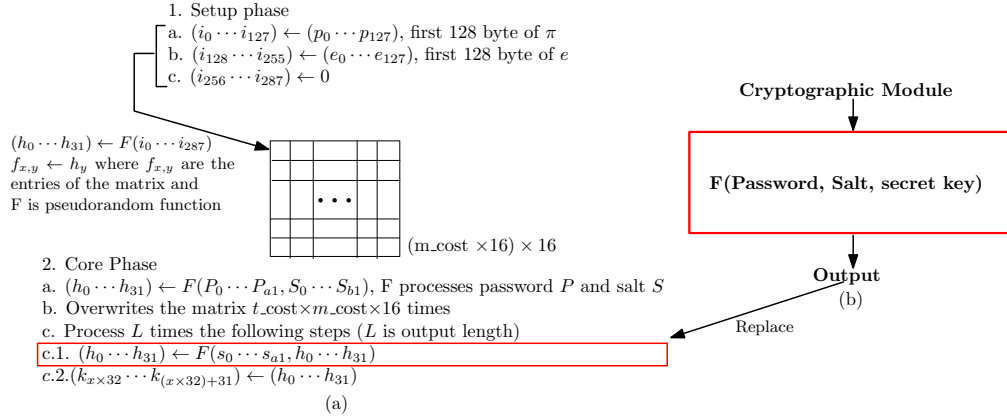


Figure 4.9: (a) Overview of the design Lanarea DF (b) The cryptographic module implementation

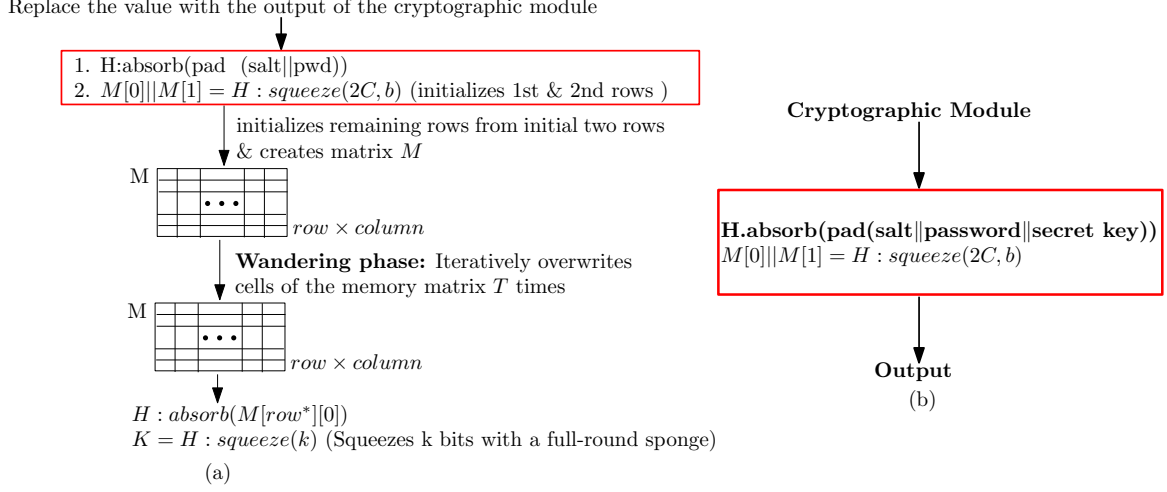


Figure 4.10: (a) Overview of the design Lyra2 (b) The cryptographic module implementation

- **Makwa [135]** The algorithm Makwa is not memory demanding therefore it is possible to implement the whole design inside the cryptographic module. Even if we implement the whole design the required modification is to make the pre-hashing computation compulsory for the password (which is optional for the design). In case of complete implementation inside the module, the final output will be the output of the algorithm Makwa. Therefore this design as explained in [135] can easily support cryptographic module based approach.
- **MCS_PHS [121]** The algorithm implements iterative hash. It initializes the value T_0 by concatenation of password, password length, salt, salt length and other parameters as shown in Fig. 4.12(a). This T_0 is proposed to be a large value but it is the input of a hash function, therefore we can use cryptographic module to calculate T_0 and T_1 . In this case we assume T_0 is small enough to fit inside the module. The suggested approach does not affect the remaining design of MCS_PHS [121].

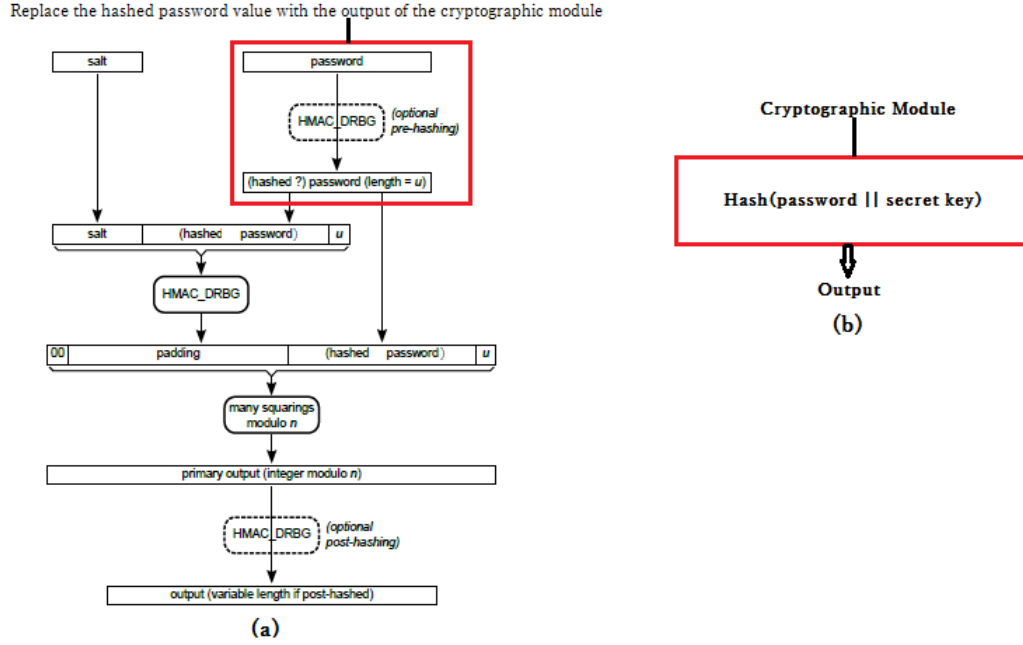


Figure 4.11: (a) Overview of the design Makwa taken from [135], (b) The cryptographic module implementation

- **Omega Crypt (ocrypt)** [79] The algorithm initializes the value Q using password and salt as shown in Fig. 4.13(a). The value Q is of length 771 byte. We can implement the initial three steps inside the cryptographic module as shown in Fig. 4.13(b). Hence the design does not require modification for suggested approach. In this case we assume the value Q is easily adjusted inside the module.

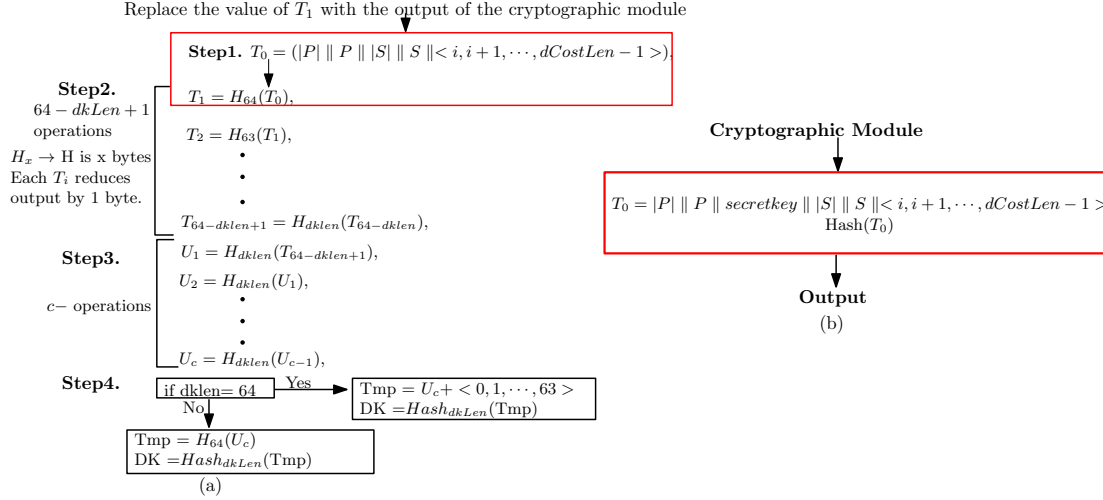


Figure 4.12: (a) Overview of the design MCS_PHS where H_k denotes output length of hash is k byte (b) The cryptographic module implementation

- **Parallel [159]** The algorithm initializes the value ‘key’ performing hash over password and salt as shown in Fig. 4.14. The overall design is not memory demanding. Therefore it is possible to implement the whole algorithm inside the cryptographic module. We have shown one possible implementation in 4.14(b) which does not need any modification of remaining design.

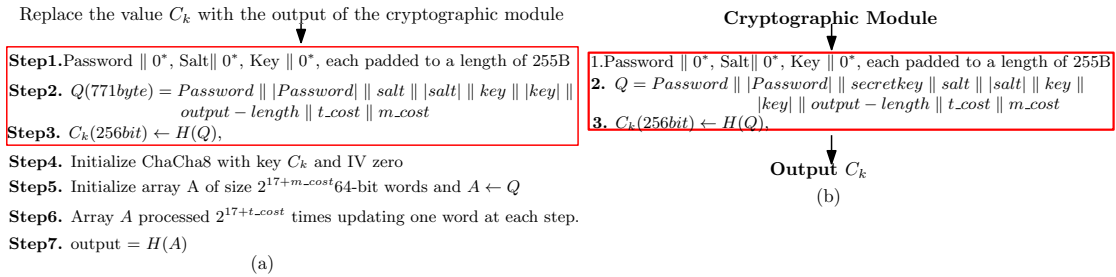


Figure 4.13: (a) Overview of the design ocrypt (b) The cryptographic module implementation

- **PolyPassHash [54]** The algorithm provides a threshold cryptosystem based technique to protect password hashes where the password hash can be verified only if a threshold of passwords are known. It basically aims to prevent an attacker from efficiently cracking individual passwords from a stolen password database. Therefore in this work we are not providing solution for PolyPassHash [54].

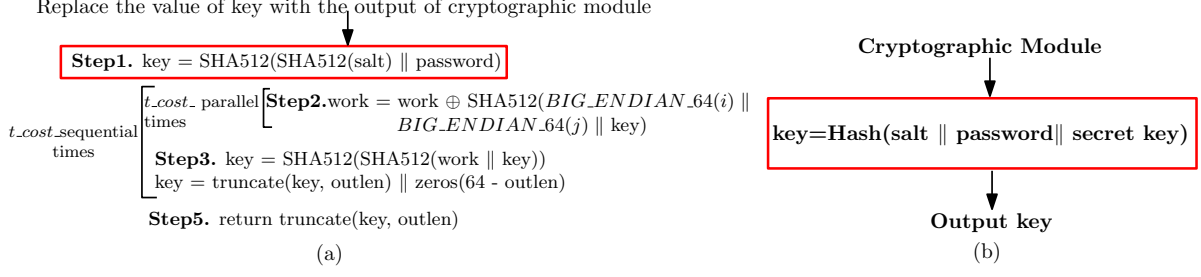


Figure 4.14: (a) Overview of the design Parallel (b) The cryptographic module implementation

- **POMELO [167]** This algorithm uses three different state updation functions that are used to update the state array which is first initialized to zero and then filled with password and salt values as shown in Fig. 4.15(a). To provide cryptographic module based approach we suggest to replace the password and salt values with the output of the module as shown in Fig. 4.15(b). This suggested modification may change the design requirement by the use of an additional cryptographic primitive which is not preferred. Therefore, in this case, we should be specific before selection of the function that can be implemented without affecting the overall design requirements. The suggested approach does not affect the remaining design of POMELO [167].
- **Pufferfish [95]** The algorithm initializes the S-box of size $2^{m_cost+5}512$ bit processing the salt and the password as shown in Fig. 4.16(a). The value password is again used to generate the value `key_hash`. Therefore we suggest to implement the hash of the password to perform inside the cryptographic module and to replace the value password with this hashed value for the computation of the steps 2 and 3 as shown in Fig. 4.16. This modification does not affect the remaining design.

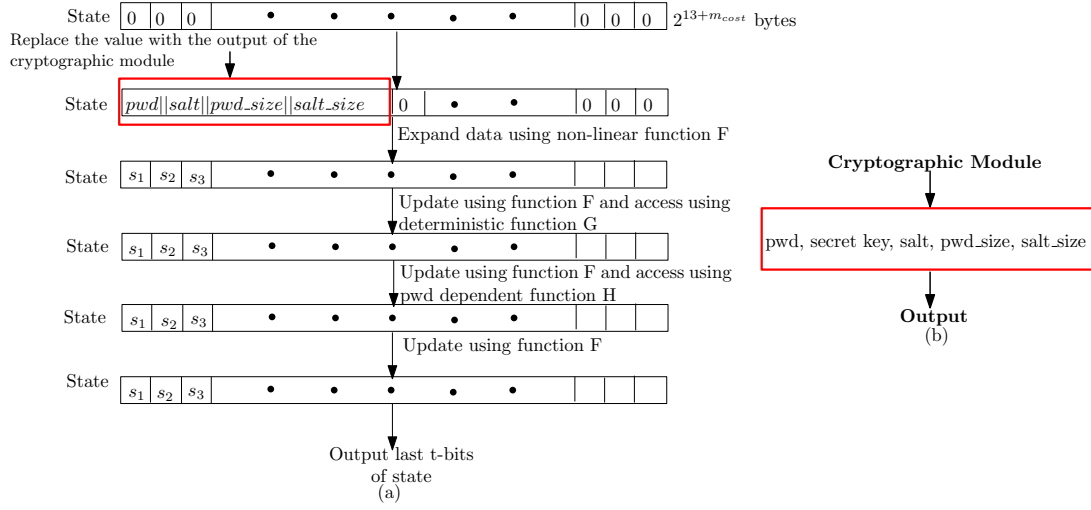


Figure 4.15: (a) Overview of the design POMELO (b) The cryptographic module implementation

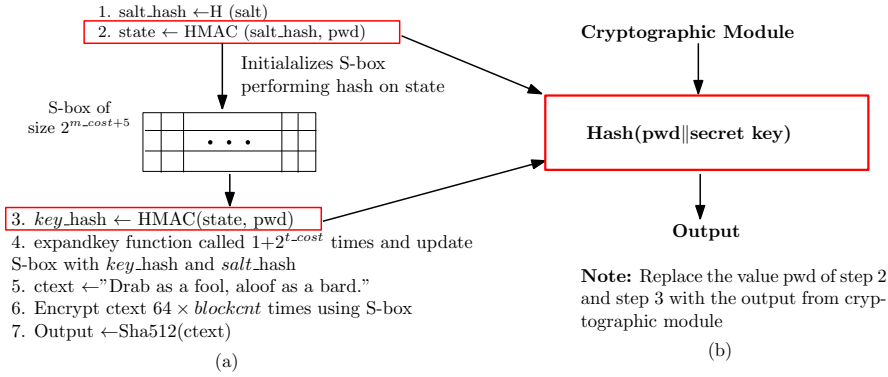


Figure 4.16: (a) Overview of the design Pufferfish (b) The cryptographic module implementation

- **Rig [59]** The initialization phase of this algorithm computes the hash of the value x derived from password, salt and other parameters and produces the value α as shown in Fig. 5.4(a). We suggest to implement this initialization phase inside the cryptographic module and this approach does not affect the remaining design of Rig.

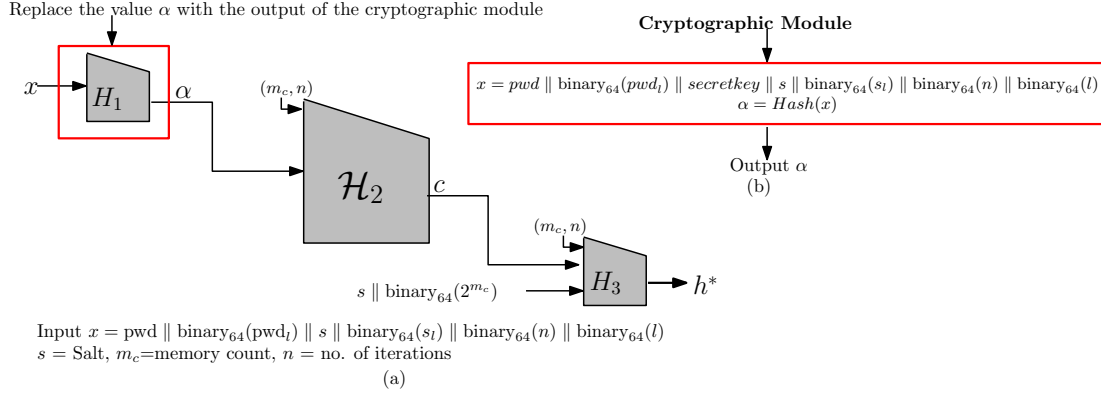


Figure 4.17: (a) Overview of the design Rig (b) The cryptographic module implementation

- **Schvrch [161]** The algorithm initializes the state array of size 2 KB with zero and then fills it with the password, salt and their length parameters and output length. The processing of the steps depends on the values m_cost and t_cost . If these values are greater than zero then the state array is updated t_cost time with function *revolve* otherwise update with function *evolve*. The whole procedure is shown in Fig. 4.18(a). For the cryptographic module based approach, we suggest the state array be initialized with the output of the cryptographic module instead of the plain-text values. In that case, we assume that the size of the array is small enough to be accommodated inside the low memory cryptographic module. The algorithm Schvrch defines different functions inside the design. Therefore it is required to put extra attention for the selection of the function inside the module to maintain the design requirements as explained in [161].
- **Tortuga [143]** The algorithm is based on Turtle design and specifically, the permutation used in this construction is the Turtle algorithm. The steps of the algorithm are enumerated in Fig. 4.19(a). It uses the password at step 2 to compute its length and the password is absorbed into the sponge at step 3. To provide the cryptographic module based approach we suggest to compute the hash of the password with the secret key inside the module and to use the hashed value in place of the password. Therefore our suggested approach does not require major modification in the design of Tortuga [143].

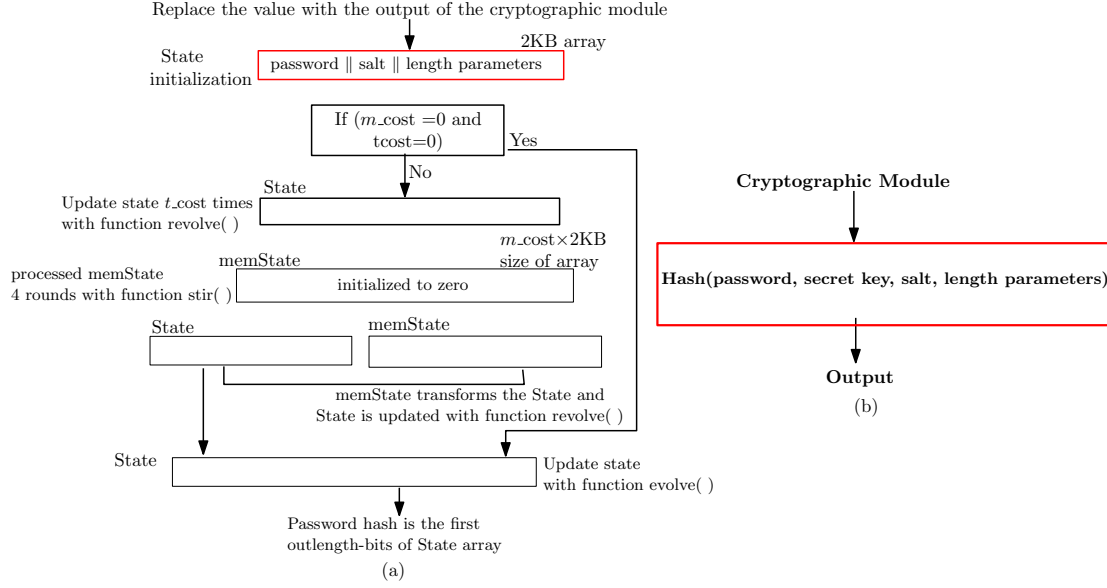


Figure 4.18: (a) Overview of the design Schvrch (b) The cryptographic module implementation

- **TwoCats** [69] The PHC [16] submission TwoCats [69] include two variants, TwoCats and SkinnyCat. We provide the overview of the common part of both these variants as our suggestion easily satisfy both. The algorithm first computes a pseudo-random key PRK from password, salt and other parameters. It initializes a state array of size 256 bits and another array mem of length 2^{m_cost+8} . Fig 4.20 shows that we can implement the generation of PRK inside the cryptographic module. Therefore proposed solution does not require modification of the design as explained in [69].
- **Yarn** [104] The algorithm has five phases and the first phase of it uses the password and generates the final state applying Blake2b as shown in Fig. 4.21(a). The second phase processes the resulting state and expands it to produce pseudo-random data for the subsequent phases. Therefore the first phase can be implemented inside the cryptographic module as shown in Fig. 4.21(b) and this approach does not affect the remaining design of Yarn [104].

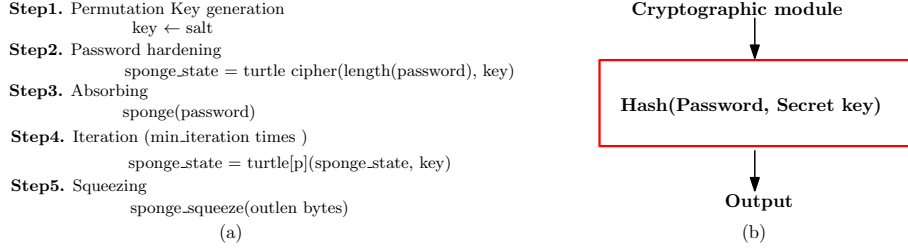


Figure 4.19: (a) Overview of the design Tortuga (b) The cryptographic module implementation

Security analysis: The security of the algorithm is based on the security of BLAKE2b function and the AES function. At the first phase, the algorithm performs repeated hash (Blake2b) on the input password. The suggested solution appends secret key with the password and performs the first phase inside the cryptographic module. This modification enhances the security of password leakage and also preserves the one-wayness. The memory-hardness and other properties of the whole design remain the same. Hence the suggested modification enhance the overall security of the algorithm.

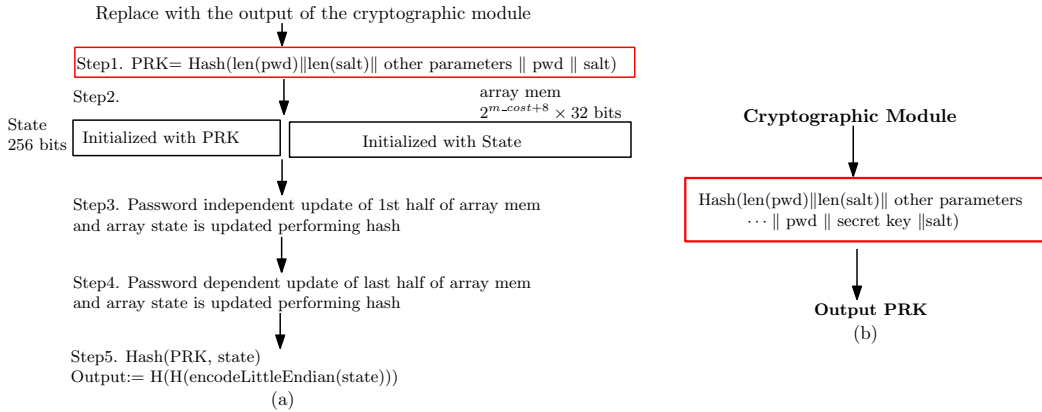


Figure 4.20: (a) Overview of the design TwoCats (b) The cryptographic module implementation

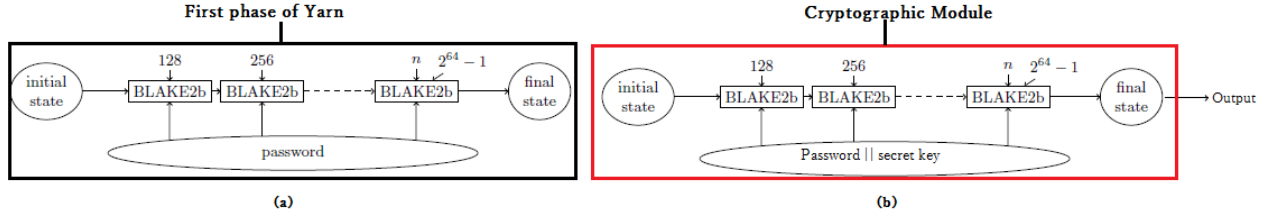


Figure 4.21: (a) Overview of the first phase design Yarn taken from [104], (b) The cryptographic module implementation

- **yescrypt** [133] yescrypt is based on the design Scrypt [132] and the significant difference is that the design of RoMix in yescrypt supports an optional pre-filled read-only lookup table (VROM, implemented in ROM). If VROM is accessed then the YESCRYPT_RW flag is set, and the other random-read lookup table (V, located in RAM) is updated partially. Password is the input to the initial call of PBKDF2 and last call of PBKDF2 uses the results of this initial call. We can implement the initial call to PBKDF2 in cryptographic module appending the secret key with the password as shown in Fig. 4.22 to enhance the security. Therefore cryptographic module based design does not require modification in the design of yescrypt as explained in [133].

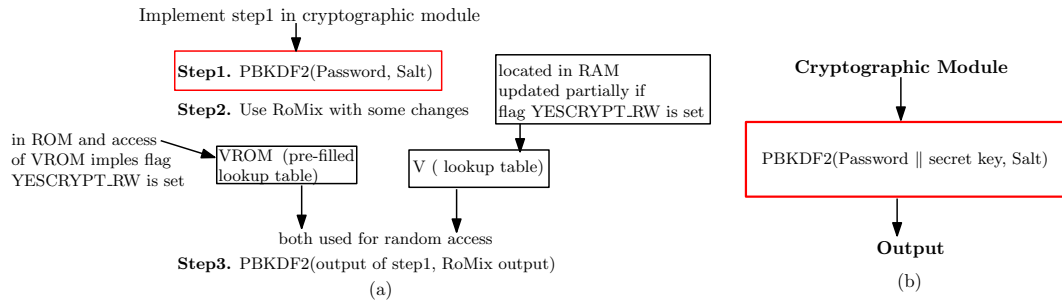


Figure 4.22: (a) Overview of the design yescrypt (b) The cryptographic module implementation

4.4.1 Categorization of the Schemes

From the analysis of the submitted PHC designs in the last section with respect to their suitability for a cryptographic module based implementation, we categorize the suggested approaches as follows.

- **Category I:** Use of the secret key with password (and other parameters, if applicable) for the initial hash computation inside the cryptographic module.

Table 4.1: Categorization of the algorithms based on the approaches taken for cryptographic module implementation.

Category I	Category II	Category III	Category IV
battercrypt	AntCrypt	Argon	Yarn
Catena	EARWORM	Lanarea DF	
CENTRIFUGE	MCS_PHS†	POMELO	
Gambit†	Omega Crypt†	Schvrch †	
Lyra2†	Pufferfish		
Makwa	Tortuga		
Parallel			
Rig			
TwoCats			
yescrypt			

†Assuming that the memory requirement for the suggested implementations is small enough to fit inside a low memory crypto-module.

- **Category II:** Replacement of the input password by the hash of the password with the secret key (and other parameters if applicable) which does not affect the overall design requirements (in terms of the cryptographic primitive used inside the cryptographic module).
- **Category III:** Replacement of the input password by the hash of the password with the secret key which may affect the design requirement (in terms of the cryptographic primitive used inside the cryptographic module).
- **Category IV:** Use of cryptographic module for initial multiple line execution (e.g., repeated use of the cryptographic primitive).

Table 1 lists the PHC algorithms based on the defined categories.

4.5 Summary

In this chapter, we provide a novel approach to enhance the security of the implementation of password hashing algorithms using a cryptographic module. We first define the cryptographic module and the levels of security provided by it and then propose the generic approach to use a cryptographic module for securing password hashing. We examined and then graphically explain how the designs of the PHC candidates fit in this architecture. Considering the emerging threats in the landscape of Cryptographic implementations, we believe our analysis will have a significant impact towards the security of password hashing algorithms. We hope that secure implementations of password hashing algorithm would

consider this cryptographic module based approach to enhance their security in appropriate use cases. A natural research extension to this work would be to apply the proposed approach to the implementation of password hashing. It would help to analyse the effect of this secure implementation of password hashing to prevent both online and offline attacks on password-based authentication.

Chapter 5

Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes

In previous chapters, we discussed the designing aspects of password hashing schemes (PHS). This chapter is dedicated to the analysis of PHS. A cryptanalytic technique known as time-memory tradeoff (TMTO) was proposed by Hellman for finding the secret key of a block cipher. This technique allows sharing the effort of key search between the two extremes of exhaustively enumerating all keys versus listing all possible ciphertext mappings produced by a given plaintext (i.e. table lookups). The TMTO technique has also been used as an effective cryptanalytic approach for PHS. The increasing threat of password leakage from compromised password hashes demands a resource consuming algorithm to prevent the pre-computation of the password hashes. A class of password hashing designs provides such a defense against time-memory tradeoff (TMTO) attack by ensuring that any reduction in the memory leads to exponential increase in runtime. These are called *Memory hard* designs and the security of such designs are defined by measuring their TMTO defense. Another important measure for a good PHS is its efficiency in utilizing all the available memory as quickly as possible, and fast running time when more than the required memory is available.

The *Memory Hardness* requirement for password hashing schemes is significant to prevent the dictionary attack. Usually, attackers use GPU clusters, FPGAs and ASICs to get tremendous amounts of computation power to brute-force frequently used *passwords* when a general purpose cryptographic hash function like SHA-1, SHA-2, BLAKE etc. is used. These constructions are extremely fast in hardware as well as software implementations thus enabling an attacker to perform billions of hashes per second. To prevent such attempt, memory hard designs are a good alternative. However, it is generally difficult to evaluate the “memory hardness” of a given password hashing design following its algorithmic description..

In this chapter, we present a simple technique to analyze the memory hardness of those algorithms whose execution can be expressed as a Directed Acyclic Graph (DAG). The nodes of the DAG correspond to the storage required by the algorithm and the edges correspond to the flow of the execution. Our proposed technique provides expected run-times at varied levels of available storage for the DAG. Most of the algorithms submitted to the PHC have

multiple variants and each can be represented as a different DAG. We experimentally show the TMTO of the algorithms representing the DAG with specific parameters. Specifically, we give a generic algorithm to traverse the DAG which allows to vary the memory storage and computes the increased algorithmic runtime (re-computation penalties) for different tradeoffs (varying memory and time) options. We show the effectiveness of our proposed technique by applying it to three designs from the “Password Hashing Competition” (PHC) - Argon2-Version 1.2.1 (the PHC winner), Catena-Version 3.2 and Rig-Version 2. Our analysis shows that Argon2i shows weak memory hardness which is also highlighted in a recent work by Corrigan-Gibbs et al. For Argon2i, the DAG representation varies depending on the output of a pseudorandom function which is non-uniform. Therefore, different DAGs for different input values are obtained. Consequently, the choice of nodes which should be kept in memory becomes probabilistic and hence difficult to analyze. Therefore, we first apply heuristic methods to find the optimal points for memory reduction and then apply the proposed traversal algorithm to obtain the TMTO results.

We analyze these PHS for performance under various settings of time and memory complexities. Our experimental results show (i) simple DAGs for PHS are efficient but not memory hard, (ii) complex DAGs for PHS are memory hard but less efficient, and (iii) combination of two simple graphs in the representation of a DAG for PHS achieves both memory hardness and efficiency.

This chapter is organized as follows: We first discuss the existing work on time-memory tradeoff analysis in Section 5.1. Next, the brief description on existing analysis of memory hard functions using graph pebbling is included in Section 5.2. We then present a brief overview of three cache-timing resistant password hashing algorithms namely, Argon2i, Catena and Rig in Section 5.3. This is followed by the preliminaries necessary for the understanding of the proposed time-memory tradeoff analysis technique in Section 5.4. The description of the proposed algorithm for TMTO is presented in Section 5.5. Subsequently, the re-computation penalties and performance analysis are presented in Section 5.6 and Section 5.7 respectively. Finally in Section 5.8, we summarize the whole chapter. The original contribution of this thesis is from Section 5.3 to Section 5.7.

5.1 Existing Results on Cryptanalytic Time-Memory Tradeoff

The idea of Time-Memory Tradeoff (TMTO) to optimize the cryptanalytic effort for a search which includes N possible solutions was introduced by Hellman in [97]. Specifically for TMTO, the cryptanalyst tries to optimize the product $T \cdot M$ searching for a correct solution among N feasible choices where T is the number of operations performed (time), M is the number of words of memory and $T \cdot M = N$. The relative cost of CPU cycles (T) is much lesser than RAM space (M), as a result most attacks attempt to reduce memory at the cost of increased algorithmic runtime.

The proposed technique of [97] for TMTO analysis considers chosen plaintext attack

scenario to find the key of a block cipher. Specifically, m uniformly random starting points are chosen and then m chains of length t of ciphertexts are computed where only the starting and ending points of the chains are stored that yields the tradeoff (see Chapter 2 for more details).

Utilizing the concept of Hellman [97], Philippe Oechslin introduced a cryptanalytic time-memory trade-off, named rainbow table [130]. This technique allows attacking a password hashing scheme by reducing the cryptanalysis time with the help of precomputed data stored in memory. It creates chains of password hashes choosing m uniformly random passwords from provided key space. It generates m chains of length t using t different reduction functions. The main advantage of rainbow table over the technique of Hellman [97] is that the chains can collide within the same table without merging (to merge, collision at the same position is required for two different chains). This is possible because rainbow table uses t different reduction functions to compute the chains. However, this technique is only applicable on password hashing schemes that do not consider salt as an input with the password (refer Chapter 2 for more details). A recent result on TMTO analysis based on precomputation method is covered in [46] for data-independent password hashing schemes and also provides a generic ranking method for data-dependent schemes. This proposed method is applied in [45] to prove the memory hardness of Argon2i. However, our analysis in Section 5.5 shows that Argon2i does not provide the required security level. Infact an attack showing the weak memory hardness of Argon2i was recently shown by Corrigan-Gibbs et al. [98].

5.2 On the Analysis of Memory-Hardness using Graph Pebbling

This section presents the existing analysis of password hashing schemes which is based on graph pebbling concepts (another significant area of research). We only provide the significant results in brief. The graph pebbling technique helps to prove the memory hardness property of password hashing schemes and recently proposed memory-hard designs of PHC [16] are analysed through graph pebbling technique implemented on their respective DAG representation.

Pebbling helps to determine the efficient way to use the registers of a CPU and/or the random-access memory of a general-purpose computer. In practice, we frequently come across the situation where the number of registers (space) available is insufficient to hold all the data on which a program operates and therefore registers need to be reused. If the space is increased, the number of computation steps (time) can generally be reduced [142]. The tradeoffs between the number of storage locations and computation time are analysed by the pebble game. In theoretical computer science, the concept of pebbling is described with a game known as pebble game or graph pebbling. For a DAG, pebbling represents the operation of assigning space to a vertex of the graph. Considering the constraint on the usage of space by the vertices of the DAG, the pebble game helps to determine a time-memory tradeoff for the given algorithm by pebbling a predetermined vertex within the

corresponding DAG. The initial assumption for the game is that the DAG has no pebble. The player performs a number of certain actions until a predefined output vertex has been pebbled. Considering the execution of an algorithm which can be represented as a DAG, a pebble on a vertex of the DAG indicates that its value is in a register.

5.2.1 Pebble Game

The goal of the pebble game is to pebble the vertices of the graph with a number of pebbles (space) and steps (time) that are minimal, i.e., neither can be reduced without increasing the other. Hence, it examines tradeoffs between the number of storage locations (memory/space) and computation time of an algorithm which can be modeled by a Directed Acyclic Graph. Specifically, the technique helps to derive lower bounds on the exchange of space S for time T and commonly evaluated in the form of product $S.T$ on the input of size N [142]. Following are the rules of the game.

Rules of the pebble game:

- A free pebble can be placed on an input vertex at any time
- A vertex can be pebbled only if all its predecessors are pebbled.
- A pebble can be removed at any time.
- Each output vertex must be pebbled at least once.

A pebbling strategy is the execution of the rules of the pebble game on the vertices of a DAG. The placement of a pebble on an input vertex represents the reading of input data and on a non-input vertex corresponds to computing the value associated with the vertex. The removal of a pebble represents the erasure or overwriting of the value associated with the vertex on which the pebble resides. When pebbles are allowed to be placed on input vertices at any time, it reflects that the inputs are readily available.

As already explained, the objective of the game is to successively pebble each vertex while minimizing the number of pebbles that are ever on the graph simultaneously. Following are the important terminologies and notations associated with the pebbling technique for a DAG.

- The size is the number of nodes N .
- The depth is the length of its longest path.
- The space S denotes the number of pebbles, and time T is the number of moves.
- ST -cost is the product of the maximum no. of simultaneously used pebbles and the number of steps needed to complete the pebbling.

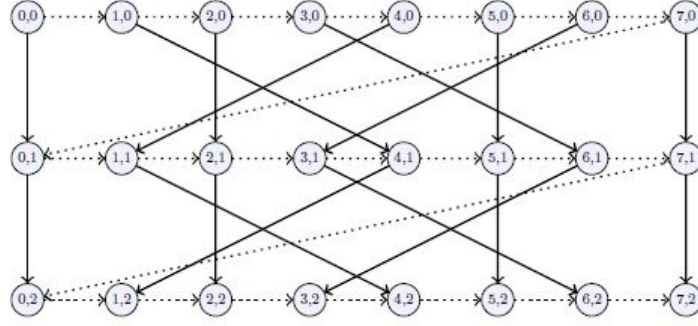


Figure 5.1: A (8, 2)-Bit-Reversal Permutation Graph

5.2.2 Important Results

In this section we list the important conjectures and theorems which play a significant role in the understanding of research in the analysis of password hashing schemes.

The following conjecture is from [118] which answers the question on how much T increases as S decreases from N toward $\Omega(N/\log N)$. It is important to note that in theoretical computer science a polynomial-time algorithms are considered as feasible, whereas algorithms whose time complexity exceeds every polynomial are not. In this sense the conjecture asserts that there are graphs for which space savings of $S = O(N/\log N)$, though possible, are infeasible.

Tradeoff Conjecture [118] There are graphs of size N that can be pebbled with $O(N/\log N)$ pebbles only in a time that grows superpolynomially in N .

Graph families whose pebbling time increases explosively from linear to superpolynomial are of special interest in the literature. Among the graphs, the bit-reversal permutation graph was studied extensively. The bit-reversal permutation is among the permutations that are most difficult to realize in serial computation schemes with restricted storage capacity. It has the property that it scatters adjacent numbers approximately evenly over the interval. A bit-reversal permutation graph with 8-vertices at each level where the total level is 2 is shown in Fig. 5.1 which is also explained in Section 5.4. The following theorem proves the time-memory tradeoff achieved for pebbling the bit-reversal permutation graph as presented in [142].

Theorem 5.2.1 *If $S \geq 2$, then pebbling the bit-reversal graph on N elements with S pebbles takes time $T > \frac{N^2}{16S}$*

Pebbling in Parallel Setting [31] The standard pebbling game is defined over the sequential computational model. Motivated through the parallelism in the modern computational system, the pebbling paradigm to a parallel setting is introduced in 2015 [31]. It

modifies the rules of the standard pebbling game to obtain a parallelised generalization. It allows the rules to be applied batch-wise instead of at most one pebble movement per step of the sequential model. Hence, a new cost measure is introduced which is called the cumulative complexity (cc) of a graph. The cc of a given execution in the (parallel) pebbling game for graph G is the sum of the number of pebbles lying on G when summed across all steps in the execution.

- **Cumulative Pebbling Complexity:** Let G be a DAG, $P = (P_0, \dots, P_t)$ be an arbitrary pebbling of G and Π be the set of all complete pebblings of G . Then the (cumulative) cost of P and then cumulative complexity (cc) of G are defined respectively to be:

$$pcost(P) := \sum_{i=0}^t |P_i|$$

$$cc(G) := \min\{pcost(P) : P \in \Pi\}$$

Pebbling of Bit-Reversal Graph in Parallel Setting In sequential setting it is shown in Theorem 5.2.1 that the tradeoff complexity of the bit-reversal graph is $ST = O(N^2)$. However, in the parallel setting, the cumulative complexity is shown over the same graph as $O(N^{1.5})$ in [31]. Instead of taking a bit-reversal graph, the result is shown in a generalized graph called a sandwich graph. A sandwich graph is a chain of N nodes (numbered 1 through N) with arbitrary additional edges connecting nodes from the first half of the chain with nodes of the second half of the chain such that no node has in-degree greater than 2. Bit-Reversal graph is a special case of Sandwich Graph.

Lemma 5.2.2 *Any Sandwich graph G of size N has*

$$cc(G) = O(N^{1.5})$$

The above lemma shows that bit-reversal permutation does not provide the expected tradeoff which was proved traditionally. Therefore, a theoretical analysis on memory hard functions has recently gained acceptance in cryptography. Several significant results on memory hardness were proposed afterward which includes the analysis on the winner of PHC, Argon2i. However, asymptotic attacks on several data Independent Memory Hard Functions (iMHFs) including Argon2i were dismissed as not practically relevant by the proponents of the quick standardization of password hashing designs. A recent result of Crypto 2016 [29] known as the AlwenBlocki attack and a proof on the design of iMHFs opens new directions of research on the design of memory hard functions. It demonstrates that no DAG with constant indegree can completely resist the attack. More precisely, they show that any iMHF is at best c -ideal for $c = \Omega(\log^{1-\epsilon} N)$ and any $\epsilon > 0$. In particular, this means that ideal iMHFs do not exist [29]. In practice, moderately hard to compute functions have proven to be useful security primitives.

Table 5.1: Bounds on the amortized energy complexity taken from [30].

Memory Hard Function (MHF)	Building Block	$E_R(\mathcal{A})$
Rig.v2 [59]	Blake2 [35]	$O(N^{1.75})$
TwoCats [69]	Blake2 [35]	$O(N^{1.75})$
Gambit [134]	Keccak Permutation [40]	$O(N^{1.75})$
Lyra2 [102]	Blake2 [35] or Keccak Permutation [40]	$O(N^{1.67})$
Pomelo [167]	Basic Operations	$O(N^{1.83})$

Memory Hardness of some Designs from PHC [16] The result from [29] shows an attack against any iMHF for which the DAG G is not depth-robust. A DAG, G is not depth-robust if there is a relatively small set S of nodes such that after removing S from G the resulting graph (denoted $G - S$) has low depth (i.e. contains only short paths). The analysis from [30] on iMHFs uses the following concept and the attack is summarized in Table 5.1.

- **AT Complexity** $AT_R(\mathcal{A})$: The AT complexity of an algorithm $\mathcal{A}$ is the product of area (A) of an implementation of the algorithm in an application specific integrated circuit (ASIC) and the time (T) it takes the circuit to produce output. AT complexity usually estimates the financial cost of implementing an algorithm. To upper-bound the cost of a brute-force attack on an Memory Hard Function (MHF), the amortized AT-complexity of the attack per instance of the MHF is computed. To make the results more independent of the technology it is used by parameterizing the complexity using a core-memory energy ratio $R \in \mathbb{R}^+$ denoting the ratio of the size of an (on-chip) implementation of the underlying function and area required to store one of its outputs.
- **Energy Complexity** $E_R(\mathcal{A})$: The energy complexity of an algorithm $\mathcal{A}$ measures the amount of electricity required by an ASIC implementing the algorithm per evaluation. To make the results more technology independent, a core-memory area parameter $\bar{R} \in \mathbb{R}^+$ which denotes the ratio between the amount of electricity used to evaluate the function and the amount of electricity needed to store the output of the function for an equivalent amount of time is adapted.
- The AT and energy quality of the iMHF can be expressed in terms of the complexity of ‘pebbling’ the graph. These complexities are upper-bounded in terms of the depth-robustness of the DAG. There should not be an algorithm that evaluates an iMHF at energy and AT complexity lower than N^2 .

5.3 Overview of Three Cache-timing Attack Resistant Password Hashing Algorithms

In this section, we briefly explain three cache timing attack resistant password hashing algorithms submitted to Password Hashing Competition (PHC) [16]. Overview of these

algorithms is provided ahead (though details of Argon2 and Rig has been explained in Chapter 3).

5.3.1 Argon2i [45]

Argon2 - Version 1.2.1 is the winner of PHC [16]. The first version was submitted as Argon and later updated to Argon2. Recently the new Version 1.3 of Argon2 has appeared which addresses the problem of memory optimization reported in [98]. For our work, we focus on the version 1.2.1 which was the PHC winner. Argon2 specifies two variants, Argon2d which follows input dependent memory access pattern and Argon2i which follows input independent memory access pattern. Both are effective for different use cases: Argon2d for computing crypto-currencies and Argon2i for password hashing, key derivation etc. [45]. Both the variants are different only at the point of index (of a matrix) computation. Our TMTO analysis is applicable to the variant Argon2i and we explain this design next.

The hash function Blake2b [35] represented as $\mathcal{H}$ and the compression function based on Blake2b permutation represented as G are used. First the variable length password P and salt S with other parameters are hashed using $\mathcal{H}$ to produce H_0 . This H_0 is used to generate a memory matrix $M_{i,j}$, $0 \leq i < p$ and $0 \leq j < q$ where p is the number of lanes (rows) and $q = m/p$ is the number of columns computed as below.

$$\begin{aligned} M_{i,0} &\leftarrow G(H_0, i \parallel 0), & 0 \leq i < p \\ M_{i,1} &\leftarrow G(H_0, i \parallel 1), & 0 \leq i < p \\ M_{i,j} &= G(M_{i,j-1}, M_{\phi(i,j)}), & 0 \leq i < p, 2 \leq j < q \\ \text{Output} &\leftarrow \mathcal{H}(M_{0,q-1} \oplus M_{1,q-1} \oplus \cdots \oplus M_{p-1,q-1}) \end{aligned}$$

where the function $\phi(i, j)$ computes the index of the matrix M and its computation is either password-dependent (Argon2d) or password-independent (Argon2i).

Index Computation $\phi(i, j)$: The memory matrix M is further partitioned in $S = 4$ slices. Intersection of a slice and a lane (row) is mentioned as the segment of length q/S . To compute the index, two round compression function G is run in counter mode with counter i . The first round input to G is a string of all-zeroes and the second round input is constructed as follows:

$$r \parallel l \parallel s \parallel m \parallel t \parallel x \parallel i \parallel 0$$

where, r is the pass number, l is the lane number, s is the slice number, m is the total number of memory blocks, t is the total number of iterations, x is 1 for Argon 2i and i is the counter starting in each segment from 1. Each application of G produces a 64-bit value. The two applications of G , therefore, produce a 128-bit value $J_1 \parallel J_2$ where $|J_1| = |J_2| = 64$. To get the memory index $\phi(i, j)$, compute $l = J_2 \bmod p$ which determines the index of the lane from which the block will be taken. If $r = s = 0$, then l is set to the current lane index. Then determine the set of indices $\mathcal{R}$ that is referenced for given $M_{i,j}$ according to the following rules as mentioned in [45]:

1. If l is the current lane, then $\mathcal{R}$ includes all blocks computed in this lane, which are not overwritten yet, excluding $M_{i,j-1}$.
2. If l is not the current lane, then $\mathcal{R}$ includes all blocks in the last $S - 1 = 3$ segments computed and finished in lane l . If $M_{i,j}$ is the first block of a segment, then the very last block from $\mathcal{R}$ is excluded.

Then take a block z from $\mathcal{R}$ by enumerating blocks in $\mathcal{R}$ in the order of construction as below. $J_1 \rightarrow |\mathcal{R}|(1 - \frac{(j_1)^2}{2^{64}})$, $x = (j_1)^2/2^{32}$, $y = (|\mathcal{R}| * x)/2^{32}$, $z = |\mathcal{R}| - 1 - y$. For detailed design of Argon, one may refer to [45].

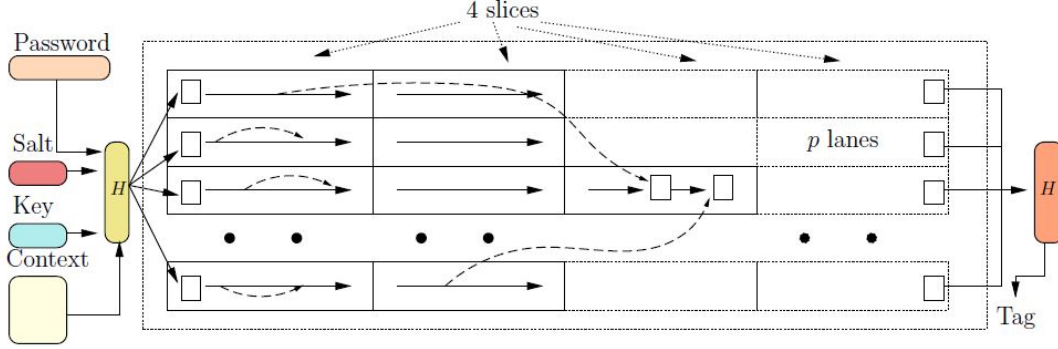


Figure 5.2: General overview of the design Single-pass Argon2 with p lanes and 4 slices, taken from [41].

5.3.2 Catena [88]

The design Catena provides two variants supporting input (password) independent memory access: Catena-Butterfly which is represented as a stack of double-butterfly graphs [51], and Catena-Dragonfly which is based on bit-reversal graphs [117]. Catena uses a function H which implements Blake2b [35], a function H' which implements a single round of Blake2b including finalization, denoted as Blake2b-1, a randomization layer τ (optional), and a “memory-hard” function F . Both the variants of Catena differ in the choice of this function F specified as F_λ^g , where F can be represented as a DAG with depth λ and 2^g nodes at each level. The variable g is called the garlic parameter. First, the algorithm initializes the variable x by setting it equal to the hash value computed on the concatenation of the three inputs: tweak t , salt and password. The garlic parameter g defines the time and memory requirements for Catena. The value x is then updated by the function $flap$ to produce the final password hash as shown in Fig. 5.3. The $flap$ function has three phases. In the first phase, a memory of size $2^g \cdot n$ bits is initialized, where $g_{low} \leq g \leq g_{high}$ and n (bits) is the output length of the underlying hash function. The second phase calls the function τ (optional) which depends on the public input γ . Finally, the third phase calls a memory-hard function F . When F is instantiated with BRH_λ^g ((g, λ) - Bit Reversal Hashing) it is

denoted as Catena-BRG and when F is instantiated with DBH_λ^g ((g, λ) - Double Butterfly Hashing) it is denoted as Catena-DBG. The overview of the design applying the function $flap$ is shown in Fig. 5.3.

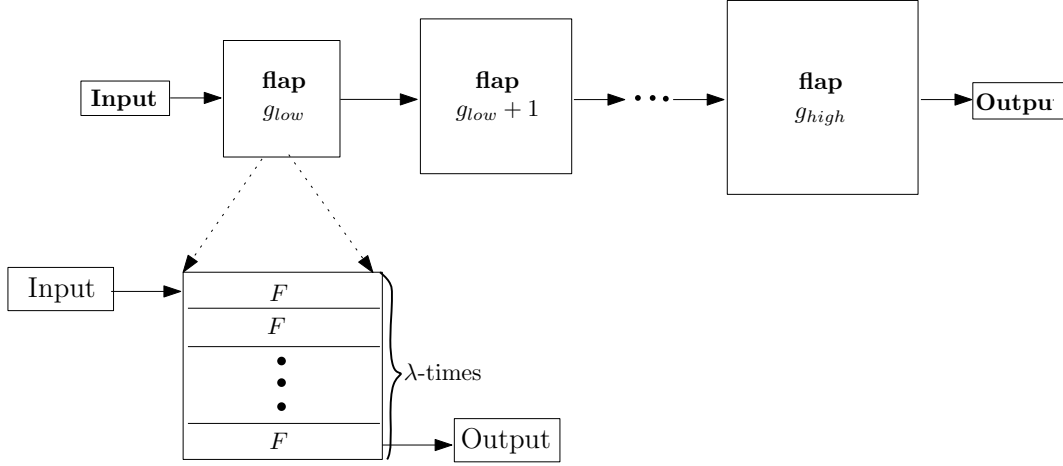


Figure 5.3: General overview of the design Catena applying the function $flap$, taken from [88]. Flap function is the λ -times implementation of the memory-hard function F , where F can be a bit-reversal graph as shown in Fig. 5.8 or a double butterfly graph as shown in Fig. 5.10.

5.3.3 Rig [59]

The algorithm Rig provides two variants where the general construction is represented as Rig $[H_1, H_2, H_3]$. The strictly sequential variant is denoted as Rig [Blake2b, BlakeCompress, Blake2b] and the optimized variant which improves the performance by performing memory operations in larger chunks is represented as Rig [BlakeExpand, BlakePerm, Blake2b]. Both the variants differ in the instantiations of the functions H_1 , H_2 and H_3 . We provide the general description of the design which is similar to the sequential variant.

The algorithm defines a round with four phases. Phase 1 is called the initialization phase which computes the hash of the value x derived from password, salt and other parameters and produces the output α . The hash function is represented as H_1 and instantiated with Blake2b. Next phase 2 is called the ‘setup phase’. This phase uses the value α and initializes two arrays k and a each of size $m = 2^{m_c}$ where m_c is taken as the input to define the required memory units. Each element of both the arrays is generated from the output of a hash function. The function H_2 is implemented as 1-round of Blake2b. Next phase is the ‘iterative transformation phase’ and is designed to update the stored array values λ -times where λ is the number of iterations. In this phase, each hash computation represented by H_2 takes input from both the arrays. Array k is accessed computing bit-reversal permutation on the indices and array a is accessed sequentially. The m - computations of H_2 at setup phase and $\lambda \times m$ computations of H_2 at iterative transformation phase altogether are denoted

as function $\mathcal{H}_2$. Specifically, the function $\mathcal{H}_2$ is the memory-hard function which accesses and updates λ -times the above mentioned arrays a and k . Both the arrays are generated from the output of a hash function, therefore, without loss of generality a single node in the graphical view of the $\mathcal{H}_2$ accommodates two elements, one each from two different arrays. Hence, the function $\mathcal{H}_2$ can be represented as λ -times implementation of function G where G is a bit-reversal straight graph as explained in Fig. 5.9. The last phase is called the ‘output generation phase’. This phase computes one hash (represented by H_3) taking salt as input with the last chaining value, produces the final output of each round. If round=1, this output is considered as the password hash otherwise the output of this phase is considered as the input to the next round. The value of m at round i , i.e. m_i is updated at round $i + 1$ as: $m_{i+1} = 2 \times m_i$ and other descriptions remain same. The overview of the Rig design where round=1, is shown in Fig. 5.4.

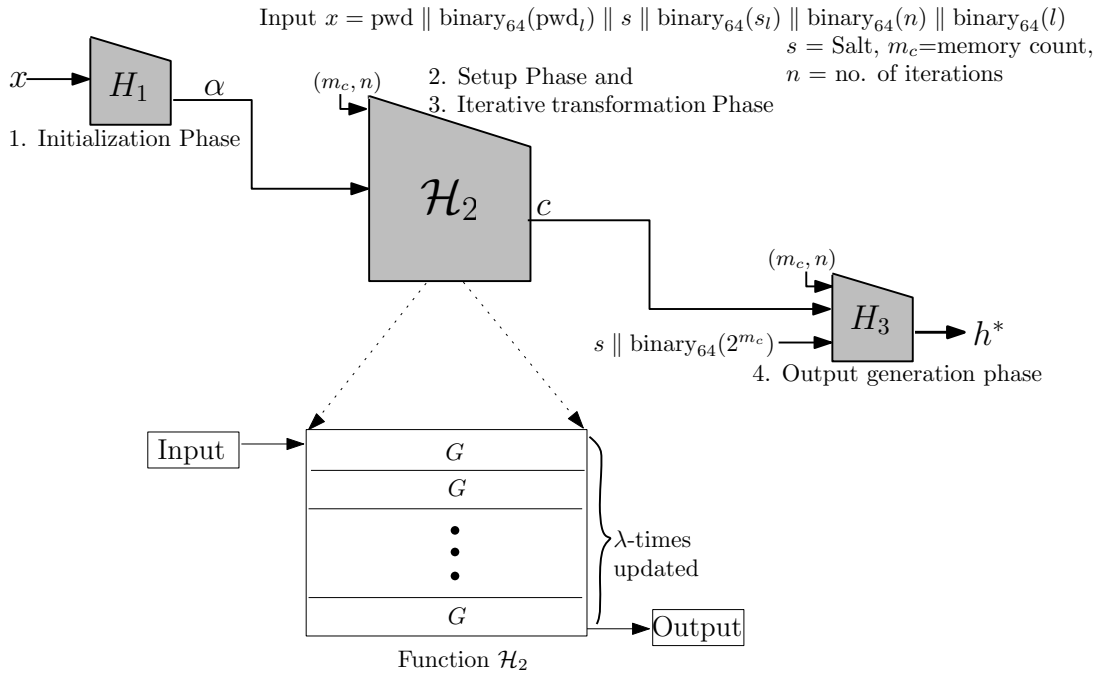


Figure 5.4: Overview of the design Rig taken from [59]. The function $\mathcal{H}_2$ internally updates λ -times the values of two arrays a and k each having m -entries applying function G where the function G is a bit-reversal straight graph as shown in Fig. 5.9.

5.4 Preliminaries

In this Section we give a brief overview of different graph structures which are the key concepts of our proposed cryptanalysis technique.

Directed Acyclic Graph (DAG) A directed graph is an ordered pair $(\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$ is a set of nodes and $\mathcal{E} \subset V \times V$. Every edge $e = (X_i, X_j)$ in the set $\mathcal{E}$ is ordered. A directed graph $\mathcal{G}$ is acyclic if it does not contain any directed cycle.

Bit-Reversal Permutation [117] A bit-reversal permutation is a permutation of a sequence of m elements with $m = 2^k$ where $k \in \mathbb{N}$. The elements are indexed from 0 to $m - 1$ and to permute the elements the bits of indices represented in binary form are reversed. Each element is then mapped to the new location as per the reversed value of indices from 0 to $m - 1$. Example, for $k = 3$ and $m = 2^3$ elements, the bit reversal graph with indices $0, 1, \dots, 7$ is shown in Fig. 5.5.

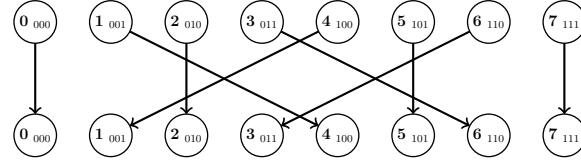


Figure 5.5: Bit-Reversal Permutation of $m = 2^3$ elements with 3-bit binary representation of indices.

5.4.1 Description of Some Directed Acyclic Graphs

In this work, we analyze DAGs consisting of a two-dimensional matrix of nodes. The characteristics of the graph are determined by the connectivity (dependency) between the nodes. We next describe some graphs which are useful for analyzing the designs Catena [88] and Rig [59]. The graphical representation of these schemes is derived from the following 4 types of graphs which are named according to the property followed by their edges as shown in Fig. 5.6.

1. The *Sequential* graph is obtained by connecting all the nodes of the graph sequentially (level-wise).
2. The *Vertical* graph is obtained by connecting all the nodes of the graph vertically (level-wise).
3. The *BitReversed* graph is obtained by applying bit-reversal permutation [117] on each node (level-wise).
4. The *Butterfly* graph [68, 51] is obtained by placing two back-to-back Fast Fourier Transformation (FFT) graphs after omitting one row in the middle.

The analyzed password hashing schemes can be graphically represented by overlaying these 4 types of graphs in various combinations as described ahead.

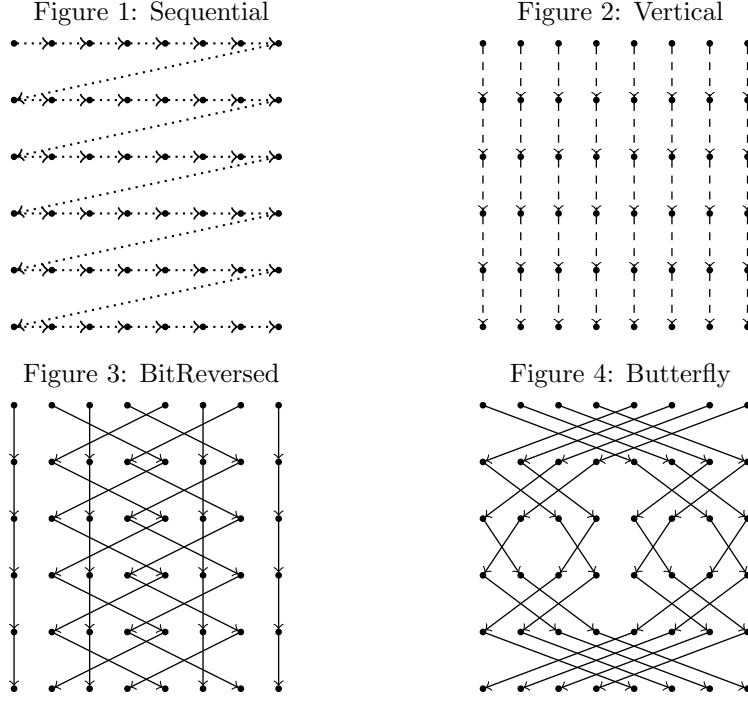


Figure 5.6: Graphs based on different types of edges

$(\mathcal{N}, \lambda)$ -Straight Graph A $(\mathcal{N}, \lambda)$ -Straight Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential* and *Vertical* edge types graphs. λ denotes the depth of the graph and $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ is the number of nodes at each layer. An example of $(8, 2)$ -Straight graph is shown in Fig. 5.7. It is a simple and symmetric graph where $\lambda = 2$ is the depth of the graph. The consecutive nodes of each level are connected sequentially and level-wise nodes are connected vertically. We use this graph to show the working of the DAG traversal algorithm (defined below) with respect to the designs Catena [88] and Rig [59].

$(\mathcal{N}, \lambda)$ -Bit-Reversal Graph (Representing the Catena-BRG Construction [88]) A $(\mathcal{N}, \lambda)$ -Bit-Reversal Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ edges can be formed by overlaying

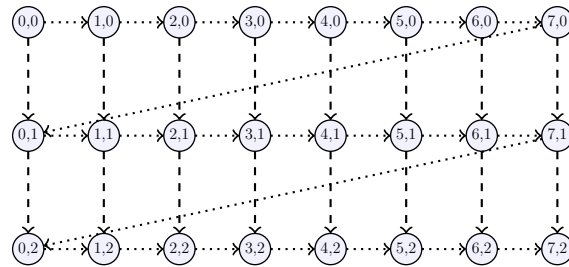


Figure 5.7: $(8, 2)$ -Straight Graph

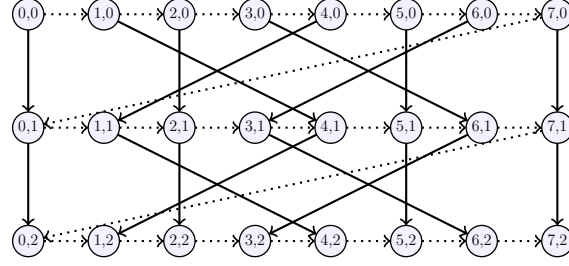


Figure 5.8: (8,2)-Bit-Reversal Graph

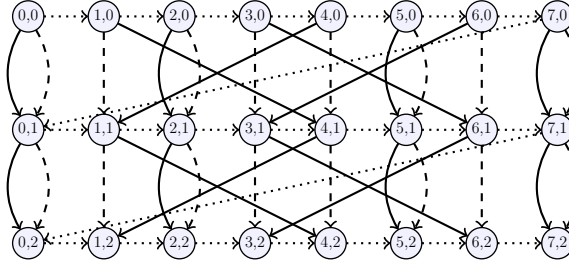


Figure 5.9: (8,2)-Bit-Reversal-Straight Graph

the *Sequential* and *BitReversed* edge types graphs. λ is the depth of the graph and at each level number of nodes $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from [59]). An example of (8,2)-Bit-Reversal graph is shown in Fig. 5.8. As per the definition, it performs bit-reversal permutation at each level or it is a stack of $\lambda = 2$ bit-reversal permutation operations.

This graph represents the directed data dependency of Catena-BRG construction [88]. Specifically, it describes the flow of the flap function (see Fig. 5.3) with respect to its core memory-hard function F as described in Section 5.3.2. The function F instantiated with bit-reversal graph and denoted as BRH^g_λ , requires three inputs: g that specifies the required number of nodes (2^g) of the graph at each level, the value x which is the input to process, and the value λ which defines the depth of the graph. Therefore, Catena-BRG can be specified by a bit-reversal graph with $\lambda \times 2^g$ nodes representing the entire computation graph where the directed edges show the flow of the execution.

$(\mathcal{N}, \lambda)$ -Bit-Reversal-Straight Graph (Representing the Rig Construction [59]) A $(\mathcal{N}, \lambda)$ -Bit-Reversal-Straight Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *BitReversed* edge types graphs. λ is the depth of the graph and at each level the number of nodes $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and (definition adapted from [88]). An example of (8,2)-Bit-Reversal-Straight graph is shown in Fig. 5.9 where $\lambda = 2$ is the depth of the graph and at each level, the number of nodes is 8.

This graph represents the directed data dependency of the $\mathcal{H}_2$ function of the Rig construction [59], depicted in Fig. 5.4. The function $\mathcal{H}_2$ accesses and updates values of two different arrays, each of size $m = 2^{m_c}$, at every level of the DAG (a level of the DAG is the

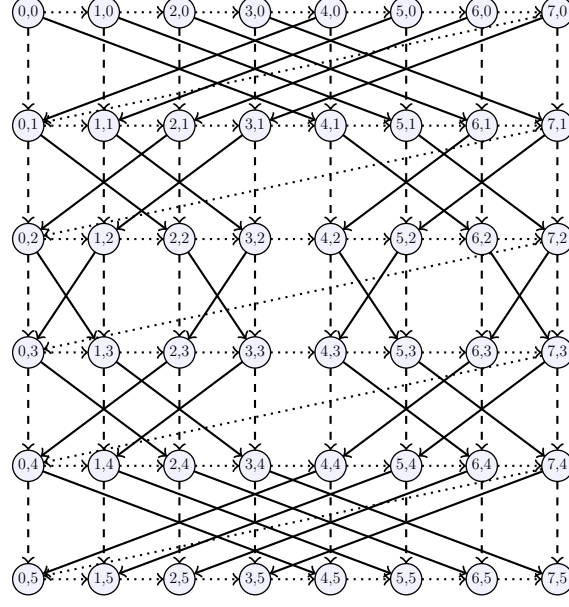


Figure 5.10: (8, 1)-Double-Butterfly Graph

horizontal dashed line in Fig 5.9, which is further explained in Section 5.3.3). To simplify the graphical view, we consider storing both the arrays at the i^{th} location of the memory arrays with a single node of the graph. Therefore $\mathcal{H}_2$ is represented as bit-reversal-straight graph with m nodes and each node of the graph accommodates two elements, one each from two different arrays. The number of iterations n defines the depth of the graph i.e., the number of times the nodes are accessed and updated. The directed edges show the flow of execution of the algorithm.

$(\mathcal{N}, \lambda)$ -Double-Butterfly Graph (Representing the Catena-DBG Construction [88])

A $(\mathcal{N}, \lambda)$ -Double-Butterfly Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *Butterfly* edge type graphs. λ is the depth of the graph and at each level the number of nodes $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from [88]). An example of (8, 1)-Double-Butterfly graph is shown in Fig. 5.10 where $\lambda = 1$ is the depth of the graph and the number of nodes at each level is 8.

This graph represents the directed data dependency of Catena-DBG construction [88]. It describes the flow of the flap function (see Fig. 5.3) with respect to its core memory-hard function F instantiated with double-butterfly graph. This function, represented as DBH^g_λ , requires three inputs: g that specifies the required number of nodes (2^g) of the graph at each level, the value x which is the input to process, and the value λ which defines the depth of the graph. Specifically, the $(\mathcal{N} = 2^g, 1)$ -Double-Butterfly graph representation is stacked λ times in Catena-DBG to create $(\mathcal{N} = 2^g, \lambda)$ -Double-Butterfly graph. The memory traversal pattern follows from the original FFT butterfly structure. Due to the significantly large number of operations and a large number of layers (for example, 5 in Fig. 5.10) at each level,

this graph traversal is significantly slower than the previous types. The re-computation effort increases exponentially with reduction in memory. This is due to the fact that in-degree of a node in the DAG corresponding to this design is high, e.g., each node has in-degree of 3 when the total number of nodes is 8 (see Fig. 5.10). The directed acyclic graph corresponding to Catena-DBG has 2^9 nodes which are arranged as λ stacks of double butterfly graphs. The directed edges in the DAG show the flow of execution of the algorithm as explained in Section 5.3.2.

5.5 Traversing a Dependency Graph to Analyse Trade-off Penalties

A password hashing design is considered memory hard, with respect to a pre-specified memory, if its implementation requires significantly larger runtime when the memory is reduced by even a fraction smaller than the pre-specified number. The expected increase in runtime is exponential in the amount of memory reduction.

Many designs in the PHC claim a strong time-memory tradeoff defense. However, there exists no general method to verify the claimed TMTO defense for a proposed algorithm. With the aim to give a solution, we provide a cryptanalytic approach and apply the technique on Catena, Rig and Argon2i. The representation of these algorithms as a directed acyclic graph, as explained in Section 5.4, accommodates the memory dependencies throughout their execution. Our proposed technique follows a simple approach to allow the flexibility to store the memory elements as per the choice of the implementor and then to perform on-the-fly computation of memory elements which are not stored at the time these elements are needed. This may increase the computation time from the usual implementation of the design. We compute the increased algorithmic runtime which we denote as re-computation penalty. This re-computation penalty provides the actual TMTO defense of the algorithm. However, an attacker is not obliged to follow the advice of the designer and may vary the memory storage to other nodes. This could potentially allow him to compute the password hash at a lower cost than the one envisaged by the designer. The algorithmic description of the proposed method is provided in Algorithm 5.1.

The graphical representation of a password hashing algorithm shows the memory dependencies between various memory elements as a password hash is computed in accordance with the design. The nodes of the graph represent the storage elements (memory) of the design and the arrows targeting the nodes show the dependencies. The Algorithm 5.1 traverses the nodes of the password hashing scheme following its actual implementation and computes the values that are not stored when required. Therefore ‘node’ plays an important role and below is the data structure defined to keep the state of the nodes during traversal.

```

structure Node
{
integer X = 0, Y = 0;

```

```

boolean MemoryAllowed = false, MemoryValid = false, Traversed = false;
array Node [ ] Dependencies;
}

```

Initially, all the values of ‘node’ are set to false. Each ‘node’ keeps track of an array which includes all the nodes that derive its value. The password hashing schemes we analyze need memory equal to the number of nodes in one row. If enough memory is available, then there is no need to do any TMTO, and the computation takes the time it needs to process all the nodes once, i.e., the time of actual implementation of the scheme. If enough storage is unavailable then it requires to perform a tradeoff between time and memory and it is expected that it will require a significantly large number of operations to compute the values that are not stored. We explain the proposed technique with examples in the following Section.

5.5.1 Description of the Proposed Technique

The nodes of a graph are represented as a tuple specifying column and row numbers. Therefore the starting value $(0, 0)$ contains the value corresponding the initial inputs of a password hashing algorithm and is assumed to be known. The algorithm takes as input the locations of the nodes that are allowed to be stored during the evaluation. The location can be all the nodes corresponding to a column or a row or some random locations throughout the graph. Therefore, there can be a large number of possible combinations of allowed-memory locations and the overall effort (computations) will depend on the allowed memory and its allocation in the complete graph. The structure *Node* has the field *MemoryAllowed* to let the algorithm know which node has storage and allow it to store the value when it is available/calculated during traversal and then mark the *MemoryValid* true to know the value is available for further computations. The algorithm starts traversing from the last node, i.e., from node $(M-1, N-1)$ and runs iteratively backward, traversing node-to-node until all the dependencies are computed. To explain the traversal we follow the following notation.

$$(N^i, N^j) \rightarrow \{ (D^i, D^j), (D^{i+1}, D^{j+2}), \dots \} \Rightarrow (N^{i+1}, N^{j+1}) \rightarrow \dots$$

Algorithm 5.1: DAG Traverse

Input: **graph** $\langle$ **Node** $\rangle$ -Dependency graph to traverse,
integer M- No. of columns, N- No. of rows

Variables: **Node** n,
stack $\langle$ **Node** $\rangle$ processing, dependency,
boolean dependencyfound,
list $\langle$ **Node** $\rangle$ traverse

Output: **list** $\langle$ **Node** $\rangle$ traverse - A list of nodes traversed by the algorithm.

1. n = graph[M-1, N-1]; ▷ each node contains all its dependencies
2. **while**(true) **do**
3. **if**(n.Traversed == false)
4. **foreach** dependency in n.Dependencies **do**
5. **if** (dependency.MemoryValid == false)
6. dependency.push(dependency)
7. **end if**
8. **end foreach**
9. n.Traversed $\leftarrow$ true
10. **else**
11. **if** dependency.count >0
12. n = dependency.pop()
13. processing.push(n)
14. **if** (n.MemoryValid == false)
15. add n to **list** traverse
16. **end if**
17. dependencyfound $\leftarrow$ true;
18. **foreach** Node d in n.Dependencies **do**
19. **if** (d.MemoryValid == false)
20. dependencyfound $\leftarrow$ false
21. **end if**
22. **end foreach**
23. **if** (dependencyfound == true)
24. **while** processing.count >0 **do**
25. temp = processing.pop()
26. **if** temp.MemoryAllowed == true
27. temp.MemoryValid $\leftarrow$ true
28. **end if**
29. **end while**
30. graph.clearAllTraversed() ▷ clear graph to process next dependencies
31. **end if**
32. **else break** ▷ when no dependency is left to process
33. **end while**
34. **return** list $\langle$ **Node** $\rangle$ traverse

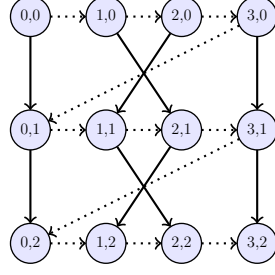


Figure 5.11: (4, 2)-Bit-Reversal Graph

Where, $(N^i, N^j), ((N^{i+1}, N^{j+1})) \dots$ are the nodes, $(D^i, D^j), (D^{i+1}, D^{j+2}), \dots$ are the dependencies discovered during the traversal and i, j are the corresponding column and row numbers. Specifically, the aim of all computations is to find out the value of node $(M-1, N-1)$ where M is the number of columns and N is the number of rows, while the input $(0,0)$ is known. All the nodes in between need to be computed on the way. Algorithm 5.1 (as defined) is considering *pointer arithmetic*, so, when node n is pushed and then popped from the stack *dependency*, any changes to n will be reflected in the initial *graph* structure. For a better understanding of the proposed approach, we provide two examples which cover two different scenarios and also prove the validity of the method.

5.5.1.1 Example 1

A (4, 2)-Bit-Reversal Graph is shown in Fig. 5.11.

Let us consider that all the nodes allow memory storage. The procedure can be performed by traversing the nodes in the following order starting with node $(3, 2)$ where 3 is the column number and 2 is the row number. The traversal of (4, 2)-Bit-Reversal Graph follows our notation of traversal including the dependencies which is explained above and needs total 12 steps covering the following path.

$$\begin{aligned}
& (3,2) \rightarrow \{(2,2), (3,1)\} \Rightarrow (3,1) \rightarrow \{(2,1), (3,0)\} \Rightarrow (3,0) \rightarrow \{(2,0)\} \\
& \Rightarrow (2,0) \rightarrow \{(1,0)\}, (1,0) \rightarrow \{(0,0)\} \Rightarrow (0,0) \rightarrow \{ \} \\
& \Rightarrow (2,1) \rightarrow \{(1,1)\} \Rightarrow (1,1) \rightarrow \{(0,1)\} \Rightarrow (0,1) \rightarrow \{ \} \\
& \Rightarrow (2,2) \rightarrow \{(1,2)\} \Rightarrow (1,2) \rightarrow \{(0,2)\} \Rightarrow (0,2) \rightarrow \{ \}
\end{aligned}$$

Considering the node $(3, 2)$ as the starting point of Fig. 5.11, the initial dependencies are $(2, 2)$ and $(3, 1)$ which again have further dependencies. For each node, the chain of its dependency nodes are backtracked. Therefore, the dependency path of node $(3, 1)$ includes $(2, 1)$ and $(3, 0)$ then from $(3, 0)$ to $(2, 0)$ which gives $(1, 0)$ and finally for $(1, 0)$ the dependency $(0, 0)$ ends the current chain of dependencies. Whenever a dependency is found it is put in the stack.

Table 5.2: Traversal for Example 1

Position	Dependency
(3,2)	(2,2), (3,1)
(3,1)	(2,1), (3,0)
(3,0)	(2,0)
(2,0)	(1,0)
(1,0)	(0,0)
(0,0)	
(2,1)	(1,1)
(1,1)	(0,1)
(0,1)	
(2,2)	(1,2)
(1,2)	(0,2)
(0,2)	

As the value of (0,0) is always known, it helps to end the dependency chain and also to compute the values of the stack by popping them one-by-one. Therefore, all the nodes of the current stack are processed. As we are considering the scenario where all the values are allowed storage, the values at the memory location (1,0), (2,0) and (3,0) are updated after their first processing. Next, the dependency (2,1) is processed which needs only the dependency (1,1) as another dependency (1,0) is known. Dependency (1,1) needs (0,1) which is known from (0,0), (3,0). Next the values (0,1), (1,1), (2,1) and (3,1) are updated. The stack is then processed again and takes the value (2,2) and the process continues until all the nodes are processed and all the dependencies are met. The method takes total 12 computations as below.

$$(3,2) \rightarrow (3,1) \rightarrow (3,0) \rightarrow (2,0) \rightarrow (1,0) \rightarrow (0,0) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (0,1) \rightarrow (2,2) \rightarrow (1,2) \rightarrow (0,2)$$

The traversal steps for Example 1 are shown in Table 5.2.

5.5.1.2 Example 2

A (4,2)-Bit-Reversal Graph is considered as shown in Fig. 5.12 where only the first column (nodes in blue) is allowed memory storage. The rest of the nodes have no memory and during traversal (*MemoryValid* set to false), they need to be re-computed every time they are encountered.

The complete traversal for a (4,2)-Bit-Reversal Graph as shown in Fig. 5.12 needs 35 steps and the steps take the following path.

$$\begin{aligned} & (3,2) \rightarrow \{(2,2), (3,1)\} \Rightarrow (3,1) \rightarrow \{(2,1), (3,0)\} \Rightarrow (3,0) \rightarrow \{(2,0)\} \\ & \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{(0,0)\} \Rightarrow (0,0) \rightarrow \{\} \\ & \Rightarrow (2,1) \rightarrow \{(1,1), (1,0)\} \Rightarrow (1,0) \rightarrow \{\} \end{aligned}$$

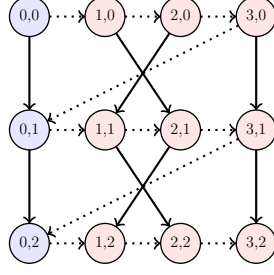


Figure 5.12: (4, 2)-Bit-Reversal Graph with only memory in the first column.

$$\begin{aligned}
&\Rightarrow (1,1) \rightarrow \{(0,1), (2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (0,1) \rightarrow \{(3,0)\} \Rightarrow (3,0) \rightarrow \{(2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (2,2) \rightarrow \{(1,2), (1,1)\} \Rightarrow (1,1) \rightarrow \{(2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (1,2) \rightarrow \{(0,2), (2,1)\} \Rightarrow (2,1) \rightarrow \{(1,1), (1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (1,1) \rightarrow \{(2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (0,2) \rightarrow \{(3,1)\} \Rightarrow (3,1) \rightarrow \{(2,1), (3,0)\} \Rightarrow (3,0) \rightarrow \{(2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (2,1) \rightarrow \{(1,1), (1,0)\} \Rightarrow (1,0) \rightarrow \{\} \\
&\Rightarrow (1,1) \rightarrow \{(2,0)\} \Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{\}
\end{aligned}$$

5.5.1.3 Limitation of the Proposed Technique

The proposed technique is a theoretical TMTO attack technique. Realistic application needs different resources and different approach which is beyond the scope of current work. However, we target to address it further in future. Therefore, currently it is difficult to provide the actual memory-hardness and time-complexity following the proposed algorithmic approach. The expected result to show memory hardness requires a higher minimum bound which is shown in Section 5.6 by explaining the results obtained from the implementation of three algorithms, Argon2i, Catena and Rig.

5.6 Results

All experiments documented in this section and in Section 5.7 follow the same setup as described ahead. In order to get consistent results for the different algorithms, we perform all the test on a single machine with the code compiled by the same compiler. The details are as follows:

- **CPU:** Intel Core i7 4770 (Turbo Boost: ON) - Working at 3.9 GHz
- **RAM:** Double Channel DDR3 16 GB (2400 MHz)
- **Compiler:** gcc / g++ v4.9.2 (-march=native and -O3 flags were set if not already in the makefiles). This would cause the compiler to use the AVX-2 instructions.

- **OS:** UBUNTU 14.04.1, on HYPER-V, on Windows-8.1 with 8 GB allocated RAM to the VM. We also performed benchmarks on native Linux OS to make sure that the virtualization does not cause any changes in the results.

For consistency, we use single-threaded versions of the algorithms. The obtained figures of re-computation penalties for algorithms Cantena and Rig are independent of the choice of input parameters. This is due to the deterministic behavior of both the algorithms. However, we experiment taking similar values for all algorithms to maintain consistency. The design Argon2i provides different graph structures for different public parameters (independent of password and salt), therefore we explain its TMTO analysis separately in Section 5.6.1. Hence, in case of Argon2i, it is important to mention the parameter choices (the exact values) to ease the verification of claimed results.

The DAG traversal algorithm provided in Section 5.5 can be used to compute the re-computation penalties for any graph by varying the memory storage. To apply Algorithm 5.1, a large number of memory configuration is possible. For example, a column of a graph can be enabled or disabled, i.e., when a column is enabled all the nodes for that column have memory storage abilities, otherwise not. We apply Algorithm 5.1 to the following cases and come up with re-computation penalties in different memory sizes. For our experiments, we only allow a limited set of configurations, i.e. columns are enabled or disabled. The considered password hashing designs are regularly structured and therefore the approach used to enable/disable columns is easy to implement. For analyzing reduced memory scenarios, we try to evenly distribute the memory along columns starting from the first column. The design Argon2i provides different graph structures for different implementation parameters, therefore we explain its TMTO analysis separately in Section 5.6.1. Following are the graphs that represent a fixed structure for all parameter choices and we compute the re-computation penalties for them.

- $(\mathcal{N}, \lambda)$ -Straight Graph (SG)
- $(\mathcal{N}, \lambda)$ -Bit-Reversal Graph (Catena BRG)
- $(\mathcal{N}, \lambda)$ -Double Butterfly Graph (Catena DBG)
- $(\mathcal{N}, \lambda)$ -Bit-Reversal-Straight Graph (Rig Graph)

The cumulated results (including Argon2i) for the graphs are as shown in Fig. 5.13. It shows the comparison of the re-computation penalty with change in allowed memory proportion. It is clear from the results that the re-computation penalty increases drastically for reductions in memory size. For the experiments, we fix $M = 64$ (columns) even though we experimented with larger values upto 512. This is because the characteristics of the DAGs do not depend significantly on the value of M and follow similar pattern of rate of growth, but the runtime becomes large. All data from Table 5.3, 5.4, 5.5 and 5.7 are included in Fig. 5.13.

Figure 5.13: Re-computation Penalties for Graphs

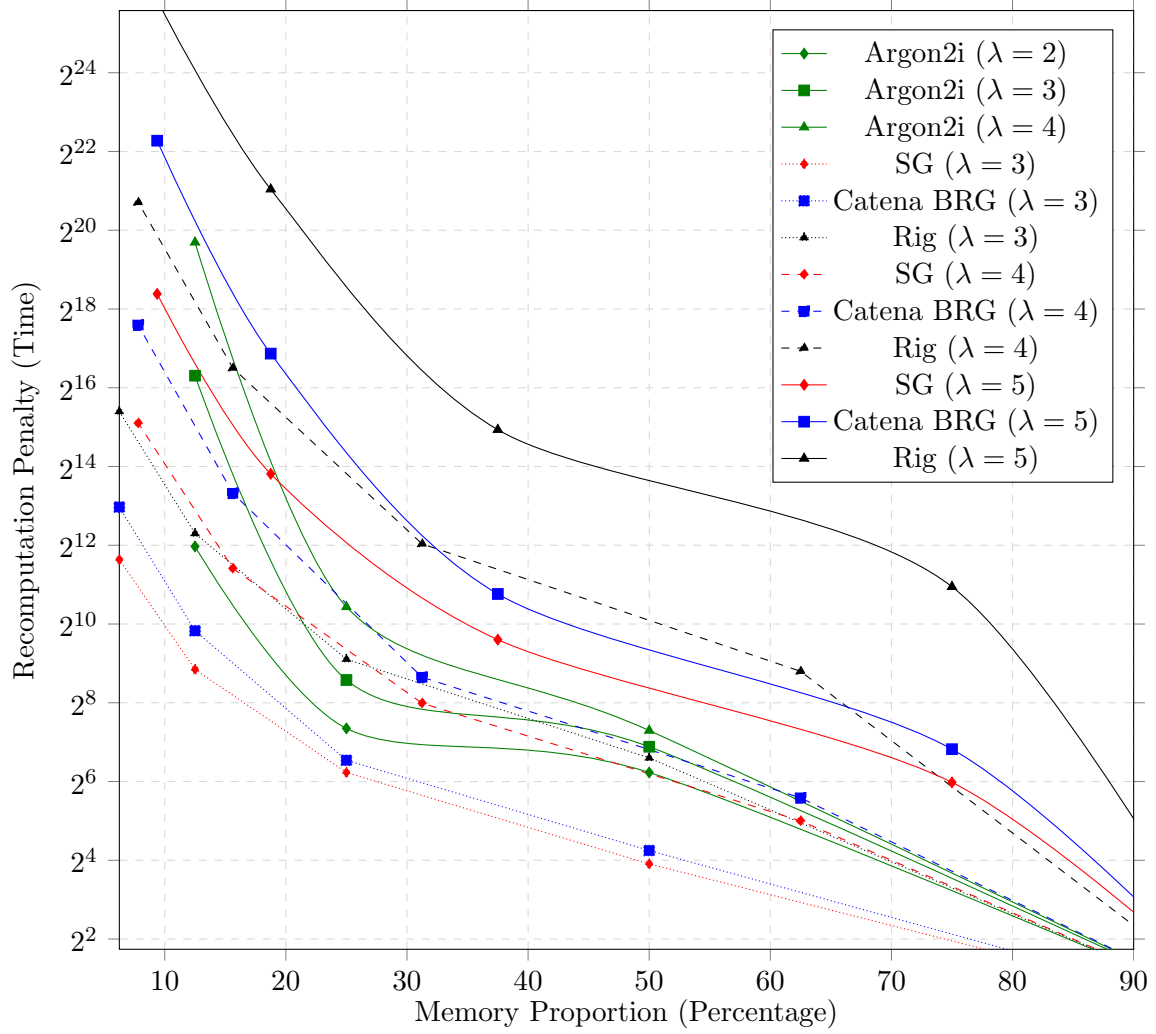


Table 5.3: Re-computation Penalties for Graphs with 4 rows ($\lambda = 3$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph (Catena BRG)	Bit-Reversal- Straight Graph (Rig)
50	15	19	97
25	75	93	551
12.5	460	909	5036
6.25	3181	8019	43143

Table 5.4: Re-computation Penalties for Graphs with 5 rows ($\lambda = 4$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal- Straight Graph
62.5	32	48	445
31.25	254	401	4190
15.625	2724	10215	92483
7.8125	35120	197389	1707950

Table 5.5: Re-computation Penalties for Graphs with 6 rows ($\lambda = 5$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal- Straight Graph
75	63	113	1971
37.5	777	1736	31270
18.75	14378	119358	2152596
9.375	341447	5060331	-

Table 5.6: Re-computation Penalties for Double-Butterfly Graph ($\lambda = 1$)

Memory Proportion (%)	Double-Butterfly Graph
50	18
25	922
12.50	60504
6.25	2043702

The algorithm uses stacks during calculations, but, the maximum size of the stack is bounded by the maximal length of a single path (plus sub paths). As a result, the algorithm does not consume a large amount of memory even for huge re-computation dependency tree calculations.

5.6.1 Re-computation Penalty for Argon2i

The design Argon2 is the winner of the PHC competition [16] and an IETF internet-draft [43] proposes it to be made a standard password hashing design for internet protocols. Therefore it is important to have detailed theoretical as well as practical analysis of this design. Our proposed approach can help analyze any password hashing scheme whose execution can be represented as a DAG. As we discuss in Section 5.3.1, Argon2i falls in the cache-timing attack resistant category for having password independent memory access pattern. However, it does not follow a fixed DAG structure over all parameter choices as the memory access depends on the pseudorandom output of the compression function G which is non-uniform. Due to the non-uniform G , we get different DAGs depending on the chosen input. The nodes of the graph at each level represent the memory elements of the corresponding row of the

memory matrix of Argon2i as described in Section 5.3.1. Except for the first two nodes of the first level, all nodes are connected to its previous node and a node value derived from the output of the compression function G . We provide three different graphs for different parameter choices in Fig. 5.14, 5.15 and 5.16. The parameters p is set to 1 which means we are considering the single threaded version of Argon2i.

A recent attack on Argon2i, presented by Corrigan-Gibbs et al. [98], shows that having a pseudorandom memory access pattern for Argon2i does not provide the advantage over fixed memory access as employed in designs like Catena and Rig. In fact, Corrigan-Gibbs et al. exploit the non-uniform distribution of memory accesses against Argon2i. We explain the main idea of the attack against Argon2i with an example. Consider the memory access pattern in the last few nodes of Argon2i in Fig. 5.14. The last 6 nodes accessed among the total 8 in this figure follow the sequence 0, 1, 1, 3, 3, 3. This implies that these 8 nodes can be computed with only 3 nodes (current node and the previous ones numbered 1 and 3). If the design was memory hard, we should have required all 8 nodes to compute the output of Argon2i but we can manage to compute this while storing only 3 nodes. Hence there is no recomputation penalty. Note that this behavior is caused by the one-to-many mapping in Argon2i (e.g. node 3 is used to compute nodes 5, 6 and 7 in Fig. 5.14). This provides the attacker with an opportunity to reduce the required number of memory units throughout the Argon2i computations without paying any penalty.

In order to make a fair comparison between Argon2i, Catena and Rig, we analyze these designs with a similar number of memory units. Since each node of Argon2i takes 1024 KiB memory while each node of Rig and Catena takes 512 KiB, we compare n nodes of Argon2i with $2n$ nodes of Catena and Rig. Argon2i creates the memory array depending on the previous node and a node derived from the pseudorandom output of function G . Thus, the memory access pattern for Argon2i at the first level itself starts to contribute to its memory hardness. On the other hand, the memory access pattern for Catena and Rig at the first level is derived from the initial input sequentially. For levels $l > 1$, the memory access depends on the previous node and a node derived from a fixed permutation. Therefore, as far as the computation effort is concerned, iteration i for Argon2i is equivalent to computations with iteration $i + 1$ of the designs Catena and Rig. We consider this equivalence in our experimental results.

To obtain the recomputation-penalty for Argon2i, we provide the dependency graph (DAG) generated from the above mentioned input parameters. To compute TMTO, the selection of nodes to allow memory are obtained after applying different heuristic approaches. This is required for Argon2i because of the non-uniform dependencies on nodes, as already mentioned. Since the selection of optimal nodes varies depending on the given input for Argon2i, it is not feasible to get a closed-form solution for it. However, following the probabilistic analysis on the execution pattern of Argon2i, it is possible to provide a randomized algorithm to get the optimal nodes considering the frequency of dependencies. We later explain the basis of our heuristics for searching the optimal nodes.

To select the nodes with a higher frequency of dependencies, we used the following strategy. First, we list the nodes by sorting them on decreasing order of dependencies, and then

Table 5.7: Re-computation Penalties for Argon2i (32KiB) at varying λ (t-cost)

Memory Proportion (%)	$\lambda = 2$	$\lambda = 3$	$\lambda = 4$
50	75	118	157
25	163	382	1383
12.5	4008	80906	845846

choose the nodes maintaining a range of distance between any two nodes. The reason for our choice is that if two nodes are close-by in the DAG, then we can compute the next node from the previous one with a small cost. This can be seen as a “greedy strategy” with regard to the locally optimal nodes in order to obtain the globally optimum solution. Our heuristic was chosen based on several experiments we performed on graphs of small order and checking the number of computations required as the memory was reduced.

Our experimental results show that the non-uniform distribution of Argon2i affects the memory-hardness of the design, causes a weakness. The results of our experiments showing the re-computation penalty for Argon2i are shown in Table 5.7.

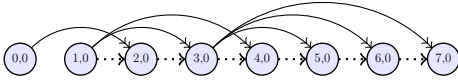


Figure 5.14: Argon2i at t=1, m= 8 blocks

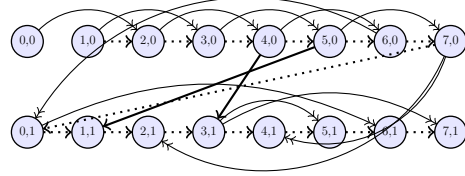


Figure 5.15: Argon2i at t=2, m= 8 blocks

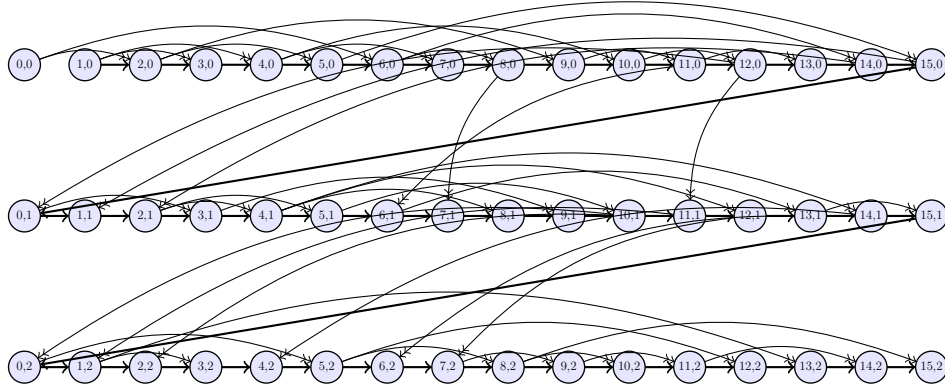


Figure 5.16: Argon2i at t=3, m= 16 blocks

5.6.1.1 Argon2i - Version 1.3 [42]

This version was developed recently to mitigate the attack shown in [98] on the previous version which was the PHC winner. This is similar to the design explained in Section 5.3.1 except for iterations $t > 1$ where for each computation of the block $M_{i,j}$, the new block is XORed to its previous value instead of just overwriting the previous one. The problem without this XOR operation was that for each block computation there was a time gap between the moment the block was used for the last time (through index computation) and the moment it is overwritten. Therefore it was possible to drop the block after last referenced. This is no longer possible in the modified version. However, this version 1.3 gives little overhead on the performance as $(t - 1) \times i \times j$ more XOR operations are performed.

5.7 Performance Analysis

For consistency, we use single-threaded versions of the algorithms. There was no significant difference between the best case and worst case running time of the algorithms. Therefore all the experiments were run 5 times and the average timings have been taken. The performance graph is shown in Fig. 5.17 which shows the execution time vs. memory for all algorithms benchmarked. All parameters and the experimental setup are same as mentioned in Section 5.6. For Argon2, we consider only the cache-timing attack resistant variant, Argon2i, for the performance analysis. The latest version of the code is cloned from [19]. The performance of Argon2i is similar to Catena Dragonfly and slower than the design Rig.

The performance figure shows that Catena-Dragonfly is much faster than Catena-Butterfly, but, the Dragonfly version is shown not to be memory-hard in [46]. The latest version of code at the time of benchmark was cloned from [17] and optimized SSE implementation is used for both the benchmarks. One of the reasons for the slow nature of the Butterfly version is the need for $2 \cdot g$ rows for processing. This property of Catena-DBG, combined with the relatively small read-writes to the RAM makes the overall structure significantly slow. Even the fastest version of Catena-Butterfly-Blake2b-1 can only achieve overall memory hashing speed of around 80 MiB/s. The Catena-Dragonfly is much faster due to the significantly reduced number of rounds as compared to Catena-Butterfly. In addition to this, every node in the Catena-BRG graph has a dependency on two previous ancestors as opposed to three in Catena-DBG. This leads to reduced number of random memory accesses and faster speeds.

For the performance of the design Rig we use the latest version from [20]. We use the optimized implementation with the Blake2b round using AVX-2 instructions. All default settings are used as described in the code and Makefile. One source code improvement is the removal of the writing of the data back to the memory in the last row, this change resulted in around 5% improvement in overall performance for small values of N .

We also include the performance of ‘Scrypt’ [132] algorithm which is the first memory-hard algorithm for password hashing. There are several implementations of Scrypt available, we use one of the fastest variants of the implementation from [18] with AVX2 implementation

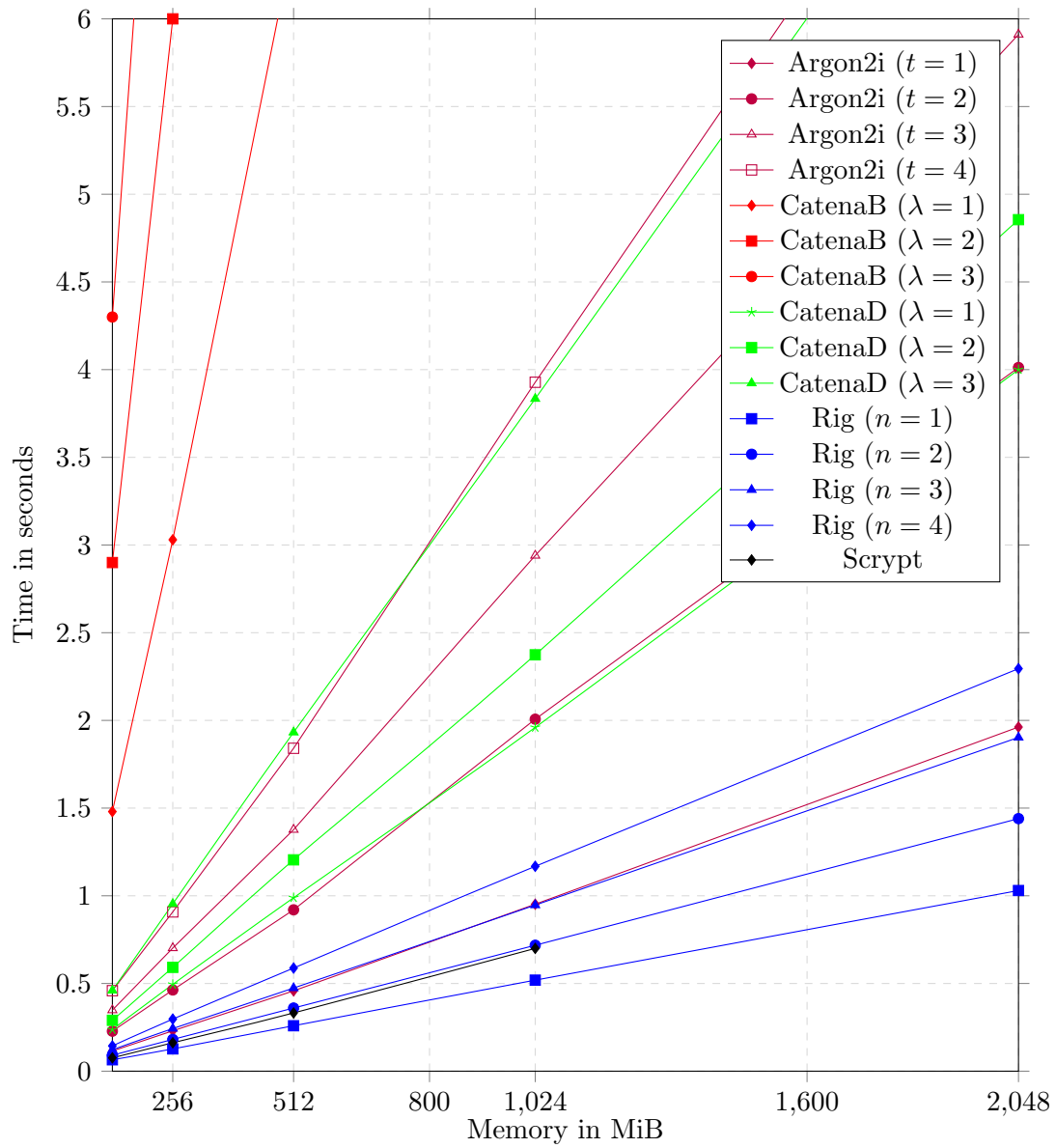


Figure 5.17: Execution time vs. memory, faster feeling of memory is better.

using Blake2b and Salsa64/8.

5.8 Summary

It is difficult to provide a general technique to analyse the time-memory tradeoff for memory hard designs. In this chapter, we propose a technique for traversal of DAGs to analyze the TMTO. Therefore, it is applicable to the designs which can be represented as a DAG. We apply the proposed technique on three cache-timing attack resistant designs namely, Argon2i, Catena and Rig by performing preliminary analysis with various parameters and TMTO options.

The performance graph in figure 5.17 shows the execution time vs. memory for all the memory-hard algorithms benchmarked. It is clear that Catena-Butterfly is the slowest and take a significant amount of time in hashing passwords with moderate to large amounts of memory. The performance of Catena is unlikely to significantly improve even with native assembly implementation.

Argon2i and Rig-Version 2 provide good performance in a wide range of use cases, though Argon2i is slower than Rig. The attack of [98] on Argon2i shows reduction over claimed TMTO defense which is also visible in our analysis as shown in Table 5.7.

It would be interesting to apply our proposed time-memory tradeoff analysis technique to evaluate other complex cryptographic designs similar to password hashing schemes where proposing a standard mathematical/theoretical model for analysis is significantly difficult. One possible extension of this work can be to apply the proposed analysis technique on other password hashing designs as well. Our experimental results show that simple DAGs are efficient in terms of performance but are not memory hard. Use of complex DAG provides expected security against time-memory tradeoff but shows poor efficiency. Combining two simple graphs in the representation of a DAG achieves both security and efficiency. Hence, to provide a general technique to combine different simple graphs for the efficient design (both in terms of TMTO defense and performance) of a password hashing scheme can be another interesting problem to explore.

Chapter 6

On the Improvement of Password Database Breach Detection Technique

The breach of password databases has been a frequently occurring phenomena in the security industry. Many a time, organizations suffering from such a breach may have an interest in not publicizing the occurrence of such an incident. However, sometimes the organizations are in the dark about such breaches even after they have taken place. This necessitates a strong breach detection mechanism. A result by Juels et al. (ACM-CCS 2013) provides a method, known as ‘Honeyword’, for detecting a password database breach. In this technique, the system generates multiple honeywords, which act as fake passwords and stores them along with the real password. Any login attempt with honeywords is identified as a compromise of the password database, as no legitimate user is expected to know the honeywords corresponding to his/her password. The main components of this technique are (i) the generation of honeywords, (ii) typo-safety measures for preventing false alarms, (iii) alarm policy upon detection, and (iv) testing robustness of the system against various attacks.

This chapter is dedicated to the analysis of existing password database detection technique. Specifically, we analyse the limitations of existing honeyword generation techniques. We propose a new attack model and call it ‘Multiple System Intersection attack considering Input’. We show that the ‘Paired Distance Protocol’ proposed by Chakraborty et al., is not secure in this attack model. We also propose new and more practical honeyword generation techniques, namely ‘evolving-password model’, ‘user-profile model’, and ‘append-secret model’. These techniques achieve ‘approximate flatness’, implying that the honeywords generated using these techniques are computationally indistinguishable from the password. In certain specific cases, these techniques can also achieve ‘perfect flatness’ where the honeywords are information theoretically indistinguishable from the password. Our proposed techniques overcome most of the risks and limitations of existing techniques. We prove the flatness of our ‘evolving-password model’ technique through experimental analysis. In our experiment, we use publicly available password databases leaked by hackers. To validate our claim, we compare our proposed techniques with the existing honeyword generation techniques on various parameters.

The roadmap for this chapter is as follows. In Section 6.2 we present an overview of the

‘Honeyword’ technique as proposed by Juels et. al. In Section 6.3, we detail the existing attacks for analyzing the security of honeyword based authentication and also explain our proposed attack model on the same. Further, analysis of existing honeyword generation techniques is included in Section 6.4. Our proposed honeyword generation techniques and security analysis are presented in Sections 6.5 and 6.6 respectively. Subsequently, the details of the experiment showing the flatness of our proposed evolving password model are documented in Section 6.7. Finally, we provide a comparison of our proposed technique with existing approaches and conclude the work in Sections 6.8 and 6.9 respectively. The original contribution of this thesis is from Section 6.4 to Section 6.8.

6.1 Introduction

There are several ways to prevent the attacker from performing dictionary attack or to increase the complexity to carry out this attack manifolds. Making the password hashing algorithm resource consuming is one way to stop the adversary from pre-computing the dictionary. This was the objective behind the Password Hashing Competition (PHC) [16]. Another approach is to introduce confusion by adding a list of fake passwords along with the correct password. This would discourage the adversary to mount dictionary attack even after compromising the database. This approach, proposed by Juels et.al. [103], of using fake passwords can help in detecting password database breaches. Specifically, any login attempt with one of the fake passwords detects the breach. The idea was influenced by some other existing techniques mentioned below. The honeypot technique [64], introduced in early 90’s, is a system or component which influences the adversary to attack the wrong targets. Honeypot accounts are fake accounts created by the system administrator to detect password database breaches. Honeytoken is a honeypot that contains fake entries like social security or credit card numbers [152] to identify malicious activity. Kamouflage [48] is a theft-resistant password manager that creates multiple decoy password lists along with the correct password list.

Frequent cases of password database breaches like that of LinkedIn in 2012 [66], Adobe in 2013 [146], eBay in 2014 [110], AshleyMadison in 2015 [67] etc., are indicative of security issues in current password-based authentication systems which are unable to ensure user privacy. In the case of LinkedIn, breach of 6.5 million passwords was reported in 2012. However, in May 2016, additional 100 million passwords were found, that were reportedly leaked in the same breach in 2012 [100]. In response to this, LinkedIn invalidated all the passwords that were not changed since 2012 [66]. No efficient solution for successfully detecting such database breaches was reported in the literature prior to [103]. This makes the Honeywords technique [103] a significant contribution in the detection of password database breaches. In this technique, the server generates multiple fake passwords called honeywords for each user, and stores them along with the actual password chosen by the user. Even if an attacker gets access to the password database, she would not be able to distinguish the actual password from honeywords. Therefore with a very high probability, she is expected to enter a honeyword to carry out the attack. If a honeyword is

entered instead of the password, the system raises an alarm, thus detecting the compromise of password database. The efficiency of this system mainly depends on the honeyword generation scheme to generate honeywords that are indistinguishable from the real password. The authors in [103], provide some heuristic honeyword generation techniques, along with detailed analysis of the system implementing the honeywords technique. Continuing in the same line of research, we provide an experimental method for quantifying the flatness of honeyword generation schemes. We also implement a distance-measure between password and honeyword using ‘Levenshtein distance’ [93] to avoid false detection when a legitimate user enters a honeyword because of a typing error.

6.2 Overview of the Honeywords Technique [103]

Consider an authentication system with n users $u_1, u_2, \dots, u_n$ where u_i is the username of the i th user. Let p_i denote the password of i th user. Any typical system maintains a file F listing pairs of usernames and passwords hashed with a password hashing algorithm (PHS) H as

$$(u_i, H(p_i)).$$

To enhance confidentiality of users and to ensure detection of breaches, a system implementing honeywords creates a list W_i of distinct words called sweetwords for each user u_i , represented as

$$W_i = (w_{i,1}, w_{i,2}, \dots, w_{i,k})$$

where k is a small integer and recommended value of $k = 20$. Exactly one of the values in the list W_i is the actual password, i.e., $\exists j$ such that $w_{i,j} = p_i$. Let $c(i)$ denote the correct index of the password p_i for username u_i in the list W_i , for instance,

$$w_{i,c(i)} = p_i.$$

The correct password is called “sugarword” and the remaining $(k - 1)$ words in the list W_i are called honeywords. These honeywords are generated by the system. Let $Gen(k)$ be the procedure to generate the list W_i and index $c(i)$. The index $c(i)$ is stored in the *honeychecker*, an auxiliary secure server incorporated within the system (the detailed information about *honeychecker* is given later). At the time of authentication, the system interacts with the *honeychecker* to get the correct index of the i th user’s password in the list W_i . The database stores the entry for each user u_i , as:

$$(u_i, H_i)$$

where H_i is a k -tuple of hash values of sweetwords. That is,

$$H_i = (H(w_{i,1}), H(w_{i,2}), \dots, H(w_{i,k})).$$

Honeychecker: It is assumed to be an auxiliary secure server where secret information can be stored. Ideally, we want the *honeychecker* to maintain minimal information about the secret state. It stores the index $c(i)$ corresponding to user u_i . As explained in [103] the server communicates with the *honeychecker* by sending encrypted and authenticated messages over dedicated channels. The *honeychecker* accepts the following two commands:

- **Set** (i, j) : Sets the index of sugarword for user i at position j in the array of sweetwords, which means $c(i) = j$.
- **Check** (i, j) : Checks if the index of sugarword for user i is at position j . It returns the result of this check to the server. It may additionally raise an alarm if the check fails.

The only job of a *honeychecker* is to securely store some information about the secret state and not the entire secret state. Note that this information is useless, unless paired with the password database. This is unlike any crypto-module that contains a key and is expected to perform cryptographic operations. Therefore, even if the *honeychecker* is compromised, the security of the system is only reduced to the level of security without incorporating honeywords and *honeychecker*.

Flatness: Let z denote the adversary’s expected probability of guessing the password from the list W_i . An adversary can guess with probability $1/k$, therefore $z \geq 1/k$. A honeyword generation method is “ ϵ -flat” for a parameter ϵ , if the maximum value of the guessing probability z for all the adversaries is ϵ and the technique is **perfectly flat** if z is equal to $1/k$. If the value of ϵ is very close to $1/k$ for a technique then it is called ‘approximately flat’.

In [103], the authors have illustrated several flat or approximately flat $Gen(k, p_i)$ procedures that are explained in the following section.

6.2.1 Honeyword Generation

Honeyword generation techniques have been divided into 2 categories in [103] depending on their effect on the user interface (UI).

1. **Legacy-UI** password changes: The choice of the password of a user is not influenced by the UI.
 - (a) **Chaffing by tweaking:** In this technique, ‘ $t \in \mathbb{N}$ ’ selected character positions in the password are tweaked to generate honeywords. Example, if the last $t=2$ selected positions are chosen, the honeywords for password abcde12 could be abcde56, abcde97 etc. Digits are replaced by randomly chosen digits, alphabets by randomly chosen alphabets and special characters by randomly chosen special characters.

- (b) **Chaffing with a password model:** To generate honeywords, this technique, makes use of a probabilistic model of real passwords which can be analysed from publically available password databases.
 - (c) **Chaffing with ‘tough nuts’:** The ‘tough nuts’ is defined as honeywords that are harder to crack than the average, e.g., a 256-bit random string. Tough nuts can be used to make the job of an attacker more difficult.
2. **Modified-UI** password changes: Choice of the password of a user is influenced by the UI in this password change procedure.
- (a) **Take-a-tail:** It is identical to the chaffing-by-tweaking method discussed above, except that the tail of the user password is randomly chosen by the system. Example from [103],

Propose a password:****;
Append 413 to make your new password: **413**;
Enter your new password:*****.

The value 413 is the random tail generated by the system for each password. Honeywords are now generated by replacing the digits ‘413’ by other randomly chosen digits.

6.3 Attacks

In this section we describe various attacks for analyzing the security of honeyword generation techniques.

6.3.1 Denial-of-Service (DoS) Attack

This attack is an attempt to make a service unavailable to its intended users. This is a potential problem associated with the honeyword generation technique. When honeywords are used for authentication and a breach is detected, the system takes action according to the chosen policy. Typically, the policy blocks the user from logging in the system at least temporarily. To impose denial of service to multiple users, an adversary can intentionally provide a honeyword for authentication without actually compromising the complete database. This is possible when the generated honeywords are easy to predict. Therefore, the $\text{Gen}(k)$ algorithm is an important deciding factor while analysing the security of a system implementing the honeyword technique.

6.3.2 Multiple System Intersection considering Output (MSIO) Attack

Humans have a tendency to choose the same password for multiple websites [83]. This behavior opens a new attack surface to compromise the user password assuming the existence of

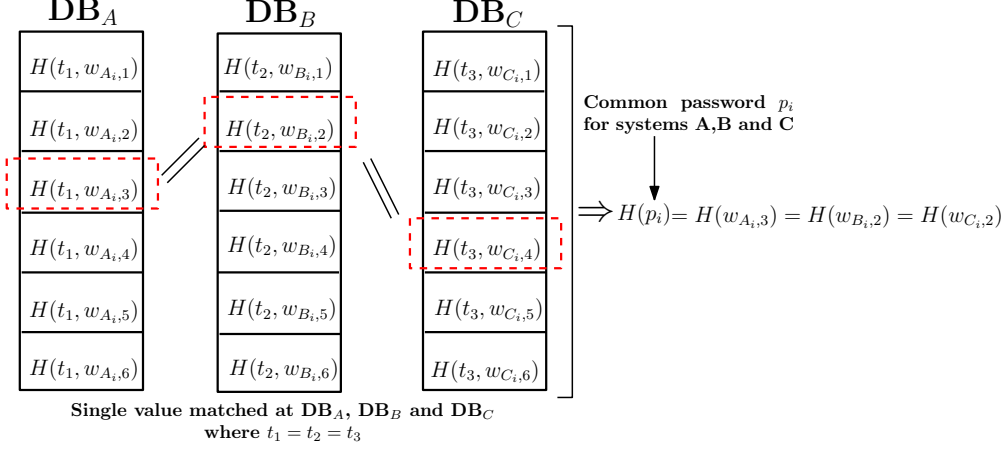


Figure 6.1: MSIO attack: The system A, B and C contains databases DB_A , DB_B and DB_C respectively including the sweetwords for the user i which are available to the attacker. Assuming password p_i is same across all the systems and honeywords are distinct, intersection of the database entries provides the password.

zero collision on the honeywords corresponding to the same password used for multiple sites. Let the databases of three different systems A , B and C as shown in Fig. 6.1 represented by DB_A , DB_B and DB_C respectively, be available to the attacker. The sweetwords containing distinct honeywords corresponding to user i are represented as $(w_{A_i,j})$, $(w_{B_i,j})$ and $(w_{C_i,j})$ where $1 \leq j \leq 6$ for system A , B and C respectively. Let, t_1, t_2, t_3 represents the tweak values corresponding the databases DB_A , DB_B and DB_C respectively which may be associated with the input password. These tweak values are fixed for a server and represent, for example, a shift by an integer value for mapping the same password to different values. We assume $t_1 = t_2 = t_3$, hence the use of tweak shows no effect on the database values. Therefore randomness of entries of databases only depends on the choice of sweetwords, not on the value of the tweak and we assume the values $t_1 = t_2 = t_3$ are known to the attacker. The list of sweetwords for each system intersects at the value $H(w_{A_i,3}) = H(w_{B_i,2}) = H(w_{C_i,2})$. Assuming honeywords are distinct, this collision helps the adversary to identify the password p_i for user i . The attack complexity is equal to the effort to get p_i from $H(w_{A_i,3})$ or $H(w_{B_i,2})$ or $H(w_{C_i,2})$. Hence, this attack defeats the detection ability of the honeyword technique, which is based on the indistinguishability of the password from the honeywords. As the intersection is taken on the output stored on multiple systems corresponding to the same user password, in this paper we specify this attack as ‘Multiple System Intersection considering Output (MSIO) Attack’ while it is defined as ‘Multiple Systems Intersection’ attack in [103].

6.3.3 Multiple System Intersection considering Input (MSII) Attack

This attack is also motivated by the tendency of humans to choose the same password for multiple websites [83]. Since different websites are assumed to have a different non-random tweak value associated with the user password, thus the same password associated with the same user is mapped to a different value depending on the tweak. Hence intersection of the databases of different websites corresponding to a fixed user does not lead to a common value. As shown in Fig. 6.2, let the databases of three different systems A , B and C are DB_A , DB_B and DB_C respectively be available to the attacker. Distinct sweetwords corresponding user i are represented as $(w_{A_i,j})$, $(w_{B_i,j})$ and $(w_{C_i,j})$ where $1 \leq j \leq 6$ for systems A , B and C respectively. The randomness of entries in the database depend on the choice of the sweetwords, since the corresponding tweak values $t_1 \neq t_2 \neq t_3$ are easy to compute if one or more of t_1 , t_2 or t_3 are known. We claim that the overall computational complexity of the MSII attack is asymptotically similar to the MSIO attack. Since the intersection of database entries does not provide any common value, the attacker can only try to guess the sweetwords for a fixed user from one of the available systems such as A . Let an attacker require effort e to get any of the $w_{A_i,j}$ from $H(w_{A_i,j})$ where $w_{A_i,j}$ is the j -th sweetword of user i on system A . For simplicity, we can assume that the effort required to compute t_1 is negligible with respect to e . Then with $k \times e$ effort all k sweetwords can be obtained where one of them is the sugarword/password and $k = 6$ for Fig. 6.2. When these k -values are tested on systems B and C , only the real password is expected to match the output, since honeywords are expected to be distinct on different systems. Hence, the overall effort only depends on the number of honeywords, which is constant times the effort required in MSIO attack. We refer to this attack as Multiple System Intersection considering Input (MSII) Attack, since it is based on the information obtained by getting an intersection on the inputs. In order to prevent this attack, the random secret must be hard to guess. This can be ensured if multiple systems map the same password to different outputs by ensuring that each system has a distinct random secret.

6.4 Existing Honeyword Generation Techniques and Their Limitations

The efficiency of database breach detection mechanism is largely dependent on the honeyword generation mechanism. Honeywords generated using perfectly flat honeyword generation techniques are indistinguishable from the real passwords. This prevents the adversary from determining the sugarword from the list of sweetwords. It is challenging to design a perfectly flat honeyword generation technique as passwords are chosen by the users and human behavior cannot be modeled using a generalised thumb rule. In this section, we discuss the limitations while suggesting possible improvements for honeyword generation and provide references when the included analysis is from the existing literature.

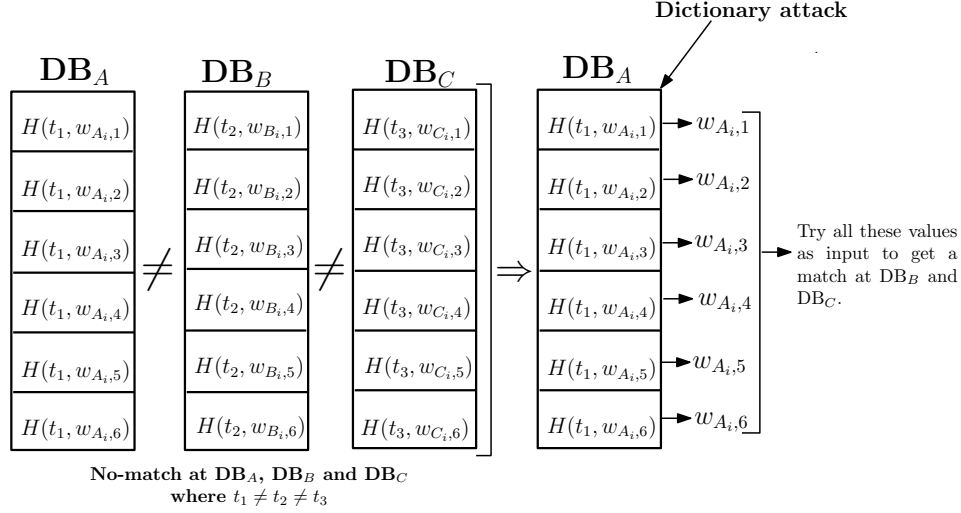


Figure 6.2: MSII attack: The system A, B and C contains databases DB_A , DB_B and DB_C respectively including the sweetwords for the user i which is available to the attacker. All values across all the databases are distinct. Assuming password p_i is same across all the systems, attacker implements dictionary attack on one of the systems (A for this case) and gets all the sweetwords at A. To get the password p_i attacker gives these sweetwords as input to get a match at DB_B and then at DB_C .

1. Following are the techniques introduced under the ‘Legacy-UI’ category of honeyword generation. The limitations of this category are that they do not prevent MSIO or MSII attacks.

- (a) **Chaffing by tweaking:** In this techniques, t -positions of the password are tweaked to generate honeywords. Characters at each of the t -positions are replaced with randomly chosen characters of the same type. In “Chaffing by tweaking digits”, the last t -positions containing digits in the password are tweaked. In [80], the authors acknowledge the bias in human behavior while selecting digits in their passwords. According to report [52], which is based on the analysis of hacked Adobe database, the digits chosen by users follow specific patterns like birthday, any important date, year, some consecutive numbers etc. Therefore, replacing them with some random number makes honeywords, easily distinguishable from the password. Ideally, such algorithms should try to ape the human behavior while replacing the characters, which is not always easy to formalize.

If username and passwords are correlated like username: Terasa and password: mother, then it is easy for the adversary to distinguish the password from honeywords. Moreover, it is difficult to prevent such correlation between the username and the password.

- (b) The “**Close-number-formation (CNF)**” method suggested in [56] is presented as an improvement over chaffing by tweaking technique. It makes use of two lists:

$\{+, -\}$ and $\{1, 2, 3\}$. User chooses password along with one character from each of the lists. The digits used in honeywords are plus/minus of 1/2/3 of the password digits. For example, if the password is 28March2000 then the appropriate range for the possible honeywords can be: 26March1998 - 30March2002, etc. Even though the authors claim that this is an improvement, in this case, a user can predict his/her own honeywords with high probability while ideally, users should not know the honeywords. False alarms would also be frequent since the honeywords are extremely close to the passwords and there is a high chance that a legitimate user made typo error. To formalize the proposed approach, it needs to consider many different parameters before providing the range like range of dates, possible range of year and sequences etc.

- (c) **Chaffing with a password model:** This model is based on publicly available password databases which might also be available to the attacker. Therefore, the attacker can also generate the same list of honeywords for a chosen password. The ‘modeling syntax’ method makes use of the same syntax as that of the password to generate honeywords. For example, the password ‘mice3blind’ can be decomposed token-wise as ‘ $A_4|D_1|A_5$ ’ where A stands for alphabet-string and D for digit-string. Lengths are also used in the suffix. Following the same syntax, a possible generated honeyword could be ‘rose5rings’. As mentioned in [80], the list of 10000 most commonly used passwords [3] includes the following

bond007 james007 007bond

The authors in [80] further explain that with the modeling syntax method, it is easy to distinguish honeywords from the above mentioned passwords if honeywords are generated by randomly replacing the tokens preserving their length. However, it is difficult to restrict the user from choosing passwords with such obvious correlations.

- (d) **Chaffing with ‘tough nuts’:** In this model, the system uses some tough nuts as honeywords to make it difficult for the adversary to crack the hash of the password. Authors in [103], suggest using this method by combining it with some other method. However, according to [80], most of the passwords are simple combinations of digits, alphabets and special characters and not the tough nuts. Therefore, the adversary can easily skip to search for the tough nuts. As a result, this strategy will reduce the overall complexity required to compromise the system, thereby defeating one of the purposes of honeywords which is to increase the complexity required to break the system. However, the use of tough nuts as explained in [103] was introduced to discourage the adversary to mount dictionary attack which is not possible in the real scenario if it is preferable for the attacker to skip them.
2. Following are the techniques introduced under the ‘Modified-UI’ category of honeyword generation. The motivation behind this category is to introduce randomness in user-chosen password to prevent MSIO and MSII attacks.

- (a) **Take-a-tail [103]**: This method appends system generated digits at the end of user-chosen passwords. This approach introduces randomness in user selection. This technique helps to mitigate the correlation problem that could exist between username and password. This is because all honeywords will differ from the password only in the appended digits. It also prevents multiple system intersection attack. However, as explained in [57] it is difficult for users to remember random digits and especially in the case when multiple sites implement this technique and each site gives the user a unique number that needs to be memorized. Also, it is highly likely for the system to raise false alarms, especially in cases where random digits are selected from a small space.
- (b) The “**Modified-tail**” is proposed in [56] as an improvement over the take-a-tail technique. Number of possible strings of length m , with m different characters is $m!$. The prefixes of each of these strings is also unique. Therefore, in this technique, the user needs to select $m - 1$ characters from the provided set of m special characters (to fix the prefix of his own choice) and system appends the remaining one at the end to form the tail of the password. The honeywords are generated by taking all possible combinations of the tail (which are $m!-1$). For example (from [56]), let the user-chosen password to be ‘tea’ and permutation of special characters chosen from the set $S = \{ @, ?, | \}$ be ‘@?|’. Then the list of honeywords are:

tea?|@, tea?@|, tea|?@, tea|@?, tea@?|, tea@|?

It is difficult to prevent false detection when a legitimate user makes a typing error in this scenario as honeywords are very close (distance-wise) to the password. Apart from the issue of typing errors, the burden of memorizing extra $m - 1$ characters from a system generated list is placed on the user. Therefore, this technique is not at all user-friendly.

- (c) The “**Caps-key based approach**” proposed in [56] also claims to be an improvement over the ‘Take-a-tail’ method. In this technique, the user is asked to choose any two characters from the password and convert them to upper case. To generate honeywords, the system then selects any two characters from the password and converts it to upper case. For a password of length l , total $\binom{l}{2} - 1$ such honeywords are possible. For example (from [56]), the password chosen by the user is ‘animal’ and the two characters selected by the user are ‘ni’. Therefore the new password becomes ‘aNI mal’. The honeyword list contains (for $k = 6$),

AnImal AnimaL aNimaL aNI mal ANimal anImAl

This technique makes the system prone to false detection in case of typing errors. It also increases the possibility of a DoS attack by making the guessing of honeywords easier.

- (d) The **Paired Distance Protocol** or **PDP** method proposed in [57] creates a secure circular list containing alphabets and digits with default 36 entries in a

random order. The user selects a random string RS of length l from the set of alphabets and digits along with the password. The distance between any two characters of the RS string is called the ‘paired distance’. This paired distance is measured for the RS string considering the position distance (clock-wise) between the same characters of the secure circular list. If the first character of RS is not known then it is difficult to guess RS if only the paired distances are stored in the database. For a circular list of size n , it provides total $(n - 1)$ possible values. This idea is used in this technique and honeywords are generated randomly by choosing different paired distances for each password. The *honeychecker* keeps the index of the correct password alongwith the record of the first character of the RS string.

Attack on PDP: As mentioned in [57]

- Same circular list could be shared among m different systems and
- User may use same RS for different accounts

MSIO attack is possible in both the above scenarios. Suppose the user chooses the same RS for different systems and the circular list is available alongwith the database entries to the attacker. In this case, the intersection of entries from multiple systems can reveal the actual password and the attacker only needs a maximum of $n - 1$ additional trials (where n is the total number of entries in the circular list) to predict the password when the first character is not known. Hence this technique fails to provide security against MSIO attack.

Now suppose different circular lists are provided to different systems and the attacker has complete access to these lists and the databases corresponding to the m -different systems. In this case, assuming that the user chooses the same RS , the attacker has to find all k sugarwords for one of the systems. Trying these sugarwords on other systems, the attacker can find a match for the password, thereby successfully mounting MSII attack. Hence, the proposed method is not secure against MSIO and MSII attack. Apart from the above attacks, it forces the user to randomize the choice of RS which is an overhead for the user to memorize.

6.5 The Proposed Techniques

In this section we propose approximately flat honeyword generation techniques considering typo-safety and other policy choices.

6.5.1 Password Policy

It is difficult to impose strict rules to enhance the entropy of user-chosen passwords. However, users should avoid using dictionary words as passwords, selecting the same password on multiple websites, using commonly used passwords, or choosing passwords correlated to the username. For example, if the username is ‘titanic’ and password is ‘rose@1997’, then there

exists a correlation which can help the attacker to uniquely identify the password. A study on password policy as explained in [84] shows that the most restrictive password policies do not provide greater security. Therefore our system imposes the following less restricted and practical to implement conditions on password selection.

1. Username or it's sub-string should not appear in the password.
2. The password should contain at least 8 characters including alphabets, special symbols and digits.

6.5.2 Typo-Safety

According to an analysis given in [63], the error patterns in mini-QWERTY keyboard show the rate of human errors as: 'Substitution' errors - 40.2%, 'Insertion' errors - 33.2%, 'Deletions errors' - 21.4% and 'Transposition' errors - 5.2%. There is a possibility that a legitimate user may enter a honeyword because of typing errors if honeyword is very similar to the password. Such issues can be resolved with a very high probability by maintaining a minimum distance between the password and each generated honeyword. We suggest using 'Levenshtein distance' [93] to compute the distance between password and honeywords. 'Levenshtein distance' is calculated by counting the number of deletions, insertions, or substitutions required to transform one string into another. It can be used to calculate distances between variable length strings. In this way, all types of human typing errors can be taken into account.

6.5.3 Proposed Honeyword Generation Techniques

We propose the following techniques for the honeyword generation.

1. Legacy-UI password changes: The password selected by the user is not influenced by the UI
 - (a) **Evolving password model:** We first define the key terms for the better understanding of the scheme. These terms are defined with respect to the available disclosed password databases.
 - **Token:** We consider token as a sequence of characters that can be treated as a single logical entity. In our context, for a given password, tokens are the alphabet-strings(A), digit-strings(D) or special-character-strings(S).
 - **Pattern:** The different combinations of tokens form patterns for a password, e.g., ADS_1 , AS_2D , S_1AS_1D etc.

Note: To create honeywords indistinguishable from user password we do not preserve the length of alphabets and digits, however, we preserve the length of special-characters. Therefore the length of the special-character is mentioned as a subscript of S in the representation of the pattern.

- **Frequency:** It is the number of occurrences of the tokens or the password pattern in the available password database.

This evolving password model uses the probabilistic model of real passwords to generate honeywords. Specifically, it pre-computes the frequencies of each password enlisted in an existing database and also computes the frequency of each individual tokens (alphabets-strings, digits-strings, special-characters-strings) of the password following the technique suggested in [164]. The frequencies of the password pattern and its tokens are stored in a file. The database of these frequencies is updated with each new user registration, therefore the frequency list evolves with each new registration. Honeywords are generated by first matching the frequency of the password pattern as a whole and then matching the frequency of the tokens of the password. To match the frequency considering the tokens, we follow the technique proposed in [164], where the probability of the honeyword is the product of the probabilities of the tokens used to derive it. The whole procedure can be covered with two different steps of computation as listed below.

- i. Algorithm to compute frequencies of password pattern and tokens
- ii. Algorithm to generate honeywords with the help of pre-computed frequencies and update the frequency lists, which means evolving the frequency lists with each new registered user password.

Next we provide an example of generating honeywords from a given password: 'abcde123 %'. To generate the honeywords compute the frequency of the pattern of abcde123 % (ADS_1) and the frequency of the tokens: 'abcde', '123' and '%'. Let frequency of the pattern of 'abcde123 %' matches with the pattern of type AS_1D . Now choose tokens with frequencies similar to the frequencies of 'abcde', '123' and '%' respectively. The token '123' matches with '9', '%' with '_' and 'abcde' with 'secret'. Therefore one of the honeywords is 'secret_9'.

In this technique, the *honeychecker* keeps the index of the correct password. At the time of authentication, the index value is verified by the server through a secure communication with the *honeychecker* as explained in [103].

- (b) **User-profile model:** It generates honeywords by combining some details from the user profile and checks the threshold of minimum distance with the password. One technique to generate honeywords is to create different sets containing tokens of each type, for instance, alphabet-strings, digit-strings and special-character-strings from provided user details. Then make possible combinations of the elements from each set of tokens. The combinations thus generated are used as honeywords. This technique is best suited when the password contains some substring correlated to the user profile. According to a 2013 report [157] by Google which includes top 10 worst password ideas, users tend to select passwords related to themselves. Hence, this technique generates honeywords indistinguishable from most of the user passwords. To provide typo-safety, a minimum distance between the honeywords and the password should be maintained and we suggest

the default value to be 3. One possible approach to create honeywords is following the steps below (considering the password policy mentioned in Section 6.5.1). For listing the special characters, we can have the list of commonly used special characters.

- i. Create a separate list of tokens named, `token_digits`, `token_alphabet`, `token_specialChar` from the information provided in the user profile.
- ii. To create k honeywords, take k different combinations of elements from each token lists, satisfying the password policy of the service.
- iii. Compare the tokens of the password with the tokens of the honeyword. Reject the honeyword if more than one token matches with the password.

As an example, let the following information about a user profile be known:

Name: Alice Wood; **DoB:** 19/07/1995; **Address:** 54 west 28 street;

Name of the first pet: Jerry

Password: Tom!54street

Then the system can generate the following for this user:

`Token_digit`={19, 07, 1995, 54, 28}

`Token_alphabet`={Alice, Wood, west, street, Jerry}

`Token_specialChar`={/, !}

Honeywords: Wood/1995; Alice_19; Jerry#19wood, Alice@28street

Flatness: As per the report [157] of Google, the top 10 worst password ideas are related to the personal details of the user. Another work on targeted user password guessing [162] leverages the tendency that majority of the population uses personal details for choosing a password. Therefore, it is easy to interpret that honeywords generated from user profile will be indistinguishable from user passwords for the majority of the population. Thus this technique is expected to provide perfect flatness. However, when passwords are not related to the user profile, it is easy to differentiate a password from honeywords. But due to the general human tendency, the probability of this happening is extremely small, thus, this proposed technique provides perfect flatness. However, in general, it is approximately flat.

In this technique, the *honeychecker* keeps the index of the correct password. At the time of authentication, the index value is verified by the server through a secure communication with the *honeychecker* as explained in [103].

2. Modified-UI password changes: The password selected by the user is influenced by the UI, while minimizing the usability of the system.
 - (a) **Append-secret model:** In this technique, at the time of registration, the user provides his/her password. The system asks for an extra entry, say l where l ranges from two to four characters in length. The system generates a random string r of default length of 3 considering digits, alphabets and special characters.

It computes $f(pwd \parallel l \parallel r)$ and outputs x where f is a collision-resistant one-way function. Honeychecker stores both the index of correct password and the value r . Considering the *honeychecker* as a secure database which manages minimum storage, we define the random secret to be a small value and recommend the default length as 3. We also recommend that a different random r is used for each user. This is to prevent the disclosure of (remaining) secrets even when the communication between the server and the *honeychecker* is compromised and the communicated secret is disclosed. The database stores ' $H(pwd \parallel x)$ ' instead of $H(pwd \parallel l)$, where H is a password hashing algorithm. Even after successful implementation of dictionary attack it is still challenging for the attacker to identify the real password. Honeywords generated by this technique differ in the choice of l which influences the value x and the final hash. To provide typo-safety we recommend the length of l from 2 – 4 to maintain 'levenshtein distance ≥ 2 '. As r is randomly selected for each site, even if the user selects the same l for multiple sites the intersection of information from multiple sites does not reveal the actual password. Therefore, it provides security against both MSIO and MSII attack.

Example:

Enter Password: abcde

Use any string of length (2-4): 1998

System generated secret: \$8d

System computes: $f(abcde \parallel 1998 \parallel \$8d) = 4e7j@$

Database stores: $H(abcde \parallel 4e7j@)$

Therefore, the dictionary attack may reveal the password: *abcde4e7j@* but not the actual input: *abcde1998*. To get the password, the value of r is required.

Honeychecker: Due to the minimal storage requirement for the *honeychecker*, we do not recommend using random secret r of length 128 bits or above which enhances security. As *honeychecker* is only for secure storage and not for secure computation, we do not allow cryptographic computation inside the *honeychecker* where the use of HMAC or other crypto-primitive was possible. If distinct key ≥ 128 bit is allowed then it requires more storage. Using a single key for all user is possible but in that case compromising a single communication compromises the whole database. To avoid the communication of the key, a possible approach is to perform all computation inside the *honeychecker*. However, *honeychecker* is not supposed to be a crypto-module. Therefore, we recommend a small and distinct value of r for each user.

We assume the communication between the *honeychecker* and the server is over dedicated channels and/or encrypted and authenticated as explained in [103]. At the time of authentication the secret r is first communicated and then the *honeychecker* verifies the command, 'Check: i, j ' as explained in Section 6.2.

Usability Issues: In our proposed technique the user needs to select and mem-

orize a string l of length 2-4 apart from the initially chosen password. The choice of length from 2-4 for l is for enhancing the security of existing password-based authentication with minimal burden on the user. Authors in [50, 47] show that humans can learn multiple strong passwords over time through spaced repetition. Thus, memorizing the string l should not be a difficult task. The constraint of spaced repetition is not satisfied immediately in the proposed scenario, however, the results disprove the perception that humans can not memorize strong passwords. Based on this existing result the burden of 2-4 length strings is not a significant one. Therefore the Append-secret model does not really have a strong tradeoff between security and usability. It demonstrates how security can be enhanced without compromising usability.

6.5.4 Alarm Policy

Different alarm policies are implemented for passwords based on different levels of hardness to guess the password. This would prevent strong actions in case of DoS attacks targeting the vulnerability of high frequency passwords. As mentioned in [103], policies need not be uniform across a user population. Following are some of the policies suggested in existing literature.

Analysis on Different Alarm Policies

- As suggested in [80], we can limit the number of login attempts that can be made using honeywords in a certain period of time. Based on the limit, an action like refreshing all existing passwords can be taken.

A single login attempt with honeywords suggests either the compromise of the password database from server or weakness in honeyword generation technique implemented by the server. However, when a threshold limit of attempts is crossed, there is a high chance that it is due to a database breach. In such a situation, refreshing the passwords is the most suitable solution even though it puts the burden on the users. Moreover, to ensure security and privacy, enforcing the change of passwords after regular intervals of time is a common practice.

- Per-user Policy: As suggested in [103] the system can generate fake accounts called honeypot accounts known only to the *honeychecker*. Any login attempt to the honeypot account with honeyword suggests a breach of password database file. An alarm must be raised in this case. Note that this case requires stronger measures as compared to the measures taken when login attempts to legitimate accounts are made using honeywords. This is because an attacker could have guessed the honeyword for a valid account, or a legitimate user could have made a typing error. The policy for handling such cases must be additionally included with the policies for other cases.

A system with dummy accounts makes it vulnerable to DoS attacks as explained in [103]. However, we can have a system where the probability of guessing the honeyword is low. If we have a system which allows k honeywords and possible options of honeywords is s for any password then we can restrict this DoS attack. This is possible if the probability to hit the honeyword is very low, which means the value of $(k - 1)/s$ is very low.

- **Selective alarm Policy:** This policy as mentioned in [103] requires the system to raise an alarm if honeyword for a sensitive account such as administrator account is used for login. Such a case can also be exploited to launch a DoS attack. Therefore, this case must also be handled differently from the general policy.

Proposed Alarm Policy

Analyzing the existing suggestions for deciding the effective alarm policy, we propose the following.

- (a) If a honeyword, corresponding to a password with very high frequency (e.g., ≥ 0.2) is entered, only inform that user to change his/her password. When honeyword corresponding to a relatively uncommon password is entered, ask all users to change the password.
- (b) If there are repeated attempts within a short span of time, count the number of attempts and send notifications to users, to change passwords, following the approach given in point (a).

On Measuring the Password Probability: The work [164] suggests a method to compute the probability based on the frequency of each token in the password. The frequency can be analysed from compromised real databases which capture the distribution of real user-chosen passwords. We can compute the probability of the tokens such as alphabet-strings(A), digit-strings(D) and special-character-strings(S) for a given password from the frequency list computed on real data. The probability of a password can be estimated as the product of the probabilities of all its tokens.

6.5.5 Managing Old Passwords

As suggested in [103] there is no need to store the hash of the old passwords. Even if the old passwords are required to be stored, they may be stored in a separate database file. This database can be accessed, to ensure that the user does not reuse an old password.

6.6 Security Analysis

In this section, we provide the security analysis of the proposed models against the attacks explained in Section 6.3, and some other common attacks.

6.6.1 Brute-Force Attack

For this attack scenario, we assume that the *honeychecker* is not compromised and the adversary gets access to the password database from the server. Then the success probability for the attacker to get the real password depends on the flatness of the honeyword generation technique. In the case where ‘user-profile’ model is used for honeyword generation, the technique will provide approximately perfect flatness since the probability of a password being related to the user profile is very high. Therefore, in this case, brute-force attack does not help to uniquely identify the password. In the case of ‘evolving-password’ model, the honeywords are generated such that their frequency is similar to that of the password selected by the user. Hence adversary gets no advantage in guessing the real password without additional information that can be gathered from other attacks such as social engineering attacks. In the case of ‘append-secret’ model, the attacker can get the value x from brute force attack as explained in Section 6.5.3. To get the actual input, the adversary needs to guess both the user and system selected random secrets. This sums up to all possible combinations considering user choices for l of length 2-4 and the random choice of length 3 made by the system. This requires extra effort of

$$((I) + (I) + (I)) \times (I)$$

computations for each password, where I is the input space for random secret and the user input.

6.6.2 Targeted Password Guessing

Personal information of a user may help an adversary in distinguishing the actual password from honeywords. However, if user-profile model is used for honeyword generation, it increases the difficulty for adversary, provided that passwords of the same user from multiple sites are unavailable to the adversary. In case of the evolving-password model, if the password is correlated to the user profile then it is highly likely that adversary will be able to distinguish the password from honeywords. Password guessing is an effective attack against weak popular passwords, passwords re-used across multiple website and passwords related to user profile as mentioned in [162]. Therefore, a hybrid approach can be taken by merging both models to generate honeywords. In that case, tokens can be generated from both user profile and password frequency and

honeywords can be generated with different combination of the tokens. However, this hybrid approach cannot prevent MSIO or MSII attacks. The append-secret model can provide security against MSIO and MSII attacks and hence prevents the disclosure of the password for a targeted user.

6.6.3 Denial-of-Service Attack

Honeyword generation from user profile model requires details of the personal information of each individual. It is not unusual for an adversary to collect personal information of any user through social engineering attacks [129]. This makes it easy for an adversary to guess honeywords and carry out a DoS attack on the system. Therefore predicting honeywords is not very challenging and hence DoS attack is easy to implement in such cases. Moreover, targeting some specific individuals would be easier, even though honeyword generation techniques provide flatness close to uniform. This is because honeywords generated from user profile are indistinguishable from password. Therefore considering the possibility of social engineering this technique is not DoS resistant. However, the probability of hitting the honeyword can be decreased as explained in Section 6.5.4 under ‘Per-user Policy’ by increasing the information space from the user profile. We can say that for user-profile model, given a password, an adversary can provide a honeyword with negligible probability. This is possible when space of honeyword is large for a targeted user else it is difficult to guess the honeywords in general. We categorize this technique as ‘moderate’ DoS resistant. Therefore, we suggest that the system must take strong actions only if the count of accounts being attacked by DOS attacks is more than a threshold value.

The evolving-password model updates the database of frequencies with each new registration. Therefore the updated database differs from any existing openly available password database files, depending on the number of new registrations. The update of database prevents the adversary from guessing the honeywords with a high probability. Further, the similarities in the frequencies of honeywords with the password provide a nearly perfect flatness. We experimentally show that guessing the password from the honeywords is difficult and the model satisfies perfect flatness.

The append-secret model makes the task of guessing honeywords more challenging, as it implements a one-way function taking input from the user and a random secret from the system to generate a value that is appended to the password.

DoS attack with Fake Account: There could be two preventive approaches to handle the case where an attacker is registering a dummy account and trying to guess the honeyword. One is a preventive approach where the solution is to make the honeyword generation perfectly flat. Hence, guessing honeyword is difficult and DoS attack can be prevented. Other possible solution is to distinguish fake accounts from real ones.

This can be implemented through classification using machine learning, for instance, by tracing some common features for fake accounts. For example, fake accounts are inactive after creation, few features that are provided by the service could commonly be untouched for fake accounts etc. Once an account is classified as fake, there is a high probability that a login using honeyword for this account could lead to a DoS attack.

6.6.4 Multiple System Intersection considering Input/Output Attack

It is a common human tendency to use the same password for multiple sites [83]. This facilitates the adversary when attacking multiple distinct systems. The adversary can take an intersection of sugarwords from various systems to guess the password, which means it can mount MSIO attack. If the same password maps to distinct values without applying distinct random secrets to different websites, MSII attack is possible. Our proposed append-secret model modifies the password chosen by the user based on a random secret generated by the system. Therefore, even if the adversary gets access to the password files from distinct systems, the same password maps to different values and hence intersection does not reveal the password. Even if the adversary can get all the sugarword corresponding to a specific system, it is not possible for her to match the entries to another system without knowing the random secret of the two systems. Hence, both MSII and MSIO attacks are not possible against our proposed ‘Append-Secret model’.

6.7 Experimental Data

We set-up an experiment to check the flatness of the technique of honeyword generation and the effectiveness of implementing Levenshtein distance to prevent false detection. We simulate our ‘evolving password model’ using publicly available password files. We generate honeywords and compare the lists of honeywords generated for two passwords of the same frequency. We also check the ease with which the honeywords can be guessed. To generate honeywords, the frequency of all tokens is preserved, while only the length of special characters is preserved.

Following is a detailed description of the implemented algorithms. For our experiment, we use multiple databases containing real user passwords which were compromised by hackers and were made publicly available. Passwords are considered as private data and hence we keep the databases used in this study confidential. Respecting ethical and legal issues in the distribution of user private information, we can provide the list to the legitimate researcher who agrees to follow the moral and legal standards of

handling private information. The databases can be obtained by contacting the authors, after agreeing to standard ethical practices in handling the same. Our combined database contains more than 4 million unique user passwords. This database is used to pre-compute the frequencies of password patterns as well as frequencies of individual tokens. These computed frequencies are stored in separate files. Our experimental process is described below. The two-step procedures to generate the honeywords as explained in Section 6.5.3 are specified in Algorithm 6.1 and Algorithm 6.2.

Algorithm 6.1: To pre-compute the frequencies of password pattern and tokens

Input: A database containing n number of user passwords

1. for each n in database do
 2. **patterns_map** = {Store the frequency of the password pattern}
 3. Split the password into separate tokens and store in different files as
 4. **alphabets_map** = {Store the frequency of the string of alphabets}
 5. **digits_map** = {Store the frequency of the string of digits}
 6. **specialChars_map** = {Store the frequency of the special character}
 7. end for
-

Let a password in the input database is ‘password_123’. Applying Algorithm 6.1 on the password, we first decide its pattern (syntax). For instance, ‘ AS_1D ’ where A = string of alphabet, S_x = special character of length x and D = string of digits. We then store the frequency in a file named patterns_map. The algorithm then splits the password into tokens: {‘password’, ‘_’, ‘123’}. The files alphabets_map, digits_map and specialChars_map stores the frequency of the alphabet-string ‘password’, the frequency of digit-string ‘123’; and the frequency of the special character ‘_’.

Using the generated files by implementing the Algorithm 6.1, we generate the honeywords following Algorithm 6.2.

Algorithm 6.2: To generate honeywords

1. Find the frequency of the password pattern from **patterns_map** and update the respective frequency.
 2. **List**₁ = {store all password patterns that have same frequency}
 3. Split the password into individual tokens to compute frequency and also update the respective token frequencies.
 4. **List**₂ = {store all strings of alphabets that have same frequency from **alphabets_map**}
 5. **List**₃ = {store all strings of digits that have same frequency from **digits_map**}
 6. **List**₄ = {store all strings of special symbols that have same frequency from **specialChars_map**}
 7. List_honeywords={}
 8. for $i = 1$ to k (where k =number of honeywords)
 9. pattern_x=randomly choose a pattern from List₁
 10. for each token in pattern_x
 11. honeyword = replace the token with elements chosen randomly from
 12. List₂, List₃ and List₄ depending on the type of token
 13. end for
 14. if levenshtein_distance (password, honeyword) ≥ 3
 15. add the honeyword to List_honeywords
 16. else do not update the value of k
 17. end for
 18. **return** List_honeywords
-

To test Algorithm 6.2, we combine the list of leaked password files of two different large organizations. These files were leaked by hackers in 2006 and 2010, respectively. Just like the training database which we already mentioned, we maintain the confidentiality of this test database as well. This database can also be provided to academic researchers after due diligence that they will use standard ethical practice in handling the same. Anyone needing the files can contact the authors.

As we generate honeywords by randomly choosing entries from list₁, list₂, list₃ and list₄ (see Algorithm 6.2) which are created from an evolving dataset, we get different set of honeywords even for same frequency passwords, with a very high probability. To test the probability, we generate honeywords for different pairs of passwords of same frequency. For our chosen set of password pairs, we observe the following number of entries (the minimum entries obtained over password-pairs of having different frequencies from very high to low):

$$|\text{List}_1| = 10; \quad |\text{List}_2| = 20; \quad |\text{List}_3| = 10; \quad |\text{List}_4| = 10$$

Therefore, the number of possible honeywords for passwords of the above case is 20000 and the probability that the two sets of honeywords for a pair of similar frequency passwords collide is .001, which is very low. This assures that it is difficult to predict

honeywords for two passwords of the same frequency. With evolving frequency counts, it is also difficult to create similar list of honeywords for an adversary even if the adversary has the access of same leaked password database.

The Flatness: Honeywords are generated analysing the frequency of the real user passwords. It is very difficult to model human behavior to generate passwords. Therefore, techniques based on the distribution of real user password are the best way to map human tendency towards password generation. Algorithm 6.2 randomly generates honeywords maintaining a threshold distance (≥ 3) with the password. Hence, the technique is perfectly flat.

6.8 Comparison with Existing Work

There exist a few works [80, 57, 56] on the analysis of honeyword based detection techniques. The comparisons of existing honeyword generation techniques and our proposed methods, considering various advantages and drawbacks, are shown in Table 6.1 and Table 6.2. Table 6.1 compares the existing protocols defined under the ‘Modified-UI’ category. It is clear from the comparison and the explanation in Section 6.5.3 that our proposed technique, ‘Append-Secret model’ is user-friendly and provides protection against ‘multiple system intersection considering Input/Output’ attack. The advantage of ‘Modified-UI’ technique is that it introduces randomness in user-chosen password and prevents the MSII/MSIO attack. The disadvantage of Take-a-tail method under this category is that the randomness is generated by the system and user needs to memorize it. Another technique PDP is an approach where randomness is selected by the user as explained in Section 7.3. However, introducing the new MSII attack model we have shown that the technique is completely broken. Specifically, it fails to provide security against both MSII and MSIO attacks. Further, this method forces users to randomize their passwords and this is a memory overhead for users. Hence, our proposed technique ‘Append-secret model’ tries to balance the disadvantages of the above mentioned techniques under ‘Modified-UI’ approach and the comparison is tabulated in Table 6.1. Table 6.2 compares all the existing honeyword generation techniques considering flatness and significant attack models. The Denial-of-Service (DoS) attack resistance is defined as ‘weak’ or ‘strong’ where ‘weak’ signifies that given a password, an adversary can provide a honeyword with non-negligible probability and ‘strong’ signifies that honeywords are indistinguishable from passwords. ‘Multiple system intersection considering Output (MSIO) and ‘Multiple system intersection considering Input (MSII)’ attack resistance means that even if accounts of the same user on different systems are compromised, it will not reveal the password. The typo-safety is considered to be taken care of only if it is experimentally performed or explicitly mentioned while describing the technique. Flatness is defined as the probability to distinguish $(k - 1)$ -honeywords from the password. A technique

Table 6.1: **Comparison of Modified-UI honeyword generation techniques.**

Take-a-tail [103]	PDP [57]	Append-sectret [this work]
System selects the random value for user	User selects the random value	User and system selects the random value
For n different sites n-different values for each user	For n different sites may use same value for each user	For n different sites n-different values for each user
Prevents MSIO attack	MSIO attack Possible	Prevents MSIO attack
Prevents MSII attack	MSII attack Possible	Prevents MSII attack

is user-friendly if the choice of password is not influenced by the UI.

6.9 Summary

In this chapter, we propose new honeyword generation techniques which overcome several limitations of the existing honeyword generation techniques. Our method produces honeywords that are almost perfectly flat. To prevent false detection, in cases where legitimate user unintentionally enters a honeyword, we implement Levenshtein distance to maintain minimum required distance (3 for our experiment) between password and honeywords. We propose a new attack model and show that the ‘Paired Distance Protocol’ defined in [57] is completely broken in our attack model. The detailed analysis of existing honeyword techniques and their comparison with our proposed techniques is also provided. It would be an interesting problem to work on a new data structure to prevent false alarm as any distance measure (e.g. Levenshtein distance) may be typo resistant, but may not be confusion resistant always. Therefore, legitimate user can provide honeyword through confusion. It would be a good future extension of the present work.

To measure the typo-safety, one possible approach that is commonly followed is by collecting data conducting a real life survey. This approach requires designing relevant survey questions, carefully choosing the target audience and then analyzing the results of the sample surveys. We consider it as an extension of our current work and a good target for our future work.

Table 6.2: **Comparison of honeyword-generation methods.** ‘weak’ DoS Resistant means that given a password, an adversary can provide a honeyword with non-negligible probability; ‘moderate’ DoS Resistant means that given a password, an adversary can provide a honeyword with non-negligible probability on special scenarios else indistinguishable from passwords; ‘strong’ DoS Resistant means honeywords are indistinguishable from passwords. MSIO attack and MSII attack resistance mean that even if accounts of the same user on different systems are compromised, it will not reveal the password. The typo-safety is considered to be taken care of only if it is experimentally performed or explicitly mentioned while describing the technique. Flatness is defined as the probability to distinguish $(k - 1)$ -honeywords from the password.

Honeyword Method	Flatness	DoS Resistant	Typo Safety	Legacy UI	MSIO Resistance	MSII Resistance	user-friendly
Tweaking [103]	$< 1/k$	weak	no	yes	no	no	yes
Password-model [103]	no	strong	no	yes	no	no	yes
Tough nuts [103]	N/A	strong	no	yes	no	no	yes
Take-a-tail [103]	$1/k$	weak	no	no	yes	yes	no
Close-number formation [56]	$1/k$	weak	no	yes	no	no	yes
Caps-key based [56] approach	$1/k$	weak	no	no	yes	yes	no
Modified-tail [56] approach	$1/k$	weak	no	no	yes	yes	yes
PDP [57]	$1/k$	strong	no	no	no	no	no
Evolving password model [this work]	$1/k$	strong	yes	yes	no	no	yes
User-profile model [this work]	$\approx 1/k$	moderate	yes	yes	no	no	yes
Append-secret model [this work]	$1/k$	strong	yes	no	yes	yes	yes

Chapter 7

The Analysis of Universal 2nd Factor (U2F) Protocol

The augmentation of a second factor to strengthen the simple password-based authentication is in recent trend. The Universal 2nd Factor (U2F) protocol is an open authentication standard to strengthen the two-factor authentication process. It is proposed by Fast IDentity Online alliance as a secure protocol. U2F augments the existing password-based infrastructure by using a specialized USB, termed as *the U2F authenticator*, as the 2nd factor. In U2F, at the time of manufacturing, the U2F authenticator is assigned fixed keys namely the device secret key and the attestation private key. These secret keys are later used by the U2F authenticator during the *Registration phase* to encrypt and provide a digital signature to crucial information that will help in proper validation of the user and the web server. However, the use of fixed keys for the above processing leaks information through side channel about both the secrets.

There exists several second factor authentication techniques based on a variety of technologies such as One Time Passwords (OTPs) [128], PKI-based hardware tokens, such as smart cards and USB tokens (e.g., RSA SecurID [4]). The second factor protocol, U2F is a new technique proposed in 2014 by a large industry alliance with more than 260 industry consortium, Fast IDentity Online (FIDO) [2]. There is a lack of work on the security analysis of this new technology. Therefore, a thorough analysis is of utmost importance and through this work we target to provide the same.

In this chapter, we first show why the U2F protocol cannot be secure against side channel attacks. We then propose a countermeasure to fix this issue. Our proposed solution is based on the rekeying technique that derives session keys from the device secret key. Use of different session keys instead of the fixed device secret key helps in keeping the device secret key confidential. The signature function that uses the attestation private key also leaks information that can be captured through side channel analysis. Usually in public key infrastructure (PKI) [154], the verification of the digital signature is satisfied by the use of certificates. Intuitively, it may seem that the use of session key will prevent the side channel attack on attestation private key as well. However, it is not feasible to get a fresh certificate from the CA for each private (session) key derived from the initial key. Hence,

it is difficult to prevent side channel attacks using public key infrastructure. Therefore, we recommend a modification to the protocol for mitigating this attack. Incorporating our proposed countermeasure and recommended modification, we then present a new variant of the existing U2F protocol that provides strong security guarantees. There is a lack of clear and comprehensive literature that describes the U2F protocol with detailed security analysis. We present a detailed analysis of the U2F protocol including its security analysis. We also try to fill the gap in the description of the protocol by providing a clear explanation of all the cryptographic operations. We also explain in brief how the side channel attacks of U2F protocol and the corresponding proposed countermeasures are similarly applicable to Universal Authentication Framework (UAF) protocol.

The roadmap for this chapter is as follows. In Section 7.2, we present an overview of the U2F protocol. In Section 7.3, we detail the existing security analysis on U2F protocol. Possible side channel attack points that can be exploited in U2F protocol are explained in Section 7.4. Subsequently, the countermeasure on these attacks and the complete description of our modified U2F protocol incorporating the countermeasure are documented in Section 7.5. This is followed by a detailed explanation of the design rationale behind our proposed modifications and its security evaluation in Sections 7.6 and 7.7 respectively. In Section 7.8, we present a discussion on the few limitations of the U2F protocol. The Section 7.9 explains in brief how the side channel attacks of U2F and the corresponding countermeasures can be applied to UAF protocol as well. Finally, in Sections 7.10, we summarize and conclude our work. The original contribution of this work is from Section 7.2 to Section 7.9.

7.1 Introduction

To prevent the offline attack, ‘Password Hashing Competition’ (PHC) [16] encouraged a resource consuming design for password hashing algorithms. The idea is to slow down the process of parallel computations by making computations resource intensive. However, this approach is not sufficient to prevent online attacks, such as phishing, Man-In-The-Middle (MITM), etc. Therefore the requirement of augmenting a second factor to strengthen the simple password based authentication is in recent trend. Few significant second factor solutions such as Time-based One-Time Password (TOTP) [10] and Short Message System (SMS) [21] are an easy target for phishing attack or MITM attack which in turn affect user privacy severely and introduce major loss [122]. Frequent cases of phishing and monetary theft due to security flaws in password-based solution have been reported. The U2F protocol proposed by Fast IDentity Online (FIDO) alliance in 2014 has been introduced as a strong augmentation that can overcome all known attacks currently faced in practice. Another protocol proposed by the FIDO alliance is called Universal Authentication Framework (UAF). The UAF protocol is an authentication protocol which supports biometric authentication to provide a unified and extensible authentication mechanism that supplants passwords [5]. As UAF supports biometric authentication, a different category of authentication when compared to U2F which supports passwords, in this work we mainly

focus on the analysis of U2F protocol. The detailed description of UAF protocol is included while the detailed analysis is left out of the scope of our work. The U2F solution is based on public-key solution ¹. Considering the claim of U2F developers that U2F is a significant contribution towards strengthening simple password-based authentication, a thorough third-party analysis of U2F solution is required to verify their assertion.

Motivation: At the time of manufacturing phase, the U2F authenticator is assigned a unique device secret key and an attestation private key. Both these keys are fixed throughout the lifetime of the token. The device secret key is used to compute a value called *Keyhandle*. The attestation private key is used to perform a signature. Both these operations are performed during the *Registration Phase* of the U2F protocol. Multiple executions of the *Registration Phase* involving computations with these fixed keys leaks information through *side channel* about both the keys.

The side channel attacks (SCA), proposed by Kocher [112, 113], exploit weaknesses in the physical implementation of cryptosystem to recover the secret key information. These attacks treat ciphers as a grey box and capitalize on the side channel leaks such as timing information [112], power consumption [113], electromagnetic leaks [138] etc. to correlate them with the internal states of the processing device which are dependent on the secret key. Several flavors of side channel attacks have been proposed depending upon the type of leakage investigated. Among them, power attacks are the most popular and extensively studied class of attacks. Power attacks analyze the power consumption of a cryptographic operation to recover the secret information. The side channel attacks are the most powerful and easy to implement attacks against cryptographic implementations [112, 113].

In general, the U2F authenticator is a specialized USB and all crucial cryptographic operations are performed inside it. Physical access to the USB is not a difficult scenario and hence side channel attacks are easy to mount. Any cryptosystem is an easy target for these kinds of attacks if suitable countermeasures are not adopted. The U2F specifications [6, 7] do not consider security against side channel attacks. However, the security of the U2F solution depends on the keys which can easily be disclosed through a side channel attack. Therefore, it is of utmost importance to protect U2F from this category of attacks.

The ‘Differential Power Analysis (DPA)’ attack [113] is a more advanced form of power attack which allows an attacker to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations [112]. When a cryptographic operation is performed over a fixed key, computation with different input-output pairs can easily reveal the key by applying DPA attack. Template attack [62] is another powerful side channel attack which can reveal the secret with very few

¹Public-key cryptosystem uses two mathematically related, but not identical, keys - a public key which is known to all and a private key which is known only to the owner. It accomplishes two functions: encryption and digital signature. For encryption, a sender encrypts with the public key of the receiver so that only the receiver can decrypt with its own private key. For digital signature, the sender signs a known message with its private key so that the receiver can verify with the public key of the sender. For more details one can refer to [154]

input-output pairs. It requires significant pre-processing to be implemented but can break implementations secure against DPA attack as well [62]. As template attack is difficult to implement in practice [25], we analyze and attempt to harden the U2F protocol to achieve DPA resistance.

Given the easy applicability of side channel attacks on a device which can be in the hands of an attacker, we consider the leakage of secret keys under DPA. We show that the use of fixed keys in U2F protocol can leak information about both the secrets by applying the DPA attack. We explore the possible cryptographic solutions to prevent repeated use of the fixed keys for U2F protocol. We show that the rekeying technique, which generates different session keys, is a suitable countermeasure to this attack.

“Rekeying” is a technique where different subkeys are derived from a master key to limit the number of operations performed under the same master key [24]. It is considered an effective approach to prevent leakage of a secret key information by limiting the number of operations performed with each generated keys. In our proposed solution, we utilize rekeying technique to protect the computations that are performed under fixed keys in the U2F protocol.

7.2 Overview of the U2F Protocol

In a password-based authentication system, a server allows n users identified with their usernames such as $u_1, u_2, \dots, u_n$, and their corresponding passwords $p_1, p_2, \dots, p_n$. For instance, u_i and p_i denote the username and password of the i th user respectively. The server maintains a database listing pairs of username and corresponding hashed password as follows.

$$(u_i, H(p_i \parallel s_i))$$

where H is the password hashing algorithm imposed by the server and s_i is the server generated salt corresponding to each user u_i which is a fixed length random value. U2F is a second-factor authentication protocol that augments this simple password-based authentication system and is explained ahead. The key notations used in this work are listed in Table 7.1.

The U2F is a protocol proposed by Fast IDentity Online (FIDO) alliance in 2014 as U2F v1.0 [6] and later in 2016 as U2F v1.1 [7]. The protocol description is similar in both the versions with the only differences being in the optimization of their implementation. Following are the listed differences between these two versions.

- The U2F protocol allows multiple tokens to be registered for the same AppId for a specific user. Therefore, for each token, different Keyhandles are generated for a specific AppId. In U2F v1.0, a separate AppId and challenge pairs are sent for every keyHandle, whereas U2F v1.1 suggests an optimization over it. The U2F v1.1 allows a single AppId and challenge pair for multiple keyhandles registered for the AppId.
- The U2F v1.1 JavaScript API specification supersedes JavaScript API of U2F v1.0. The major difference between these two versions is the way requests to be signed are formatted between the relying party and the client.

Table 7.1: Notations

Relying Party/Origin	Web server or Server
FIDO Client	Web browser of the client
FIDO Authenticator	U2F token (hardware/software)
AppId	URL of the web server supporting U2F
DS_K	Unique secret key assigned to the U2F token
K_i	i^{th} session key generated from DS_K and the counter i
PK_u	Public key of the AppId corresponding to the username u
SK_u	Private key of the AppId corresponding to the username u
PK_M	Attestation public key of the manufacturer
SK_M	Attestation private key of the manufacturer
PK_T	Public key of the U2F token
SK_T	Private key of the U2F token
$Sign_x(m)$	Signing operation on input message m , using secret key x
ACert	Attestation certificate issued by a trusted CA to the manufacturer
$\parallel$	concatenation operator
$\mathcal{K}(\cdot)$	a randomized asymmetric keypair generation function
s	salt, a fixed length public random value
$f^{(i)}(K_0)$	i times recursive call of function f starting from initial value K_0
$x \xleftarrow{\$} \{0, 1\}^n$	randomly generated n -bit value x
$DB \xleftarrow{add} x$	adding the value x to the database DB
R	An r -bit challenge randomly generated by the server
Kh	A key container called Keyhandle generated by the U2F token at the time of ‘Registration’ which contains AppId and private key SK_u
$H(x)$	Computing hash on the value x where H is any cryptographic hash function
CA	Trusted Certificate Authority under PKI infrastructure
S_db	A database maintained by the server to keep records
U2F_db	A database maintained by the U2F token to keep records

The U2F protocol allows online services to enhance the security of existing password infrastructure by adding a strong second factor, called ‘Universal 2nd Factor’ (hence the name U2F), at the time of user login. The augmentation is claimed to be a secure and user friendly solution. To use U2F, the user needs to login with a username and password and then perform a simple button-press on an associated device (which usually happens to be a USB in the case of U2F). A single U2F authenticator can be used across all online services that support the protocol with built-in support in the web browser of the client.

The protocol allows the U2F authenticator to be either a secure hardware or a software. In the following description we use a common term ‘U2F token’ to express both the hardware

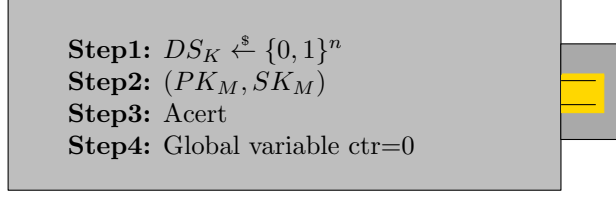


Figure 7.1: **U2F Manufacturing Phase.** The key DS_K is a randomly generated n -bit secret. The keypair (PK_M, SK_M) is the asymmetric keypair of the manufacturer shared with the token. Acert is the certificate signed by a trusted CA which includes PK_M . At the time of registration phase, a signature with SK_M is verified with PK_M from Acert by the server to prove the genuineness of the U2F token. The global variable ctr is initialized to zero and incremented after each successful authentication as explained in Section 7.2.3

and the software. The U2F protocol supports two generic steps for online authentication, namely, ‘Registration’ and ‘Authentication’. Both of these processes involve three parties: U2F token (hardware/software) which is also referred to as the ‘FIDO Authenticator’, the web browser of the client which is called ‘FIDO client’ and the web server which is referred to as the ‘Relying Party’. The U2F token is a crucial part for the analysis of the protocol. Therefore, we explain the protocol in three steps, appending the process of assigning the secret keys to the U2F token by the manufacturer at the time of manufacturing as the first step. Following is the description of these three steps of the protocol.

7.2.1 U2F Manufacturing Phase

This phase is executed at the time of manufacturing of the U2F token as shown in Fig 7.1. In this phase, the U2F token is provided with a randomly generated unique secret called Device Secret Key DS_K at Step1. At Step2, a public-private key pair (PK_M, SK_M) is provided to the token. The (PK_M, SK_M) keypair is not a one-time generation. Once randomly generated from a key generation function, the manufacturer provides the same (PK_M, SK_M) pair to all the tokens of a specific model manufactured by it. Note that the manufacturer can keep the same key pair for all the tokens manufactured by it (that is, it can treat all the tokens manufactured by it as the same model). At Step3, a certificate Acert issued by a trusted CA which includes PK_M (the public key of the manufacturer) is provided to the token. During the registration phase, the signature with SK_M is verified with PK_M which is extracted from Acert by the server. This verification proves the genuineness of the U2F token. Therefore, both DS_K and (PK_M, SK_M) are fixed for a token while the same (PK_M, SK_M) values are provided to multiple tokens which preserve the anonymity of the token. An integer counter ‘ctr’ is provided at Step4. This ‘ctr’ is initialized to value zero and gets incremented after each successful execution of the authentication phase as explained in Section 7.2.3.

7.2.2 U2F Registration Phase

The registration procedure is explained in Fig 7.2. Initially the username u and password p are communicated to the server at Step1. At Step2 the server verifies the values u and $H(p \parallel s)$ which is the hash of the password where s is the salt. On successful verification, the server generates a r bit random challenge R to differentiate each request, else it rejects the registration request. The server sends its identity as AppId and the value R to the browser at Step3. At Step4 the values are forwarded to the U2F token along with the (TLS) ‘channelId’ of the browser for the server. The TLS channelId is a TLS extension, originally proposed in [75] as Origin-Bound Certificate and its refined version is available as an IETF Internet-Draft [71]. Receiving the information, at Step5, the U2F token first generates a website specific public-private key pair represented as (PK_u, SK_u) by applying a random key generation function $\mathcal{K}(\cdot)$. The function $\mathcal{K}(\cdot)$ can be an openssl key pair generation library which needs as input an elliptic curve such as the NIST standard P-256 curve. We denote a function $f(\cdot)$ which depends on a secret key K as $f_K(\cdot)$. The U2F token computes the function with key DS_K and outputs $f_{DS_K}(\text{AppId} \parallel SK_u)$ which is called the Keyhandle Kh . The above function can be instantiated with any block cipher such as AES in CBC mode or HMAC as shown in Fig. 7.3 and 7.4 respectively. The values (PK_u, SK_u) and Kh can optionally be stored in a database denoted as U2F_db inside the U2F token. It then computes a signature $\mathcal{S}$ as $\text{Sign}_{SK_M}(\text{AppId}, R, \text{ChannelId}, PK_u, Kh)$. At Step6, the token sends the values PK_u , Kh and $\mathcal{S}$ along with ACert to the browser. The browser forwards the received values to the server at Step7. After receiving the values, the server first verifies the signature $\mathcal{S}$ with PK_M from the certificate ACert. This verification proves the genuineness of the U2F token. On verification, the server updates its database S_db with the values (PK_u, Kh, Acert) corresponding to the username u and it shows the successful registration of the U2F token else the server rejects the registration request. The U2F protocol allows a user to register multiple tokens with the same account.

7.2.3 U2F Authentication Phase

The authentication procedure is explained in Fig. 7.5. Once second factor authentication with U2F is registered, for example, when the values $(u, H(p \parallel s), PK_u, Kh, \text{Acert})$ are registered with the server for a particular user u , the subsequent login to the website needs to verify the registered values by communicating with the U2F token through browser. The steps are as follows. After receiving username u and password p at Step1, the server checks its database to retrieve the Keyhandle Kh for u at Step2. On successful verification, the server generates a r bit random challenge R and sends it along with Kh and its AppId to the browser at Step3. The browser forwards the received information with ChannelId to the U2F token at Step4. At Step5, the U2F token first verifies the received values by performing inverse computation $f_{DS_K}^{-1}(Kh) = (\text{AppId}' \parallel SK'_u)$. It then compares the received AppId' with the stored AppId. Similar checks can be done if $f_K(\cdot)$ is a keyed hash as shown in Fig. 7.4. If a successful match happens, the token increments a counter represented as ‘ctr’. This ‘ctr’ can be a global or a local variable. If it is local variable then each AppId gets its

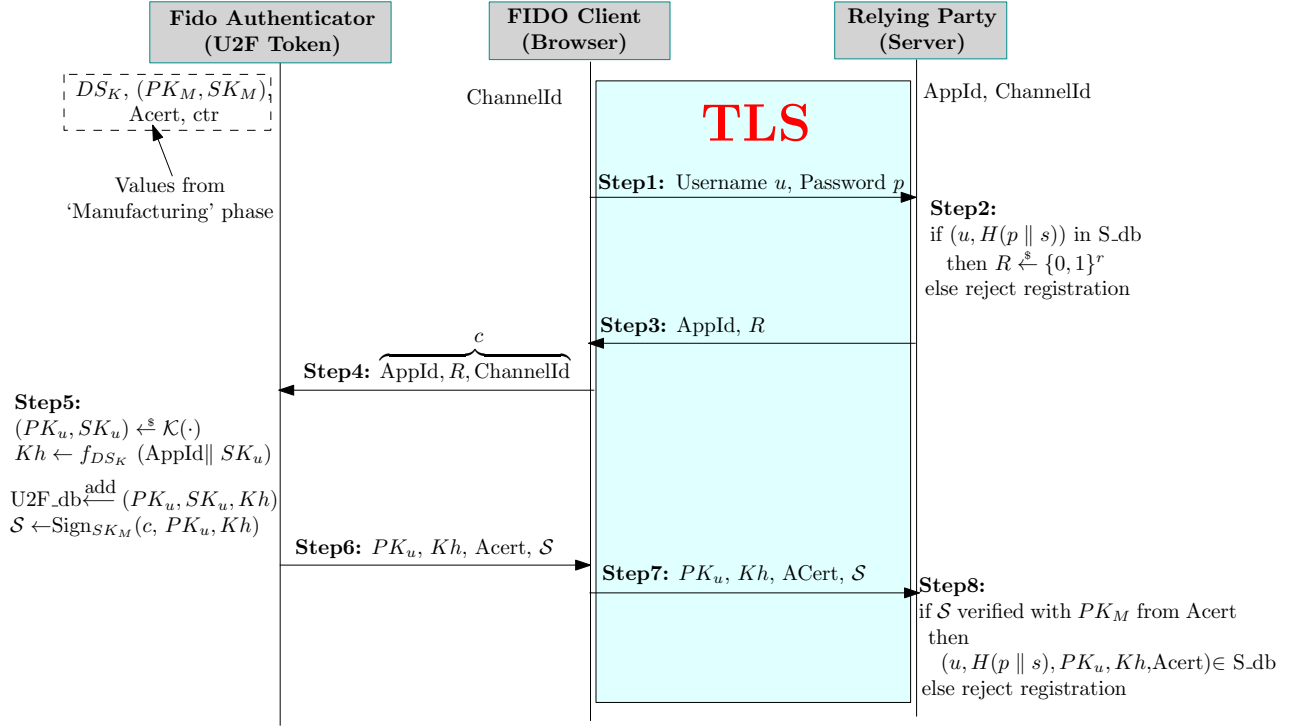


Figure 7.2: **U2F Registration Phase.** The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate Acert which includes PK_M and the counter 'ctr' are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId . S.db is the server database. R is a r bit random number generated by server. $\mathcal{K}(\cdot)$ is a random key generation function which generates the keypair (PK_u, SK_u) . $f_{DS_K}(\cdot)$ with key DS_K is the function to generate Keyhandle Kh as shown in Fig. 7.3 and 7.4, U2F_db is the database at U2F token.

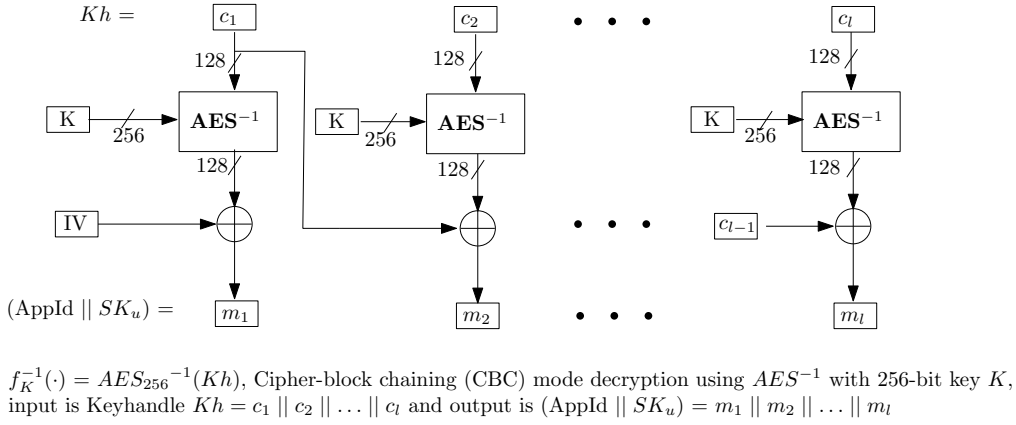
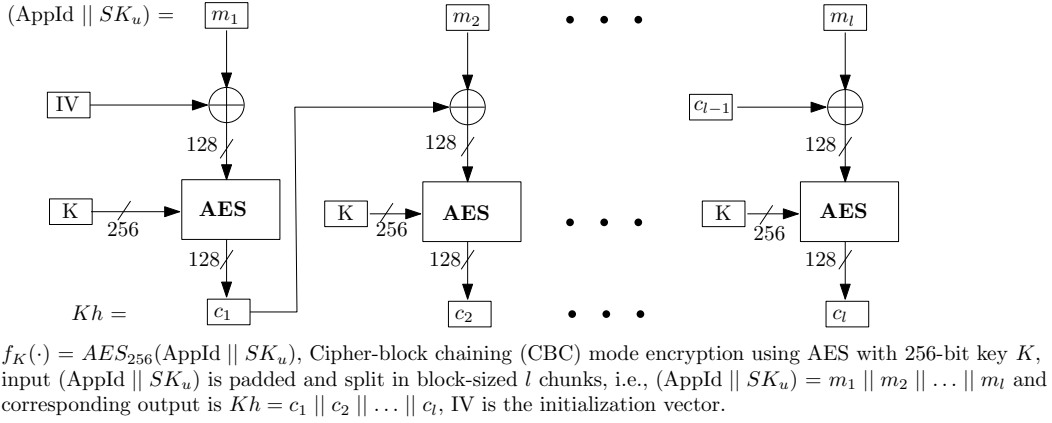


Figure 7.3: Instantiation of $f_K(\cdot)$ and $f_K^{-1}(\cdot)$ using AES.

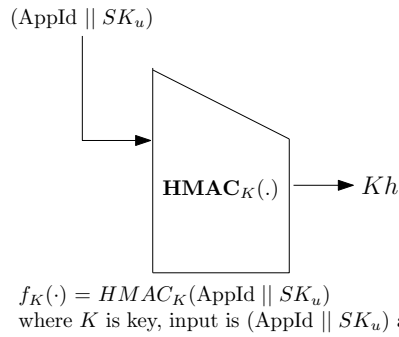


Figure 7.4: Instantiation of $f_K(\cdot)$ using HMAC.

own ‘ctr’ else a single ‘ctr’ is used across all registered AppId. Throughout the explanation, we consider ‘ctr’ as a global integer variable. The value of the ‘ctr’ is incremented after each successful authentication by the U2F token. This value ‘ctr’ is introduced to detect cloning of the U2F token.

A common danger with lightweight user-held devices like tokens is the duplication or the ‘cloning’ of the device. A cloned U2F device can be created by copying all the information, including the current ‘ctr’, from the original device. The U2F protocol thwarts the ‘cloning attack’ as follows. At Step5, the U2F token signs the values received from step4 and ‘ctr’ with the signing key SK_u , in addition to the generation of the Keyhandle. The signature generated at this step is represented as $\mathcal{S}$. The U2F token sends $\mathcal{S}$ and ctr to the browser at Step6. The browser in turn forwards these values to the server at Step7. Finally, the server verifies the received signature $\mathcal{S}$ with the stored key PK_u for the username u from S_db . To detect cloning, the server checks if the received ‘ctr’ is greater than previously stored ‘ctr’. If this is not the case then an inconsistency in ‘ctr’ is detected. This could be an attempt to clone a legitimate token and this detection is possible when the legitimate token is successfully authenticated at least once by the server after being cloned. Only on successful verification does the server store the updated ‘ctr’ value in the database.

7.3 Existing Security Analysis of U2F Protocol

This section presents the current security analysis of U2F protocol.

7.3.1 Denial of Service (DoS) Attack

Denial-of-service attacks are characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service [78]. We explain some ways in which such an attack can be launched against the U2F protocol.

1. Following the explanation of the protocol in Section 7.2, a user specific public-key and Keyhandle (PK_u and Kh) are generated by the U2F token at the time of registration. The secret key is kept inside the token and the public key is stored on the server, to identify the Keyhandle later. To bind a Keyhandle with a token, a database of the public-key PK_u and the Keyhandle Kh is maintained at the server.

If an attacker can modify the stored public key PK_u at the server then it will cause a DoS attack for a legitimate user at the time of authentication [120] since access will be denied to the legitimate user after verification in Step8 of Fig. 7.11. In order to eliminate this attack, secure storage of PK_u is required. Note that use of certificates (as in traditional PKI setting) is not feasible here since the asymmetric keypairs are generated by the token and getting certificate from a trusted CA for each new registration is not practical for the token.

2. The U2F specification [6] states that “The relying party’s dependence on passwords is reduced. The password can even be simplified to a 4 digit PIN. End users carry a single

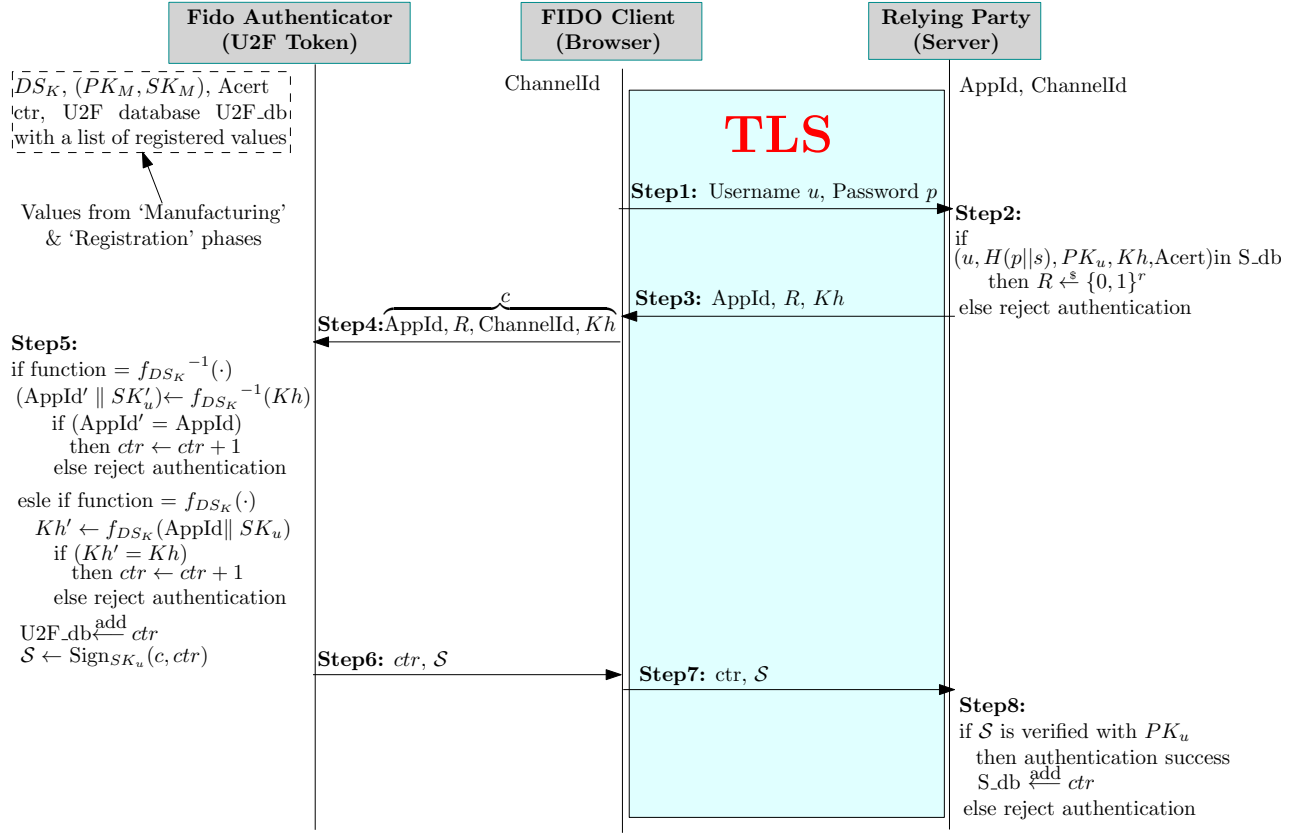


Figure 7.5: **U2F Authentication Phase.** The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate Acert which includes PK_M and counter 'ctr' are provided at the manufacturing phase. The U2F_db contains all the registered values from registration phase. The server database S_db contains all the required values generated at the registration phase corresponding to a registered username. R is a r bit random number generated by the server. AppId is the URL of the server. ChannelId is the TLS ChannelId. The value Kh is the Keyhandle. The verification of received Kh for both the cases of Step5 is shown in Fig. 7.3 and 7.4. 'ctr' is the counter used to record the number of authentication requests successfully satisfied by the U2F token.

U2F device which works with any relying party supporting the protocol”. Therefore U2F claims to provide enhanced security even with a weak password. At client side, an attacker with access to a different U2F token may register U2F authentication for an account of a third party without his notice. This is possible with the assumption that the password of a user account is known to the attacker (or the password is easy to guess). This way the legitimate user would be prevented to access his own account for not having the registered token with him. This attack is difficult to prevent as it needs the passwords to be difficult to guess and known only to its user.

3. Another possibility of DoS attack is when the attacker gets access to the U2F token of a legitimate user. Since any hardware token has a fixed and limited memory, an attacker may attempt to exhaust all of it. For example, an attacker can register multiple websites of its own choice, leaving no storage for any other request. As there is no way to know the list of websites registered under a U2F token by its user, it is difficult for a legitimate user to notice this attack. The user can only receive a denial for new registration requests. It shows that keeping the U2F token safe is mandatory to resist this DoS attack.

7.3.2 Parallel Signing Request attack

Consider a user interacting with two U2F enabled web pages [122] and assume that an attacker can detect the web page with which the user is interacting at a given instance. In this case, the attacker can synchronize the two U2F operations initiated for two different websites by the user. As per the specification of U2F v1.0, there is no way to notify the user about which operation is performed by the U2F token. The user merely performs a button-touch for both ‘Registration’ and ‘Authentication’ processes since only the user-consent is mandatory for the execution of both these operations.

Suppose a website is requesting for U2F authentication and is visible in the foreground of the user’s display, and at the same time a transaction page (initiated by the attacker) is in the background (opened as an iframe). The user provides his consent for the displayed website while the first request that reaches to the U2F could be for confirming the transaction. This scenario shows a successful execution of a transaction without the knowledge of the user. Therefore the second-factor authentication with U2F is not safe for monetary transactions and such transactions could be at high risk.

With growing popularity of Internet banking, cases of online frauds have also increased manifold resulting in considerable financial losses [163]. The banking frauds have increased 93% in 2009-2010 [22], and 30% in 2012-2013 [23] and these are ever growing. Internet banking or online transaction frauds are difficult to analyze and detect because the attack techniques are rapidly evolving while their detection is usually delayed [163]. Therefore, transaction-nonrepudiation² is an essential property for any secure authentication scheme.

²Nonrepudiation is the assurance that a participating entity in a protocol cannot deny an action done by it later [37].

FIDO alliance has already mentioned in the protocol specification that U2F does not claim transaction non-repudiation. Therefore, the scenario described above is technically not an attack on U2F. However, existing second-factor authentications, such as SMS or OTP do provide this assurance. Even the version U2F v1.1 recommends to have a notification for each operation but it is not mandatory. Therefore, with the current specifications, U2F is not a secure solution for password-based authentication. Augmenting OTP and U2F together can protect the transaction, however, it adds overhead and contradicts one of the goals of U2F which is to be user-friendly. As U2F lacks transaction non-repudiation, it is not a secure solution to protect password-based authentication.

7.3.3 Phishing attack

The concept of the phishing attack was first introduced in 1987 at Interex conference by Jerry Felix and Chris Hauck [82, 139]. NIST in [115] defines phishing as ‘a technique which refers to use of deceptive computer-based means to trick individuals into disclosing sensitive personal information’. Commonly, it is a form of social engineering that uses email or malicious websites to solicit personal information from an individual or company by posing as a trustworthy organization or entity.

U2F claims to be phishing resistant solution and two different scenarios prove the claim by showing how phishing can easily be detected at the time of authentication phase of the U2F protocol [122]. In one scenario, when the U2F is registered with a genuine website (AppId) and at authentication phase if a phishing website (AppId') forwards the Keyhandle and challenge from that genuine website (corresponding AppId), the U2F can easily detect the phishing attempt with origin mismatch. Following Fig. 7.5, the scenario can be described as the U2F receives AppId' at Step4 and the Keyhandle provides AppId at Step5 and hence both AppIds differ in value. Hence phishing is detected at the U2F token. In the second scenario, if an attacker forwards challenge (R) from a genuine website and Keyhandle (Kh') from a phishing website, U2F token signs the challenge with the corresponding private key of the received Keyhandle. If the signature is forwarded to the genuine website, it will reject it as the signature would not be verified (Step8 of Fig. 7.5). Therefore, because of the signature and origin-specific Keyhandle, existing approaches for phishing attack fail against the U2F protocol.

7.3.4 Man-In-The-Middle (MITM) attack

In the MITM attack, the common scenario involves two legitimate endpoints (victims), and a third party (attacker) [65]. The attacker has access to the communication channel between two endpoints, and can eavesdrop, insert, modify or simply replay their messages. According to [65], the term ‘MITM attack’ was first mentioned by Bellovin et al. in [38].

The U2F specifications [6, 7] claim that this technique prevents MITM attack where the MITM attacker intermediates between the user and the webserver. As per the U2F specifications, the MITM attack is possible when the following two conditions are satisfied.

1. MITM attacker is able to get a server certificate for the origin name issued by a trusted CA, and
2. Channel IDs are not supported by the browser.

However, the MITM attack can be carried out even when these conditions are not satisfied, contradicting the developers claim. In [105], a MITM attack is shown on TLS with Channel ID support which is based on the following conditions.

1. MITM attacker is able to get a server certificate for the origin name issued by a trusted CA, and
2. Channel ID is supported by the browser.

The attack is similar to the attack in [165] which is also applicable to the U2F based authentication and called ‘Man-In-The-Middle-Script-In-The-Browser (MITM-SITB)’ [105]. The MITM-SITB attack works as follows. It assumes that TLS Channel ID is supported by the browser and the communicated webserver possesses valid/invalid certificate as it works with invalid certificates as well. This is due to the fact that the web browser only sends warning of invalid certificate to the user and proceeds according to the user action. In many cases, users ignore such warnings. Hence, even invalid certificate can allow the attacker to launch the attack. The details of the attack are provided next.

We assume that an attacker knows the website request of the user in advance and hence compromises the DNS server to divert the route of that legitimate website to a malicious website which contains same origin address (URL) but which is hosted on a different server (with different IP address). Next, when the user initiates authentication to the compromised website using U2F, it establishes a TLS connection with the malicious website. The malicious website pushes a malicious JavaScript code to the browser, without the notice of the user browser, and terminates the connection. The browser re-establishes a fresh TLS connection but with the legitimate website, as the attacker places the legitimate IP at DNS server, for subsequent communication with the website. The attacker becomes passive after injecting the code and the user authentication is performed over the connection between the browser and the legitimate website. Browsers commonly support ‘Single Origin Policy (SOP)’, i.e., the webpages can interact with each other only if they belong to the same origin (URL). The browser thus prevents interaction between multiple websites with different origins (URL’s). Since the injected malicious code contains the origin of the legitimate website, the browser allows the code to access the webpages of the legitimate website. It is difficult to detect this attack as the TLS channel ID will be the same for both the browser and the server. The U2F only verifies the Channel ID of the browser which is origin-specific and hence cannot detect the attack.

Real world attacks are multi-faceted. All the above attacks are based on standard attacks employed on browsers and internet traffic, and none of these attacks require tampering of the U2F token or exploiting the U2F protocol. However, there are other classes of attacks which can exploit the U2F token/protocol. One such class of attacks is the side channel attack. In

the next section, we describe an attack on the U2F protocol which is based on the powerful side channel analysis. When a cryptographic device can get in the hands of the adversary or otherwise become easily accessible then one needs to pay close attention to side-channel attacks. The U2F solution is based on a hardware token (in common practice) and hence can easily be accessible to the attacker. As side channel attacks are easy to implement and the most successful cryptographic attacks to compromise secrets, it is of utmost importance to protect U2F from side channel attacks.

7.4 Possible Side Channel Attacks on U2F Protocol

The side channel attack is one of the most powerful attacks since the seminal work of 1996 by Paul Kocher [112]. In this class of attack, information is gained from the physical implementation of a cryptosystem [89]. There exist different forms of leakage under side channel. For example, timing information [112], power consumption [113], electromagnetic leaks [138] etc., provide information about the secrets that can be exploited. It is considered essential that the implementations of cryptographic algorithms are side channel resistant, especially when the device could be in the hands of the attacker. For some specific cryptographic operations, algorithmic countermeasures are available.

Kocher’s first major contribution included exploiting the correlation of cryptographic secret keys with the time to perform certain cryptographic operations [112]. In a subsequent work, he showed how the secrets from a device can be extracted by utilizing the ‘Differential Power Analysis (DPA)’ attack [113]. DPA is a more advanced form of power analysis which allows an attacker to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations [112]. When a cryptographic operation is performed over a fixed key, computation with different input-output pairs can easily reveal the key. In the following section, our proposed attack is based on the observation that different input-output pairs in U2F protocols are computed over a fixed key. Therefore, we claim that the DPA attack is possible on the implementation of U2F protocol. The ‘Template attack’ [62] is another powerful side channel attack which can break implementations secure against DPA attack but it is difficult to implement in practice [25]. The template attack requires significant pre-processing to be implemented. Specifically, it requires a great number of traces to be preprocessed before the attack can begin. Given the current state-of-the-art in side channel attacks, it is necessary to have a cryptographic implementation to be DPA resistant at the very least.

Next we explain how a DPA based side channel attack can be launched on the U2F protocol.

7.4.1 Side Channel Key Recovery attack on Device Secret Key DS_K

According to the specifications [6, 7], all U2F tokens contain a unique device secret key DS_K which is used to generate Keyhandle Kh as shown in Fig. 7.3 and 7.4. Specifically, at the

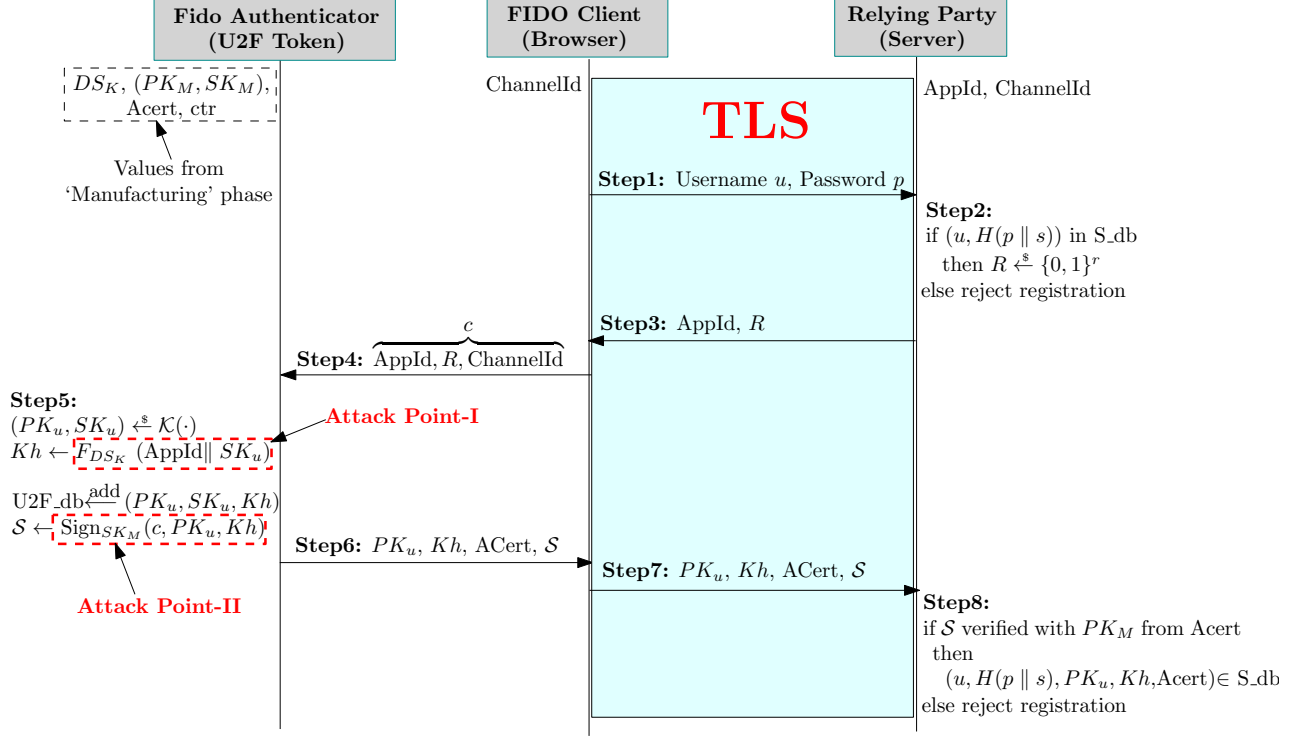


Figure 7.6: **Side channel attack points at Registration Phase.** The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate $Acert$ which includes PK_M and the counter 'ctr' are from the manufacturing phase. $AppId$ is the URL of the server. $ChannelId$ is the TLS $ChannelId$. S_db is the server database. R is a r bit random number generated by server. $\mathcal{K}(\cdot)$ is a random key generation function which generates the keypair (PK_u, SK_u) . The function $f_{DS_K}(\cdot)$ with key DS_K generates Keyhandle Kh as shown in Fig. 7.3 and 7.4. $U2F_db$ is the database at U2F token. The Attack Point-I is to compromise the key DS_K and the Attack Point-II is to compromise the key SK_M .

time of registration, the U2F token computes $Kh \leftarrow f_{DS_K}(\text{AppId} \parallel SK_u)$ where AppId is the URL of the server and SK_u is the private key for the username u corresponding to the AppId. With the access to the U2F token, an attacker can collect multiple power traces for different input-output pairs operated under the function $f_{DS_K}(\cdot)$ with fixed but unknown key DS_K . This is possible when the attacker gets the U2F token and requests registration for different AppIds for U2F authentication. We assume that the communication between the browser and U2F token is easy to monitor by the attacker. This assumption is based on the fact that the information received or sent by the browser can be easily captured through the Document Object Model (DOM) tree created by the browser as explained in [91]. This allows the attacker to collect many different input-output patterns (the AppId and the corresponding Keyhandle) and thus enables him to launch a successful DPA attack to extract DS_K from $f_{DS_K}(\cdot)$. The computation at which the DPA can be launched at the time of registration of the token is shown in Fig. 7.6 as ‘Attack point-I’.

The DPA attack has proven successful in efficiently attacking smart cards and many dedicated embedded systems which perform computations on the basis of a stored secret key [168]. In the classical form of DPA [81], an adversary collects multiple power traces and ciphertexts corresponding to a computation under a fixed key. It then chooses an intermediate value of the computation which depends both on the input and a small part of the secret key. The adversary then predicts the key bits and the corresponding power consumption. If the guessed key bits are correct then the predicted power consumption should match the observed one. On the other hand, if the key guess is wrong then the power consumption trace should be different from the observed one with a very high probability. As explained above, such an attack can be performed against the U2F token.

To prevent this attack, repeated use of fixed DS_K for different input-output (AppId-Keyhandle) pairs should be prevented. Use of session key is a standard solution [24] and our proposed countermeasure is based on the same idea.

7.4.2 Side Channel Key Recovery Attack On Attestation Private Key SK_M

As explained in Section 7.2.1, the attestation private key SK_M along with the certificate Acert, which contains PK_M signed by some trusted CA, are provided by the manufacturer to the U2F token at the time of manufacturing. This key SK_M is used as a signature key at the time of registration to prove genuineness of the U2F token to the server. The same attestation key is provided by the manufacturer to a large number of tokens generated during some specific time period to preserve anonymity of the tokens. The signature $\mathcal{S} = \text{Sign}_{SK_M}(\text{AppId}, R, \text{ChannelId}, PK_u, Kh)$ with signing key SK_M which is provided at the time of registration also leaks information about the key SK_M through side channels. The computation which may leak information about SK_M is represented as ‘Attack Point-II’ in Fig. 7.6. Therefore, an attacker with access to the U2F token can collect multiple traces for different inputs (containing different values of AppId, R , ChannelId, PK_u, Kh) and outputs (signature) pairs signed with a fixed but unknown key SK_M . It is easy to launch

a DPA attack in a manner similar to the one explained in Section 7.4.1. For a particular token, all the signatures generated at the time of registration are produced using this fixed key SK_M . Moreover, as already mentioned, a large number of devices are using this same key SK_M . Therefore compromising a single SK_M on a U2F token is enough to compromise a large number of tokens which contain the same key SK_M . After extracting SK_M , an attacker can simulate the U2F token in a software program and use it to prove the possession of the token. It is almost impossible to detect which device has been compromised and therefore the only remedy is to invalidate all tokens with the compromised SK_M . Use of different public-private key pair for attestation can prevent this mass compromise.

There are two common approaches to develop a side channel attack resistant system. In the first approach, masking and other similar techniques are implemented at the hardware level to prevent the leakage of data which can be exploited by an attacker. In the second direction of work, algorithmic modifications are made such that the data which is available to the attacker becomes essentially random. In the following sections, we present an algorithmic solution to prevent/mitigate DPA based side channel attacks on U2F protocol.

7.5 Proposed Countermeasures

We describe a countermeasure to prevent the recovery of the device secret key from side channel attacks. We use a counter-based re-keying process, a concept which was first analysed in [24] and then an efficient tree-based approach to implement it was described in [111]. We also suggest a minor modification in the approach of assigning attestation private key to the U2F tokens to mitigate the attack mentioned in Section 7.4 which allows recovery of the key SK_M .

7.5.1 Countermeasure to protect Device Secret Key

Our proposal to protect the device secret key DS_K from SCA is to generate session keys sequentially from DS_K , a concept known as “re-keying” in the literature [24]. Our counter based re-keying technique is explained in Algorithm 7.1. The function g of the proposed algorithm takes the previous session key and current counter as inputs and generates the current session key. The use of counter for re-keying is a natural way to avoid repetition of keys. However, the counter-based re-keying technique has various drawbacks depending on the setting and application. In the case of client-server model, which is the most common setting, this method suffers from synchronization issues when the session key is generated by the client and the corresponding counter is communicated to the server. The server can compute the same secret only when it knows the initial secret and the current counter. Therefore, a necessary assumption in this technique is that the initial secret is known to both the client and the server.

Considering the U2F protocol where sharing of keys with the server is not required, a significant problem is the computation overhead inside the U2F token. The overhead

increases with the value of the counter since the computation starts from the initial counter value and sequentially proceeds till the current counter value. We address how to reduce the overhead of this sequential computation next.

Generally, there are two approaches to derive session keys. In the parallel approach, the i th session key K_i is generated as $g(K, i)$ when K is the initial secret and g is some function. In the sequential approach, the i th session key is generated by applying a function on the $(i - 1)$ th session key [24]. The sequential approach is better to avoid DPA as it changes the key at each execution depending on the previously generated key. In the parallel approach, every execution depends on the initial secret and potentially leak information about the initial secret via side channels. The sequential approach also provides forward secrecy, for instance, the disclosure of the current key does not reveal the previously generated session keys. However, this approach is inefficient. The value of the counter reveals the number of computations required to derive the session key starting from the initial secret. If there is a significant difference in the current key counter and the initial counter, and if the previous values are not stored then a huge number of computations are needed to reach the desired value. For example, to compute the i^{th} session key, it requires a single computation if the $(i - 1)^{th}$ session key is stored. However, if the j^{th} session key is required to be re-generated and the last stored subkey is the i^{th} one, with $i \gg j$, then we need j computations starting from the initial secret to re-derive the j^{th} session key. This is a clear overhead if $i \gg j$.

We discuss how to reduce this computational overhead in our proposed techniques and also explain the usability of each proposal below.

Algorithm 7.1: To generate session key from device secret key

Input: AppID, *Counter*, Device secret key DS_k , Previous session key, U2F database U2F_db

Output: A tuple (Keyhandle, *Counter*)

1. Global var *Counter* initialized at 1
2. $K_0 \leftarrow DS_K$
3. Set $i = Counter$
4. $(PK_u, SK_u) \xleftarrow{\$} \mathcal{K}(\cdot)$
5. $K_i \leftarrow g(K_{i-1}, i)$
6. $Kh_i \leftarrow f_{K_i}(\text{AppId} \parallel SK_u)$
7. $\text{U2F_db} \xleftarrow{add} (Kh_i, i)$
8. $Counter = Counter + 1$
9. return (Kh_i, i)

We provide some practical approaches to instantiate the function g of Algorithm 7.1. We represent the function g as $g(\text{key}, \text{counter})$ where key is an n -bit secret value and *Counter* is an integer.

Proposed Technique 1: In this case $g(\text{key}, Counter) = \text{HMAC}_{key}(Counter)$. The function g is the standard HMAC with any cryptographically secure hash function, such as SHA256. The key is the previous n -bit session key K_{i-1} and the input is the current m -bit

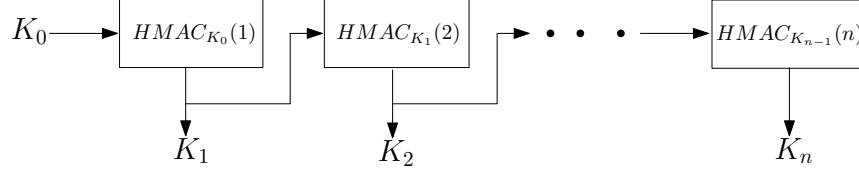


Figure 7.7: **The function** $g(K_{i-1}, i) = \text{HMAC}_{K_{i-1}}(i) = K_i$ where $K_0 = DS_K$. It generates n session keys sequentially.

counter i which produces the next n -bit session key $\text{HMAC}_{K_{i-1}}(i)$ as shown in Fig. 7.7. Initially a system-wide global variable *Counter* i is initialized at value one. It gets incremented at each registration request satisfied by the U2F token. The *Counter* is handled internally by the token therefore no external control is possible. As *Counter* i is incremented for each new registration, we get different session keys K_i . This K_i is then used for computing Keyhandle $Kh_i \leftarrow f_{K_i}(\text{AppId} \parallel SK_u)$. Therefore, the value i is associated with the value Kh_i and the pair (Kh_i, i) is stored internally inside a database of U2F token represented as U2F_db. At the time of authentication, the value of Kh_i is verified only if the correct value of i is provided.

Security: The function $\text{HMAC}(\text{key}, \text{Counter})$ works as a pseudorandom number generator³. Therefore, it increases the lifetime of the device-secret key and achieves the provable security gains in practice [24]. This sequential approach of key generation also provides forward secrecy, i.e., disclosure of the current key does not reveal the previously generated session keys. To protect future keys, use of random nonce instead of *Counter* is advisable. In that case nonce value should not be shared outside the U2F token and the keyhandle can be used as the index of the database inside the U2F token.

Usability: Our aim is to achieve DPA resistance and at the same time have efficient computation inside the U2F token. The *Counter* based approach avoids repeating the same session key for each registration but it is a huge computational overhead to compute the session key every time. When the current *Counter* is much larger than the initial value, it increases the required number of computations significantly if only the initial secret is available.

The computation of every new session key needs one extra hash computation if the previous session key is stored. Therefore, at the time of registration, the overhead is just one hash computation. We assume that the current counter value is i , the corresponding session key K_i is stored at the U2F token, and at this instance, an authentication request for j^{th} registered value with $j \ll i$ reaches the token. As $(j - 1)^{\text{th}}$ session key is not stored, it requires j computations to verify the compute the session key starting from the initial counter. Therefore, at the time of authentication, if the current counter value is much

³A pseudorandom number generator is an algorithm which generates a sequence of numbers that are computationally indistinguishable from true random numbers [116].

larger than the initial counter, the computation overhead is also significant to re-derive the corresponding session key (assuming that the computation starts with the initial secret). However, we can greatly reduce the overhead of authentication phase. One possible solution is estimate what will be a good gap between the stored session keys, that is, every k th session key can be stored rather than only the first one, for a suitable value of k . Since the tokens have a limited memory, and the number of registrations is upper bounded by available memory, storing multiple session keys does not cause significant overhead. This is due to the fact that k is likely to be quite large as we explain next.

The value of k can be estimated by considering the time to perform a single HMAC computation and estimating the number of feasible HMAC computations that can be performed in an allowable time. The session keys can then be stored while maintaining a fixed gap between different counters. Keeping every k th session key, at most $k - 1$ HMAC computations are needed to regenerate any session key, and this is within allowable time period due to the choice of k .

We assume that the computation of both Keyhandle and session key requires the same cryptographic primitive (HMAC). Since Keyhandle generation logic is already supposed to be implemented in the token, no extra implementation overhead is caused for the computation of session keys. We recommend to store the Keyhandle and the corresponding *Counter* for each registered relying party at the U2F token. The requirement of this storage is explained in detail in section 7.7. Considering the existing storage capacity of any hardware token, space overhead due to (Keyhandle, *Counter*) pair inside the U2F token is not significant.

Proposed Technique 2: This technique to generate session keys is due to Kocher [111]. An initial secret key, which happens to be DS_K in the case of U2F, is used in conjunction with a *Counter* i to derive a session key K_i . Two invertible functions F_A and F_B , and their corresponding inverses F_A^{-1} and F_B^{-1} respectively, are also needed in this method. The function g of Algorithm 7.1 is instantiated by one of the 4 functions F_A , F_B , F_A^{-1} , or F_B^{-1} , depending on the value of counter i [111]. The technique is briefly explained next.

Fig 7.8 shows the sequence of states (shown as dots) indexed with counters starting from the initial value 1. A state with counter i represents the session key K_i and a method to derive it. The index depth D is an additional parameter, which is a predefined constant, and represents the ‘levels’ of session keys present. The value of D is fixed by estimating the possible number of session keys that are needed to be derived. For example, Fig 7.8 is for the case $D = 5$, meaning that five levels of key values are present. Each rectangular box in Fig. 7.8 represents a specific session key. Thus, multiple dots in a rectangular box represent different states sharing the same session key.

The top level (row 0) in the figure has a single box containing states 1, 31 and 61 all of which share the same value of session key. The next level (row 1) contains two boxes, both of which contain 3 states. The left box contains states 2, 16 and 30 which share the same value of session key. Similarly, the right box contains states 32, 46 and 60 which share a common value of session key different from the left one. Similarly, the other levels and boxes represent different session keys. There is a unique session key per box.

There is only one box which has one node at level 0. There are 2 boxes containing 3 nodes each at level 1. Going forward, there are 4 boxes each of which contains 3 nodes at level 2. In general, all the levels N except the last one (that is $1 \leq N < D - 1$) contain 2^N boxes each of which contains 3 session keys. The last level ($N = D - 1$) contains 2^N boxes and each box has a single key. Thus, the total number of keys for a session key structure with depth D described here is given by Eqn. 7.1. For the example figure presented here, D is 5 and it results in 61 session keys according to this equation. These 61 states are seen in Fig. 7.8. It is clear from the description that any particular session key is used no more than 3 times within the structure.

$$\text{No. of session keys} \leq 2^{D-1} + \sum_{i=0}^{D-2} 3(2^i) = 2^{D-1} + 3(2^{D-1} - 1) = 2^{D+1} - 3 \quad (7.1)$$

The session keys are derived iteratively following the relation given below.

$$\begin{aligned} K_1 &= g(DS_K) \\ K_i &= g(K_{i-1}) \text{ for } 2 \leq i \leq (2^{D+1} - 3), \text{ where } g \text{ is as defined in Eqn. 7.2.} \end{aligned}$$

$$g = \begin{cases} F_A & \text{if moving downwards from a parent node to the child node on the left} \\ F_B & \text{if moving downwards from a parent node to the child node on the right} \\ F_A^{-1} & \text{if moving from the left node to its parent} \\ F_B^{-1} & \text{if moving from the right node to its parent.} \end{cases} \quad (7.2)$$

Security: This approach of re-keying can be seen as a hybrid approach incorporating sequential and parallel approaches [24]. The session key is derived by applying a series of computations depending on the value of the *Counter*. A session key is not repeated more than three times in this construction, therefore, it is hard to gather enough traces for leakage analysis of the session keys. The proposal requires the use of a reversible function and any secure block cipher can be used. The security analysis of the scheme in [24] shows that it is a secure approach.

Usability: The implementation needs four functions $F_A, F_A^{-1}, F_B, F_B^{-1}$ which can be easily instantiated with a single block cipher like AES computed with the previous session key to generate the current session key. At the time of registration, the computation for a session key requires one AES evaluation if the previous session key is already stored. This is a small computational overhead which is not an issue. At the time of authentication, the maximum number of AES evaluations is equal to the depth D , i.e. the complexity is $\log(n)$ where n is the total number of possible session keys. When $D = 39$, the number of derived session keys is more than 1 trillion ($\approx 10^{12}$) which means that practically we never require more than

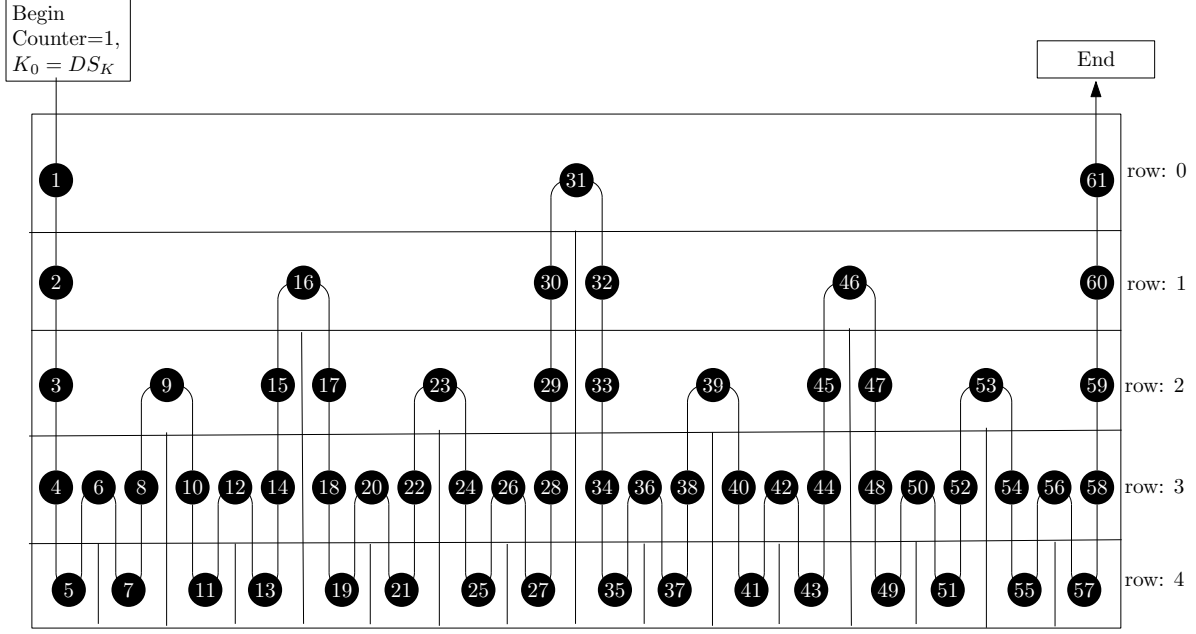


Figure 7.8: Kocher’s session key generation method with function $g(K_{i-1}, i) = K_i$ and index depth $D = 5$, taken from [111]. The function g of Algorithm 7.1 is instantiated by one of the 4 functions F_A , F_B , F_A^{-1} or F_B^{-1} in a sequence depending on the *Counter* i as explained in the text.

39 AES evaluations to derive a session key. This overhead is not significant if the AES-NI instruction set extension is available on the token. Current implementations of AES-128 on Intel processors supporting AES-NI can allow approximately 2^{29} operations per second [166].

With the assumption that the computation of Keyhandle requires the same block cipher (AES), no additional hardware/software is required for session key computations. Storage for the registered (Keyhandle, *Counter*) pair inside the U2F token is required. Considering the existing storage capacity of hardware token, this overhead is not significant.

7.5.2 Countermeasure to minimize the effect of Attestation Private Key Recovery

The signature function that uses the attestation private key also leaks information that can be captured through side channel analysis as explained in Section 7.4.2. Usually, in public key infrastructure (PKI) [154], the verification of the digital signature is satisfied by the use of a certificate. Intuitively, it may seem that use of session key will prevent the side channel attack on attestation private key as well. However, getting a valid certificate for each session key generated from attestation private key is not practical in an asymmetric cryptosystem. Hence, it is difficult to prevent side channel attacks using the existing U2F design for signature generation. Therefore, we suggest a minor modification in the current approach for assigning the attestation private key to the U2F token. The approach cannot prevent the

recovery of the attestation private key but minimizes the effect of the attack. Before explaining our proposed recommendation, we explain the existing issues with attestation private key for the U2F token.

Trust on FIDO The attestation certificate *Acert* to prove the genuineness of the U2F token is issued to the manufacturer by some trusted CA. The FIDO server maintains a centralized database of all valid certificates. One of the goals as mentioned in the specifications [6, 7] for U2F is to make it a standard solution. A standard should be acceptable to all. This means FIDO needs to acquire global acceptance with a trust on this centralized approach. However, it is difficult to develop trust globally. One possible solution could be a decentralized approach. If a government agency under each country takes the responsibility to maintain a server with a trusted list of CAs and list of valid certificates, the citizens can be expected to trust it. The FIDO server should also include all the valid certificates and in that case, the U2F approach could be used as a standard. To prove the genuineness of the token, a relying party can verify with the country-level trusted server or the FIDO server or both. This can lead to a standard U2F solution which is globally accepted.

Genuineness or Anonymity? In the specifications [6, 7], it is mentioned that a manufacturer of a U2F token provides an attestation certificate signed by some trusted CA along with the attestation private key, permanently written inside the token. The attestation certificate and signature with signing key prove the genuineness of manufacturer and not the token. This is because a large bundle of U2F tokens is provided with the same attestation private key and corresponding certificate to preserve the anonymity of the token. However, if a single device is compromised, then all the tokens with the same identity are compromised. This can be prevented if each device has its own attestation public-private key pair. This attestation certificate is required to show the genuineness of the token. As U2F is a second-factor authentication device, and the user association is already established with the username and password, there is no enhanced security obtained with the anonymity of the token. Therefore, the token should only prove its legitimacy which is the main goal of the tokens. Therefore, we see a trade-off between security and anonymity in the case of U2F protocol. Since the compromise of a single attestation private key impacts a large number of tokens, we believe that security should be preferred over anonymity.

Countermeasure to minimize the effect of Attestation Private Key Recovery

The main idea behind the ‘private key’ of asymmetric cryptosystem is to keep it private with the owner and not to share it with any third party. However, the scenario of sharing same attestation private key with a huge number of tokens can be visualized as the manufacturer keeping its own private key for all the tokens manufactured by it. Therefore the manufacturer should not distribute the same attestation private key among a large number of U2F tokens. Instead, it should have its own private key and a corresponding certificate signed by a trusted CA. A unique public-private keypair should be generated by the manufacturer for each token. To prove genuineness in this scenario, the manufacturer should sign the public

key of the token. The signature by the manufacturer together with the certificate by a trusted CA including the manufacturer public key proves the genuineness of the U2F token. This two layers of signature prevent the sharing of the private key and prove the genuineness of the U2F token which is the goal behind the attestation concept of U2F tokens. This also enhances the security as attacker has to forge two signatures now. The unique attestation private key is still vulnerable to the attack which is shown as Attack point-II in Fig. 7.6 but limited to target a single U2F token at a time instead of a large number of tokens.

7.5.3 Overview of the U2F protocol with Our Proposed Modifications

In this section, we briefly explain the U2F protocol incorporating the proposed countermeasure and the recommendation.

Our Modified U2F Manufacturing Phase: Unique Secrets for U2F Token This phase is executed at the time of manufacturing of the U2F token as shown in Fig 7.9. In this phase, the U2F token is provided with a randomly generated unique secret key called Device Secret Key DS_K at Step1. At Step2, a unique public-private key pair (PK_T, SK_T) is generated from a key generation function $\mathcal{K}(\cdot)$ for the token. The asymmetric keypair of the manufacturer is represented as (PK_M, SK_M) . At Step3, a certificate $Acert$ which certifies the key PK_M (public key of the manufacturer) is provided to the U2F token thus verifying the signature $\mathcal{S}_M$ of the manufacturer. The $Acert$ is issued by a trusted CA to the manufacturer. At Step4, the key PK_T is signed by the manufacturer using his secret key SK_M . The signature obtained in this step is denoted as $\mathcal{S}_M$. Since the corresponding public key of the manufacturer is certified by a trusted CA, the genuineness of the token is established by the chain of trust. Both DS_K and (PK_T, SK_T) are fixed and unique for a token. A global variable ctr initialized to zero is provided at Step5. This helps to detect cloning as explained in Section 7.2.3 during the authentication phase. A global variable *Counter* is provided at Step6 which is initialized to 1 and used to generate session keys from DS_K to prevent repeated use of DS_K at the following registration phase.

As can be seen from the above description, our proposal adds two extra steps (namely Step 2 and Step 6), and makes a minor modification in Step4 of the original protocol. The asymmetric key pair (PK_T, SK_T) supplied with each token solves the problem of sharing the private key which is described in Section 7.5 earlier.

Our Modified U2F Registration Phase: Use of Session Key The registration process is explained in Fig. 7.10. At Step1, the username and password are provided to the server. On verification of the received values, the server generates a r -bit random value R at Step2. At Step3, the value R and server AppId are sent to the browser. Adding the ChannelId, the browser forwards the received values to the U2F token at Step4. At Step5, the U2F token first generates the site-specific key pairs (PK_u, SK_u) applying a random key

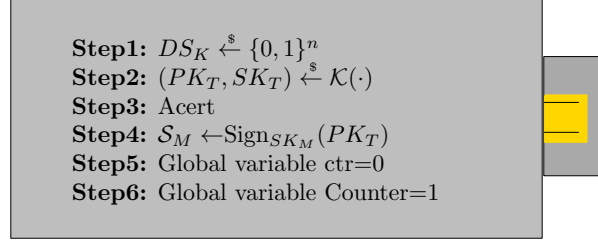


Figure 7.9: **Our Modified U2F Manufacturing Phase**, DS_K is a randomly generated n bit device secret. $\mathcal{K}(\cdot)$ is a random key generation function to generate a unique asymmetric keypair (PK_T, SK_T) for the token. The asymmetric keypair of the manufacturer is (PK_M, SK_M) . Acert is the certificate issued by a trusted CA which contains PK_M . $\mathcal{S}_M$ is a signature on PK_T by the manufacturer with signing key SK_M . At registration phase, a signature with SK_T is verified with PK_T which is again verified from $\mathcal{S}_M$ and Acert. The global variable ctr is initialized at value zero which is incremented after each successful authentication as explained in Section 7.2.3. The global variable *Counter* is initialized at value one which is incremented after each successful registration as explained in Section 7.2.2.

generation function $\mathcal{K}(\cdot)$. The value of *Counter* is assigned to a variable i which gets incremented after each registration. It then computes the session key K_i following the steps of Algorithm 7.1. The session key generation function $g(K_{i-1}, i)$ can be instantiated with either the proposed hashed based approach or using Kocher's technique as explained in Section 7.5 and shown in Fig. 7.7 and 7.8 respectively. Next, the Keyhandle Kh_i is generated from the function $f_{K_i}(\cdot)$ following one of the approaches as explained in Fig. 7.3 and 7.4. Finally, the token adds the values (PK_u, SK_u, Kh_i, i) in a database represented as U2F_db which is stored in the token. It computes a signature $\mathcal{S}$ with the key SK_T on the values (c, PK_u, Kh_i, i) . At Step6 the token sends the values PK_u, Kh_i, i , Acert, $\mathcal{S}_M$ to the browser. The browser forwards the received values to the server at Step7. On receiving the values the server first verifies the signature $\mathcal{S}$ with PK_T after verification of $\mathcal{S}_M$ with Acert. If verified it keeps the values $(u, H(p \parallel s), PK_u, Kh_i, i, \text{Acert}, \mathcal{S}_M)$ in its database, else it rejects the registration request.

As can be seen from the above description, our proposal modifies Step5 by adding the computation of session key K_i and updating the U2F database with values (Kh_i, i) . The signature $\mathcal{S}$ is performed by the token specific signing key SK_T instead of SK_M . The modifications of Step5 correspondingly make a minor modification in Step6 and Step7 of the original protocol. At Step8 the genuineness of token is established by verifying two levels of signature. The use of session key for the generation of each keyhandle prevents the side channel attack mentioned in Section 7.4.1.

Our Modified U2F Authentication Phase At the time of authentication, receiving the request with username u and password p at Step2, the server checks for the corresponding registered values. If registered, the server sends the values AppId, Challenge R , Keyhandle Kh_i and *Counter* i to the browser at Step3. The browser forwards the received values along

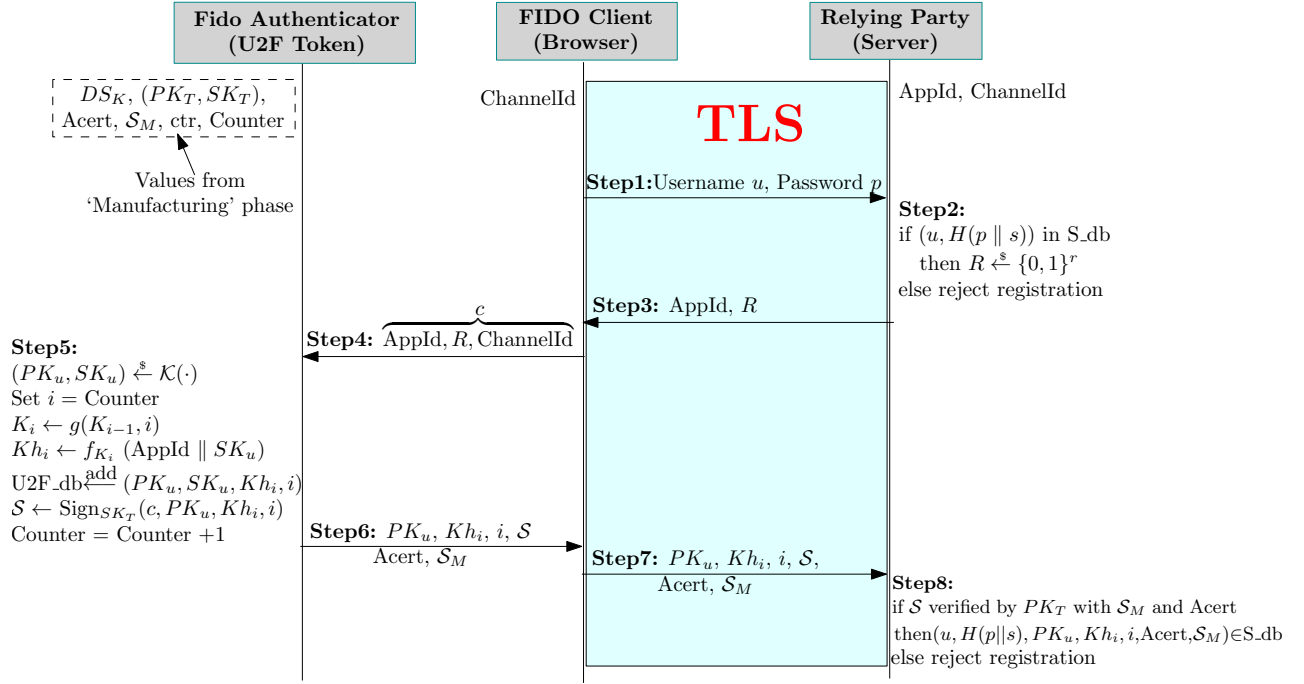


Figure 7.10: **Our Modified Registration Phase.** the key DS_K , the asymmetric keypair (PK_T, SK_T) with a signature $\mathcal{S}_M$ containing PK_T , Acert, counters 'ctr' and *Counter* are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId. S_db is the server database described in Section 7.6.4. R is a r bit random number generated by the server. $\mathcal{K}(\cdot)$ is a key generation function to generate site-specific random keypairs (PK_u, SK_u) . The function g generates the session key K_i as shown in Fig. 7.7 and 7.8. $f_{K_i}(\cdot)$ is the function to generate Keyhandle Kh_i as shown in Fig. 7.3 and 7.4, U2F_db is the database at U2F token as explained in Section 7.6.3.

with TLS ChannelId to the U2F token at Step4. At Step5 the U2F token first checks for the entry of the pair (Kh_i, i) in its database. If the entry is verified, it re-derives the session key K_i from function $g(K_{i-1}, i)$ following the re-keying technique explained in Section 7.5 and then verifies the received value Kh_i . If the function is $f_{K_i}^{-1}(\cdot)$, the U2F token computes $F_{K_i}^{-1}(Kh_i)$ which provides the output $(AppId' \parallel SK'_u)$. It then verifies if $AppId' = AppId$. Else for the case when the function is $f_{K_i}(\cdot)$, it computes $F_{K_i}(AppId \parallel SK_u) = Kh'_i$ and verifies if $Kh'_i = Kh_i$. Both the cases are shown in Fig. 7.3 and 7.4 respectively. On verification for both cases, the token increments a counter, we represent it as 'ctr' which is initialized at value zero as mentioned in [6, 7]. The value ctr is incremented and at Step6, a signature $\mathcal{S}$ including ctr are sent to the browser by the U2F token. The browser forwards the received values to the server at Step7. Finally, at Step8 the server verifies the signature $\mathcal{S}$ with PK_u and after verification, it updates the value of ctr else the authentication is rejected.

As can be seen from the above description, our proposal makes minor modifications at Step3 and Step4 by adding *Counter i* which was introduced at the modified registration phase. The verification process of Step5 needs one extra computation to re-derive the session key K_i . The signature $\mathcal{S}$, which was used to sign the values received from Step4 originally, now additionally signs the value i and the corresponding changes are reflected in Steps 6 and 7. The use of token specific private key SK_T mitigates the attack mentioned in Section 7.4.2 and the use of session key at Step5 helps in secure authentication of each registered user.

7.6 Design Rationale

In this section we analyse the role of both the counter values, namely *Counter* and 'ctr', and the databases S_db and U2F_db for the execution of our modified U2F protocol.

7.6.1 Role of *Counter*

The global variable *Counter* is initialized at 1 and gets incremented after each successful U2F token registration to a website. Specifically the value *Counter* is crucial to generate the value Keyhandle as shown in Fig. 7.10. The value of *Counter* signifies the number of entries in the U2F database U2F_db as explained in Section 7.6.3. As described in Fig. 7.10, counter i affects the generation of the session key K_i and subsequently the generation of the Keyhandle Kh_i . Thus, the key K_i is a function of the *Counter*. The pair (Kh_i, i) corresponding to a username u is maintained in both the server and the U2F token databases. At the time of authentication the server sends the pair (Kh_i, i) to the U2F token but before initiating any computations, the U2F token first checks for the valid entry of the (Kh_i, i) pair in its database. It proceeds only if the entries match.

Problem with invalid (Kh_i, i) pair: At the time of authentication, an attacker can provide any randomly chosen value of *Counter i* which will generate a specific session key K_i . The key K_i is fixed for this i but unknown to the attacker. Note that the attacker cannot

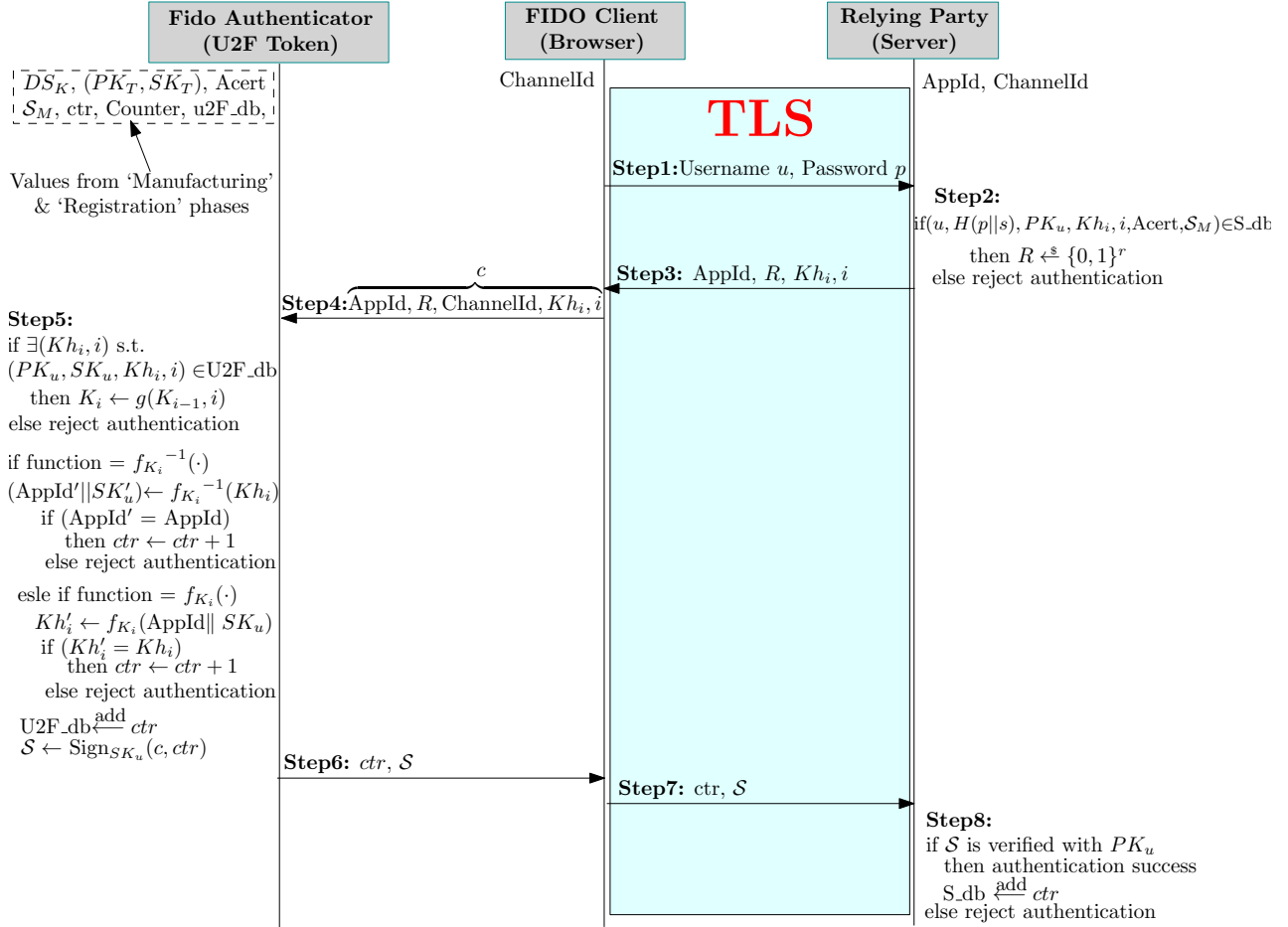


Figure 7.11: **Our Modified Authentication Phase.** The key DS_K and the asymmetric keypair (PK_T, SK_T) with signature S_M which includes PK_T , $Acert$, the counters 'ctr' and $Counter$ are from manufacturing phase. The database $U2F_db$ as explained in Section 7.6.3 contains all registered values from registration phase. S_db is the server database explained in Section 7.6.4. R is a r bit random number generated by the server. $AppId$ is the ULR of the server. $ChannelId$ is the TLS $ChannelId$. Kh_i is the Keyhandle corresponding to the $Counter$ i . The function $g(K_{i-1}, i)$ generates the session key K_i as shown in Fig. 7.7 and 7.8. The function $f_{K_i}(\cdot)$ is shown in Fig. 7.3 and 7.4. 'ctr' is the counter used to record the number of authentication requests successfully satisfied by the U2F token.

succeed in computing or guessing the Keyhandle Kh_i except with a negligible probability. If the Keyhandle was not being checked (as in the original protocol), an attacker can make multiple attempts with any counter i with randomly chosen (incorrect) Keyhandle Kh'_i and collect power traces to extract information about the session key K_i . This attack will yield information about the correct key K_i as this key depends only on the *Counter* i .

To prevent such a malicious attempt, it is required to allow only the legitimate (Kh_i, i) pairs to initiate authentication requests. Therefore, we recommend to keep all the registered (Kh_i, i) pairs in a database inside the U2F token. Since the U2F token is assumed to be a secure device [6], this database can be considered safe. When an invalid pair is provided to the token, the authentication request is refused. Hence, our proposed modification prevents the collection of traces by an attacker not having valid credentials. The problem of an attacker deliberately trying to overflow the database is not an issue with the U2F token and is discussed while describing technique 1 in Section 7.5.

7.6.2 Role of ‘ctr’

The use of ‘ctr’ is proposed in original specifications of the U2F protocol [6, 7]. As explained in the specifications, ‘ctr’ can be a global or a local (specific to a website) variable. It gets incremented after each successful execution of authentication phase of the U2F protocol. Without loss of generality, we consider ‘ctr’ as a global variable. After each successful authentication request, the value of the ‘ctr’ gets incremented and the server keeps the updated value in its database. If the received value of ‘ctr’ at Step8 of Fig. 7.11 is less than the last recorded ‘ctr’ at the server database, it shows a possible cloning of the U2F token. Thus the use of the counter ‘ctr’ may help detect cloning of the U2F token. As mentioned in [6, 7], it is not a strong solution to detect cloning due to synchronization issues, false detection etc.

7.6.3 Role of U2F_db

The database at U2F token is represented as U2F_db. It contains the value generated at registration phase corresponding to each registration. For each registration, the U2F_db keeps a tuple containing site-specific keypairs (PK_u, SK_u) and the Keyhandle Kh_i corresponding to *Counter* i , for instance the tuple (PK_u, SK_u, Kh_i, i) as shown in Fig. 7.10. The value of *Counter* shows the total number of entries in the U2F_db. The use of the database cannot be optional to provide a secure authentication against side channel attack. To make the protocol side channel resistant, it is required to allow only the legitimate computations inside the U2F token. Therefore, valid entries are verified from the U2F_db before initiating any computation. At our modified authentication phase, information leakage is possible due to illegitimate computation providing wrong counter and Keyhandle pair (Kh_i, i) as explained in Section 7.6.1. This can be prevented by verifying the existence of the pair (Kh_i, i) in the U2F_db before initiating the computation.

7.6.4 Role of S_db

The database at the server is represented as S_db. It contains the values generated at registration and authentication phases corresponding to each registered username. For each username u , a tuple including hash of the password with salt $H(p \parallel s)$, public key PK_u for u , Keyhandle Kh_i corresponding to *Counter* i , the certificate Acert, the signature $\mathcal{S}_M$ and the value of ‘ctr’ to record number of successful authentication corresponding to the U2F token is kept in the database. The tuple $(u, H(p \parallel s), PK_u, Kh_i, i, \text{Acert}, \mathcal{S}_M, \text{ctr})$ is as described in Fig. 7.11. At the time of authentication, if the server receives a ‘ctr’ from the U2F token which is less than the value of ‘ctr’ in S_db, it marks the U2F token as compromised. Thus, this helps detect a cloning attack even though it is not a perfect solution due to synchronization issues, false detection etc.

7.7 Evaluation

We evaluate our proposed modified U2F protocol considering different scenarios under the modified registration and authentication phases. We evaluate the protocol by considering a different combination of values that can be tampered and their effect on the security of the overall system.

7.7.1 Modified Registration Phase

In this phase, the values forwarded by the browser to the U2F token at Step3 in Fig. 7.10 contain AppId, ChannelId and random challenge R . Only the value AppId is useful for the computations of Keyhandle which is used in the authentication phase.

Scenario-I (AppId, R , ChannelId) This is the case of an authentic request for registration. Valid Kh_i would be registered when corresponding *Counter* is i . Multiple registration requests with legitimate inputs may reveal the signing key SK_T as discussed in Section 7.4 through side channel, however, a single signature is safe.

Scenario-II (AppId', R , ChannelId) This is the case when the authentic AppId is replaced with a tampered AppId' at the time of registration request. The Keyhandle at the U2F token would contain AppId' and not AppId. Using this malicious approach, a user would get authenticated for fake AppId' instead of the real AppId, without her even noticing it. When the real user attempts to access AppId, it leads to denial of service since the token has been registered for AppId'.

7.7.2 Modified Authentication Phase

We evaluate possible scenarios by tampering the values (AppId, Kh_i, i) corresponding to *Counter* i . These values are provided by the browser to the U2F token at Step4 of Fig. 7.11 and the verification in Step5 is performed using them.

Scenario-I (AppId, Kh_i, i): This is the case when valid inputs corresponding to *Counter* i are provided to the U2F token. As the inputs are not tampered, validation at Step5 and beyond succeeds. This is the scenario of successful authentication of the U2F token.

Scenario-II (AppId, Kh_i, i'): This is the case when *Counter* i corresponding to the Kh_i is tampered to a fake value i' . In this scenario, if the check for entries at U2F_db is not performed, incorrect computation corresponding to i' produces $K_{i'}$ and subsequently affects the computation of $Kh_{i'}$. The detection is possible with the verification of Kh_i having legitimate AppId as explained in Fig. 7.11. When multiple requests with i' and different legitimate (AppId , Keyhandle) pairs are provided, then some information about the key $K_{i'}$ may be leaked. By compromising $K_{i'}$, an attacker can reveal all the K_i s where $i > i'$. Since the function $f_K(\cdot)$, as shown in Fig. 7.3, is invertible and K is the disclosed secret K_i , it also discloses the value SK_u (see Fig. 7.11). The signature at each authentication phase is performed with this signing key SK_u whereas at registration phase the signing key is SK_T . Therefore, it is easy to simulate the functionality of the U2F token including SK_u without the possession of the token at the time of authentication. However, it is not possible to simulate the registration process as registration needs the signing key SK_T . The attack against the authentication phase described in this scenario can be thwarted by verifying the tuple (AppId, Kh_i, i) with the database U2F_db.

Scenario-III (AppId, Kh'_i, i): This is the case when Kh_i corresponding to *Counter* i is tampered to a value Kh'_i . In this scenario, if the check for entries at U2F_db is not performed, computations corresponding to i produce K_i and then Kh_i . The U2F token then verifies the received Kh'_i with the computed Kh_i . The mismatch in values signals authentication failure by the token. Therefore, the modification or tampering of Kh_i results in denial of service (DoS) attack after performing unnecessary computations.

Apart from the DoS attack, this scenario is similar to Scenario-II above. Multiple tampered Kh'_i corresponding to a fixed i may disclose the key K_i through side-channel attack. If the function $f_K(\cdot)$, as shown in Fig. 7.3, is invertible it further discloses SK_u . If the modification of Kh_i can be detected early then such side channel disclosure can be prevented or limited. Verifying the input tuple with the valid entries of the U2F database is required to disallow tempered inputs.

Scenario-IV (AppId, Kh'_i, i'): This is the case when both the Keyhandle and the *Counter* values are tampered. Without the check for valid entries, both the scenarios described in II and III are possible. Verification of inputs with the U2F database is required in this case as well.

Scenario-V (AppId', Kh_i, i): This is the case when AppId corresponding to valid registered Kh_i, i is tampered to another value AppId'. This modification fails to satisfy the verification in Step5 of Fig 7.11. Hence, this scenario causes a DoS attack since the registration data contains AppId which does not match with the tampered AppId'.

Thus storing the tuple (AppId, Kh_i, i) at the U2F database is required to prevent the DoS attack.

Scenario-VI $(\text{AppId}', Kh_i, i')$: This is the case when *Counter* i corresponding to the Keyhandle Kh_i is tampered to another value i' and AppId is modified to AppId'. In this scenario, if the check for entries at U2F_db is not performed, incorrect computations corresponding to i' produce $K_{i'}$ and subsequent computations produce $Kh_{i'}$. This leads to denial of service to a valid U2F token.

Moreover, these unnecessary computations leak information about $K_{i'}$ via side channels. Multiple such attempts with valid Keyhandle but incorrect i' may compromise $K_{i'}$ and an attacker can reveal all K_i s where $i > i'$. This may also disclose the value of SK_u as explained in Scenario-II earlier. Hence, software simulation of the U2F authentication phase is feasible. Therefore, verification of the inputs with U2F database is required to prevent this scenario.

Scenario-VII $(\text{AppId}', Kh'_i, i)$: This is the case when Kh_i corresponding to *Counter* i is tampered to value Kh'_i and AppId to AppId'. In this scenario, if the check for entries at U2F_db is not performed, computation corresponding to i produces K_i and subsequently Kh_i . The detection is possible with the check that $Kh_i \neq Kh'_i$. This is a case of DoS attack and causing unnecessary computations. As explained earlier in Scenario-II, these unnecessary computations may reveal keys K_i and SK_u . Verification with the database is needed to prevent this case.

Scenario-VIII $(\text{AppId}', Kh'_i, i')$: This is the case when all the three values, the Keyhandle, the *Counter* and the AppId are tampered. Without the check for valid inputs, both the Scenarios II and III explained earlier are possible. Further, repeated inputs with incorrect i' may leak information about $K_{i'}$ and all K_i s where $i > i'$. Similar to the Scenario-II, SK_u may be disclosed when function, as shown in Fig. 7.3, is $f_{K_i}^{-1}(\cdot)$ and correct Kh_i corresponding to i is known. This allows an attacker to simulate future authentications by the token. Verification of the inputs with a database is required to prevent this case as well.

Scenario-IX $(\text{AppId}, Kh_i, i, R, \text{ChannelId})$: The random value R is fixed for a particular session but changes at each authentication request. When AppId, Kh_i , and i are genuine, then all the computations of Step5 in Fig. 7.11 are verified. Afterwards, signature $\mathcal{S}$ is computed utilizing R .

Since R is a one-time random input, an attacker with access to the U2F token can initiate multiple computations corresponding to the legitimate AppId, Kh_i and i values. The attacker can thus generate signatures on different inputs (as R varies) with the same secret key SK_u . Collection of power traces corresponding to different input-output pairs again leaks information about this key SK_u which may be extracted using side channel attacks.

This scenario describes an inherent limitation of the U2F protocol and the only way it can be prevented is to restrict the use of the token by an attacker with all valid inputs with multiple random values of R .

7.8 Limitations of U2F

At the authentication phase of the U2F protocol, a signature $\mathcal{S}$ is obtained with the site-specific signing key SK_u as shown at Step5 of Fig. 7.11. The inputs to the signing algorithm are fixed except R which is randomly generated at each authentication request initiated by the token. As explained above, in Scenario-IX, this changing R provides different input-output pairs computed under the same secret key SK_u to an attacker, who may utilize side channel attacks to learn this secret. For example, it is easy for an attacker to initiate a legitimate authentication request multiple times and compromise the key. Then it can simulate the legitimate authentication without possession of the token. This attack is not possible to prevent with existing U2F solution. The possible countermeasure is to restrict computations and release of data only to the genuine users of the token. One possibility is to take biometric input to first authenticate a legitimate user and then initiate further computations. Therefore, the scenario above shows a limitation of the U2F solution. Another limitation is transaction non-repudiation as mentioned in the specifications [6, 7] and explained in Section 7.3.

7.9 Possible Side Channel Attacks on Universal Authentication Framework Protocol and Mitigation

The Universal Authentication Framework (UAF) protocol is an authentication protocol proposed by FIDO alliance which supports biometric authentication to provide a unified and extensible authentication mechanism that supplants passwords [5]. It involves five entities as shown in Fig. 7.12. We describe these entities next. The first entity is the user of the protocol and the second entity is the UAF authenticator which is similar to a U2F token but also includes a database and a verifier to authenticate the biometric data of users. The next entity, Authenticator Specific Module (ASM) is the software to work as an interface between the UAF authenticator and the browser. The fourth entity, the user browser is called the FIDO client. The fifth entity includes both the web server and the FIDO server. The web server is also called the relying party. The FIDO server is an additional server which runs on the relying party's infrastructure. This FIDO server maintains a database which includes information coming from the UAF authenticator. Similar to the U2F protocol, the UAF protocol can be explained in three phases:

1. *The UAF Manufacturing Phase* that assigns device secret key DS_K and an asymmetric keypair (PK_M, SK_M) to the UAF authenticator, apart from performing additional steps to handle the biometric input.

2. *The UAF Registration Phase* that registers a user for future authentication.
3. *The UAF Authentication Phase* that authenticates the registered users.

Following is the description of the three phases of the UAF protocol.

7.9.1 UAF Manufacturing Phase

This phase is executed at the time of manufacturing of the UAF token as shown in Fig 7.13. In this phase, the UAF token is provided with a randomly generated unique secret called Device Secret Key DS_K at Step1. At Step2, a public-private key pair (PK_M, SK_M) is provided to the token. The (PK_M, SK_M) keypair is not a one-time generation. Once randomly generated from a key generation function, the manufacturer provides the same (PK_M, SK_M) pair to all the tokens of a specific model manufactured by it. Note that the manufacturer can keep the same key pair for all the tokens manufactured by it (that is, it can treat all the tokens manufactured by it as the same model). At Step3, a certificate Acert issued by a trusted CA which includes PK_M (the public key of the manufacturer) is provided to the token. During the registration phase, the signature with SK_M is verified with PK_M which is extracted from Acert by the server. This verification proves the genuineness of the UAF token. Therefore, both DS_K and (PK_M, SK_M) are fixed for a token while the same (PK_M, SK_M) values are provided to multiple tokens which preserve the anonymity of the token. An integer counter ‘ctr’ is provided at Step4. This ‘ctr’ is initialized to value zero and gets incremented after each successful execution of the authentication phase.

7.9.2 UAF Registration Phase

The registration procedure is explained in Fig 7.14. Initially the username u and password p are communicated to the FIDO client at Step1 which is forwarded to the server at Step2. At Step3 the server verifies the values u and $H(p \parallel s)$ which is the hash of the password where s is the salt. On successful verification, the server triggers the UAF registration request to FIDO server at Step4. At Step5 the FIDO server generates a r bit random challenge R to differentiate each request. The FIDO server sends the UAF registration policy which includes the authentication modes supported by it and the value R to the server at Step6 which is forwarded at Step7 to the FIDO client. At Step8 the FIDO client forwards the identity of the server as AppId and the value R alongwith the registration policy to the ASM. ASM generates an n -bit random value called the kid and computes a value called keyhandle access token (KT) which is the hash of the AppId and kid . The ASM stores the pair (kid, KT) in its database represented as A_db. At Step9, the ASM forwards the received values of Step8 and the pair (kid, KT) to the UAF token. At Step10 the UAF token requests user verification satisfying the UAF registration policy communicated by the FIDO server. On successful user verification, at Step11 the UAF token first generates a website specific public-private key pair represented as (PK_u, SK_u) by applying a random key generation function $\mathcal{K}(\cdot)$. The function $\mathcal{K}(\cdot)$ can be an OpenSSL key pair generation library which

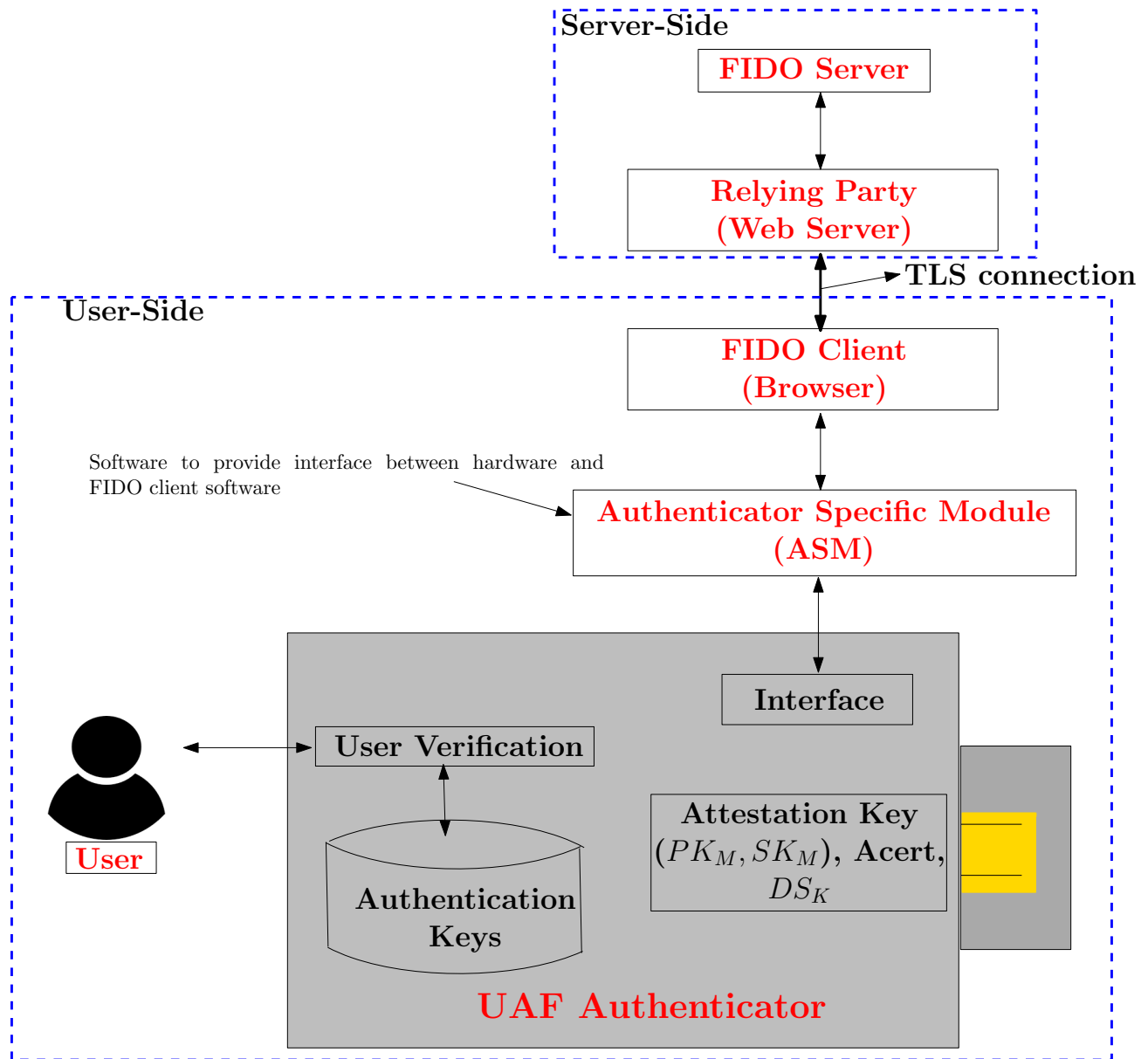


Figure 7.12: **UAF protocol entities.** UAF authenticator is the UAF token provided by manufacturer. The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate $Acert$ which includes PK_M are provided by the manufacturer. ASM is the software to provide interface between hardware token and client browser.

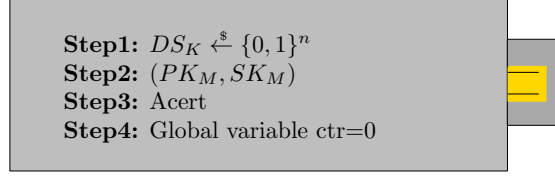


Figure 7.13: **UAF Manufacturing Phase.** The key DS_K is a randomly generated n -bit secret. The keypair (PK_M, SK_M) is the asymmetric keypair of the manufacturer shared with the token. Acert is the certificate signed by a trusted CA which includes PK_M . At the time of registration phase, a signature with SK_M is verified with PK_M from Acert by the server to prove the genuineness of the UAF token. The global variable ctr is initialized to zero and incremented after each successful authentication.

needs as input an elliptic curve such as the NIST standard P-256 curve. We define a key dependent function $f_K(\cdot)$ where K is the secret key. The UAF token computes the function with key DS_K and outputs $f_{DS_K}(SK_u, KT, u)$ which is called the Keyhandle Kh . The above function can be instantiated with any block cipher such as AES in CBC mode or HMAC. It then computes a signature $\mathcal{S} \leftarrow \text{Sign}_{SK_M}(R, \text{AppId}, \text{ChannelId}, \text{AAID}, kid, PK_u)$ where Authenticator Attestation ID (AAID) is a unique identifier assigned to a model of the FIDO authenticators. At Step12, the token sends the values $PK_u, C, \text{AAID}, kid$ and $\mathcal{S}$ along with ACert to the ASM. The ASM forwards the received values to the browser at Step13. The browser forwards the received values to the server at Step14. After receiving the values, the server forwards the values to the FIDO server at Step15. At Step16, the FIDO server first verifies the signature $\mathcal{S}$ with PK_M from the certificate Acert. This verification proves the genuineness of the UAF token. On verification, the FIDO server updates its database FIDO_db with the values (AAID, kid) corresponding to the username u and it shows the successful registration of the UAF token else the server rejects the registration request. The UAF protocol allows a user to register multiple tokens with the same account.

7.9.3 UAF Authentication Phase

The authentication procedure is explained in Fig 7.15. Once second factor authentication with UAF is registered, for example, when the values $(\text{AAID}, PK_u, kid, \text{Acert})$ corresponding the username u are registered with the FIDO server, the subsequent login to the website needs to verify the registered values by communicating with the UAF token through the browser. The steps are as follows. After receiving username u at Step1, the browser forwards the value to the server at Step2. On receiving the value, the server requests UAF authentication to the FIDO server at Step3. The FIDO server checks its database to retrieve the values AAID and kid . At Step4 the FIDO server generates a r bit random challenge R and sends it along with AAID, kid and the policy to the server at Step5. The server appends its AppId and forwards the received values to the browser at Step6. The browser forwards the received information with ChannelId to the ASM at Step7. Checking the policy, the ASM selects the authenticator

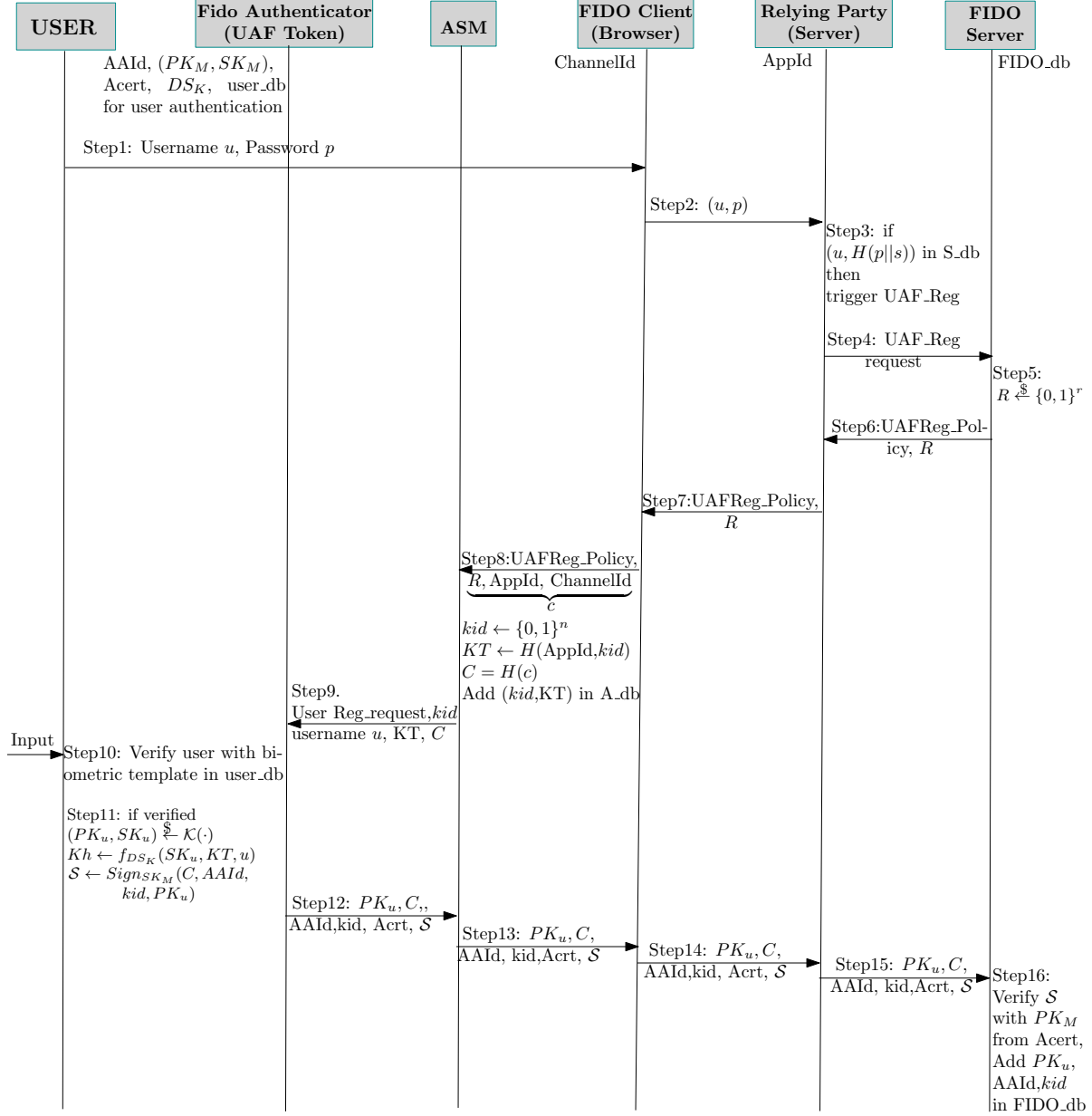


Figure 7.14: **UAF Registration Phase.** The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate $Acert$ which includes PK_M and the counter ‘ctr’ are from the manufacturing phase. $AppId$ is the URL of the server. $ChannelId$ is the TLS $ChannelId$. $FIDO_db$ is the FIDO server database. R is a r bit random number generated by FIDO server. $\mathcal{K}(\cdot)$ is a random key generation function which generates the keypair (PK_u, SK_u) . $f_{DS_K}(\cdot)$ with key DS_K is the function to generate Keyhandle Kh , A_db is the database of ASM.

supported by the attached UAF token. It then computes the keyhandle access token KT from the received AppId and kid and a value C which is the hash of values (AppId, R , ChannelId) at Step8. At Step9, the ASM sends the values AAIId, KT and C to the UAF token. On receiving the values, the UAF token requests the user verification at Step10. It first verifies the user with the stored biometric template at Step11. On successful verification, it verifies Kh corresponding the received KT at Step12. Specifically, it performs inverse computation $f_{DS_K}^{-1}(Kh) = (SK'_u, KT', u')$. It then compares the received KT with the computed KT' . If a successful match happens, the token increments a counter represented as 'ctr'. This 'ctr' can be a global or a local variable. If it is local variable then each AppId gets its own 'ctr' else a single 'ctr' is used across all registered AppId. Throughout the explanation we consider 'ctr' as a global integer variable. The value of the 'ctr' is incremented after each successful authentication by the UAF token. This value 'ctr' is introduced to detect cloning of the UAF token as explained in Section 7.2.3 for U2F. The UAF token signs the values (AAId, C , ctr) with SK_u which is represented as $\mathcal{S}$ and sends the values $\mathcal{S}$ and ctr to the ASM at Step13. The ASM then forwards the values to the browser at Step14 and similarly the browser forwards to the server at Step15. At Step16 the server sends the values to the FIDO server. Finally, the FIDO server verifies the received signature $\mathcal{S}$ with stored key PK_u for the username u from FIDO.db. On successful verification, it keeps the 'ctr' value with the database and this shows the successful completion of the authentication process.

7.9.4 Attack on UAF

The UAF authenticator, similar to the U2F authenticator, is assigned a unique device secret key DS_K and an asymmetric keypair (PK_M, SK_M) with attestation certificate Acert at the Manufacturing phase by the manufacturer. The key DS_K is used to compute Keyhandle and can be compromised by applying side channel attacks following the explanation of the Section 7.4.1. Similarly, the attestation private key SK_M , which is used as the signing key to perform signature during the UAF registration phase, can be compromised as explained in Section 7.4.2. We recommend the same countermeasure explained in Section 7.5 to protect the DS_K in case of UAF and to provide unique (PK_M, SK_M) pair to each UAF authenticator to mitigate the attack on SK_M . Providing side channel attack resistant solution is crucial for any hardware that can be easily accessible to an attacker. Therefore, prevention or mitigation of SCA is of utmost importance considering UAF as well.

7.10 Summary

In this chapter, we observe that the side channel attack is possible on U2F protocol which may compromise the device secret key DS_k and attestation private key SK_M following the explanation of the specifications [6, 7]. Both of these keys are assigned to the U2F token at the manufacturing phase of the U2F protocol. This side channel attack can completely break the second factor U2F solution. We suggest countermeasure to protect the DS_k and suggest modification in the protocol to mitigate the attack of compromising the SK_M . We

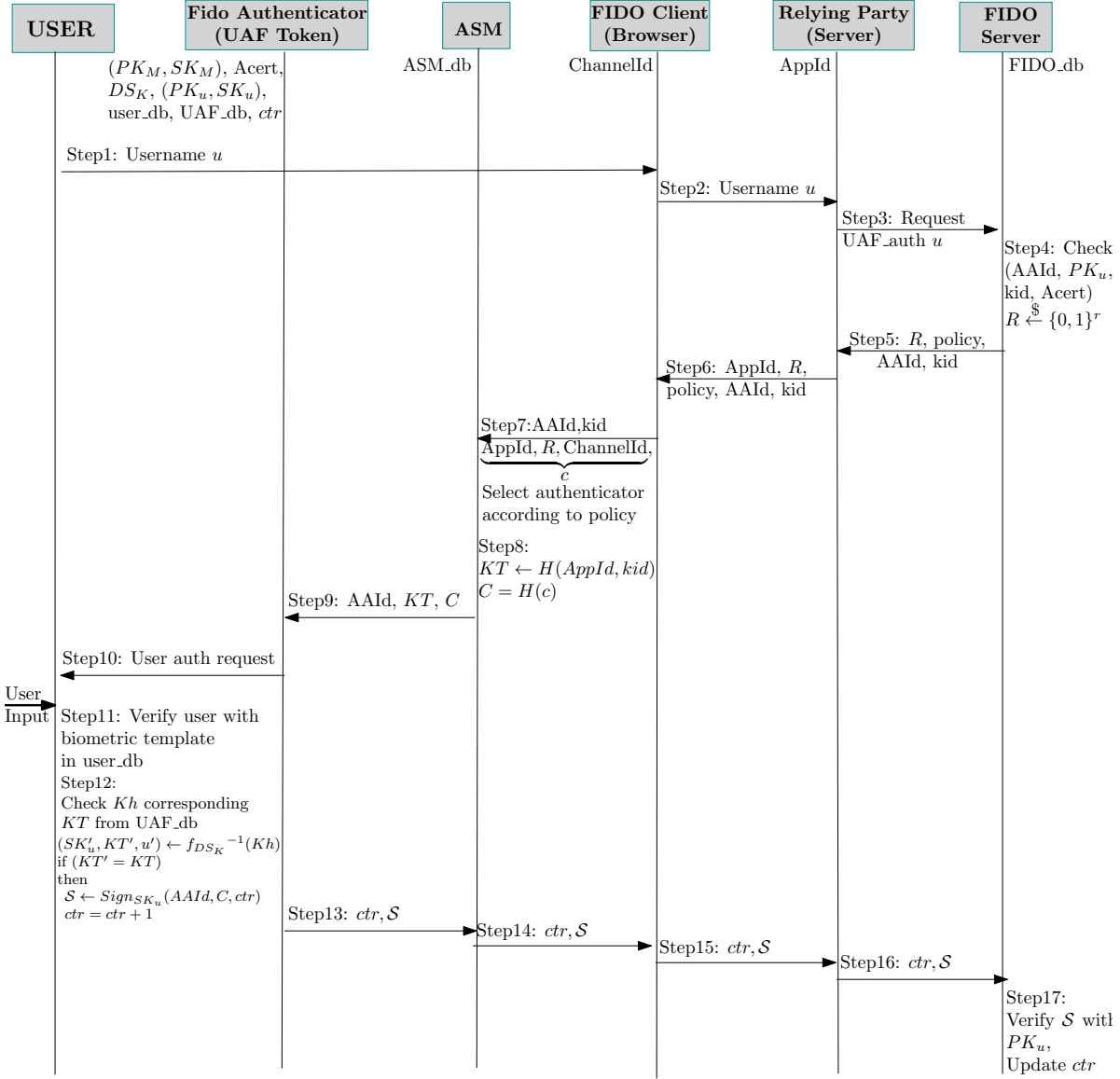


Figure 7.15: **UAF Authentication Phase.** The key DS_K , the asymmetric keypair (PK_M, SK_M) with attestation certificate $Acert$ which includes PK_M and counter ‘ctr’ are provided at the manufacturing phase. The FIDO server database $FIDO_db$ contains all the required values generated at the registration phase corresponding to a registered username. R is a r bit random number generated by the server. $AppId$ is the URL of the server. $ChannelId$ is the TLS $ChannelId$. The value Kh is the Keyhandle. ‘ctr’ is the counter used to record the number of authentication requests successfully satisfied by the UAF token.

also present the detailed analysis of the U2F protocol including the existing security analysis. The side channel attacks can be applied to any such secure device that uses a single fixed key either for encryption or signature and hence the UAF protocol is also an easy target of this side channel attack. We provide a brief overview of the UAF protocol showing how the similar solution for U2F can be applied to UAF as well.

Incorporation of different cryptographic primitives in a protocol makes the security proof challenging. Therefore, to provide the proof for overall security of the U2F protocol needs more expertise. Our current approach was to examine the protocol component-wise. Even though, each component is secure, it does not imply that the overall protocol is secure. The approach of universal composability could be a way towards providing the overall security of the protocol which is a good target for future work.

The open problem for the security of U2F protocol is to find a signature scheme secure against side channel attack. This is an interesting problem which can help to secure many implementations that leak private key through side channel. Working on the countermeasure to overcome the limitations of U2F as explained in Section 7.8, is another interesting problem to explore.

Chapter 8

Conclusion

In this chapter, we summarize our results presented in this thesis and give some possible directions for future research.

8.1 Summary

In this thesis, we focused on the design and analysis of password-based authentication system. We design a password hashing scheme *Rig* and present a Time-Memory Tradeoff (TMTO) analysis technique for password hashing. We present a work on the improvement of current password database breach detection technique, *Honeywords*. The second-factor solution Universal 2nd Factor framework (U2F) is a significant contribution which lacks thorough security analysis on the proposal. We present our analysis on this U2F protocol.

1. We studied the existing significant designs for password hashing, namely bcrypt, pbkdf2, Scrypt and design our scheme Rig following the requirements mentioned at the page of Password Hashing Competition. Existing analysis on *Rig* shows it is a good design considering the current state-of-the-art solutions. It prevents all known attacks and provides good defence against TMTO attack. Considering the online and offline attacks on password hashing, we propose a cryptographic module based implementation for secure implementation of password hashing. It is a significant contribution considering the recommendation of NIST which says all crypto-primitive should be implemented inside crypto-module to provide relevant security.
2. We next reviewed the TMTO analysis of PHS and presented a simple graph-based approach to measure the TMTO for three designs, namely, Argon2-Version 1.2.1 (the PHC winner), Catena-Version 3.2 and Rig-Version 2 by performing preliminary analysis with various parameters and TMTO options. Specifically, the technique is applicable to the designs which can be represented as a DAG. The proposed DAG traversal technique is flexible enough to be applied (may require minor simplification) on various other complex cryptographic designs for which making a mathematical model is significantly difficult. Our TMTO analysis shows that Argon2i does not provide the

required memory hardness. We also present the performance graph capturing the execution time vs. memory for all the memory-hard algorithms benchmarked. It is clear that Catena-Butterfly is the slowest and take significant amount of time in hashing passwords with moderate to large amounts of memory. The performance of Catena is unlikely to significantly improve even with native assembly implementation. Argon2i and Rig-Version 2 provide good performance in a wide range of use cases, though Argon2i is slower than Rig. The attack of [98] on Argon2i shows reduction over claimed TMTO defense which is also visible in our analysis. In brief, our experimental results show that simple DAGs are efficient in terms of performance but are not memory hard. Use of complex DAG provides expected security against time-memory tradeoff but shows poor efficiency. Combining two simple graphs in the representation of a DAG achieves both security and efficiency.

3. We then switch to the analysis of the work *honeyword* which is a password database breach detection technique. Chapters 6 includes our contribution on the analysis and improvement of the existing proposals on honeywords. We propose new honeyword generation methods that are perfectly flat. We conducted an experiment to show the flatness of our evolving-password model technique. To prevent false detection, in cases where legitimate user unintentionally enters a honeyword, we implement Levenshtein distance to maintain minimum required distance (3 for our experiment) between password and honeywords. We propose a new attack model called MSII attack and show that the ‘Paired Distance Protocol’ is completely broken in this MSII attack model. The detailed analysis of existing honeyword techniques and their comparison with our proposed techniques is also provided.
4. We then discussed our analysis on the recently proposed second factor authentication, U2F protocol in Chapters 7. It is a protocol proposed by FIDO alliance with the aim 1) ease of use, 2) privacy and security, and 3) standardization. Therefore, a third party analysis is of utmost importance to verify the security claims. We provide our analysis which shows U2F is not a strong solution with current specification. We show the ease in compromising the secrets of U2F token through side channel attack. This was one of the crucial works done by us which shows how user privacy can be compromised when all existing attacks are not considered while designing a solution. We also provide the countermeasure to mitigate the attacks on U2F. We also observe that similar problem exists in the proposal of another authentication technique called UAF. We conclude our contribution considering both the U2F and UAF protocols. Hence, designing a secure solution is ever challenging.

8.2 Future Work

In this section, we attempt to identify future research directions in which we can try applying the topics discussed in this thesis.

1. Recently in CRYPTO'16, asymptotic attacks on the memory hardness of password hashing designs have been proposed. Therefore, designing a memory hard design looks a challenging direction to work on.
2. We have proposed a general technique to secure the implementation of password hashing algorithm with the use of a cryptographic module. A natural research extension to this work would be to apply the proposed approach to the implementation of password hashing. It would help to analyse the effect of this secure implementation of password hashing to prevent both online and offline attacks for password-based authentication.
3. It would be interesting to apply our proposed time-memory tradeoff analysis technique to evaluate other complex cryptographic designs similar to password hashing schemes where proposing a standard mathematical/theoretical model for analysis is significantly difficult. One possible extension of this work can be to apply the proposed analysis technique on other password hashing designs as well.

The algorithm Rig is represented by a DAG which is a combination of two simple graphs namely, straight graph and bit-reversal graph. The variant of Catena, Catena-DBG, is represented with double-butterfly graph which is memory-hard. However, considering the performance with respect to time vs memory consumption, Catena-DBG is very slow in comparison with Argon2i and Rig. Even Argon2i shows weak memory hardness and follows a complex DAG representation. Therefore, our experimental results show that the combination of simple graphs for the design of a password hashing scheme is more efficient in terms of performance and time-memory tradeoff defense than the algorithm with complex graphical representation. Hence, to provide a general technique to combine different simple graphs for the efficient design (both in terms of TMTO defense and performance) of a password hashing scheme can be another interesting problem to explore.

4. It is challenging to prevent false alarm for the detection of password database breach. We have presented the use of Levenshtein distance for the same. However, it would be an interesting problem to work on a new data structure to prevent false alarm as any distance measure (e.g. Levenshtein distance) may be typo resistant, but may not be confusion resistant. Therefore, legitimate user can provide honeyword through confusion. It would be a good future extension of the present work.

To measure the typo-safety, one possible approach that is commonly followed is by collecting data conducting a real life survey. This approach requires designing relevant survey questions, carefully choosing the target audience and then analyzing the results of the sample surveys. We consider it as an extension of our current work and a good target for the future work.

5. The open problem for the security of U2F protocol is to find a signature scheme secure against side channel attack. This is an interesting problem which can help to secure many implementations that leak private key through side channel attack. Working on

the countermeasure to overcome the limitations of U2F as explained in Section 7.8, is another interesting problem to explore.

Incorporation of different cryptographic primitives in a protocol makes the security proof challenging. Therefore, to provide the proof for overall security of the U2F protocol needs more expertise. Our current approach was to examine the protocol component-wise. Eventhough, each component is secure, it does not imply that the overall protocol is secure. The approach of universal composability could be a way towards providing the overall security of the protocol which is a good target as a future work.

Bibliography

- [1] Crypt(3), freebsd library functions manual. <https://www.freebsd.org/cgi/man.cgi?query=crypt&apropos=0&sektion=3&manpath=FreeBSD+9.1-stable&arch=default&format=ascii>.
- [2] FIDO Alliance. <https://fidoalliance.org/about/what-is-fido/>.
- [3] Most common passwords list. Available at: <http://www.passwordrandom.com/most-popular-passwords/page/89>.
- [4] RSA. <https://www.rsa.com/en-us/products/rsa-securid-suite/rsa-securid-access/securid-hardware-tokens>.
- [5] FIDO UAF Protocol Specification. FIDO Alliance Implementation Draft, 02 February 2017. Available at: <https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf>.
- [6] Universal 2nd factor (u2f) overview. FIDO Alliance Proposed Standard, 14 May 2015. Available at: <https://fidoalliance.org/specs/fido-u2f-overview-ps-20150514.pdf>.
- [7] Universal 2nd factor (u2f) overview. FIDO Alliance Proposed Standard, 15 September 2016. Available at: <https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf>.
- [8] Fips pub 140-2, security requirements for cryptographic modules. Federal Information Processing Standards Publication (Supercedes FIPS PUB 140-1, 1994 January 11), 2001. <http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf>.
- [9] *30th IEEE Symposium on Security and Privacy (S&P 2009)*, 17-20 May 2009, Oakland, California, USA (2009), IEEE Computer Society.
- [10] Totp: Time-based one-time password algorithm. rfc: 6238, 2011. [Online; accessed 24-May-2017].
- [11] U.S. Department of Labor Bureau of Labor Statistics, CPI Inflation Calculator, 2012. https://www.bls.gov/data/inflation_calculator.htm.

- [12] XE Inc. Universal Currency Converter, 2012. <http://www.xe.com/currencyconverter/>.
- [13] Entropy: How Password Strength Is Measured , 2013. <http://akgupta.ca/blog/2013/06/06/entropy-how-password-strength-is-measured/>.
- [14] Security of key derivation functions, 2013. <http://lyra-kdf.net/Presentation.pdf>.
- [15] Comments on phc mailing list, 2014. <http://article.gmane.org/gmane.comp.security.phc/2172>.
- [16] Password Hashing Competition (PHC), 2014. <https://password-hashing.net/index.html>.
- [17] Reference implementation of catena, a memory-consuming password scrambler, 2015. <https://github.com/medsec/catena> (75 commits).
- [18] A flexible implementation of colin percival's scrypt, 2016. <https://github.com/floodyberry/scrypt-jane> (58 commits).
- [19] Memory-hard scheme argon2, 2016. <https://github.com/khovratovich/Argon2> (162 commits).
- [20] Reference and optimized implementations of rig, a simple, secure and flexible design for password hashing, 2016. <https://github.com/arpanj/Rig>.
- [21] NIST Special Publication 800-63B, Digital Identity Guidelines, 2017. [Online; accessed 24-May-2017].
- [22] Symantec Internet Security Threat Report trends for 2010. Technical report, Symantec, April 2011. Available at: http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.
- [23] Internet Security Threat Report 2013. Technical report, Symantec, April 2013. Available at: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.
- [24] ABDALLA, M., AND BELLARE, M. Increasing the lifetime of a key: A comparative analysis of the security of re-keying techniques. In *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings* (2000), T. Okamoto, Ed., vol. 1976 of *Lecture Notes in Computer Science*, Springer, pp. 546–559.

- [25] AGRAWAL, D., RAO, J. R., ROHATGI, P., AND SCHRAMM, K. Templates as master keys. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings* (2005), J. R. Rao and B. Sunar, Eds., vol. 3659 of *Lecture Notes in Computer Science*, Springer, pp. 15–29.
- [26] AKSHIMA, CHANG, D., GOEL, A., MISHRA, S., AND SANADHYA, S. K. Generation of secure and reliable honeywords, preventing false detection. *Submitted to Transactions on Dependable and Secure Computing* (2017).
- [27] ALMEIDA, L. C., ANDRADE, E. R., BARRETO, P. S. L. M., AND JR., M. A. S. Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs. *IACR Cryptology ePrint Archive 2014* (2014), 30.
- [28] ALVAREZ, R. CENTRIFUGE - A password hashing algorithm. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Centrifuge-v0.pdf>.
- [29] ALWEN, J., AND BLOCKI, J. Efficiently computing data-independent memory-hard functions. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II* (2016), M. Robshaw and J. Katz, Eds., vol. 9815 of *Lecture Notes in Computer Science*, Springer, pp. 241–271.
- [30] ALWEN, J., GAZI, P., KAMATH, C., KLEIN, K., OSANG, G., PIETRZAK, K., REYZIN, L., ROLÍNEK, M., AND RYBÁR, M. On the memory-hardness of data-independent password-hashing functions. *IACR Cryptology ePrint Archive 2016* (2016), 783.
- [31] ALWEN, J., AND SERBINENKO, V. High parallel complexity graphs and memory-hard functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015* (2015), R. A. Servedio and R. Rubinfeld, Eds., ACM, pp. 595–603.
- [32] ATTRIDGE, J. An Overview of Hardware Security Modules. SANS Institute InfoSec Reading Room, 2002.
- [33] AUMASSON, J., GUO, J., KNELLWOLF, S., MATUSIEWICZ, K., AND MEIER, W. Differential and invertibility properties of BLAKE. In *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers* (2010), pp. 318–332.
- [34] AUMASSON, J. P. Password hashing: the future is now, 2013. <https://media.blackhat.com/us-13/US-13-Aumasson-Password-Hashing-the-Future-is-Now-WP.pdf>.

- [35] AUMASSON, J. P., NEVES, S., WILCOX-O’HEARN, Z., AND WINNERLEIN, C. BLAKE2: Simpler, Smaller, Fast as MD5. In *ACNS (2013)*, M. J. J. Jr., M. E. Lo-casto, P. Mohassel, and R. Safavi-Naini, Eds., vol. 7954 of *Lecture Notes in Computer Science*, Springer, pp. 119–135.
- [36] BARKER, E. Nist special publication 800-57 part 1, revision 4, recommendation for key management, part 1: General, 2016. <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf>.
- [37] BELLA, G., AND PAULSON, L. C. Mechanical proofs about a non-repudiation protocol. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings (2001)*, R. J. Boulton and P. B. Jackson, Eds., vol. 2152 of *Lecture Notes in Computer Science*, Springer, pp. 91–104.
- [38] BELLOVIN, S. M., AND MERRITT, M. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 4-6, 1992 (1992)*, IEEE Computer Society, pp. 72–84.
- [39] BERTONI, G., DAEMEN, J., PEETERS, M., AND ASSCHE, G. V. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers (2011)*, Springer.
- [40] BERTONI, G., DAEMEN, J., PEETERS, M., AND ASSCHE, G. V. Keccak. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings (2013)*, T. Johansson and P. Q. Nguyen, Eds., vol. 7881 of *Lecture Notes in Computer Science*, Springer, pp. 313–314.
- [41] BIRYUKOV, A., DINU, D., AND KHOVRATOVICH, D. Argon2 for password hashing and cryptocurrencies, 2016. <https://www.ietf.org/proceedings/96/slides/slides-96-cfrg-1.pdf>.
- [42] BIRYUKOV, A., DINU, D., AND KHOVRATOVICH, D. Argon2: the memory-hard function for password hashing and other applications , version 1.3, 2016. <https://www.cryptolux.org/images/0/0d/Argon2.pdf>.
- [43] BIRYUKOV, A., DINU, D., KHOVRATOVICH, D., AND JOSEFSSON, S. The memory-hard Argon2 password hash and proof-of-work function . IRTF Crypto Forum Research Group Active Internet-Draft (cfrg RG): draft-irtf-cfrg-argon2-00, 2016. <https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/>.

- [44] BIRYUKOV, A., AND KHOVRATOVICH, D. ARGON v1: Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Argon-v1.pdf>.
- [45] BIRYUKOV, A., AND KHOVRATOVICH, D. Tradeoff cryptanalysis of memory-hard functions. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II* (2015), T. Iwata and J. H. Cheon, Eds., vol. 9453 of *Lecture Notes in Computer Science*, Springer, pp. 633–657.
- [46] BIRYUKOV, A., AND KHOVRATOVICH, D. Tradeoff cryptanalysis of memory-hard functions. *IACR Cryptology ePrint Archive 2015* (2015), 227.
- [47] BLOCKI, J., KOMANDURI, S., CRANOR, L. F., AND DATTA, A. Spaced repetition and mnemonics enable recall of multiple strong passwords. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015).
- [48] BOJINOV, H., BURSZEIN, E., BOYEN, X., AND BONEH, D. Kamouflage: Loss-resistant password management. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings* (2010), D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., vol. 6345 of *Lecture Notes in Computer Science*, Springer, pp. 286–302.
- [49] BONNEAU, J. Guessing human-chosen secrets. *Technical Report, University of Cambridge, Computer Laboratory 819* (May 2012).
- [50] BONNEAU, J., AND SCHECHTER, S. E. Towards reliable storage of 56-bit secrets in human memory. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (2014), pp. 607–623.
- [51] BRADLEY, W. F. Superconcentration on a pair of butterflies. *arXiv preprint arXiv:1401.7263* (2014).
- [52] BURNETT, M. The Pathetic Reality of Adobe Password Hints, 2013. Available at: <https://xato.net/the-pathetic-reality-of-adobe-password-hints-bb40fd92220f#.u8deumwif>.
- [53] BURR, W. E., DODSON, D. F., NEWTON, E. M., PERLNER, R. A., POLK, W. T., GUPTA, S., AND NABBUS, E. A. Nist special publication 800-63-2, electronic authentication guideline, part 1: General, August 2013. <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf>.
- [54] CAPPOS, J. PolyPassHash: Protecting Passwords In The Event Of A Password File Disclosure. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/PolyPassHash-v0.pdf>.

- [55] CARVALHO, C. The gap between processor and memory speeds. *Proc. of IEEE International Conference on Control and Automation*, 2002.
- [56] CHAKRABORTY, N., AND MONDAL, S. Few notes towards making honeyword system more secure and usable. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015* (2015), O. B. Makarevich, R. Poet, A. Elçi, M. S. Gaur, M. A. Orgun, L. K. Babenko, M. S. Ferdous, A. T. S. Ho, V. Laxmi, and J. Pieprzyk, Eds., ACM.
- [57] CHAKRABORTY, N., AND MONDAL, S. A new storage optimized honeyword generation approach for enhancing security and usability. *CoRR abs/1509.06094* (2015).
- [58] CHANG, D., JATI, A., MISHRA, S., AND SANADHYA, S. K. Cryptographic module based approach for password hashing schemes. In *Mjølsnes [125]*, pp. 39–57.
- [59] CHANG, D., JATI, A., MISHRA, S., AND SANADHYA, S. K. Rig: A simple, secure and flexible design for password hashing. In *Information Security and Cryptology - 10th International Conference, Inscrypt 2014, Beijing, China, December 13-15, 2014, Revised Selected Papers* (2014), D. Lin, M. Yung, and J. Zhou, Eds., vol. 8957 of *Lecture Notes in Computer Science*, Springer, pp. 361–381.
- [60] CHANG, D., JATI, A., MISHRA, S., AND SANADHYA, S. K. Cryptanalytic time-memory tradeoff for password hashing schemes. *Submitted to International Journal of Information Security* (2016).
- [61] CHANG, D., MISHRA, S., SANADHYA, S. K., AND SINGH, A. P. On making u2f protocol leakage-resilient via re-keying. *Submitted to ACM Transactions on Information and System Security* (2017).
- [62] CHARI, S., RAO, J. R., AND ROHATGI, P. Template attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers* (2002), B. S. K. Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 13–28.
- [63] CLARKSON, E., CLAWSON, J., LYONS, K., AND STARNER, T. An empirical study of typing rates on mini-qwerty keyboards. In *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005* (2005), G. C. van der Veer and C. Gale, Eds., ACM, pp. 1288–1291.
- [64] COHEN, F. The use of deception techniques: Honeypots and decoys. http://all.net/journal/deception/Deception_Techniques_.pdf.
- [65] CONTI, M., DRAGONI, N., AND LESYK, V. A survey of man in the middle attacks. *IEEE Communications Surveys and Tutorials* 18, 3 (2016).

- [66] CONTRIBUTORS, W. 2012 LinkedIn hack. Wikipedia, The Free Encyclopedia, Date retrieved: 29 May 2016. Available at: https://en.wikipedia.org/w/index.php?title=2012_LinkedIn_hack&oldid=722095159.
- [67] CONTRIBUTORS, W. Ashley Madison data breach. Wikipedia, The Free Encyclopedia, Date retrieved: 29 May 2016. Available at: https://en.wikipedia.org/w/index.php?title=Ashley_Madison_data_breach&oldid=721001290.
- [68] COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.
- [69] COX, B. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/TwoCats-v0.pdf>.
- [70] CRISTOFARO, E. D., DU, H., FREUDIGER, J., AND NORCIE, G. Two-Factor or not Two-Factor? A Comparative Usability Study of Two-Factor Authentication. *CoRR abs/1309.5344* (2013).
- [71] D., B., AND R., H. Transport Layer Security (TLS) Channel IDs, v01 (IETF Internet-Draft) , 01, 2013. Available at: <http://tools.ietf.org/html/draft-balfanz-tlschannelid>.
- [72] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
- [73] DAEMEN, J., AND RIJMEN, V. A new MAC construction ALRED and a specific instance ALPHA-MAC. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers* (2005), pp. 1–17.
- [74] DES. Data encryption standard. In *FIPS PUB 46, Federal Information Processing Standards Publication* (1977), pp. 2–46.
- [75] DIETZ, M., CZESKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012* (2012), T. Kohno, Ed., USENIX Association, pp. 317–331.
- [76] DÜRMUTH, M., GÜNEYSU, T., KASPER, M., PAAR, C., YALÇIN, T., AND ZIMMERMANN, R. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In *ESORICS* (2012), S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 716–733.
- [77] DÜRMUTH, M., AND ZIMMERMANN, R. AntCrypt. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/AntCryptv0.pdf>.

- [78] ELLEITHY, K. M., BLAGOVIC, D., CHENG, W., AND SIDELEAU, P. Denial of Service Attack Techniques: Analysis, Implementation and Comparison. Systemics, Cybernetics and Informatics, Volume 3 - Number 1, 2006.
- [79] ENRIGHT, B. Omega Crypt (ocrypt). Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/OmegaCryptv0.pdf>.
- [80] ERGULER, I. Achieving flatness: Selecting the honeywords from existing user passwords. *IEEE Trans. Dependable Sec. Comput.* 13, 2 (2016), 284–295.
- [81] FAN, J., GUO, X., MULDER, E. D., SCHAUMONT, P., PRENEEL, B., AND VERBAUWHEDE, I. State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In *HOST 2010, Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 13-14 June 2010, Anaheim Convention Center, California, USA* (2010), J. Plusquellic and K. Mai, Eds., IEEE Computer Society, pp. 76–87.
- [82] FELIX, J., AND HAUCK, C. System security: A hacker’s perspective, September 1987.
- [83] FLORÊNCIO, D. A. F., AND HERLEY, C. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007* (2007), C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, Eds., ACM, pp. 657–666.
- [84] FLORÊNCIO, D. A. F., AND HERLEY, C. Where do security policies come from? In *Proceedings of the Sixth Symposium on Usable Privacy and Security, SOUPS 2010, Redmond, Washington, USA, July 14-16, 2010* (2010).
- [85] FORLER, C., LIST, E., LUCKS, S., AND WENZEL, J. Overview of the candidates for the password hashing competition - and their resistance against garbage-collector attacks. In Mjølshes [125], pp. 3–18.
- [86] FORLER, C., LUCKS, S., AND WENZEL, J. The Catena Password Scrambler. Submission to Password Hashing Competition (PHC), 2014.
- [87] FORLER, C., LUCKS, S., AND WENZEL, J. The Catena Password-Scrambling Framework. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Catena-v2.pdf>.
- [88] FORLER, C., LUCKS, S., AND WENZEL, J. The Catena Password-Scrambling Framework . Submission to Password Hashing Competition (PHC), 2015.
- [89] FRANCOIS-XAVIER STANDAERT. Introduction to Side-Channel Attacks. Available at: <https://perso.uclouvain.be/fstandae/PUBLIS/42.pdf>.

- [90] FRANKE, D. The EARWORM Password Hashing Algorithm. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/EARWORM-v0.pdf>.
- [91] GARSIEL, T., AND IRISH, P. How browsers work: Behind the scenes of modern web browsers, 2011. [Online; accessed 4-June-2017].
- [92] GELERNTER, N., KALMA, S., MAGNEZI, B., AND PORCILAN, H. The Password Reset MitM Attack. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), IEEE Computer Society, pp. 251–267.
- [93] GILLELAND, M. Levenshtein Distance, in Three Flavors. Available at: <http://people.cs.pitt.edu/~kirk/cs1501/assignments/editdistance/Levenshtein%20Distance.htm>.
- [94] GILMORE, J. Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design. *Electronic Frontier Foundation* (1998).
- [95] GOSNEY, J. M. The Pufferfish Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Pufferfish-v0.pdf>.
- [96] GRAY, J., AND SHENOY, P. Rules of Thumb in Data Engineering, MS-TR-99-100, Microsoft Research, Advanced Technology Division. Technical Report, December 1999, Revised March 2000.
- [97] HELLMAN, M. E. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on* 26, 4 (1980), 401–406.
- [98] HENRY CORRIGAN-GIBBS AND DAN BONEH AND STUART SCHECHTER. Balloon Hashing: Provably Space-Hard Hash Functions with Data-Independent Access Patterns . *IACR Cryptology ePrint Archive* (2016).
- [99] HERLEY, C., AND VAN OORSCHOT, P. C. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy* 10, 1 (2012), 28–36.
- [100] HUNT, T. Observations and thoughts on the LinkedIn data breach, 2015. Available at: <https://www.troyhunt.com/observations-and-thoughts-on-the-linkedin-data-breach/>.
- [101] JOSEPH BONNEAU. Guessing human-chosen secrets. *Technical Report, University of Cambridge* (May, 2012).
- [102] JR, M. A. S., ALMEIDA, L. C., ANDRADE, E. R., DOS SANTOS, P. C. F., AND BARRETO, P. S. L. M. The Lyra2 reference guide. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Lyra2-v1.pdf>.

- [103] JUELS, A., AND RIVEST, R. L. Honeywords: making password-cracking detectable. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), A. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM.
- [104] KAPUN, E. Yarn password hashing function. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Yarnv2.pdf>.
- [105] KARAPANOS, N., AND CAPKUN, S. On the effective prevention of TLS man-in-the-middle attacks in web applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. (2014), K. Fu and J. Jung, Eds., USENIX Association, pp. 671–686.
- [106] KASABOV, A., VAN KERKWIJK, J., SMEETS, M., AND VAN VEEN, M. Distributed GPU Password Cracking. *Research Project, Universiteit Van Amsterdam* (May, 2011).
- [107] KATZ, J., AND LINDELL, Y. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.
- [108] KEHOE, B. P. Zen and the Art of the Internet, A Beginner’s Guide to the Internet, First Edition, January 1992. [Online; accessed 2-July-2017].
- [109] KELSEY, J., SCHNEIER, B., HALL, C., AND WAGNER, D. Secure applications of low-entropy keys. In *Information Security, First International Workshop, ISW '97, Tatsunokuchi, Japan, September 17-19, 1997, Proceedings* (1997), E. Okamoto, G. I. Davida, and M. Mambo, Eds., vol. 1396 of *Lecture Notes in Computer Science*, Springer, pp. 121–134.
- [110] KHANDELWAL, S. Hacking any eBay Account in just 1 Minute, 2014. Available at: <http://thehackernews.com/2014/09/hacking-ebay-accounts.html>.
- [111] KOCHER, P. Leak resistant cryptographic indexed key update. US Patent 6539092.
- [112] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 104–113.
- [113] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings* (1999), M. J. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 388–397.
- [114] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. RFC 2104, HMAC-MD5: Keyed-MD5 for Message Authentication, 1996. [Online; accessed 22-July-2017].

- [115] LAB), C. S. D. I. T. Guide to malware incident prevention and handling. NIST Special Publication 800-83 Revision 1.
- [116] LAGARIAS, J. C. Pseudorandom numbers, 1993. Statistical Science.
- [117] LENGAUER, T., AND TARJAN, R. E. Upper and lower bounds on time-space tradeoffs. In *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA* (1979), pp. 262–277.
- [118] LENGAUER, T., AND TARJAN, R. E. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM* 29, 4 (1982), 1087–1130.
- [119] LEONG, P. H. W., AND THAM, C. UNIX password encryption considered insecure. In *Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991* (1991), USENIX Association, pp. 269–280.
- [120] LEOUTSARAKOS, N. What’s wrong with FIDO?, 2015. Available at: http://www.zeropasswords.com/pdfs/WHATisWRONG_FIDO.pdf.
- [121] MASLENNIKOV, M. Password Hashing Scheme MCS PHS. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/MCS_PHS-v2.pdf.
- [122] MAURY, F., AND BERGEM, M. A first glance at the U2F protocol. SSTIC2016, 2016. Available at: https://www.sstic.org/media/SSTIC2016/SSTIC-actes/a_first_glance_at_the_u2f_protocol/SSTIC2016-Article-a_first_glance_at_the_u2f_protocol-maury_bergem.pdf.
- [123] MENTENS, N., BATINA, L., PRENEEL, B., AND VERBAUWHEDE, I. Time-memory trade-off attack on FPGA platforms: UNIX password cracking. In *Reconfigurable Computing: Architectures and Applications, Second International Workshop, ARC 2006, Delft, The Netherlands, March 1-3, 2006, Revised Selected Papers* (2006), K. Bertels, J. M. P. Cardoso, and S. Vassiliadis, Eds., vol. 3985 of *Lecture Notes in Computer Science*, Springer, pp. 323–334.
- [124] MERKLE, R. Secrecy, authentication, and public key systems. *Technical Report*, 1979.
- [125] MJØLSNES, S. F., Ed. *Technology and Practice of Passwords - International Conference on Passwords, PASSWORDS’14, Trondheim, Norway, December 8-10, 2014, Revised Selected Papers* (2015), vol. 9393 of *Lecture Notes in Computer Science*, Springer.
- [126] MORRIS, R., AND THOMPSON, K. Password security: A case history, 1979. <http://cs-www.cs.yale.edu/homes/arvind/cs422/doc/unix-sec.pdf>.
- [127] MUBARAK, H. Lanarea DF. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Lanarea-v0.pdf>.

- [128] MULLINER, C., BORGAONKAR, R., STEWIN, P., AND SEIFERT, J. Sms-based one-time passwords: Attacks and defense - (short paper). In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings* (2013), K. Rieck, P. Stewin, and J. Seifert, Eds., vol. 7967 of *Lecture Notes in Computer Science*, Springer, pp. 150–159.
- [129] NARAYANAN, A., AND SHMATIKOV, V. De-anonymizing social networks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA* [9], pp. 173–187.
- [130] OECHSLIN, P. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (2003), D. Boneh, Ed., vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 617–630.
- [131] O’GORMAN, L. Comparing passwords, tokens, and biometrics for user authentication. In *Proceedings of the IEEE 91*, 4 (2003), 2021–2040.
- [132] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. In *BSDCon* (2009). http://www.bsdcn.org/2009/schedule/attachments/87_script.pdf.
- [133] PESLYAK, A. yescript - a password hashing competition submission. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/yescript-v0.pdf>.
- [134] PINTÉR, K. Gambit - A sponge based, memory hard key derivation function. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Gambit-v1.pdf>.
- [135] PORIN, T. The MAKWA Password Hashing Function. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Makwa-v0.pdf>.
- [136] PRENEEL, B. The first 30 years of cryptographic hash functions and the NIST SHA-3 competition. In *Topics in Cryptology - CT-RSA 2010, The Cryptographers’ Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings* (2010), J. Pieprzyk, Ed., vol. 5985 of *Lecture Notes in Computer Science*, Springer, pp. 1–14.
- [137] PROVOS, N., AND MAZIÈRES, D. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track* (1999), USENIX, pp. 81–91.

- [138] QUISQUATER, J., AND SAMYDE, D. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings* (2001), I. Attali and T. P. Jensen, Eds., vol. 2140 of *Lecture Notes in Computer Science*, Springer, pp. 200–210.
- [139] RAMZAN, Z. Phishing attacks and countermeasures. In *Handbook of Information and Communication Security*, P. P. Stavroulakis and M. Stamp, Eds. Springer, 2010, pp. 433–448.
- [140] RAY, M., AND ZAVERUCHA, G. What Microsoft Would like from the Password Hashing Competition. A presentation at Passwords14, 2014. <http://crypto.systema.ru/PHC/Microsoft-PHC-Passwords14.pdf>.
- [141] SALTZER, J. H. Protection and the control of information sharing in MULTICS. In *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973* (1973), H. Schorr, A. J. Perlis, P. Weiner, and W. D. Frazer, Eds., ACM.
- [142] SAVAGE, J. E. Space-Time Tradeoffs, Models of Computation Exploring the Power of Computing . http://cs.brown.edu/~jes/book/pdfs/ModelsOfComputation_Chapter10.pdf.
- [143] SCH, T. Tortuga - Password hashing based on the Turtle algorithm. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/Tortuga-v0.pdf>.
- [144] SCHALLER, R. R. Moore’s law: past, present, and future. *IEEE Spectrum* June 1997.
- [145] SCHNEIER, B. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop, Cambridge, UK, December 9-11, 1993, Proceedings* (1993), R. J. Anderson, Ed., vol. 809 of *Lecture Notes in Computer Science*, Springer, pp. 191–204.
- [146] SCHNEIER, B. Cryptographic blunders revealed by adobe’s password leak. Schneier on Security, 2013. Available at: https://www.schneier.com/blog/archives/2013/11/cryptographic_b.html.
- [147] SCHNEIER, B. Frequent Password Changes Is a Bad Security Idea. *Crypto-Gram* (August 5, 2016).
- [148] SCHNEIER, B. Choosing a Secure Password. *Crypto-Gram* (February 25, 2014).
- [149] SHANNON, C. E. Communication theory of secrecy systems. *Bell System Technical Journal*, Vol 28, pp. 656715 (October 1949).

- [150] SHANNON, C. E. Prediction and entropy of printed english. *Bell System Technical Journal* 30, 1 (1951), 50–64.
- [151] SHANNON, C. E. A mathematical theory of communication. *Mobile Computing and Communications Review* 5, 1 (2001), 3–55.
- [152] SPITZNER, L. Honeytokens: The other honeypot, 2003. <http://www.symantec.com/connect/articles/honeytokens-other-honeypot>.
- [153] SPRENGERS, M. GPU-based Password Cracking. *Master's thesis, Radboud University Nijmegen* (2011).
- [154] STALLINGS, W. *Cryptography and network security - principles and practice (3. ed.)*. Prentice Hall, 2003.
- [155] STEUBE, J. oclhashcat-plus - advanced password recovery password, Accessed December 15, 2015. <http://hashcat.net/oclhashcat/>.
- [156] TASEVSKI, P. Password Generation Strategy . <http://courses.cs.ut.ee/2011/security-seminar-spring/uploads/Main/pedrag-draft.pdf>.
- [157] TECHLICIOUS. Google Reveals the 10 Worst Password Ideas, 2013. <http://techland.time.com/2013/08/08/google-reveals-the-10-worst-password-ideas/>.
- [158] THOMAS, S. battcrypt (Blowfish All The Things). Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/battcrypt-v0.pdf>.
- [159] THOMAS, S. Parallel. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Parallel-v0.pdf>.
- [160] TURAN, M. S., BARKER, E., BURR, W., AND CHEN, L. NIST: Special Publication 800-132, Recommendation for Password-Based Key Derivation. Computer Security Division Information Technology Laboratory. <http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf>.
- [161] VUCKOVAC, R. schvrch. Submission to Password Hashing Competition (PHC), 2014. <https://password-hashing.net/submissions/specs/Schvrch-v0.pdf>.
- [162] WANG, D., ZHANG, Z., WANG, P., YAN, J., AND HUANG, X. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, pp. 1242–1254.
- [163] WEI, W., LI, J., CAO, L., OU, Y., AND CHEN, J. Effective detection of sophisticated online banking fraud on extremely imbalanced data. *World Wide Web* 16, 4 (2013).

- [164] WEIR, M., AGGARWAL, S., DE MEDEIROS, B., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, 17-20 May 2009, Oakland, California, USA [9].
- [165] WHITEHOUSE, K., KARLOF, C., AND CULLER, D. E. A practical evaluation of radio signal strength for ranging-based localization. *Mobile Computing and Communications Review* 11 (2007).
- [166] WIKIPEDIA. Advanced encryption standard — wikipedia, the free encyclopedia, 2017. [Online; accessed 8-June-2017].
- [167] WU, H. POMELO: A Password Hashing Algorithm. Submission to Password Hashing Competition (PHC), 2014. <https://passwordhashing.net/submissions/specs/POMELO-v1.pdf>.
- [168] ZHOU, Y., AND FENG, D. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive 2005* (2005), 388.