Design and Analysis of Cryptographic Algorithms for Authentication

Jakob Wenzel

Inauguraldissertation

zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

Oktober 2017

Gutachter: Prof. Dr. Stefan Lucks
Bauhaus-Universität Weimar

Prof. Dr. Gregor Leander
Ruhr-Universität Bochum

A child born today will grow up with no conception of privacy at all. They'll never know what it means to have a private moment to themselves, an unrecorded, unanalysed thought. And that's a problem because privacy matters, privacy is what allows us to determine who we are and who we want to be.

EDWARD JOSEPH SNOWDEN

Abstract

During the previous decades, the upcoming demand for security in the digital world, e.g., the Internet, lead to numerous groundbreaking research topics in the field of cryptography. This thesis focuses on the design and analysis of cryptographic primitives and schemes to be used for authentication of data and communication endpoints, i.e., users. It is structured into three parts, where we present the first freely scalable multi-block-length block-cipher-based compression function (Counter-bDM) in the first part. The presented design is accompanied by a thorough security analysis regarding its preimage and collision security.

The second and major part is devoted to password hashing. It is motivated by the large amount of leaked password during the last years and our discovery of side-channel attacks on `scrypt` – the first modern password scrambler that allowed to parameterize the amount of memory required to compute a password hash. After summarizing which properties we expect from a modern password scrambler, we (1) describe a cache-timing attack on `scrypt` based on its password-dependent memory-access pattern and (2) outline an additional attack vector – garbage-collector attacks – that exploits optimization which may disregard to overwrite the internally used memory. Based on our observations, we introduce Catena – the first memory-demanding password-scrambling framework that allows a password-independent memory-access pattern for resistance to the aforementioned attacks. Catena was submitted to the Password Hashing Competition (PHC) and, after two years of rigorous analysis, ended up as a finalist gaining special recognition for its agile framework approach and side-channel resistance. We provide six instances of Catena suitable for a variety of applications. We close the second part of this thesis with an overview of modern password scramblers regarding their functional, security, and general properties; supported by a brief analysis of their resistance to garbage-collector attacks.

The third part of this thesis is dedicated to the integrity (authenticity of data) of nonce-based authenticated encryption schemes (NAE). We introduce the so-called j-IV-Collision Attack, allowing to obtain an upper bound for an adversary that is provided with a first successful forgery and tries to efficiently compute j additional forgeries for a particular NAE scheme (in short: reforgeability). Additionally, we introduce the corresponding security notion j-INT-CTXT and provide a comparative analysis (regarding j-INT-CTXT security) of the third-round submission to the CAESAR competition and the four classical and widely used NAE schemes CWC, CCM, EAX, and GCM.

Zusammenfassung

Die fortschreitende Digitalisierung in den letzten Jahrzehnten hat dazu geführt, dass sich das Forschungsfeld der Kryptographie bedeutsam weiterentwickelt hat. Diese, im Wesentlichen aus drei Teilen bestehende Dissertation, widmet sich dem Design und der Analyse von kryptographischen Primitiven und Modi zur Authentifizierung von Daten und Kommunikationspartnern.

Der erste Teil beschäftigt sich dabei mit blockchiffrenbasierten Kompressionsfunktionen, die in ressourcenbeschränkten Anwendungsbereichen eine wichtige Rolle spielen. Im Rahmen dieser Arbeit präsentieren wir die erste frei skalierbare und sichere blockchiffrenbasierte Kompressionsfunktion Counter-*b*DM und erweitern somit flexibel die erreichbare Sicherheit solcher Konstruktionen.

Der zweite Teil und wichtigste Teil dieser Dissertation widmet sich Passwort-Hashing-Verfahren. Zum einen ist dieser motiviert durch die große Anzahl von Angriffen auf Passwortdatenbanken großer Internet-Unternehmen. Zum anderen bot die Password Hashing Competition (PHC) die Möglichkeit, unter Aufmerksamkeit der Expertengemeinschaft die Sicherheit bestehender Verfahren zu hinterfragen, sowie neue sichere Verfahren zu entwerfen. Im Rahmen des zweiten Teils entwarfen wir Anforderungen an moderne Passwort-Hashing-Verfahren und beschreiben drei Arten von Seitenkanal-Angriffen (Cache-Timing-, Weak Garbage-Collector- und Garbage-Collector-Angriffe) auf `scrypt` – das erste moderne Password-Hashing-Verfahren welches erlaubte, den benötigten Speicheraufwand zur Berechnung eines Passworthashes frei zu wählen.

Basierend auf unseren Beobachtungen und Angriffen, stellen wir das erste moderne Password-Hashing-Framework Catena vor, welches für gewählte Instanzen passwortunabhängige Speicherzugriffe und somit Sicherheit gegen oben genannte Angriffe garantiert. Catena erlangte im Rahmen des PHC-Wettbewerbs besondere Anerkennung für seine Agilität und Resistenz gegen Seitenkanal-Angriffe. Wir präsentieren sechs Instanzen des Frameworks, welche für eine Vielzahl von Anwendungen geeignet sind. Abgerundet wird der zweite Teil dieser Arbeit mit einem vergleichenden Überblick von modernen Passwort-Hashing-Verfahren hinsichtlich ihrer funktionalen, sicherheitstechnischen und allgemeinen Eigenschaften. Dieser Vergleich wird unterstützt durch eine kurze Analyse bezüglich ihrer Resistenz gegen (Weak) Garbage-Collector-Angriffe.

Der dritte teil dieser Arbeit widmet sich der Integrität von Daten, genauer, der Sicherheit sogenannter Nonce-basierten authentisierten Verschlüsselungsverfahren (NAE-Verfahren), welche ebenso wie Passwort-Hashing-Verfahren in der heutigen Sicherheitsinfrastruktur des Internets eine wichtige Rolle spielen. Während Standard-Definitionen keine Sicherheit nach dem Fund einer ersten erfolgreich gefälschten Nachricht betrachten, erweitern wir die Sicherheitsanforderungen dahingehend wie schwer es ist, weitere Fälschungen zu ermitteln. Wir abstrahieren die Funktionsweise von NAE-Verfahren in Klassen, analysieren diese systematisch und klassifizieren die Dritt-Runden-Kandidaten des CAESAR-Wettbewerbs, sowie vier weit verbreitete NAE-Verfahren CWC, CCM, EAX und GCM.

Acknowledgements

Many people have helped me to realize this thesis. First of all, I would like to thank my supervisor Prof. Stefan Lucks, who gave me the opportunity to work on many interesting and exciting topics. His support and suggestions of research topics and the possibility to discuss all ideas I had in mind, paved the way to this thesis, especially to the core of this thesis that regards password hashing. I would like to thank Prof. Gregor Leander for agreeing to review my thesis as a second supervisor.

Special thanks goes to Eik List, Christian Forler, and Ewan Fleischmann for their support, willingness to discuss, and contributions to that thesis. They made writing scientific papers together an enjoyable job and were always be on the spot with supporting me and my work. I am deeply grateful that Eik sacrificed so much of his time for supporting me and I really appreciate his comments on my thesis text.

I would also like to thank Sascha Schmidt and Heinrich Schilling, who made important contributions to the core part of this thesis, and all of my co-workers from the cryptographic community for their fruitful discussions and contributions: Farzaneh Abed, Ewan Fleischmann, Scott Fluhrer, Christian Forler, Michael Gorski, Thomas Knapke, Eik List, Stefan Lucks, and David McGrew.

The support of our administrative staff should not be forgotten: thanks to Nadin Glaser, Maria-Theresa Hansens, Carla Högemann, Anja Loudovici, and Christin Oehmichen for their patient and professional handling of all the work-related and also private concerns I had.

A warm thank you goes to my parents Annemarie and Joachim and my brothers Justus and Jan, who always provided me with unconditional support. Finally, I would like to thank Klara, who always beliefs in me and supports me in every possible situation.

Jakob Wenzel, October 2017

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Teile der Arbeit, die bereits Gegenstand von Prüfungsarbeiten waren, sind im Folgenden erwähnt:

- Kapitel 6.3 bis 6.5 sind in Zusammenarbeit mit Christian Forler entstanden und bereits in ähnlicher Art Bestandteil seiner Dissertation[1].
- Die grundlegende Idee von Kapitel 7 ist in Zusammenarbeit mit Christian Forler entstanden und in verminderter Form bereits Bestandteil seiner Dissertation.
- Die beiden in Kapitel 9.1.1 und 9.1.4 beschriebenden Algorithmen wurden in Zusammenarbeit mit Christian Forler ausgewählt und sind bereits Bestandteil seiner Dissertation.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt. Ich versichere, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Ort, Datum　　　　　　　　　　　　　　　　　　　　　　　　Unterschrift

[1] *Christian Forler. Analysis Design & Applications of Cryptographic Building Blocks. Doctoral thesis, Bauhaus-Universität Weimar, 2015.*

Table of Contents

Page

1 Introduction **1**

I Foundations 7

2 Block Ciphers and Block-Cipher-Based Compression Functions **9**

3 Hash Functions and Password Hashing **15**

4 Authenticated Encryption **23**

II Multi-Block-Length Hash Functions 27

5 Counter-bDM **29**

5.1 Related Work 30
5.2 Specification of COUNTER-bDM 32
5.3 Collision-Security Analysis of COUNTER-bDM 33
5.4 Preimage-Security Analysis of COUNTER-bDM 37
5.5 Summary 40

III Password Hashing 41

6 Password Scramblers and Attacks on `scrypt` **43**

6.1 A Short History of Password Hashing 43
6.2 Properties of Modern Password Scramblers 45
6.3 The Password Scrambler `scrypt` 47
6.4 Cache-Timing Attack on `scrypt` 49
6.5 (Weak) Garbage-Collector Attacks on `scrypt` 50
6.6 Summary 52

7 The Catena Password-Scrambling Framework **53**

7.1 Design Decisions 53
7.2 Specification of CATENA 55
7.3 Functional Properties 60
7.4 Security Properties 61
7.5 Usage 62

7.6 Summary 65

8 Catena – Security Analysis of the Framework 67

8.1 Password-Recovery Resistance 67
8.2 PRF Security of Catena 68
8.3 PRF Security of Catena-KG 69
8.4 Summary 69

9 Catena – Instances of its Components 71

9.1 Instances of F_λ, Γ, and Φ 71
9.2 Security Analysis of the Instances of F_λ 79
9.3 Instances of H and H' 89
9.4 Summary 95

10 Catena – Instances of the Framework 97

10.1 Default Instances (Catena-Dragonfly, Catena-Butterfly) 98
10.2 Catena-Stonefly – An ASIC-Resistant Instance of Catena 99
10.3 Catena-Horsefly – A High-Throughput Instance of Catena 99
10.4 Catena-Mydasfly – A TMTO-Resistant Instance of Catena 100
10.5 Catena-Lanternfly – A Hybrid Instance of Catena 100
10.6 Summary 101

11 Comparison of Modern Password Scramblers 103

11.1 Resistance of PHC Candidates against (W)GC Attacks 107
11.2 Sequential vs. Parallel Model 115
11.3 Summary 117

IV Authenticated Encryption 119

12 Reforgeability of Authenticated Encryption Schemes 121

12.1 Classification of AE Schemes 123
12.2 j-INT-CTXT Analysis of NAE Schemes 125
12.3 Stronger Forgery Attacks 131
12.4 Countermeasures to j-IV-Collision Attacks 135
12.5 Summary 135

V Epilog 137

13 Conclusion 139

13.1 Summary 139
13.2 Future Work 140

Bibliography

List of Tables

Table **Page**

5.1 Comparison of security results on DBL compression functions. 30
5.2 Minimum number of queries q required for a collision for $H^{\text{C}b\text{DM}}$. 37
5.3 Minimum number of queries q required for a preimage for $H^{\text{C}b\text{DM}}$. 39

6.1 Comparison of state-of-the-art password scramblers. 45

9.1 Penalty for an SBRG_2^{12} and an SBRG_3^{12} depending on the shift constant c. 74
9.2 Penalties depending on the shift of the sampling points and the depth λ. 83
9.3 Memory requirement dep. on garlic g, attack, graph, and depth λ. 84
9.4 Penalties dep. on garlic g, attack, graph, and the depth λ. 84
9.5 Penalties dep. on the position of Γ within F_λ, the graph, and the depth λ. 85
9.6 Penalties dep. on garlic g, attack, the graph with option Γ, and depth λ. 85
9.7 Comparison of BLAKE2b-1, BLAKE2b-1 w/o initialization, and BLAKE2b. 91
9.8 Measurements of Catena-Dragonfly for different hash functions H'. 94

10.1 Comparison of instances of Catena. 101

11.1 Overview of modern password scramblers – general properties (Part 1). 105
11.2 Overview of modern password scramblers – general properties (Part 2). 106
11.3 Overview of modern password scramblers – functional/security properties (Part 1). 108
11.4 Overview of modern password scramblers – functional/security properties (Part 2). 109
11.5 Overview of the pebbling complexities in the pROM. 115

12.1 Expected #oracle queries for j forgeries for the considered NAE schemes. 123
12.2 Overview of accepted classes. 125
12.3 The NAE schemes considered in [193] according to our classification. 125
12.4 Claimed INT-CTXT bounds of third-round CAESAR and classical AE schemes. 127
12.5 j-IV-CA-Resistance of third-round CAESAR and classical AE schemes. 129
12.6 Stronger forgery attacks on NAE schemes. 131

List of Figures

Figure **Page**

5.1 Two exemplary compression functions H^{C3DM} and H^{C4DM}. 33

7.1 The general idea of Catena employing its core function *flap*. 57
7.2 The general idea of *flap* with its core F_λ. 58

9.1 A BRG_1^3 and an SBRG_1^3. 73
9.2 A $\mathrm{GRG2}_1^3$ and a $\mathrm{GRG3}_1^3$. 75
9.3 A Cooley-Tukey FFT graph with eight input and output vertices. 75
9.4 Types of edges as we use them in our definitions. 77
9.5 A $(3,1)$-double-butterfly graph (DBG_1^3.). 77
9.6 Required memory for the recomputation method shown in [55]. 83

12.1 Generic AE scheme as considered in our analysis. 124
12.2 Simplified schematic illustration of the encryption process in AES-OTRv3.1. 132

List of Acronyms

AE Authenticated Encryption

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

ASIC Application-Specific Integrated Circuit

BRG Bit-Reversal Graph

BRH Bit-Reversal Hashing

CAESAR Competition for Authenticated Encryption: Security, Applicability, and Robustness

CIU Client-Independent Update

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CTA Cache-Timing Attack

DAG Directed Acyclic Graph

DBG Double-Butterfly Graph

DBH Double-Butterfly Hashing

DBL Double-Block-Length

DES Data Encryption Standard

DM Davies-Meyer

EPRE Everywhere-Preimage Security

FPGA Field-Programmable Gate Array

FPO Floating-Point Operation

GC Generic Composition

GCA Garbage-Collector Attack

GHS Generic Hashing Scheme

GGS Generic Graph Scheme

GPU Graphical Processing Unit

GRG Gray-Reverse Graph

GRH Gray-Reverse Hashing

HSM Hardware Security Module

IND-CCA Indistinguishability under Chosen-Ciphertext Attack

IND-CPA Indistinguishability under Chosen-Plaintext Attack

INT-CTXT Integrity of Ciphertext

INT-CTXT-M Integrity of Ciphertext under Multiple Forgery Attempts

INT-PTXT-M Integrity of Plaintext under Multiple Forgery Attempts

IoT Internet of Things

j-INT-CTXT Integrity of Ciphertext for j valid Forgeries

j-IV-CA j-IV-Collision Attack

KDF Key-Derivation Function

LMH λ-Memory Hardness

LSB Least Significant Bit

MAC Message Authentication Code

MSB Most Significant Bit

MBL Multi-Block-Length

NIST National Institute of Standards and Technology

NAE Nonce-Based Authenticated Encryption

NTLM NT LAN Manager

PBKDF2 Password-Based Key Derivation Function 2

PC Personal Computer

PHC Password Hashing Competition

PIN Personal Identification Number

PRF Pseudorandom Function

PRP Pseudorandom Permutation

pROM Parallel Random-Oracle Model

PS Password Scrambler

PSF Password-Scrambling Framework

RAM Random-Access Memory

ROM Read-Only Memory

RNG Random-Number Generator

SCA Side-Channel Attack

SBL Single-Block-Length

SBRG Shifted Bit-Reversal Graph

SBRH Shifted Bit-Reversal Hashing

SMH Sequential Memory Hardness

SMP Symmetric Multiprocessing

SPRP Strong Pseudorandom Permutation

SR Server Relief

TLS Transport Layer Security

TMTO Time-Memory Tradeoff

VoIP Voice over IP

WGCA Weak Garbage-Collector Attack

Notational Conventions

x^ℓ	ℓ-bit string of values x
x^*	bit string of values x of arbitrary length
X_i	i-th block of a value X
X^i	the i-th element of a set
$X \in \{0,1\}^n$	X is a bit string of n bits
$X \in \{0,1\}^*$	X is a bit string of arbitrary length
$X \in \{0,1\}^+$	X is a bit string of at least one bit
$X \in (\{0,1\}^n)^\ell$	X is a bit string of $\ell \cdot n$ bits
$X \in (\{0,1\}^n)^+$	X is a bit string of $\ell \cdot n$ bits with $\ell \geq 1$
$X \mid\mid Y$	concatenation of two bit strings X and Y
$\lvert X \rvert$	size of X in bits or size of a set X
$X \mid\mid 10^*$	X is padded with a 1-bit and trailing 0-bits until $\lvert X \mid\mid 10^* \rvert \equiv 0 \pmod n$
$\mathcal{X}$	set or family
$X \leftarrow Y$	assignment of the value Y to X
$X \twoheadleftarrow \mathcal{X}$	X is a uniformly at random chosen sample from $\mathcal{X}$
$\overline{X}$	bitwise complement of the bit string X
$X \oplus Y$	exclusive-or of the first α MSBs of X and Y with $\alpha = \min\{\lvert X \rvert, \lvert Y \rvert\}$
$X \mid Y$	X is a multiple of Y

Chapter 1

Introduction

If you can't fly, run; if you can't run, walk; if you can't walk, crawl; but by all means keep moving.

Martin Luther King, Jr.

Authentication and Authenticated Encryption (AE) are (or should be) used almost always when people act in the digital world. Be that logging into an E-Mail or an online-banking account, into a user account on a local Personal Computer (PC), or, in general, when there happens to be communication between a sender and a receiver transmitting data over an insecure channel, e.g., the Internet.

In this context, the possibility that the legit receiver is able to check whether a message sent to him was sent by the actual intended party is ensured by *authentication*; to prohibit any adversary from reading the actual content is ensured by *encryption*; and the assurance that a message was not manipulated during the transmission is denoted by *integrity*. These three form the everlasting main goals of *Cryptology*[1], concerning the "*communication in the presence of adversaries.*" (*Cryptology*, by Rivest in [169]). The history of cryptography dates back several thousand years, where people encrypted letters using simple substitution of characters or words (secret writing). Nevertheless, the field of cryptography evolved over and over and yielded to the field of *Modern Cryptography* back in the 1970s, giving rise to mathematical definitions of security, precise mathematical assumptions, and proofs of security.

Therefore, an increased rate of communication transmitting crucial data over the Internet uses an encrypted (and authenticated) channel in-between. That is motivated by the existence of adversaries, which either try to passively eavesdrop and/or to actively manipulate that very data. As a remedy, modern cryptography provides AE, a term referring to algorithms providing authentication, encryption, and integrity.

[1]Cryptology itself splits into *Cryptography* and *Cryptanalysis*, where the former concerns the design and the latter the analysis of cryptographic algorithms. Note that we usually use the term *Cryptography* when referring to *Cryptology*.

While these security goals never changed, the technology to achieve or to break such algorithms providing these goals did (e.g., Moore's Law [190]). Therefore, cryptographers are always encouraged to design algorithms utilizing the state-of-the-art technology to provide a reasonable security against adversaries that are able to use the exact same technology.

This thesis tackles three areas of modern cryptography: *block-cipher-based compression functions*, *password hashing*, and *authenticated encryption*, which are briefly introduced next. Moreover, we cover both the design and the analysis part of cryptography.

Block-Cipher-Based Compression Functions. Between 2007 and 2012, the cryptographic community mainly focused on the design of cryptographic hash functions (SHA-3 Competition [204]). While the SHA-3 Competition has encouraged many new interesting ideas for designing hash and compression functions (e.g., the sponge framework [47]), one of the most popular approaches still is to use a given block cipher and to turn it into a one-way function. While the roots to this simple principle can be tracked back to Rabin [220] at the end of the 70s, the knowledge about it is still highly relevant today. For instance, the standardized SHA-1 [206] and SHA-2 [207] hash function families base on the SHACAL-1/2 ciphers [126]. But also submissions for the SHA-3 competition, such as – MD6 [224], Skein [200], or SHAvite-3 [51] – are built on block ciphers. The advantages are obvious: compression-function designers can profit from the pseudorandomness of an Indistinguishability under Chosen-Ciphertext Attack (IND-CCA)-secure cipher and further, they also require only a single primitive for both encryption and hashing – an important matter when designing hardware for resource-constrained devices. So far, the community focused mainly on single-block- and Double-Block-Length (DBL) (block-cipher-based) compression functions, e.g., [217, 232, 247, 185, 96, 133], providing an n- or $2n$-bit output, respectively, where n denotes the output size of the underlying block cipher. While DBL hashing can offer an acceptable collision security, a variety of applications demand secure Multi-Block-Length (MBL) functions with a freely scalable output of the compression function. For instance, public-key signature schemes expect inputs of the exact length of the signing key. Moreover, in the era of SHA-3, hash values with a length of more than 256 bits are standard; however, it is an open research question to create provably secure b-block-length compression functions for $b > 2$, which we have addressed in this thesis.

Password Hashing. Today, in despite of numerous evolving technologies and techniques used for authentication, passwords[2] have been and continue to be the most widely used form of in the digital world. We describe them as user-memorizable secrets that are commonly used for user authentication and cryptographic key derivation. The former is straightforward. The latter is motivated by the demand of all cryptographic (authenticated) encryption schemes for a pseudorandom secret key that is usually required by the security proof of the underlying primitive, e.g., a block cipher. Such a key can, for example, be derived from a password.

By now, it is common wisdom to store a one-way hash of a password instead of the password itself.[3] The reason for the latter is straightforward: if an adversary gets in possession of the file containing the password hashes (usually assigned to their corresponding accounts), a password stored in plain would grant that adversary already full access to those accounts.

Moreover, humans are weak password generators and tend to favor recognizability over security. Therefore, user-chosen passwords usually suffer from low entropy and can be attacked by trying out all possible candidates in order of likelihood until the right one has been found (so-called brute-force/dictionary attack). Since it is unlikely that users will start to choose stronger passwords,

[2] In our context, "passphrases" and Personal Identification Numbers (PINs) are also "passwords".
[3] Nevertheless, many large companies still do not follow that simple rule – see [75, 153, 125] for examples.

the security must be provided by the algorithm processing a password, which we call a Password Scrambler (PS). Although there are other approaches to enhance password-based authentication, e.g., dedicated cryptographic protocols defeating "off-line" password guessing [43], we focus on password scramblers (resp. Key-Derivation Functions (KDFs)) in this thesis.

The technology and the knowledge of adversaries trying to gather information about user passwords constantly evolved over time. Current password-cracking systems utilize the capabilities of Graphical Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs), allowing for a massively parallel testing of password candidates [250]. Therefore, in 2012, the community started the Password Hashing Competition (PHC) [31] with the hope to obtain new designs thwarting state-of-the-art adversaries. The reason is an increasing asymmetry between the computational devices the typical "defender" is using, and those devices available for potential adversaries. Even without special-purpose hardware, GPUs with hundreds of cores [202] have become a commodity. By making plenty of computational resources available, GPUs are excellent tools for password cracking since each core can try another password candidate, and all cores run at full speed. However, the amount of memory – and, especially that of fast "cache" memory – of a common GPU is comparable to that of a typical Central Processing Unit (CPU) as used by the defenders. Therefore, modern password scramblers focus not only on a reasonable computational time, but also on a high memory consumption to thwart GPUs.

Besides GPU-based attacks, other threats to password scramblers exist: Cache-Timing Attacks (CTAs) [45, 108], Garbage-Collector Attacks (GCAs) [101], Weak Garbage-Collector Attacks (WGCAs) [101], and Time-Memory Tradeoff (TMTO) attacks [52, 55], which will be tackled in this thesis.

Authenticated Encryption. As mentioned above, the communication between two parties over an insecure channel often not only requires encryption (confidentiality), but also authentication and integrity. The simultaneous supply of these goals is the objective of AE schemes (and Authenticated Encryption with Associated Data (AEAD) schemes). In 2000, Bellare and Namprempre considered AE schemes from Generic Composition (GC) [39]. Schemes following this strategy require an encryption scheme providing privacy and a secure Message Authentication Code (MAC) which are combined under independent keys. In contrast, as a remedy to the usual efficiency disadvantage of GC-based AE schemes, the evolution of dedicated AE schemes started with RPC [151], IAPM [147], XCBC [116], and OCB [230]. This development has lead the cryptographic community to the ongoing Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [44], which was started in 2013, giving rise to a large variety of different design choices for dedicated AE schemes. The standard security requirement for AE schemes is to prevent leakage of any information about secured messages except for their respective lengths. However, stateless and deterministic encryption schemes would enable adversaries to detect if the same associated data and message has been encrypted before under the current key. Thus, Rogaway proposed nonce-based encryption [228], where the user must provide an additional nonce for every message it wants to process – a number used once (*nonce*). An AE scheme requiring a nonce as input is called a Nonce-Based Authenticated Encryption (NAE) scheme. In general, most existing block-cipher based AE schemes, such as EAX [42], GCM [180], or OCB3 [161], provide decent resistance to *nonce-respecting* adversaries, but fail badly when nonces are reused (*nonce misuse*). Therefore, a modern AE scheme should provide a second line of defense under nonce misuse, i.e., a decent level of security even when nonces repeat. Additionally, the standard AE security notions assume that an adversary learns nothing about the would-be message whenever a ciphertext fails the authenticity check (*decryption misuse*).

In this thesis, we focus on the integrity of NAE schemes. More detailed, we elaborate on the resistance of a given scheme to an adversary attempting to find multiple forgeries provided one single forgery was already found and called it resistance to *Reforgeability*. By considering all third-round CAESAR candidates and widely used NAE schemes, e.g., EAX and GCM, we provide a decent contribution to the cryptanalytical understanding and progress of designing modern NAE schemes.

List of Publications

This section contains all publications achieved during my doctoral studies, ordered by topic (first) and date (second). This also includes publications that are not discussed in this thesis.

Block-Cipher-Based Compression Functions

[8] *Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Counter-bDM: A Provably Secure Family of Multi-Block-Length Compression Functions. In David Pointcheval and Damien Vergnaud, editors, AFRICACRYPT, volume 8469 of Lecture Notes in Computer Science, pages 440-458. Springer, 2014.*

[98] *Ewan Fleischmann, Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. Weimar-DM: A Highly Secure Double-Length Compression Function. In Willy Susilo, Yi Mu, and Jennifer Seberry, editors, ACISP 2012, volume 7372 of Lecture Notes in Computer Science, pages 152-165. Springer, 2012.*

Password Hashing

[174] *Stefan Lucks, and* ***Jakob Wenzel****. Catena Variants - Different Instantiations for an Extremely Flexible Password-Hashing Framework. In Frank Stajano, Stig Fr. Mjølsnes, Graeme Jenkinson, and Per Thorsheim, editors, PASSWORDS, volume 9551 of Lecture Notes in Computer Science, pages 95-119. Springer, 2015.*

[109] *Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. The Catena Password-Scrambling Framework. Final Submission to the Password Hashing Competition, 2015.*

[108] *Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. Memory-Demanding Password Scrambling. In Palash Sarkar and Tetsu Iwata, editors, ASIACRYPT, volume 8874, Springer 2014.*

[101] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Overview of the Candidates for the Password Hashing Competition - And Their Resistance Against Garbage-Collector Attacks. In Stig Fr. Mjølsnes, editors, PASSWORDS, volume 9393 of Lecture Notes in Computer Science, pages 3-18. Springer, 2014.*

[4] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Overview of the Candidates for the Password Hashing Competition - And Their Resistance Against Garbage-Collector Attacks. IACR Cryptology ePrint Archive, 2014:881, 2014.* (Full Version of [101])

[107] *Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. Catena: A Memory-Consuming Password Scrambler. IACR Cryptology ePrint Archive, 2013:525, 2013.*

Design and Analysis of Authenticated Encryption Schemes

[105] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Reforgeability of Authenticated Encryption Schemes. Paul Watters and Julian Jang-Jaccard, editors, ACISP, volume ..., to appear, Springer, 2017.*

[102] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Efficient Beyond-Birthday-Bound-Secure Deterministic Authenticated Encryption with Minimal Stretch. In Joseph K. Liu and Ron Steinfeld, editors, ACISP (II), volume 9723 of Lecture Notes in Computer Science, pages 317-332. Springer, 2016.*

[104] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. POEx: A Beyond-Birthday-Bound-Secure On-Line Cipher. In Tor Helleseth, organizer, ArcticCrypt, volume 1, 2016. 16 pages.*

[103] *Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Efficient Beyond-Birthday-Bound-Secure Deterministic Authenticated Encryption with Minimal Stretch. IACR Cryptology ePrint Archive, 2016:395, 2016.* (Full Version of [102])

[9] *Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. RIV for Robust Authenticated Encryption. In Thomas Peyrin, editor, FSE, volume 9783 of Lecture Notes in Computer Science, pages 23-42. Springer, 2016.*

[110] *Christian Forler, David McGrew, Stefan Lucks, and* ***Jakob Wenzel****. COFFE: Ciphertext Output Feedback Faithful Encryption. IACR Cryptology ePrint Archive, 2014:1003, 2014.*

[5] *Farzaneh Abed, Scott Fluhrer, John Foley, Christian Forler, Eik List, Stefan Lucks, David McGrew, and* ***Jakob Wenzel****. The POET Family of On-Line Authenticated Encryption Schemes. 2nd-Round Submission to the CAESAR Competition, 2014.*

[10] *Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and* ***Jakob Wenzel****. Pipelineable On-line Encryption. In Carlos Cid and Christian Rechberger, editors, FSE, volume 8540 of Lecture Notes in Computer Science, pages 205-223, Springer, 2014.* (Underlying Encryption Scheme of POET [5])

[4] *Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and* ***Jakob Wenzel****. Pipelineable On-Line Encryption. IACR Cryptology ePrint Archive, 2014:297, 2014.* (Full Version of [10])

[106] *Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. Designing the API for a Cryptographic Library - A Misuse-Resistant Application Programming Interface. In Mats Brorsson and Luís Miguel Pinho, editors, Ada-Europe, volume 7308, Springer, 2012.*

[97] *Ewan Fleischmann, Christian Forler, Stefan Lucks, and* ***Jakob Wenzel****. McOE: A Foolproof On-Line Authenticated Encryption Scheme. IACR Cryptology ePrint Archive, 2011:644, 2011.*

Cryptanalysis of Block Ciphers

[13] *Farzaneh Abed, Eik List, Stefan Lucks, and* ***Jakob Wenzel****. Differential Cryptanalysis of Round-Reduced SIMON and Speck. In Carlos Cid and Christian Rechberger, editors, FSE, volume 8540 of Lecture Notes in Computer Science, pages 525-545. Springer, 2014.*

[7] *Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and **Jakob Wenzel**. A Framework for Automated Independent-Biclique Cryptanalysis. In Carlos Cid and Christian Rechberger, editors, FSE, volume 8424 of Lecture Notes in Computer Science, pages 561-581. Springer, 2013.*

[11] *Farzaneh Abed, Eik List, Stefan Lucks, and **Jakob Wenzel**. Cryptanalysis of the Speck Family of Block Ciphers. IACR Cryptology ePrint Archive, 2013:568, 2013.* (Full Version for the Analysis of Speck of [13])

[12] *Farzaneh Abed, Eik List, Stefan Lucks, and **Jakob Wenzel**. Differential Cryptanalysis of Reduced-Round SIMON. IACR Cryptology ePrint Archive, 2013:526, 2013.* (Full Version for the Analysis of Simon of [13])

[6] *Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and **Jakob Wenzel**. Biclique Cryptanalysis of the PRESENT and LED Lightweight Ciphers. IACR Cryptology ePrint Archive, 2012:591, 2012.*

[119] *Michael Gorski, Thomas Knapke, Eik List, Stefan Lucks, and **Jakob Wenzel**. Mars Attacks! Revisited: Differential Attack on 12 Rounds of the MARS Core and Defeating the Complex MARS Key-Schedule. In Daniel J. Bernstein and Sanjit Chatterjee, editors, IndoCrypt, volume 7107 of Lecture Notes in Computer Science, pages 94-113. Springer, 2011.*

Outline

This thesis consists of five parts, where Part I (Chapters 2 - 4) introduces the foundations (definition concepts, and security definitions) required for understanding the remainder. Part II to Part IV form the core of this work, where we also mention which publication from the list above was used as a basis for the respective chapter:

Part II (Chapter 5) To increase the flexibility of a block-cipher-based compression function, we introduce the Counter-bDM family of MBL compression functions [8] in Chapter 5, which supports a scalable output size making it suitable for a wide range of applications. Part II is adopted almost literally from [8].

Part III (Chapters 6 - 11) We start the third part with a brief survey on the history of password hashing, discuss several properties which should be provided by modern password scramblers, and show that the PS is vulnerable to CTAs, GCAs, and WGCAs, in Chapter 6, where the both latter are attack types introduced by us. That chapter builds the basis for introducing Catena in Chapter 7, a highly flexible and memory-demanding modern Password-Scrambling Framework (PSF), for which we also provide a security analysis in Chapter 8. We proceed with presenting and discussing several instances of the underlying functions of Catena in Chapter 9. In Chapter 10, we provide instances of Catena for a variety of applications. We close Part III by providing a comparison of modern password scramblers in Chapter 11.

Part IV (Chapter 12) In Chapter 12, we analyze AE schemes which are either widely used and/or were submitted to the CAESAR regarding to their resistance to reforgeability attacks. Part IV is adopted almost literally from [105].

In Part V (Chapter 13), this thesis is concluded by providing a summary and a discussion about possible future work.

Part I

Foundations

Chapter 2

Block Ciphers and Block-Cipher-Based Compression Functions

The lesson here is that it is insufficient to protect ourselves with laws; we need to protect ourselves with mathematics. Encryption is too important to be left solely to governments.

Bruce Schneier

Block Ciphers

A block cipher is a key-dependent function E mapping a fixed-length input of n bit to a fixed-length output of the same size. It is used in many cryptographic algorithms as a building block and we define it formally in Definition 2.1.

Definition 2.1 (Block Cipher). *We define a (k, n)-bit block cipher as a keyed family of permutations, which consists of an encryption function $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$, and its inverse (decryption) function $D = E^{-1} : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$. Both take a k-bit key K and an n-bit input block X, and produce an n-bit output Y, where $D_K(E_K(X)) = X$, for all $X \in \{0,1\}^n, K \in \{0,1\}^k$. For positive k and n, we denote by $\texttt{Block}(k, n)$ the set of all (k, n)-bit block ciphers.*

Security Notions and Adversaries

In cryptography, the security of an algorithm is almost always proven under some specific assumption applying to the underlying primitives. For block ciphers, we consider two security models which can be applied, namely, the *Ideal-Cipher Model* and the *(Strong) Pseudorandom Permutation*

(SPRP/PRP) Model. The Ideal-Cipher Model is an idealized model describing a block cipher as family of 2^k random permutation $\mathcal{P}$. To prove security in that model, one must show that a block cipher E_K is indistinguishable from a randomly chosen element of $\mathcal{P}$, i.e., $\mathcal{P}_i \twoheadleftarrow \mathcal{P}$ (see [59, 166, 168, 249] for examples). Note that there exist cryptographic schemes which provide security in the *artificial* Ideal-Cipher Model but fail badly if instantiated with any existing block cipher [56], e.g., the Advanced Encryption Standard (AES) [197] or the Data Encryption Standard (DES) [205].

On the other hand, one could determine the security of a block cipher E_K by showing that it is computationally indistinguishable from an n-bit permutation, rendering it a Pseudorandom Permutation (PRP) or a Strong Pseudorandom Permutation (SPRP). Before we introduce the two latter in Definition 2.2 and 2.3, we show how we model an adversary $\mathbf{A}$.

An adversary $\mathbf{A}$ is an algorithm that interacts with a given set of oracles that appear as black boxes to $\mathbf{A}$. We denote by $\mathbf{A}^{\mathcal{O}}$ the output of $\mathbf{A}$ after interacting with some oracle $\mathcal{O}$. We write $\mathbf{Adv}_F^{\mathrm{X}}(\mathbf{A})$ for the advantage of an adversary $\mathbf{A}$ against a security notion X on a function/scheme F. All probabilities are defined over the random coins of the oracles and those of the adversary, if any. We write $\mathbf{Adv}_F^{\mathrm{X}}(q) = \max_{\mathbf{A}}\{\mathbf{Adv}_F^{\mathrm{X}}(\mathbf{A})\}$ to refer to the maximal advantage over all X-adversaries $\mathbf{A}$ on a given scheme/function F which are only bounded by the number of oracle queries q they are allowed to ask to the given oracles. WLOG, we assume that an adversary $\mathbf{A}$ never asks queries to which it already knows the answer, and by $\mathcal{O}_1 \not\hookrightarrow \mathcal{O}_2$ we denote that $\mathbf{A}$ never queries $\mathcal{O}_2$ with the output of $\mathcal{O}_1$.

During the query phase, we say that an adversary $\mathbf{A}$ maintains a query history $\mathcal{Q}$ collecting all requests together with their corresponding answer. We write $\mathcal{Q}_{|x}$, if we refer only to all entries of value type x, e.g., message, ciphertext, or a tag, in the query history. For example, $N_i \notin \mathcal{Q}_{|N}$ denotes that the nonce[1] N_i is not contained in the set of nonces already in the query history $\mathcal{Q}$.

In terms of block ciphers, an adversary $\mathbf{A}$ is allowed to ask either a forward (encryption) query $E_K(X) = Y$, or a backward (decryption) query $X = D_K(Y)$, where $X, Y \in \{0,1\}^n$ and $\forall\, X :$ $D_K(E_K(X)) = X$. Each query Q^i is stored as a tuple (X_i, Y_i, K_i) in $\mathcal{Q}$, where we denote by $\mathcal{Q}_i$ the state of the $\mathcal{Q}$ after i queries have been asked by the adversary, for $1 \leq i \leq q$.

Definition 2.2 (PRP Security). *Let $E \in \mathtt{Block}(k, n)$ denote a block cipher. Let $\mathbf{Perm}_n$ be the set of all n-bit permutations. The PRP advantage of $\mathbf{A}$ against E is then defined by*

$$\mathbf{Adv}_E^{\mathsf{PRP}}(\mathbf{A}) \leq \left|\Pr\left[\mathbf{A}^{E(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathbf{A}^{\pi(\cdot)} \Rightarrow 1\right]\right|,$$

where the probabilities are taken over $K \twoheadleftarrow \{0,1\}^k$ and $\pi \twoheadleftarrow \mathbf{Perm}_n$. We define $\mathbf{Adv}_E^{\mathsf{PRP}}(q)$ as the maximum advantage over all PRP-adversaries $\mathbf{A}$ on E that ask at most q queries to the given oracles.

Definition 2.3 (SPRP Security). *Let $E \in \mathtt{Block}(k, n)$ denote a block cipher and D its inverse. Let $\mathbf{Perm}_n$ be the set of all n-bit permutations. The SPRP advantage of $\mathbf{A}$ against E is then defined by*

$$\mathbf{Adv}_{E,D}^{\mathsf{SPRP}}(\mathbf{A}) \leq \left|\Pr\left[\mathbf{A}^{E(\cdot),D(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathbf{A}^{\pi(\cdot),\pi^{-1}(\cdot)} \Rightarrow 1\right]\right|,$$

[1] A nonce (**n**umber used **once**) is a value which must never repeat in a given context.

where the probabilities are taken over $K \twoheadleftarrow \{0,1\}^k$ *and* $\pi \twoheadleftarrow \mathbf{Perm}_n$*. We define* $\mathbf{Adv}^{\mathsf{SPRP}}_{E,D}(q)$ *as the maximum advantage over all* SPRP*-adversaries* $\mathbf{A}$ *on* E *that ask at most* q *queries to the given oracles.*

Block-Cipher-Based Compression Functions

One use case for applying a block cipher as a building block is given by so-called block-cipher-based compression functions, which take two fixed-length inputs of n bit each, and output one n-bit output. A formal description is given in Definition 2.4[2]

Definition 2.4 (SBL Compression Function). *A Single-Block-Length (SBL) block-cipher-based compression function is a mapping*

$$H^{SBL} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n,$$

which uses a block cipher from $\mathtt{Block}(n,n)$*.*

The idea of a block-cipher-based compression function was first discussed in the literature by Rabin [220]. Most SBL functions use a block cipher from $\mathtt{Block}(n,n)$ and compress a $2n$-bit string to an n-bit string. A popular example is the Davies-Meyer (DM) [262] mode:

$$H^{\mathrm{DM}}(M,U) = E_M(U) \oplus U,$$

which is, for example, used twice inside Hirose-DM [134]. A more generalized form of a single-block-length compression function can be described by a function compressing a $(b+1)n$-bit input to a fixed-length bn-bit output. That construction is called MBL compression function and is formally defined in Definition 2.5.

Definition 2.5 (MBL Compression Function). *A Multi-Block-Length (MBL) block-cipher-based compression function is a mapping*

$$H^{MBL} : \{0,1\}^{bn} \times \{0,1\}^n \to \{0,1\}^{bn},$$

which takes a bn*-bit chaining value and an* n*-bit message, and outputs a new* bn*-bit chaining value. It internally employs a block cipher from* $\mathtt{Block}(bn,n)$*.*

For our security analysis, we assume the underlying block cipher to be secure, e.g.,, it behaves like an ideal block cipher. Therefore, we show the security of a block-cipher-based compression function by analyzing the construction itself and any security architect should be aware of choosing a secure

[2]Note that, in comparison to common literature, e.g., [216], we refer to a more restrictive definition of a block cipher by fixing the size of the key.

block cipher. It follows that we usually consider the *ideal-cipher model*, wherein a block cipher is modeled as a family of random permutations $\{E_K\}$ and the permutation E that is used in the compression function is chosen at random from $\texttt{Block}(k, n)$: $E \twoheadleftarrow \texttt{Block}(k, n)$. We borrow two usual assumptions about **A** from [98]:

1. If **A** has successfully found a collision or a preimage for H^{MBL}, it has obtained the necessary encryption or decryption results to E only by making queries to the oracle E.
2. **A** does not make queries to which it already knows the answer, e.g., if **A** already knows the answer to a forward query $Y = E_K(X)$, it will not request the answer to the corresponding backward query $D_K(Y)$ – which will necessarily return X – and vice versa.

Collision Security. We define the collision security of our compression function H^{CbDM} by the advantage of an adversary **A** to win Experiment 2.6, which is described in the following.

Experiment 2.6 (Collision-Finding Experiment).

1. *An adversary* **A** *is given oracle access to a block cipher* $E \twoheadleftarrow \texttt{Block}(bn, n)$.
2. *After asking at most q forward (encryption) or backward (decryption) queries* (X_i, K_i) *or* (Y_i, K_i), *where* $Y_i = E_{K_i}(X_i)$ *and* $X_i = D_{K_i}(Y_i)$, *respectively, collecting q tuples* (X_i, Y_i, K_i) *for* $1 \leq i \leq q$, *it outputs a pair* $(M, U_1, \ldots, U_b), (M', U'_1, \ldots, U'_b) \in \{0,1\}^{(b+1)n} \times \{0,1\}^{(b+1)n}$.
3. *The adversary wins the experiment iff its output is a valid collision for* H^{CbDM}, *i.e.,*

$$H^{CbDM}(M, U_1, \ldots, U_b) = H^{CbDM}(M', U'_1, \ldots, U'_b) \text{ and }$$
$$(M, U_1, \ldots, U_b) \neq (M', U'_1, \ldots, U'_b).$$

Otherwise, **A** *loses the experiment.*

The advantage of **A** for finding a collision is defined by the maximal advantage taken from all possible adversaries that request at most q queries, or formally written:

$$\mathbf{Adv}^{\text{COLL}}_{H^{\text{MBL}}}(q) := \max \left\{ \mathbf{Adv}^{\text{COLL}}_{H^{\text{MBL}}}(\mathbf{A}) \right\}.$$

Preimage Security. There are various notions considering preimage security (see [231] for example). We adapt that of Everywhere-Preimage Security (EPRE), which was introduced by Rogaway and Shrimpton in [231]. There, the adversary commits to a hash value before it makes any queries to the oracle. The preimage security of H^{MBL} is therefore defined by the advantage that an adversary **A** wins Experiment 2.7.

Experiment 2.7 (Preimage-Finding Experiment).

1. *An adversary* **A** *is given oracle access to a block cipher* $E \twoheadleftarrow \texttt{Block}(bn, n)$. *Before it makes any queries, it announces a hash value* $(V_1, \ldots, V_b) \in \{0,1\}^{bn}$.

2. *After asking at most q forward (encryption) or backward (decryption) queries (X_i, K_i) or (Y_i, K_i), where $Y_i = E_{K_i}(X_i)$ and $X_i = D_{K_i}(Y_i)$, respectively, collecting q tuples (X_i, Y_i, K_i) for $1 \leq i \leq q$, it outputs a $(b+1)$-tuple $(M, U_1, \ldots, U_b) \in \{0,1\}^{(b+1)n}$.*

3. *The adversary wins the experiment iff its output is a valid preimage for $(V_1, \ldots, V_b)$ and H^{CbDM}, i.e.,*

$$H^{CbDM}(M, U_1, \ldots, U_b) \ = \ (V_1, \ldots, V_b).$$

Otherwise, **A** *loses the experiment.*

The advantage of **A** for finding a preimage is defined by the maximal advantage taken from all possible adversaries that ask at most q queries:

$$\mathbf{Adv}^{\mathrm{EPRE}}_{H^{\mathrm{MBL}}}(q) := \max \left\{ \mathbf{Adv}^{\mathrm{EPRE}}_{H^{\mathrm{MBL}}}(\mathbf{A}) \right\}.$$

In the above defined Preimage-Finding Experiment, an adversary **A** is forced to commit itself to a hash value $(V_1, \ldots, V_b)$ for which it has to find a preimage. In contrast, we denote by

$$\mathbf{Adv}^{\mathrm{PRE}}_{H^{\mathrm{MBL}}}(q) := \max \left\{ \mathbf{Adv}^{\mathrm{PRE}}_{H^{\mathrm{MBL}}}(\mathbf{A}) \right\}$$

the advantage of an adversary **A** trying to find a preimage for an arbitrarily chosen hash value.

Chapter 3

Hash Functions and Password Hashing

Knowledge is power. Information is liberating. Education is the premise of progress. In every society, in every family.

Kofi Atta Annan

Hash Functions

The concept of a hash function was introduced by Knuth in the 1950s [157] and informally describes a function which compresses an arbitrarily large input to an output of fixed length (*hash, hash value, or digest*). For this thesis, we borrow and slightly adapt the definition of an *unkeyed hash function* by Rogaway [229].

Definition 3.1 (Hash Function). *An n-bit hash function H is a function*

$$H : \{0,1\}^* \to \{0,1\}^n,\ n \in \mathbb{N}^+,$$

where $\mathbb{N}^+$ denotes the set of positive integers.

Security Models. An idealized model for proving the security of a hash function is described by the *Random-Oracle Model* as introduced by Bellare and Rogaway [40]. To be considered as security, the hash function must be *indistinguishable* from a so-called random function (random oracle), which is an abstract primitive returning a randomly chosen fixed-length bitstring for each new input. That model could for example be applied in the case when considering the inner functionality of the hash function, e.g., SHA-1 [206] or SHA-2 [207], would make a security analysis too complex. Note that similar to the Ideal-Cipher Model discussed in Chapter 2, there exist schemes which provide

security in the Random-Oracle Model but are insecure when considering any particular instance (see [35, 118] for example).

Given a random oracle $\$: \{0,1\}^* \to \{0,1\}^n$, we say that a hash function H_K with $H : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n$ is Pseudorandom Function (PRF)-secure if it is indistinguishable from \$, where $K \twoheadleftarrow \mathcal{K}$ denotes a uniformly distributed secret chosen at random (see Definition 3.2 for a formal definition).

Definition 3.2 (PRF Security). *Let $H : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n$ be a function with a secret $K \twoheadleftarrow \{0,1\}^k$. Let $\mathbf{A}$ be an adversary that is allowed to ask at most q queries to an oracle. Further, let $\$: \{0,1\}^* \to \{0,1\}^n$ be a random function. Then, the PRF Advantage of $\mathbf{A}$ wrt. to H is defined by*

$$\mathbf{Adv}_H^{PRF}(\mathbf{A}) = \left| \Pr\left[\mathbf{A}^{H(K,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathbf{A}^{\$(\cdot)} \Rightarrow 1\right] \right|.$$

Furthermore, by $\mathbf{Adv}_H^{PRF}(q)$ we denote the maximum advantage taken over all adversaries that ask at most q queries to the given oracle.

In contrast to the idealized model described above, the security of a hash function can be measured by considering three assumption provided by the so-called *Standard Model*, i.e., *Collision Resistance*, *Preimage Resistance*, and *2nd-Preimage Resistance*. A hash function fulfilling those requirements is called a *cryptographic hash function* (see Definition 3.3).

Definition 3.3 (Cryptographic Hash Function). *Let H be as defined in Definition 3.1. We call H a cryptographic hash function if it fulfills the following three properties:*

- ***Collision Resistance**: It is practically infeasible to find two distinct inputs $x \neq x' \in \{0,1\}^*$ which lead to $H(x) = H(x')$.*
- ***Preimage Resistance**: Given a hash value $h \leftarrow H(x)$ with $x \in \{0,1\}^*$, it is practically infeasible to find x.*
- ***Second-Preimage Resistance**: Given a hash value $h \leftarrow H(x)$ and its corresponding input x with $x \in \{0,1\}^*$, it is practically infeasible to find a value $x' \in \{0,1\}^*$ with $x' \neq x$ so that $H(x') = H(x)$ holds.*

Password Hashing

We start this section by providing a generic definition (see Definition 3.4) of a password scrambler (PS). The reason for that rather loose definition is caused by the large variety of existing password-hashing designs, which makes a more strict definition impossible.

Definition 3.4 (Password Scrambler). *Let $\mathcal{S} \subset \{0,1\}^*$ and $\mathcal{PWD} \subset \{0,1\}^*$ define the set of all possible salts and passwords. Further, let $\mathcal{T} \subset \{0,1\}^*$ denote an arbitrarily large set of additional (optional) information and parameters. Then, a Password Scrambler (PS) P is a mapping*

$$P : \mathcal{S} \times \mathcal{PWD} \times \mathcal{T} \rightarrow \{0,1\}^n.$$

The optional set $\mathcal{T}$ could, for example, contain parameters regarding the time and memory consumption as well as a tweak, a header AD (associated data), or other additional information and parameters – always depending on the particular algorithm which is considered.

Note that we assume the salt value $s \in \mathcal{S}$ and the password $pwd \in \mathcal{PWD}$ to be compulsory inputs to P (even if $s, pwd \in \{0,1\}^*$ allows for the empty string). The reason for the password is obvious, whereas the reason for the salt value is based on the fact that it has become a commonly used (and also a mandatory) input since the first password scrambler `crypt` was introduced in UNIX-based systems [191] resting upon the ideas of Wilkes et al. [261]. The value s refers to a publicly known value which is processed together with the password. The basic idea is to employ a fresh value of s for each password pwd: assume $pwd_1 = pwd_2$ to be two user-given passwords. Further, let $s_1 \neq s_2$ denote two distinct salts. Then, it is highly likely that $P(s_1, pwd_1, \cdot) \neq P(s_2, pwd_2, \cdot)$. This technique, effectively thwarting "off-line" password-guessing adversaries, was introduced to reduce the impact of so-called dictionary attacks that base on a large number of precomputed password candidates. The currently most advanced member of that kind of attacks is given by *rainbow tables* published by Oechslin in 2003 [203]. To make such attacks even more complicated, Manber suggested to hold some bits of the salt secret to the adversary, turning then into *pepper* [177].

We go on with describing and defining the desired properties a modern PS should achieve. Apart from the introduction of `scrypt` [211] – the first memory-demanding PS – the topic of a flexible memory requirement of a PS come to the fore during the PHC. To make it more tangible, we introduce the parameter *garlic* (g, see Definition 3.4). In general, the parameter g denotes that one invocation of the PS requires about $G = 2^g \cdot n$ bits of memory, where n usually denotes the output size in bits of the underlying primitive, e.g., $n = 512$ for SHA-512 [196].[1]

Even if the basic idea behind a cryptographic hash function and a PS seems to be quite similar, a PS has greater demands. Therefore, we introduce seven properties which should be fulfilled by every modern PS: Memory Hardness, Preimage Security, Client-Independent Update (CIU), Server Relief (SR), Resistance to Cache-Timing Attacks (CTAs), being a Key-Derivation Function (KDF), and Resistance to GCAs and WGCAs.

Memory Hardness. To describe memory requirements, we adopt and slightly change the notion from [211]. The intuition is that for any parallelized attack, using b cores, the required memory per core is decreased by a factor of $1/b$, and vice versa.

[1]Note that the output size of n bits of the underlying primitive is also almost always the same as that of the resulting hash value.

Definition 3.5 (Memory-Hard Function [211]). *Let g denote the memory-cost factor. For all $\alpha > 0$, a memory-hard function f can be computed on a Random-Access Machine using $S(g)$ space and $T(g)$ operations, where $S(g) \in \Omega(T(g)^{1-\alpha})$.*

Loosely speaking, a memory-hard algorithm requires a certain number of memory units to be stored when computing an output. A closely linked approach is given by a so-called Time-Memory Tradeoff (TMTO) introduced by Hellman in [131], which describes an approach to trade memory/space S against time T when considering generic attacks on cryptographic algorithms. Thereby, an adversary with access to that algorithm aims at a sweet spot for minimizing $S \cdot T$. Thus, the running time T of a function f designed to allocate $G = 2^g$ units of memory is significantly increased whenever $S < G$ units of memory are available. For example, let S denote the available memory and T the required time, then, the quadratic TMTO $S \cdot T = G^2$, when using b cores, implies

$$\left(\frac{1}{b} \cdot S\right) \cdot (b \cdot T) = G^2.$$

Further, we introduce a formal generalization of this notion as shown in Definition 3.6.

Definition 3.6 (λ-Memory-Hard Function). *Let g denote the memory cost factor. For a λ-memory-hard function f, which is computed on a Random-Access Machine using $S(g)$ space and $T(g)$ operations with $G = 2^g$, it holds that*

$$T(g) = \Omega\left(\frac{G^{\lambda+1}}{S^{\lambda}(g)}\right).$$

Therefore, for a λ-memory-hard function f, the relation $S(g) \cdot T(g)$ is always in $\Omega(G^{\lambda+1})$. Furthermore, when considering b cores, we can state that

$$\left(\frac{1}{b} \cdot S^{\lambda}\right) \cdot (b \cdot T) = G^{\lambda+1}.$$

An slightly different requirement is given by Sequential Memory Hardness (SMH) that is described in Definition 3.7 (adapted version of Definition 2 in [211]).

Definition 3.7 (Sequential Memory-Hard Function (adapted from [211])). *Let g denote the memory-cost factor. Let f be a memory-hard function as defined in Definition 3.5. Then, f is sequential memory-hard if it* ***cannot*** *be computed on a Parallel Random-Access Machine with $S(g)$ space with expected number of $T(g)$ operations, where $S(g) \in \mathcal{O}(T(g)^{1-\alpha})$ for any $\alpha > 0$.*

That definition says that providing parallelism does neither decrease the computational time nor the required memory of a function satisfying sequential-memory-hardness.

Password Recovery (Preimage Security). For a modern password scrambler, it should hold that the advantage of an adversary (modeled as a computationally unbounded but always-halting algorithm) for guessing a valid password should be reasonably small, i.e., not higher than for trying out all possible candidates. Therefore, given a password scrambler P, we define the password-recovery advantage of an adversary $\mathbf{A}$ as in Definition 3.8.

Definition 3.8 (Password-Recovery Advantage). *Let $s \in \{0,1\}^*$ denote a randomly chosen salt value, $\mathcal{Q}$ be an entropy source, and pwd be a password chosen at random from $\mathcal{Q}$. Then, we define the password-recovery advantage of an adversary $\mathbf{A}$ against a password scrambler P as*

$$\mathbf{Adv}_P^{REC}(\mathbf{A}) = \Pr\left[pwd \twoheadleftarrow \mathcal{Q}, h \leftarrow P(s, pwd, \cdot) : x \twoheadleftarrow \mathbf{A}^{P,s,h} : P(s, x, \cdot) \stackrel{?}{=} h\right].$$

Furthermore, we denote by $\mathbf{Adv}_P^{REC}(q)$ the maximum advantage taken over all adversaries asking at most q queries to P.

Client-Independent Update. According to Moore's Law [189], the available resources of an adversary increase continually over time – and so do the legitimate user's resources. Thus, a security parameter chosen once may be too weak after some time and needs to be updated. This can easily be done immediately after the user has entered its password the next time. However, in many cases, a significant number of user accounts are inactive or rarely used, e.g., 70.1% of all Facebook accounts experience zero updates per month [198] and 73% of all Twitter accounts do not have at least one tweet per month [235]. It is desirable to be able to compute a new password hash (with a higher security parameter) from the old one (with the old and weaker security parameter), without requiring user interaction, i.e., without having to know the password. We call this feature a Client-Independent Update (CIU) of the password hash. When key stretching is realized by iterating an operation, CIUs may or may not be possible, depending on the details of the operation. For instance, when the original password is one of the inputs for every iteration, CIUs are impossible.

Server Relief. A slow and – even worse – memory-demanding password-based log-in process may be too much of a burden for many service providers. A way to overcome this problem is to shift the effort from the side of the server to the side of the client, which is described in [199] and more recently in [79]. We realized this idea by splitting the Password Scrambler (PS) into two parts: (1) a slow (and possibly memory-demanding) one-way function F and (2) an efficient one-way function H. By default, the server computes the password hash $h = H(F(s, pwd, \cdot))$ from a salt s and a password *pwd*. Alternatively, the server sends s to the client who responds with $y = F(s, pwd, \cdot)$. Finally, the server just computes $h = H(y)$. While it is probably easy to write a generic Server Relief (SR) protocol, only a small amount of the existing password scramblers has been designed to naturally support this property, e.g., Argon2 [52]. Note that this property is optional, e.g., the idea of SR makes no sense for the proof-of-work scenario[2] since the whole effort should be already on the side of the client.

[2] In such a scenario, the client (prover) has to solve a challenging task, e.g., to find a preimage *pwd* for a Password Scrambler (PS) P provided only with $P(pwd)$, to get access to the server.

Resistance to Cache-Timing Attacks. Consider the implementation of a PS, where data is read from or written to a password-dependent address $a = f(pwd)$. If, for another password pwd', we would get $f(pwd') \neq a$ *and* the adversary could observe whether we access the data at address a or not, then it could use this information to filter password 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. Thus, we provide a formal definition for resistance of a PS P to Cache-Timing Attacks (CTAs).

Definition 3.9 (Resistance against Cache-Timing Attacks). *Let P be a PS as defined in Definition 3.4 and $pwd \in \{0,1\}^*$ be a secret input to P. We call P resistant to cache-timing attacks iff neither its control nor its data flow depend on pwd.*

Note that CTAs are not only applicable to password scramblers (that restriction is only given for that very thesis). See [45, 210, 256] for more examples.

Key Derivation. Beyond authentication, passwords are used also to derive symmetric keys. Obviously, one can just use the output of a PS as a symmetric key – perhaps after truncating it to the required key size. This is a disadvantage if one needs either a key longer than the password hash or multiple keys. Therefore, it is prudent to consider a Key-Derivation Function (KDF) as a tool of its own right – with the option to derive more than one key and with the security requirement that compromising some keys does not endanger the other ones. Note that it is required for a KDF that its outputs cannot be distinguished from that of a random function (random oracle) (see Definition 3.2).

Resistance to (Weak) Garbage-Collector Attacks. The basic idea of this attack type is to exploit that a PS leaves the internal state or (efficiently computable) password-dependent values in memory for a *long* time during their computation. More detailed, the goal of an adversary is to find a password filter for password candidates from observing the memory used by an algorithm, where the password filter requires significantly less time/memory in comparison to the original algorithm. Next, we formally define the term GCA.

Definition 3.10 (Garbage-Collector Attack). *Let P be as defined in Definition 3.4 depending on a memory-cost parameter g and let c be a positive constant. Furthermore, let v denote the memory units which were allocated by P and remain in memory after its termination. Let $\mathbf{A}$ be a computationally unbounded adversary. We say that $\mathbf{A}$ is successful in conducting a Garbage-Collector Attack (GCA) if some knowledge about v reduces the runtime of $\mathbf{A}$ for testing a password candidate x from $\mathcal{O}(t_P)$ to $\mathcal{O}(t_f)$ with $\mathcal{O}(t_f) \lll \mathcal{O}(t_P)/c$ and $\forall x \in \{0,1\}^*$, where $\mathcal{O}(t_f)$ denotes the runtime of the password-testing function $f(x)$ of $\mathbf{A}$ and $\mathcal{O}(t_P)$ the runtime of $P(\cdot, x, \cdot)$.*

In the following, we define the Weak Garbage-Collector Attack (WGCA).

Definition 3.11 (Weak Garbage-Collector Attack). *Let P be as defined in Definition 3.4 depending on a memory-cost parameter g, and let $R(\cdot)$ be an underlying function of P that can be computed significantly more efficient than P itself. We say P is vulnerable to a Weak Garbage-Collector Attack (WGCA) if a value $y = R(pwd)$ remains in memory during (almost) the entire runtime of $P(\cdot, pwd, \cdot)$, where pwd denotes the secret input.*

An adversary that is capable of reading the internal memory of a password scrambler during its invocation gains knowledge about y. Thus, it can reduce the effort for filtering invalid password candidates by just computing $y' = R(x)$ and checking whether $y = y'$, where x denotes the current password candidate. Note that the function R can also be the identity function. Then, the plain password remains in memory, rendering WGCAs trivial.

CHAPTER 4

Authenticated Encryption

If you are the smartest person in the room, you are in the wrong room.

ANONYMOUS

Nonce-Based AE Schemes. A formal description of a Nonce-Based Authenticated Encryption (NAE) scheme (with associated data) [226] is provided in Definition 4.1, where the associated data denotes additional data which is only authenticated but not encrypted, e.g., a header of a package containing routing information and the size of the payload.

Definition 4.1 (NAE Scheme with Associated Data [226]). *A Nonce-Based Authenticated Encryption (NAE) scheme (with associated data) is a tuple* $\Pi = (\mathcal{E}, \mathcal{D})$ *of a deterministic encryption algorithm* $\mathcal{E} : \mathcal{K} \times \mathcal{A} \times \mathcal{N} \times \mathcal{M} \to \mathcal{C} \times \mathcal{T}$, *and a deterministic decryption algorithm* $\mathcal{D} : \mathcal{K} \times \mathcal{A} \times \mathcal{N} \times \mathcal{C} \times \mathcal{T} \to \mathcal{M} \cup \{\perp\}$, *where* $\perp$ *denotes the invalid symbol,* $\mathcal{K}$ *the associated non-empty key space,* $\mathcal{A} \subseteq \{0,1\}^*$ *the associated-data space,* $\mathcal{N}$ *the non-empty nonce space,* $\mathcal{T} = \{0,1\}^\tau$ *the non-empty tag space for a security parameter* τ, *and* $\mathcal{M}$, $\mathcal{C} \subseteq \{0,1\}^*$ *denote the message and ciphertext space, respectively. We write*

$$(AD, N, C, T) \leftarrow \mathcal{E}(K, AD, N, M) \quad \text{and}$$
$$M \vee \perp \leftarrow \mathcal{D}(K, AD, N, C, T)$$

to state that $\mathcal{E}$ *always outputs a ciphertext* C *and the authentication tag* T *for the tuple* (AD, N, M), *and* $\mathcal{D}$ *outputs the decryption of* (AD, N, C) *if the given tag is valid or* $\perp$ *otherwise.*

We assume that for all $K \in \mathcal{K}$, $A \in \mathcal{A}$, $N \in \mathcal{N}$, $C \in \mathcal{C}$, $T \in \mathcal{T}$, and $M \in \mathcal{M}$, Π preserves:

Stretch-Preservation: If $\mathcal{E}_K^{A,N}(M) = (C, T)$, then $|C| = |M|$ and $|T| = \tau$,

Correctness: If $\mathcal{E}_K^{A,N}(M) = (C,T)$, then $\mathcal{D}_K^{A,N}(C,T) = M$.

Tidiness: If $\mathcal{D}_K^{A,N}(C,T) = M \neq \perp$, then $\mathcal{E}_K^{A,N}(M) = (C,T)$.

Further, we will often use $\mathcal{E}_K^{AD,N}(M)$ and $\mathcal{D}_K^{AD,N}(C,T)$ as short forms of $\mathcal{E}(K, AD, N, M)$ and $\mathcal{D}(K, AD, N, C, T)$, respectively.

Security Notion for Integrity. Now, we introduce the Integrity of Ciphertext (INT-CTXT) experiment (see Experiment 4.2, based on the INT-CTXT definitions given in [39, 41, 151]) that builds the basis for the Integrity of Ciphertext for j valid Forgeries (j-INT-CTXT) experiment introduced next (see Experiment 4.4). The INT-CTXT advantage of an adversary $\mathbf{A}$ is given in Definition 4.3.

Experiment 4.2 (INT-CTXT Experiment).

1. *An adversary* $\mathbf{A}$ *is given oracle access to an encryption function* $\mathcal{E}$ *and a decryption function* $\mathcal{D}$*, where key* $K \twoheadleftarrow K$.
2. *It asks at most* q *forward (encryption) or backward (decryption) queries* (A_i, N_i, M_i) *or* (A_i, N_i, C_i, T_i)*, where* $(C_i, T_i) = \mathcal{E}_K(A_i, N_i, M_i)$ *and* $(M \vee \perp) = \mathcal{D}_K(A_i, N_i, C_i, T_i)$*, where* $\mathbf{A}$ *never queries* $\mathcal{E} \hookrightarrow \mathcal{D}$.
3. $\mathbf{A}$ *wins the experiment iff it outputs a decryption query* (A, N, C, T) *such that* $\mathcal{D}_K(A, N, C, T) \neq \perp$*. Otherwise,* $\mathbf{A}$ *loses the experiment.*

Definition 4.3 (INT-CTXT Advantage). *Let* $\Pi = (\mathcal{E}, \mathcal{D})$ *be a nonce-based AE scheme,* $K \twoheadleftarrow \mathcal{K}$*, and* $\mathbf{A}$ *be a computationally bounded adversary on* Π *with access to two oracles* $\mathcal{E}$ *and* $\mathcal{D}$ *such that* $\mathbf{A}$ *never queries* $\mathcal{E} \hookrightarrow \mathcal{D}$*. Then, the INT-CTXT advantage of* $\mathbf{A}$ *on* Π *is defined as*

$$\mathbf{Adv}_{\Pi}^{\textsf{INT-CTXT}}(\mathbf{A}) := \Pr\left[\mathbf{A}^{\mathcal{E},\mathcal{D}} \textit{ forges}\right],$$

where "forges" means that $\mathcal{D}_K$ *returns anything other than* $\perp$ *for a query of* $\mathbf{A}$.

Experiment 4.4 (j-INT-CTXT Experiment).

1. *An adversary* $\mathbf{A}$ *is given oracle access to an encryption function* $\mathcal{E}$ *and a decryption function* $\mathcal{D}$*, where key* $K \twoheadleftarrow K$.
2. *It asks at most* q *forward (encryption) or backward (decryption) queries* (A_i, N_i, M_i) *or* (A_i, N_i, C_i, T_i)*, where* $(C_i, T_i) = \mathcal{E}_K(A_i, N_i, M_i)$ *and* $(M \vee \perp) = \mathcal{D}_K(A_i, N_i, C_i, T_i)$*, where* $\mathbf{A}$ *never queries* $\mathcal{E} \hookrightarrow \mathcal{D}$.

3. **A** *wins the experiment iff it outputs* j *a decryption queries* $(A_\ell, N_\ell, C_\ell, T_\ell)$, *with* $1 \leq \ell \leq j$, *such that for all* ℓ *it holds* $\mathcal{D}_K(A_\ell, N_\ell, C_\ell, T_\ell) \neq \bot$. *Otherwise,* **A** *loses the experiment.*

Security Notion for Reforgeability. In 2004, Bellare et al. introduced the two security notions Integrity of Plaintext under Multiple Forgery Attempts (INT-PTXT-M) and Integrity of Ciphertext under Multiple Forgery Attempts (INT-CTXT-M) [37]; however, these notions capture the setting that an adversary can pose multiple verification queries for a *single* forgery. In contrast, we are interested in finding *multiple* (in general $j \geq 1$) forgeries based on multiple verification queries. In the scenario of INT-CTXT (see Experiment 4.2), an adversary wins if it can find any valid forgery, that is a tuple (A, N, C, T) for which the decryption returns anything different from the invalid symbol $\bot$ and which has not been previously obtained by **A** as response of the encryption oracle. The j-INT-CTXT security notion, as shown in Experiment 4.4, is derived from INT-CTXT in the sense that **A** now has to provide j distinct valid forgeries that all have not been obtained from the encryption oracle. In the following, we define the j-INT-CTXT Advantage of an adversary.

Definition 4.5 (j-INT-CTXT Advantage). *Let* $\Pi = (\mathcal{E}, \mathcal{D})$ *be a nonce-based AE scheme,* $K \twoheadleftarrow \mathcal{K}$, *and* **A** *be a computationally bounded adversary on* Π *with access to two oracles* $\mathcal{E}$ *and* $\mathcal{D}$ *such that* **A** *never queries* $\mathcal{E} \hookrightarrow \mathcal{D}$. *Then, the j-INT-CTXT advantage of* **A** *on* Π *is defined as*

$$\mathbf{Adv}_{\Pi}^{\textsf{j-INT-CTXT}}(\mathbf{A}) := \Pr\left[\mathbf{A}^{\mathcal{E},\mathcal{D}} \text{ forges } j \text{ times}\right],$$

where "forges" means that $\mathcal{D}_K$ *returns anything other than* $\bot$ *for a query of* **A**, *and "forges* j *times" means that* **A** *provides* j *distinct decryption queries* (A_i, N_i, C_i, T_i), $1 \leq i \leq j$, *such that* $\mathcal{D}_K(A_i, N_i, C_i, T_i) \neq \bot$ *for all* $1 \leq i \leq j$.

Part II

Multi-Block-Length Hash Functions

Counter-bDM

Peace cannot be kept by force; it can only be achieved by understanding.

Albert Einstein

A Provably Secure Family of MBL Compression Functions

Block-cipher-based compression functions serve an important purpose in cryptography since they allow to turn a given block cipher into a one-way hash function. The best understood principle are so-called Single-Block-Length (SBL) constructions, which compress a $2n$-bit input to an n-bit output, with n being the state size of the cipher. However, the output length of, e.g., the AES [76] (one of the currently most widely used block ciphers) is only 128 bits, which is too small for an acceptable collision security for most applications. As a consequence, one is usually interested in Double-Block-Length (DBL) constructions or, more generally, Multi-Block-Length (MBL) block-cipher-based hash functions[1], which take an $(b+1)n$-bit input and produce a bn-bit output, for $b \geq 2$.

While there are a large number of secure DBL compression functions, there is little research on generalized constructions. As a remedy, this chapter first introduces the class MBL^{bn} for MBL compression functions that employ a (bn, n)-bit keyed block cipher $E : \{0,1\}^{bn} \times \{0,1\}^n \rightarrow \{0,1\}^n$, and produce a bn-bit chaining value. Based on MBL^{bn}, we designed an MBL compression function, called Counter-bDM, which, to the best of our knowledge, is the first provably secure MBL compression function with freely scalable output size, i.e., it allows to choose $b > 2$. It is a generalization of the DBL compression function Hirose-DM [134]. Our description of Counter-bDM is accompanied by a thorough security analysis regarding its collision and preimage security, which employs the idea of super queries [28] for proofs of collision and preimage security. Similar approaches were presented by Mennink [184] and Lee [164]. For $b = 2$, our resulting collision-security bound shows that every adversary that wants to find a collision with advantage $1/2$ requires $2^{125.18}$ queries, which is

[1] Note that one can build an MBL block-cipher-based *hash function* by simply iterating an MBL block-cipher-based *compression function* over the message, where each call to the compression function processes one n-bit message block.

Compr. Function		Collision Bound		Preimage Bound	
WEIMAR-DM	[98]	$2^{126.73}$	[94]	2^{251}	[98]
HIROSE-DM	[133]	$2^{125.23}$	[99]	2^{251}	[28]
Counter-2DM	[Sec. 5.2]	$\mathbf{2^{125.18}}$	[Sec. 5.3]	$\mathbf{2^{251}}$	[Sec. 5.4]
ABREAST-DM	[163]	$2^{124.42}$	[100, 165]	2^{246}	[28]
TANDEM-DM	[163]	$2^{120.87}$	[168]	2^{246}	[28]
ADD-K-DM	[100]	$2^{127-k'}$	[100]	$\approx 2^{128}$	[100, 165]
CUBE-DM	[100]	$2^{125.41}$	[100]	$\approx 2^{128}$	[100, 165]
CYCLIC-DM ($k > 2$)	[100]	2^{127-k}	[100]	$\approx 2^{128}$	[100, 165]
CYCLIC-DM ($k = 2$)	[100]	$2^{124.55}$	[100]	$\approx 2^{128}$	[100, 165]
LEE/KWON	[166]	$2^{125.0}$	[165]	$\approx 2^{128}$	[100, 165]

Table 5.1: Comparison of security results on DBL compression (**Compr.**) functions, evaluated for $n = 128$ bits and a success probability of 1/2. The value k for CYCLIC-DM refers to the given cycle length.

comparable to the currently best collision-security bound of WEIMAR-DM [98]. Concerning preimage security, we obtain a near-optimal bound of 2^{251} queries, which is equivalent to the currently best bound of WEIMAR-DM and HIROSE-DM. Table 5.1 compares our results with previous DBL compression functions.

5.1 Related Work

Single-Block-Length Schemes. One of the most noteworthy works concerning SBL hash functions is that by Preneel, Govaerts, and Vandewalle [217] from CRYPTO'93. In their paper, the authors analyzed the security of SBL compression functions of the form $H_i = E_X(Y) \oplus Z$, where $X, Y, Z \in \{M_i, H_{i-1}, M_i \oplus H_{i-1}, const\}$, M_i denotes the currently processed message block, and H_{i-1} the previous chaining value. The authors identified twelve secure constructions among the 4^3 possibilities, which were resistant to relevant attacks. Though, formal proofs were shown first at CRYPTO'02 by Black, Rogaway, and Shrimpton [60], who concluded that 20 constructions, including the twelve identified by Preneel et al., provided optimal collision- and preimage-resistance. At FSE'04, Rogaway and Shrimpton discussed in detail the security notions for cryptographic hash functions and relationships among them [231]. One year later, Black, Cochran, and Shrimpton [58] showed that no rate-1 hash function, which makes only a single call for every message block to a fixed-key cipher, can achieve an optimal collision-resistance. At CRYPTO'08, Rogaway and Steinberger [232] presented the LP_{mkr} family of compression functions compressing an $m \cdot n$-bit input to an $r \cdot n$-bit input by using a sequence of k calls to a fixed-key block cipher. At FSE'09, Stam [247] generalized the considerations of Preneel et al., where he wrapped the block-cipher call with pre- and post-processing functions, considered chopped, overloaded, and supercharged versions of compression functions, and showed proofs for the collision- and preimage-security.

Double-Block-Length Schemes. The essentially first DBL hash functions were presented by Merkle [185], who proposed three constructions based on the DES. Today, there are four so-called "classical" DBL constructions, which were introduced in the early 1990s: MDC-2, MDC-4,

Abreast-DM, and Tandem-DM. MDC-2 and MDC-4 [72, 140] are (n, n)-bit DBL hash functions with rates 1/2 and 1/4, respectively. For MDC-2, Steinberger [248] proved in 2006 that no adversary asking less than $2^{74.9}$ queries will obtain a significant advantage at finding a collision. In a sophisticated proof, it was shown by Fleischmann, Forler, and Lucks [96] in 2012, that for MDC-4, an adversary requires at least $2^{74.7}$ queries to find a collision with an advantage of 1/2.

Concerning rate-1 DBL hash functions, Lucks [173] presented a first construction at Dagstuhl'07. Stam [247] also proposed a rate-1 single-call DBL function, for which he showed an almost-optimal collision-resistance, up to a logarithmic factor. However, while Lucks and Stam claimed a rate-1 property for their constructions, those are actually much slower, as pointed out by Luo and Lai [175]. At CRYPTO'93, Hohl et al. [136] analyzed the security of compression functions of rate-1/2 DBL hash functions. In 1998, Knudsen, Lai, and Preneel [155] discussed the security of rate-1 DBL hash functions. In 1999, Satoh, Haga, and Kurosawa [236] as well as Hattori, Hirose, and Yoshida [129] in 2003 attacked rate-1 DBL hash functions. At FSE'05, Nandi et al. [195] presented a rate-2/3 compression function, which was later analyzed by Knudsen and Muller at ASIACRYPT'05 [156]. At CT-RSA'11, Lee and Stam [167] presented a faster alternative to MDC-2, called MJH.

Double-Block-Length Schemes with Birthday-Type Collision Security. Abreast-DM and Tandem-DM base on the famous Davies-Meyer scheme, and have been presented by Lai and Massey [163] at EUROCRYPT'92. In 2004, Hirose added a large class of rate-1/2 DBL hash functions, composed of two independent $(2n, n)$-bit block ciphers, with $2n$ being the key and n the block size [132] . At FSE'06, he proposed a new scheme called Hirose-DM [133], which dropped the requirement of independent ciphers, and for which he provided a collision-security proof in the ideal-cipher model, stating that no adversary asking less than $2^{124.55}$ queries can find a collision with probability $\geq 1/2$.

In [213], Peyrin et al. analyzed techniques to construct larger compression functions by combining smaller ones. The authors proposed $3n$-to-$2n$-bit and $4n$-to-$2n$-bit constructions composed of five public functions, yet they did not show proofs for their concepts.

In 2008, Chang et al. introduced a generic framework for purf-based MBL constructions [68], where purf denotes a public random function.

Considering Tandem-DM, Fleischmann, Gorski, and Lucks [99] gave a collision-security proof at FSE'09, showing that no adversary can obtain a significant advantage without making at least $2^{120.4}$ queries. In 2010, Lee, Stam, and Steinberger [168] showed that the proof of Fleischmann et al. had several non-trivial flaws. Further, they provided a bound of $2^{120.87}$ queries for a collision adversary.

For Abreast-DM, Fleischmann et al. [100] as well as Lee and Kwon [165] presented, independent from each other, a collision-security bound of $2^{124.42}$ queries. More general, [100] introduced the class notion of Cyclic-DL, which included the constructions Abreast-DM, Cyclic-DM, Add-k-DM, and Cube-DM, and applied similar proofs for these. At IMA'09, Özen and Stam [208] proposed a framework for DBL hash functions by extending the generalized framework by Stam at FSE'09 for single-call hash functions. Still, their framework based on the usage of two independent block ciphers. At ProvSec'10, Fleischmann et al. [95] extended their general classification of DBL hash functions by the classes Generic-DL, Serial-DL, and Parallel-DL. For the framework by Özen and Stam, they relaxed the requirement of distinct independent block ciphers and gave collision bounds for Tandem-DM and Cyclic-DM. In [159], Krause, Armknecht, and Fleischmann provided techniques for proving asymptotically optimal preimage-resistance bounds for block-cipher-based double-length, double-call hash functions. They introduced a new Davies-Meyer DBL hash function for which they proved that no adversary asking less than 2^{2n-5} queries can find a preimage with probability $\geq 1/2$.

At ACISP'12, Fleischmann et al. [98] showed a very similar Davies-Meyer construction – called WEIMAR-DM – for which they could prove the currently best collision-security bound of $2^{126.23}$ queries, and the currently best preimage-security bound among the previously known DBL hash functions.

5.2 Specification of Counter-bDM

The design of a cryptographic algorithm, besides aspects of, e.g., performance and its cryptographic footprint, almost always aims at achieving the maximum possible security while maintaining an easy-to-grasp structure. For an MBL compression function, providing a security proof can be simplified greatly if one can ensure that the b outputs of the individual block-cipher calls in one invocation of the compression function are independent and distinct from each other. Previous DBL constructions achieve this requirement by the use of one of the following approaches:

Distinct Permutations. By using b independent permutations in the compression function. This approach is used, e.g., by the early construction of Hirose [132] or those by Rogaway and Steinberger [232].

Distinct Keys. By guaranteeing that all keys K_i used for the block-cipher calls inside one compression-function call are different: $K_i \neq K_j,\ 1 \leq i < j \leq b$, which results de facto in having different permutations. This approach is used, e.g., by WEIMAR-DM [98].

Distinct Plaintexts. By guaranteeing that all b plaintext inputs X_i used within the block-cipher calls in one compression function call are different: $X_i \neq X_j,\ 1 \leq i < j \leq b$. This approach is used, e.g., by CUBE-DM [100] or HIROSE-DM [134].

It is easy to see that the first approach is unpractical since it requires multiple permutation implementations of the class MBL^{bn}. The further two approaches are very similar. However, using a different key in every block-cipher call implies the potential need of running the key schedule of the underlying block cipher multiple times, which can become quite costly when providing a freely scalable output. Therefore, for the compression function COUNTER-bDM (short: H^{CbDM}), which is defined in this section, we employ the latter strategy ensuring that all plaintext inputs to the block-cipher calls are different (see Definition 5.1).

Definition 5.1 (Counter-bDM). *Let E be a block cipher from* $\texttt{Block}(bn, n)$*. The compression function* $H^{CbDM} : \{0,1\}^{bn} \times \{0,1\}^n \to \{0,1\}^{bn}$ *is defined as*

$$H^{CbDM}(M, U_1, \ldots, U_b) = (V_1, \ldots, V_b),$$

where the outputs V_i result from $V_i = E_K(U_1 \oplus (i-1)) \oplus U_1$, with $K = U_2 \mid\mid \ldots \mid\mid U_b \mid\mid M$.

Two concrete examples of our MBL compression-function family, COUNTER-3DM (left) and COUNTER-4DM (right), are illustrated in Figure 5.1. However, in our security analysis in Sections 5.3 and 5.4, we consider the generic COUNTER-bDM (H^{CbDM}).
It is easy to see that, due to the XOR with the counter $i-1$, all plaintext inputs X_i to the block-cipher calls are pair-wise distinct. Additionally, since all values $i-1$ are in the range of $[0, \ldots, b-1]$,

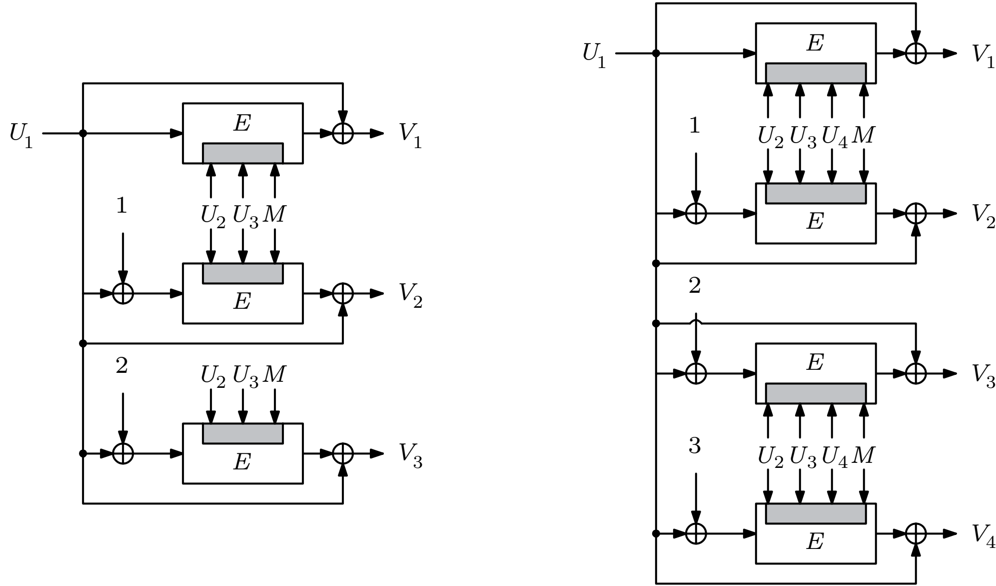

Figure 5.1: Two exemplary compression functions H^{C3DM} (left) and H^{C4DM} (right) from the family of compression functions $H^{\mathrm{C}b\mathrm{DM}}$.

the counter values affect only the least significant $\lceil log_2(b)\rceil$ bits of the plaintexts. Based on that, we denote the most significant $n - \lceil log_2(b)\rceil$ bits of each plaintext processed within one compression function as *common prefix* (see Definition 5.2).

Definition 5.2 (Common-Prefix Property). *Let $X = X_{pre} \mid\mid X_{post}$, $X \in \{0,1\}^n$ be an n-bit integer, where X_{pre} denotes the $n - \lceil log_2(b)\rceil$ most significant, and X_{post} the $\lceil log_2(b)\rceil$ least significant bits of X. Further, let $X_i = X \oplus (i-1)$ (with $1 \leq i \leq b$) denote the values which are used as plaintext inputs to the block-cipher calls in one invocation of H^{CbDM}. Then, all values X_i share the same common prefix $X_{pre} \in \{0,1\}^{n-\lceil log_2(b)\rceil}$.*

Remark 5.3. *Throughout the remainder of this paper, we denote by $c = 2^{\lceil log_2(b)\rceil} \geq b$ the maximal number of plaintexts $X = X_{pre} \mid\mid X_{post}$ which can share the same prefix X_{pre}.*

Later, we will see that both the pair-wise distinct plaintexts and the common-prefix property will be beneficial for an easy-to-grasp security analysis of Counter-bDM.

5.3 Collision-Security Analysis of Counter-bDM

We start with an adversary **A** that tries to find a collision provided with oracle access to a block cipher $E \twoheadleftarrow \texttt{Block}(bn, n)$ (see Experiment 2.6, Section 2). In between the communication of **A** and E, we construct another adversary **A′** which simulates **A**, but is sometimes allowed to make

additional queries to E that are not taken into account. Since $\mathbf{A}'$ is more powerful than $\mathbf{A}$, it suffices to upper bound the success probability of $\mathbf{A}'$. Thereby, we say that an adversary $\mathbf{A}$ or $\mathbf{A}'$ is *successful* if its query history contains the means of computing a collision for $H^{\mathrm{Cb}D\mathrm{M}}$.

Attack Setting. During the attack, $\mathbf{A}$ maintains a query history $\mathcal{Q}$ wherein it stores all queries it poses to E. An entry in the query history of $\mathbf{A}$ is a tuple (K, X, Y), where $Y = E_K(X)$. In the meantime, $\mathbf{A}'$ maintains a query list $\mathcal{L}$ which contains all input/output pairs to the compression function $H^{\mathrm{Cb}D\mathrm{M}}$ that can be computed by $\mathbf{A}$. An entry $L \in \mathcal{L}$ is a tuple $(K, X, Y_1, \ldots, Y_c) \in \{0,1\}^{(b+1+c)n}$, where $K \in \{0,1\}^{bn}$, $X \in \{0,1\}^n$ is the input to the compression function $H^{\mathrm{Cb}D\mathrm{M}}$, and $c = 2^{\lceil log_2(b) \rceil}$ (see Remark 5.3). The values $Y_i \in \{0,1\}^n$ are given as the results of the forward queries $Y_i = E_K(X \oplus (i-1))$, for $1 \leq i \leq c$.

Collision Events. When E is modeled as an ideal cipher, we run into problems when $\mathbf{A}$ is allowed to ask close to or even more than $q = 2^n$ queries. In the case when $\mathbf{A}$ asks q queries under the same key to E and q reaches $2^n - 1$, E loses its randomness. As a remedy to this problem, Armknecht et al. proposed the idea of *super queries* [28]; given some key K, $\mathbf{A}'$ can pose regular queries to E or D until $N/2$ queries with the same key K have been added to its query list $\mathcal{L}$, where $N = 2^n$.

If $\mathcal{L}$ contains $N/2$ queries for a key K and $\mathbf{A}$ requests another query for the key K from $\mathbf{A}'$, then, $\mathbf{A}'$ poses all remaining queries $(K, *, *)$ under this key to E at once. In this case, we say that a *super query* occurred. All queries that are part of a super query *are not taken into account*, i.e., they do not add to q, the number of queries $\mathbf{A}$ is allowed to ask. Since these free queries are asked at once, one no longer has to consider the success probability of a single query; instead, one can consider the event that $\mathbf{A}'$ is successful with any of the contained queries. Thus, E does not lose its randomness.

> **Remark 5.4.** *Note that a tuple $L \in \mathcal{L}$ consists of $c = 2^{\lceil log_2(b) \rceil}$ query results. Since c always divides $N/2$, i.e., $c \,|\, N/2$, each tuple L is either part of a normal query or a super query, but never both.*

In the following, we distinguish between three events for the case when $\mathbf{A}'$ is successful.

NormalQueryWin($\mathcal{L}$): This describes the case when $\mathbf{A}'$ finds a collision from its current query L^j and a query L^r that was already in its list, where L^j was a normal query.

SuperQueryWin($\mathcal{L}$): This describes the case when $\mathbf{A}'$ finds a collision with its current query L^j and a query L^r that was already in its list, where L^j was part of a super query.

SameQueryWin($\mathcal{L}$): This describes the case when $\mathbf{A}'$ finds a collision within one entry of the query history.

Since the adversary can only win if it finds a collision using either one of the mentioned events, it is sufficient for us to upper bound the sum of the probabilities. Thus, it holds that

$$\mathbf{Adv}^{\mathrm{COLL}}_{H^{\mathrm{Cb}D\mathrm{M}}}(q) \leq \Pr[\mathsf{NormalQueryWin}(\mathcal{L})] + \Pr[\mathsf{SuperQueryWin}(\mathcal{L})] + \Pr[\mathsf{SameQueryWin}(\mathcal{L})]. \quad (5.1)$$

Before we present our bound, we describe more precisely what we mean by $\mathbf{A}'$ has found a collision for $H^{\mathrm{C}b\mathrm{DM}}$. Let $L^r = (K^r, X^r, Y_1^r, \ldots, Y_c^r)$ represent the r-th entry in $\mathcal{L}$, and $L^j = (K^j, X^j, Y_1^j, \ldots, Y_c^j)$ the j-th entry in $\mathcal{L}$, where $1 \leq r < j$. We say that L^r and L^j provide the means for computing a collision if $\exists\, \ell, m \in \{0, \ldots, c-1\}$ so that b equations of the following form hold:

$$\begin{aligned} E_{K^r}(X^r \oplus m \oplus 0) \oplus X^r &= E_{K^j}(X^j \oplus \ell \oplus 0) \oplus X^j, \\ E_{K^r}(X^r \oplus m \oplus 1) \oplus X^r &= E_{K^j}(X^j \oplus \ell \oplus 1) \oplus X^j, \\ &\vdots \\ E_{K^r}(X^r \oplus m \oplus (b-1)) \oplus X^r &= E_{K^j}(X^j \oplus \ell \oplus (b-1)) \oplus X^j. \end{aligned}$$

Theorem 5.5 (Collision Security of $H^{\mathbf{C}b\mathbf{DM}}$). *Let $N = 2^n$ and let $\mathbf{A}$ be an adversary as defined in Experiment 2.6 with oracle access to a block cipher $E \twoheadleftarrow \mathtt{Block}(bn, n)$, so that $\mathbf{A}$ tries to find a collision for H^{CbDM} as given in Definition 5.1. Then, it applies that*

$$\mathbf{Adv}^{COLL}_{H^{CbDM}}(q) \;\leq\; \frac{c^2 \cdot 2^b \cdot q^2}{N^b} + \frac{c^3 \cdot 2^{b+2} \cdot q^2}{N^{b+1}},$$

where q is the maximum number of queries $\mathbf{A}$ is allowed to ask to E.

Proof. After $\mathbf{A}$ has mounted a (normal) forward query $Y = E_K(X)$ or a (normal) backward query $X = D_K(Y)$, $\mathbf{A}'$ checks if $\mathcal{L}_{j-1}$ already contains an entry $L = (K, X_{pre} \mid\mid *, *, \ldots, *)$, where X_{pre} denotes the prefix of X (see Definition 5.2) and $*$ denotes arbitrary values. In the following, we analyze the possible cases that $\mathbf{A}'$ can be confronted with and upper bound their individual success probabilities.

***Case 1:* L is not in $\mathcal{L}_{j-1}$.** In this case, $\mathbf{A}'$ labels Y as Y_1 and asks $(c-1)$ further queries to E that are not taken into account:

$$\forall i \in \{2, \ldots, c\}: \quad Y_i = E_K(X \oplus (i-1)).$$

$\mathbf{A}'$ creates the tuple $L^j = (K, X, Y_1, \ldots, Y_c)$ and appends it to its query list, i.e., $\mathcal{L}_j = \mathcal{L}_{j-1} \cup \{L^j\}$. Now, we have to upper bound the success probability of $\mathbf{A}'$ to find a collision for $H^{\mathrm{C}b\mathrm{DM}}$, i.e., the success probabilities for the events mentioned above.

***Subcase 1.1:* NormalQueryWin($\mathcal{L}$).** In this case, the adversary finds a collision using a normal query L^j and a query L^r that was already contained in $\mathcal{L}$. While super queries may have occurred for different keys before, the query history of $\mathbf{A}'$ may contain at most $N/2 - c$ plaintext-ciphertext pairs for the current key K^j. So, our random permutation E samples the query responses $Y_1^j, \ldots, Y_c^j$ for the current query at random from a set of size of at least $N/2 + c \geq N/2$ elements. Hence, the probability that one equation from above holds for some fixed ℓ and m can be upper bounded by $1/(N/2)$; and the probability for b equations to hold is then given by

$$\frac{1}{(N/2)^b} = \frac{2^b}{N^b}.$$

There are c^2 possible combinations for ℓ and m, s.t. b values V_i^j can form a valid collision with b values V_i^r, with $i \in \{0, \dots, b-1\}$. Thus, $\mathbf{A}'$ has a success probability for finding a collision for $H^{\mathrm{C}b\mathrm{DM}}$ with the j-th query and one specific previous query of

$$\frac{c^2}{(N/2)^b} = \frac{c^2 \cdot 2^b}{N^b}.$$

The j-th query can form a collision with any of the previous entries in $\mathcal{L}_{j-1}$. So, we have to determine the maximum number of queries in $\mathcal{L}_{j-1}$. If $\mathbf{A}'$ obtained a super query for each key it queried before, $\mathcal{L}_{j-1}$ may contain up to $2(j-1)$ entries. For q total queries, there may be at most q normal queries and up to $q-1$ queries (without the current one) resulting from super queries in the history. This would imply that one had to sum up the probabilities up to $2q-1$:

$$\sum_{j=1}^{2q-1} \frac{2(j-1) \cdot c^2 \cdot 2^b}{N^b}.$$

However, we can do better. In the $\mathsf{NormalQueryWin}(\mathcal{L})$ case, $\mathbf{A}'$ will not win when its last (winning) query was part of a super query. Hence, we do not need to test if any of the super queries will produce a collision with any of their respective previous queries, and we have to test only the q normal queries. Nevertheless, $\mathbf{A}'$ still has to test each of the q normal queries if they collide with any of the at most $2q$ previous queries (including those which were part of a super query). Therefore, the success probability of $\mathbf{A}'$ to find a collision for $H^{\mathrm{C}b\mathrm{DM}}$ can be upper bounded by

$$\Pr[\mathsf{NormalQueryWin}(\mathcal{L})] \leq \sum_{j=1}^{q} \frac{2(j-1) \cdot c^2 \cdot 2^b}{N^b} \leq \frac{q^2 \cdot c^2 \cdot 2^b}{N^b}. \tag{5.2}$$

***Subcase 1.2:* SuperQueryWin($\mathcal{L}$).** In this case, $\mathbf{A}'$ wins with a super query, i.e., it has asked the $(N/2+1)$-th query for K^j, triggering a super query to occur. We can reuse the argument from Subcase 1.1 that the success probability for b colliding equations from above is given by

$$\frac{c^2}{(N/2)^b} = \frac{c^2 \cdot 2^b}{N^b}.$$

Here, the query history $\mathcal{L}_j$ contains at most $2q$ queries. However, this time we do not have to test if any of the q normal queries produce a collision with any of their respective predecessors. Hence, we can upper bound the success probability of $\mathbf{A}'$ to find a collision for $H^{\mathrm{C}b\mathrm{DM}}$ with one super query by

$$\frac{2q \cdot c^2 \cdot 2^b}{N^b}.$$

For a super query to occur, $\mathbf{A}$ has to pose at least $N/(2c)$ regular queries. Thus, there can be at most $q/(N/2c)$ super queries and we obtain

$$\Pr[\mathsf{SuperQueryWin}(\mathcal{L})] \leq \frac{2q \cdot c^2 \cdot 2^b}{N^b} \cdot \frac{q}{N/2c} = \frac{c^3 \cdot 2^{b+2} \cdot q^2}{N^{b+1}}. \tag{5.3}$$

***Subcase 1.3:* SameQueryWin($\mathcal{L}$).** In this case, $\mathbf{A}'$ wins if it finds some integers $\ell, m \in \{0, \dots, c-1\}$ with $\ell \neq m$ s.t.:

$$\begin{aligned}
E_{K^j}(X^j \oplus m \oplus 0) \oplus X^j &= E_{K^j}(X^j \oplus \ell \oplus 0) \oplus X^j, \\
E_{K^j}(X^j \oplus m \oplus 1) \oplus X^j &= E_{K^j}(X^j \oplus \ell \oplus 1) \oplus X^j, \\
&\vdots \\
E_{K^j}(X^r \oplus m \oplus (b-1)) \oplus X^j &= E_{K^j}(X^j \oplus \ell \oplus (b-1)) \oplus X^j.
\end{aligned}$$

n = 64			n = 128		
#blocks b	#queries q	optimal bound $2^{bn/2}$	#blocks b	#queries q	optimal bound $2^{bn/2}$
2	$2^{61.50}$	2^{64}	2	$2^{125.50}$	2^{128}
3	$2^{92.00}$	2^{96}	3	$2^{188.00}$	2^{192}
4	$2^{123.50}$	2^{128}	4	$2^{251.50}$	2^{256}
5	$2^{154.00}$	2^{160}	5	$2^{314.00}$	2^{320}
6	$2^{185.50}$	2^{192}	6	$2^{377.50}$	2^{384}
7	$2^{217.00}$	2^{224}	7	$2^{441.00}$	2^{448}
8	$2^{248.50}$	2^{256}	8	$2^{504.50}$	2^{512}

Table 5.2: Minimum number of queries q which an adversary is required to ask to an oracle E in order to find a collision for $H^{\mathrm{C}b\mathrm{DM}}$ with advantage 1/2, depending on the choice of b and n.

However, due to the XOR with the distinct values $i-1$, all plaintext inputs $X^j \oplus (i-1)$ in one compression-function call differ from each other. Furthermore, since all plaintext inputs are encrypted under the same key K^j, their corresponding outputs Y_i are all different and uniformly distributed, and so are the values $Y_i \oplus X$ after the feed-forward operation. Hence, it is not possible for $\mathbf{A}'$ to find a collision for $H^{\mathrm{C}b\mathrm{DM}}$ among the values $Y_i \oplus X$:

$$\Pr[\mathsf{SameQueryWin}(\mathcal{L})] = 0. \tag{5.4}$$

***Case 2:* L is in $\mathcal{L}_{j-1}$.** In this case, the key K and the plaintext prefix X_{pre} of $\mathbf{A}$'s current query $(K, X_{pre} \mid\mid X_{post'})$ are already stored in some entry $L \in \mathcal{L}_{j-1}$, where L is given by $(K, X_{pre} \mid\mid X_{post}, Y_1, \ldots, Y_c)$. $\mathbf{A}'$ just extracts $Y_{(X_{post} \oplus X_{post'})+1}$ from L, and passes it to $\mathbf{A}$. This implies that $\mathbf{A}$ can learn only information which $\mathbf{A}'$ already possesses.

Based on Equation (5.1), our claim is given by summing up Equations (5.2), (5.3), and (5.4). □

In Table 5.2, we show the minimal number of queries q an adversary has to ask in order to obtain an advantage of $\mathbf{Adv}^{\mathrm{COLL}}_{H^{\mathrm{C}b\mathrm{DM}}}(q) = 1/2$ for the most practical block lengths $n \in \{64, 128\}$ and depending on b.

5.4 Preimage-Security Analysis of Counter-bDM

Attack Setting. Let $(V_1, \ldots, V_b) \in \{0,1\}^{bn}$ be the point to invert (see Definition 5.1), chosen by an adversary $\mathbf{A}$ before it makes any query to E. We define that $\mathbf{A}$ has the goal to find a preimage for the point $(V_1, \ldots, V_b)$ as described in Experiment 2.7 (see Section 2). For our preimage-security analysis, we adapt the procedure from our collision analysis, i.e., we construct another adversary $\mathbf{A}'$, which simulates $\mathbf{A}$, but sometimes is allowed to make additional queries to E that are not taken into account. Again, since $\mathbf{A}'$ is more powerful than $\mathbf{A}$, it suffices to upper bound the success probability of $\mathbf{A}'$. Here, we say that an adversary is *successful* if its query history $\mathcal{Q}$ contains the means of computing a preimage.

The procedures of $\mathbf{A}$ and $\mathbf{A}'$ asking queries to the oracle E and building the query histories $\mathcal{Q}$ and $\mathcal{L}$ are the same as that described in our collision-security proof. Furthermore, we adopt

the events $\mathsf{NormalQueryWin}(\mathcal{L})$ and $\mathsf{SuperQueryWin}(\mathcal{L})$ from there, which in this context, cover all possible winning events for $\mathbf{A}'$. Thus, it holds that

$$\mathbf{Adv}^{\mathrm{EPRE}}_{H^{\mathrm{CbDM}}}(q) \;\leq\; \Pr[\mathsf{NormalQueryWin}(\mathcal{L})] + \Pr[\mathsf{SuperQueryWin}(\mathcal{L})]. \tag{5.5}$$

Before we present our bound, we describe more precisely what is meant by $\mathbf{A}'$ has found a preimage for $H^{\mathrm{C}b\mathrm{DM}}$. Let $L^j = (K^j, X^j, Y^j_1, \ldots, Y^j_c)$ represent the j-th entry in $\mathcal{L}$. We say that L^j contains the means of computing a preimage if $\exists\, \ell \in \{0, \ldots, c-1\}$, so that the following b equations hold:

$$\begin{aligned} E_{K^j}(X^j \oplus \ell) \oplus X^j &= V_1 \\ E_{K^j}(X^j \oplus \ell \oplus 1) \oplus X^j &= V_2 \\ &\vdots \\ E_{K^j}(X^j \oplus \ell \oplus (b-1)) \oplus X^j &= V_b. \end{aligned}$$

Theorem 5.6 (Preimage Security of $H^{\mathbf{C}b\mathbf{DM}}$). *Let $N = 2^n$ and let $\mathbf{A}$ be an adversary as defined in Experiment 2.7 with oracle access to a block cipher $E \twoheadleftarrow \mathtt{Block}(bn, n)$, where $\mathbf{A}$ tries to find a collision for H^{CbDM} as given in Definition 5.1. Then, it applies that*

$$\mathbf{Adv}^{EPRE}_{H^{CbDM}}(q) \;\leq\; \frac{c \cdot 2^{b+1} \cdot q}{N^b}.$$

Proof. After $\mathbf{A}$ has mounted a (normal) forward query $Y = E_K(X)$ or a (normal) backward query $X = D_K(Y)$, $\mathbf{A}'$ checks if $\mathcal{L}_{j-1}$ already contains an entry $L = (K, X_{pre} \mid\mid *, *, \ldots, *)$, where X_{pre} denotes the prefix of X. In the following, we analyze the possible cases and upper bound their success probabilities separately.

***Case 1:* L is not in $\mathcal{L}_{j-1}$.** In this case, $\mathbf{A}'$ labels Y as Y_1 and asks $c-1$ further queries to E that are not taken into account:

$$\forall i \in \{2, \ldots, c\}: \quad Y_i \;=\; E_K(X \oplus (i-1)).$$

Then, $\mathbf{A}'$ creates the tuple $L^j = (K, X, Y_1, \ldots, Y_c)$ and appends it to its query list, i.e., $\mathcal{L}_j = \mathcal{L}_{j-1} \cup \{L^j\}$. Note that due to the XOR with $i-1$, all plaintexts X_i are different and thus, all ciphertexts Y_i and the results of all feed-forward operations $Y_i \oplus X$ are always uniformly distributed.

Now, we have to upper bound the success probability of $\mathbf{A}'$ to find a preimage for $H^{\mathrm{C}b\mathrm{DM}}$ using either a normal query or a super query.

***Subcase 1.1:* $\mathsf{NormalQueryWin}(\mathcal{L})$.** Since we assume that the winning query is a normal one, $\mathbf{A}'$ can have collected at most $N/2 - c$ queries for the current key K^j. Thus, E samples the query responses $Y^j_1, \ldots, Y^j_c$ at random from a set of size of at least $N/2 + c \geq N/2$ elements. From the c values Y_i of L^j, the probability that one equation $E_{K^j}(X^j \oplus \ell) \oplus (X^j \oplus \ell) \;=\; V_i$ from above holds for some fixed value of ℓ, can therefore be upper bounded by $1/(N/2)$ – and the probability that b equations from above hold for a fixed ℓ by $1/(N/2)^b$. Since there are now c possible values for ℓ, the probability to obtain a preimage with the j-th query is given by

$$\frac{c}{(N/2)^b} \;=\; \frac{c \cdot 2^b}{N^b}.$$

n = 64			**n = 128**		
#blocks b	**#queries** q	**optimal bound** 2^{bn}	**#blocks** b	**#queries** q	**optimal bound** 2^{bn}
2	2^{123}	2^{128}	2	2^{251}	2^{256}
3	2^{185}	2^{192}	3	2^{377}	2^{384}
4	2^{248}	2^{256}	4	2^{504}	2^{512}
5	2^{310}	2^{320}	5	2^{630}	2^{640}
6	2^{373}	2^{384}	6	2^{757}	2^{768}
7	2^{436}	2^{448}	7	2^{884}	2^{896}
8	2^{499}	2^{512}	8	2^{1011}	2^{1024}

Table 5.3: Minimum number of queries q, which an adversary is required to ask to an oracle E in order to find a preimage for H^{CbDM} with advantage 1/2, depending on the choice of b and n.

Since $\mathbf{A}'$ is allowed to ask at most q queries, it applies that

$$\Pr[\mathsf{NormalQueryWin}(\mathcal{L})] \leq \frac{c \cdot 2^b \cdot q}{N^b}. \tag{5.6}$$

Subcase 1.2: **SuperQueryWin($\mathcal{L}$).** In this case, $\mathbf{A}'$ has already posed and stored $N/2c$ queries for the key K^j of its winning query. From the super query, it obtains the remaining $N/2c$ queries for K^j. We denote the latter set of queries by $\mathcal{SQ}$. From above, we already know that the probability that one point $L^j \in \mathcal{SQ}$ satisfies the preimage property can be upper bounded by

$$\frac{c}{(N/2)^b} = \frac{c \cdot 2^b}{N^b}.$$

Since the adversary obtains $N/2c$ points from the super query, the success probability is given by

$$\frac{N}{2c} \cdot \frac{c \cdot 2^b}{N^b} = \frac{2^{b-1}}{N^{b-1}}.$$

For every super query to occur, $\mathbf{A}'$ has to collect $N/2c$ queries in advance. Thus, there are at most $q/(N/2c)$ super queries and we obtain

$$\Pr[\mathsf{SuperQueryWin}(\mathcal{L})] \leq \frac{q}{N/2c} \cdot \frac{2^{b-1}}{N^{b-1}} = \frac{c \cdot 2^b \cdot q}{N^b}. \tag{5.7}$$

Case 2: L **is in** $\mathcal{L}_{j-1}$**.** Like in the Case 2 of our collision-security proof, the key K and the plaintext prefix X_{pre} of $\mathbf{A}$'s current query $(K, X_{pre} \;||\; X_{post'})$ are already stored in some entry $L \in \mathcal{L}_{j-1}$, where $L = (K, X_{pre} \;||\; X_{post},\, Y_1, \ldots, Y_c)$. Again, $\mathbf{A}'$ extracts $Y_{(X_{post} \oplus X_{post'})+1}$ from L, and passes it to $\mathbf{A}$, so that $\mathbf{A}$ can learn only information which $\mathbf{A}'$ already possesses.

Based on Equation (5.5), our claim is given by summing up Equations (5.6) and (5.7). □

For $n = 128$ and $\mathbf{Adv}^{\text{EPRE}}_{H^{\text{CbDM}}}(q) = 1/2$, we list in Table 5.3 the amounts of queries q an adversary has to make, depending on the value of b.

5.5 Summary

This chapter introduced Counter-bDM – the first provably secure family of MBL compression function, that maps $(b+1)n$-bit inputs to bn-bit outputs for arbitrary $b \geq 2$. With Counter-bDM, we propose a simple, though, very neat design, that not only avoids costly requirements such as the need of having independent ciphers, or having to run the key schedule multiple times, but also simplifies the analysis greatly. In our collision- and preimage-security analysis, we provided proofs for arbitrary block lengths $b \geq 2$. Moreover, to prove collision and preimage resistance beyond the birthday bound, we employed the idea of super queries by Armknecht et al. [28].

Part III

Password Hashing

CHAPTER 6

Password Scramblers and Attacks on `scrypt`

Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.

ALBERT EINSTEIN

Within this chapter, we first provide a survey of the history of password hashing. Afterwards, we elaborate on the desired properties for a modern Password Scrambler (PS). This can be seen as an extension of the properties presented already in Chapter 3. Subsequently, we introduce three attacks on the widely used PS `scrypt` [211]), namely, Cache-Timing Attack (CTA), Garbage-Collector Attack (GCA), and Weak Garbage-Collector Attack (WGCA), that – among other things – built the basic motivation for designing CATENA and its instances presented in the remainder of this part. Note that Sections 6.3 to 6.5 are adopted almost literally from [108] and [101], respectively.

6.1 A Short History of Password Hashing

Long before PCs became popular in everyone's home, Wilkes realized in the late 1960s that storing an authentication password in plaintext form is insecure [261]. About 10 years later, UNIX systems integrated some of Wilkes ideas [191] by deploying the DES-based [78] one-way encryption function [117] `crypt` which hashes a user-given password. The function `crypt` already fulfilled three of the main goals of a well-designed PS:

- `crypt` makes use of a **salt** value s which is a uniformly at random chosen public input stored alongside of the password pwd and is used to thwart adversaries which are in the possession of password-hash values from many different users. For example, they protect well against the usage of rainbow tables [203].
- `crypt` is a **one-way function**, i.e., there is no efficient way to recover the password pwd from the given hash $h \leftarrow$ `crypt`(pwd, s), where s is distinct for each user, thus, providing unique

password hashes h.[1]

- `crypt` provides **key stretching**. Briefly explained, by iterating the underlying (cryptographic) primitive many times, it protects low-entropy passwords by increasing the computational time to obtain its corresponding hash. On the one hand, the impact on the defender is given by only slightly increase costs while, on the other hand, a typical brute-force attack of all password candidates experiences a strong slow-down (see Section 6.2 for more details). In 1997, Kelsey et al. [152] discussed the approach of hiding the iteration count r, which has to be guessed by an adversary. A user-specific count r was proposed by Boyen in 2007 [63]. There, the algorithm would run until the user stops it by hand (pressing a button on the keyboard).

Now, we discuss frequently used password scramblers which build the basis for the modern password scramblers discussed in the next section. Table 6.1 provides an overview of password scramblers that are or have been in frequent use. It indicates the amount of memory used, the cost factor to generate a password hash, and issues from which the considered PS suffered.

Hash-Function-Based Password Scramblers. After the introduction of `crypt`, which was based on the block cipher DES, almost all password scramblers started to utilize cryptographic hash functions as the underlying primitive. The PS that started that development in 1995 was `md5crypt` [149] by Kamp, which is based on the hash function MD5 [223] and performs key stretching with 1000 runs. It was the common PS used in Free-BSD and Linux-based systems until its own author did not consider it secure anymore [149]. The decision was mainly motivated by its performance.[2] Therefore, it was superseded by `sha512crypt` [82], which employs the cryptographic hash function SHA-512 [207] and allows to freely choose the iteration count r (default: $r = 5000$) to render it more secure and flexible than `md5crypt`.

In contrast to Linux- and UNIX-based systems, the operating system Windows hashed their passwords with algorithms from the NT LAN Manager (NTLM) suite of security protocols, i.e., `NTLMv1` and `NTLMv2` [115].[3] Due to their high performance [30], even in comparison to `md5crypt` or `sha512crypt`, we do not recommend its usage.

In 2000, the design Password-Based Key Derivation Function 2 (PBKDF2) [148] was published and later, in 2010, standardized by the National Institute of Standards and Technology (NIST) [258]. It is widely used either as a Key-Derivation Function (KDF) (e.g., in WPA, WPA2, OpenOffice, or WinZip) or as a PS (e.g., in Mac OS X, LastPass). The security of PBKDF2 relies on c iterations of HMAC-SHA-1 [160], where c is set to 1000 for default.

Next, we have a brief look at the first two password scramblers which actually require a certain amount of memory for computing the password hash, i.e., `bcrypt` [219] and `scrypt`. Note that we consider `bcrypt` not as a modern PS since it does not consider the memory to be of variable size, whereas `scrypt` does.

In contract to the aforementioned, `bcrypt` [219] is not based on a (cryptographic) hash function but on the block cipher Blowfish [240]. The PS `bcrypt` maintains an internal state of 4,168 bytes which is generated by employing the Blowfish key schedule. Thus, even if it does not allow user-chosen values for the size of that state, `bcrypt` provides some resistance to adversaries utilizing GPUs, e.g., an AMD HD 7970 can only compute about 4,000 password hashes per second [250].

[1] Although, `crypt` offers a mechanism to avoid storing plain passwords, many companies do not seem to have interest in protecting confidential data of their users, e.g., Yahoo, RockYou, CSDN (China Software Developer Network), and many others, which used to store the passwords of their users in plain [176].

[2] For example, it is possible to compute about five million password hashes per second on an AMD HD 6990 [250].

[3] `NTLMv1` was used until Windows 2000 included the native support for `NTLMv2`. Today, Microsoft recommends not to use the latter anymore since it "does not support any recent cryptographic methods, such as AES or SHA-256." [187].

Algorithm	Time	Memory	Issues
`crypt`	25	small	“too fast”
`md5crypt`	1,000	small	“too fast”
`sha512crypt`	1,000–999,999 (5,000)	small	small memory
`NTLMv1/NTLMv2`	1	small	“too fast”
`PBKDF2`	1–∞ (1,000)	small	small memory
`bcrypt`	2^4–2^{99} (2^6, 2^8)	4,168 bytes	constant memory
`scrypt`	1–∞ (2^{14}, 2^{20})	flexible, big	CTA, GCA, WGCA

Table 6.1: Comparison of state-of-the-art password scramblers. Note that all of the mentioned algorithms support salt values. The term $1-\infty$ denotes that the cost factor can be chosen arbitrarily large. A comparison of *modern* password scramblers can be found in Chapter 11. Furthermore, neither of the above provide Server Relief (SR) or Client-Independent Updates (CIUs).

However, due to the inexorable growth of fast cache memory in upcoming CPUs and GPUs, `bcrypt` will be efficiently computable in the future. For key stretching, `bcrypt` invokes the Blowfish key schedule 2^c times, e.g., OpenBSD uses $c = 6$ for users and $c = 8$ for the superuser [219].

It is easy to see that occupying a lot of memory hinders attacks using special-purpose hardware (memory is expensive) and GPUs. Due to the provision of (sequential) memory hardness (see Definitions 3.5 and 3.7) and the possibility to adjust the required memory, the PS `scrypt` [211] can be seen as the first *modern* PS.[4] A description of `scrypt` and attacks exploiting its memory handling are provided in Section 6.3.

6.2 Properties of Modern Password Scramblers

Before the publication of `scrypt`, the most extensive form of protection one could guarantee for securing a password was given by performing *key stretching* (see Section 6.1), i.e., the iterative invocation of a (cryptographic) hash function processing the password to obtain the password hash. That technique is employed in widely used password scramblers, such as `md5crypt` [149] and `sha512crypt` [82].

Key Stretching. In general, we assume so-called “off-line” adversaries which are in the possession of the password hash of the original password. Let x be the original password with b bits of entropy, and let H be a cryptographic hash function. An adversary who knows the password hash $y_1 = H(x)$ can expect to find x by trying out about 2^{b-1} password candidates in average. To slow down the adversary by a factor of 2^c, one iterates the hash function 2^c times by computing $y_i = H(y_{i-1})$ for $i \in \{2, \ldots, 2^c\}$ and uses y_{2^c} as the final password hash. This forces the adversary to call the hash function 2^{c+b}, rather than 2^b times. But, the defender is also slowed down by 2^c. Note that the computational time for scrambling a password is bounded by the tolerance of the user, and so is the choice of the parameter c. Thus, there is no protection against password-cracking adversaries for users with low-entropy passwords[5]. Furthermore, in the rare case that a user has a high-entropy password (say, $c > 100$), key stretching is unnecessary. But, for users with mid-entropy passwords,

[4] There was HEKS [221], but it was broken by the author of `scrypt`.

[5] A study from 2012 reports a min-entropy of $c < 7$ bits for typical user groups [62]. For any such group, an adversary trying the group’s most frequent password succeeds for $\approx 1\,\%$ of the users.

key stretching can provide a decent protection. Thus, we define the basic conditions for any password scrambler PS as follows[6]

(1) Given a password *pwd*, computing $P(pwd)$ should be "fast enough" for the user.

(2) Computing $P(pwd)$ should be "as slow as possible" without contradicting (1).

(3) Given $y \leftarrow P(pwd)$, there must be no significantly faster way to test q password candidates $x_1, \ldots, x_q$ for $P(x_i) = y$ than by actually computing $P(x_i)$ for each candidate x_i.

Memory-Demanding Key Stretching. Due to the fact that the quality of passwords did not increase significantly over the last years [62], and due to the vast number of leaked password hashes [176], the established approach of performing key stretching by iterating a conventional primitive many times has become less useful. The reason is an increasing asymmetry between the computational devices the typical "defender" is using, and those devices available for potential adversaries. Even without special-purpose hardware, GPUs with hundreds of cores [202] have become a commodity. By making plenty of computational resources available, GPUs are excellent tools for password cracking since each core can try another password candidate, and all cores can run at full speed. However, the memory – and, especially, fast ("cache") memory – on a typical GPU is about as large (at least by the order of magnitude) as the memory and cache on a typical CPU as used by typical defenders. So, the idea behind a memory-demanding PS is to perform key stretching with the following requirements:

(4) Scrambling a password in time T needs S units of memory (and causes a strong slow-down when given less than S units of memory).

(5) Scrambling p passwords in parallel needs $p \cdot S$ units of memory (or causes a strong slow-down accordingly with less memory).

(6) Scrambling a password on p parallel cores is not (much) faster than on a single core, even if S units of memory are available.

Note that a defender can determine S and T by selecting appropriate parameters.

Simplicity and Resilience. The first published memory-demanding password scrambler (implicitly based on the six conditions above) is `scrypt` [211]. However, we considered two aspects of `scrypt` as critical issues: first, `scrypt` is complex since it combines two independent cryptographic primitives (the SHA-256 hash function and the core operation of the stream cipher Salsa20/8 [46]) and four generic functions (HMAC, PBKDF2, `BlockMix`, and `ROMix`). Second, the data flow of the `ROMix` operation is data-dependent, i.e., `ROMix` reads data from password-dependent addresses. This renders `ROMix`, and therefore `scrypt`, vulnerable to Cache-Timing Attacks (CTAs) as described in Section 6.4. Moreover, in Section 6.5, we have shown that `scrypt` is also vulnerable to Garbage-Collector Attack (GCA). Based on these findings, we pose that a modern PS should be:

(7) easy to analyze,

(8) resilient to Cache-Timing Attacks, and

(9) resilient to Garbage-Collector Attacks.

[6] Note that, for simplicity, we ignore the additional inputs of salt s and the optional set $\mathcal{T}$ as given in Definition 3.4.

"Easy to analyze" can mean a lot of things and we do not restrict this term to a certain property. Nevertheless, with the high complexity of `scrypt` in mind, one possible approach could be the dependence of a PS on a single cryptographic primitive. To satisfy Property (8), one has to ensure that neither the control flow nor the data flow depend on secret inputs, e.g., the password. One way to satisfy Property (9) is to read and (over)write the memory a couple of times during the scrambling operation. An adversary that tries to gain information from the utilized memory will then learn only the information written at the end of the invocation of the PS.

Overview of Properties. To provide a comprehensive overview of all desired properties, we briefly restate and summarize those from Chapter 3 together with the requirements stated by the PHC [31]:

Length of the Password: Allow passwords of any length between 0 and 128 bytes.

Length of the Salt: Allow salts of 16 bytes.

Output Length: Produce (but not limited to) 32-byte outputs.

Optional Inputs: Support inputs such a personalization string, a secret key, or any application-specific parameter.

Client-Independent Update: Allow to adjust (increase) the security parameters, even without knowing the password.

Server Relief: The option to shift the main memory and time effort from an authentication server to the client, without burdening the server,

Key-Derivation Function: The output of a PS should not be distinguishable from a random string of the same length.

Pepper and Garlic: PS should provide security parameters to adjust time and memory requirements,

Cryptographic Security: Provide preimage resistance, collision resistance, immunity to length extension, and infeasibility to distinguish outputs from random.

6.3 The Password Scrambler scrypt

As mentioned before, `scrypt` is based on a complex twist of cryptographic primitives. Nevertheless, this section mainly focuses on its core operation `ROMix`, which is, together with `scrypt`, depicted in Algorithm 1. To support inputs and outputs of arbitrary lengths, `scrypt` employs a so-called pre- and post-processing by invoking PBKDF2 [148]. The core function `ROMix` utilizes a hash function H with n output bits, where n is the size of a cache line (at current machines usually 64 bytes [73]). To support hash functions with smaller output sizes, [211] proposes to instantiate H by a function called `BlockMix`, which we will not elaborate on. For our security analysis of `ROMix`, we model H as a random oracle.

`ROMix` takes two inputs: an initial state x, which depends on both salt and password, and the array size G that defines the required number of memory units. In the first phase (Lines 20-22 of Algorithm 1), `ROMix` initializes an array v. More detailed, the array variables $v_0, v_1 \ldots, v_{G-1}$ are

Algorithm 1 The `scrypt` algorithm and its core operation `ROMix` [211].

Require:
- pwd ▷ password
- s ▷ salt
- G ▷ cost parameter

Ensure: x ▷ password hash

procedure `scrypt` (pwd,s,G)
10: $x \leftarrow \text{PBKDF2}(pwd, s, 1, 1)$
11: $x \leftarrow \texttt{ROMix}(x, G)$
12: $x \leftarrow \text{PBKDF2}(pwd, x, 1, 1)$
13: **return** x

Require:
- x ▷ initial state
- G ▷ cost parameter

Ensure: x ▷ hash value

procedure `ROMix` (x, G)
20: **for** $i = 0, \ldots, G-1$ **do**
21: $\quad v_i \leftarrow x$
22: $\quad x \leftarrow H(x)$
23: **for** $i = 0, \ldots, G-1$ **do**
24: $\quad j \leftarrow x \bmod G$
25: $\quad x \leftarrow H(x \oplus v_j)$
26: **return** x

Algorithm 2 The algorithm `ROMixMC`, performing `ROMix` with K/G memory.

Require:
- x ▷ initial state
- G ▷ 1st cost parameter
- K ▷ 2nd cost parameter

Ensure: x ▷ hash value

procedure `ROMixMC` (x, G, K)
1: **for** $i = 0, \ldots, G-1$ **do**
2: $\quad$ **if** $i \bmod K = 0$ **then**
3: $\quad\quad v_i \leftarrow x$
4: $\quad x \leftarrow H(x)$
7: **for** $i = 0, \ldots, G-1$ **do**
8: $\quad j \leftarrow x \bmod G$
9: $\quad \ell \leftarrow K \cdot \lfloor j/K \rfloor$
10: $\quad y \leftarrow v_\ell$
11: $\quad$ **for** $m = \ell+1, \ldots, j$ **do**
12: $\quad\quad y \leftarrow H(y)$ ▷ invariant: $y \leftarrow v_m$
13: $\quad x \leftarrow H(x \oplus y)$
14: **return** x

set to $x, H(x), \ldots, H(\ldots(H(x)))$, respectively. In the second phase (Lines 23-25), `ROMix` updates x depending on v_j. After G updates, the final value of x is returned and undergoes the post-processing.

A minor issue is that `scrypt` uses the password pwd as one of the inputs for post-processing. Thus, it has to stay in memory during the entire password-scrambling process. This may become a risk if there is any chance that the memory can be compromised during the time `scrypt` is running. Compromising the memory should not happen, anyway, but this issue could be easily fixed without any bad effect on the security of `scrypt`, e.g., one could replace Line 12 of Algorithm 1 by $x \leftarrow \text{PBKDF2}(x, s, 1, 1)$.

6.3.1 Brief Analysis of `ROMix`

The following part deals with a variant of `ROMix` which can be computed with less than G units of memory. Suppose, we have only $S \ll G$ units of memory for the values in v. For convenience, we assume that G is a multiple of S and define $K \leftarrow G/S$. As it will turn out, the memory-constrained algorithm `ROMixMC` (see Algorithm 2) generates the same result as `ROMix` with less than G memory units and is $\Theta(K)$ times slower than `ROMix`. From the array v, we will only store the values v_0, v_K, v_{2K}, $\ldots$, $v_{(S-1)K}$ using all the S memory units available.

At Line 9, the variable ℓ is assigned the biggest multiple of K less or equal j. By verifying the invariant at Line 12, it is clearly recognizable that `ROMixMC` computes the same hash value as the original `ROMix`, except that v_j is computed on-the-fly, beginning with v_ℓ. These computations call the random oracle on the average $(K-1)/2$ times. Thus, the second phase of `ROMixMC` requires

about $(K+1)/2$ times the effort than the second phase of ROMix, which dominates the workload for ROMixMC.

Next, we briefly discuss why ROMix fulfills Sequential Memory Hardness (SMH) (for the full proof see [211]). The intuition is as follows: The indices j are determined by the output of the random oracle H and thus, essentially uniformly distributed random values over $\{0, \ldots, G-1\}$. With no practical way to anticipate the next j, the most suitable strategy is to minimize the size of the "gaps", i.e., the number of consecutively unknown v_j. This is indeed what ROMixMC does, by storing one v_i every K-th step.

6.4 Cache-Timing Attack on scrypt

The motivation behind the attacks discussed in the remainder of this chapter stems from the existence of Side-Channel Attacks (SCAs) which are able to, e.g., (1) extract cryptographic secrets exploiting a buffer over-read in the implementation of the Transport Layer Security (TLS) protocol (Heartbleed) [69] of the OpenSSL library [268], (2) extract sensitive data on single-core architectures [14, 15, 16, 17, 123, 210], (3) gain coarse cache-based data on Symmetric Multiprocessing (SMP) architectures [222], and (4) to attack SMP architectures extracting a secret key over a cross-VM side channel [269].

Now, we have a deeper look at scrypt and ROMix as shown in Algorithm 1 regarding to their vulnerability to Cache-Timing Attacks (CTAs). CTAs are one form of SCAs and require the adversary to have physical access to the targets machine. Once this access is granted, the adversary will implement a so-called *spy process* as described next.

The Spy Process. As it turns out, the idea to compute a "random" index j and then ask for the value v_j, which is immensely useful for SMH, 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, but, 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 (Line 23 of Algorithm 1), the spy process reads from a bunch of addresses to repress all the values v_i that are still in the cache. Thereupon, ROMix is allowed to run for another short time.
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's has been read by ROMix in the meantime.

So, the spy process reveals the indices j for which v_j has been read, and with this information, we can mount the following CTA.

Preliminary Cache-Timing Attack. Let x be the output of PBKDF2($pwd, s, 1, 1$), where *pwd* denotes the current password candidate and s the salt. Then, we can apply the following password-candidate sieve.

1. Run the first phase of ROMix without storing the values v_i (i.e., skip Line 21 of Algorithm 1).
2. Compute the index $j \leftarrow x \bmod G$.
3. If v_j is among the values that have been read by ROMix, store *pwd* in a list.

4. Otherwise, conclude that *pwd* is a wrong password.

This sieve can run in parallel on any number of cores, where each core tests another password candidate *pwd*. Note that each core needs only a small constant amount of memory – the data structure to decide if j is one of the indices being read with v_j can be shared between all the cores. Thus, we can use exactly the kind of hardware that `scrypt` has been designed to hinder.

Next, we discuss the gain of this attack. Let r denote the number of iterations the loop in Lines 23-25 of Algorithm 1 has performed before the second interrupt by the spy process. So, there are at most r indices j with v_j being read. That means, we expect this approach to sort out all but r/G candidates. If our spy process manages to interrupt very soon, after allowing it to run again, we have $r \ll G$. This may enable us to use Commercial Off-The-Shelf (COTS) to run full `ROMix` to search for the correct password among the candidates on the list.

Final Cache-Timing Attack. In this attack, we allow the second interrupt to arrive very late – maybe even as late as the termination time of `ROMix`. So, the loop in Lines 23-25 of `ROMix` has been run $r = G$ times. As it seems, each v_i has been read once. But actually, this is only true *on average*; some values v_i have been read more then once, and we expect about $(1/e)G \approx 0.37\,G$ array elements v_i not to have been read at all. So applying the basic attack allows us to eliminate about 37 % of all password candidates – a rather small gain for such hard work.

In the following, we introduce a way to push the attack further, inspired by Algorithm 2, the memory-constrained `ROMixMC`. Our final CTA on `scrypt` needs only the smallest possible amount of memory: $S = 1, K = G/S = G$, and thus, we have only to store the single value v_0. Like the second phase of `ROMixMC`, we compute the values v_j on-the-fly when needed. Unlike `ROMixMC`, we stop execution whenever one of our values j is such that v_j has not been read by `ROMix` (according to the information from our spy process).

Thus, if the first j has not been read, we immediately stop the execution without any on-the-fly computation; if the first j has been read, but not the second, we need one on-the-fly computation of v_j, and so forth.

Since a fraction (i.e., $1/e$) of all values v_i has not been read, we will need about $1/(1 - 1/e) \approx 1.58$ on-the-fly computations of some v_j, each at the average price of $(G - 1)/2$ times calling H. Additionally, each iteration needs one call to H for computing $x \leftarrow H(x \oplus v_j)$. With regards to the effort for the first phase, i.e., G calls to H, the expected number of calls to reject a wrong password is about

$$G + 1.58 * \left(1 + \frac{G-1}{2}\right) \approx 1.79\,G.$$

As it turns out, rejecting a wrong password with constant memory is faster than computing ordinary `ROMix` with all the required memory, which actually makes $2G$ calls to H, without computing any v_i on-the-fly. We stress that the ability to abort the computation, thanks to the information gathered by the spy process, is crucial.

6.5 (Weak) Garbage-Collector Attacks on scrypt

Memory-demanding password scramblers, and especially those fulfilling SMH, provide a decent level of security against adversaries utilizing massively parallel hardware such as GPUs. Since it is actually impossible to provide a Swiss army knife of password hashing providing security against all kinds of adversaries, every act of protecting against a certain threat will open the door to another one. This is also the case for memory-demanding password scramblers. Assume that a PS allocates a

large amount of memory blocks $v_0, v_1, \ldots, v_{G-1}$ when processing a password. These blocks becomes "garbage" after the final hash value is produced, and will remain in memory; maybe for later reuse. One usually assumes that the adversary learns the hash of the secret. The Garbage-Collector Attack (GCA) assumes that the adversary additionally learns the memory content, i.e., the values v_i, after termination of the PS (see Definition 3.10).

Garbage-Collector Attack on `ROMix`. The function `ROMix` takes the initial state x and the memory-cost parameter G as inputs. First, `ROMix` initializes an array v of size $G \cdot n$ by iteratively applying a cryptographic hash function H (see Algorithm 1, Lines 20-22), where n denotes the output size of H in bits. Second, `ROMix` accesses the internal state at randomly computed points j to update the password hash (see Lines 23-25).

It is easy to observe that the value $v_0 \leftarrow H(x)$ is the result of a single call to H, i.e., it is a plain hash of the original secret x. Assuming an adversary has access to v_0, it can easily bypass `ROMix` and directly test every password candidate x' by testing $H(x') \stackrel{?}{=} v_0$, i.e., the time and memory complexity per candidate is significantly decreased to $\mathcal{O}(1)$. Furthermore, if the adversary fails to learn v_0, but any of the other values $v_i \leftarrow H(v_{i-1})$, the computational effort grows to $\mathcal{O}(i)$, but the memory complexity is still $\mathcal{O}(1)$.

As a possible countermeasure, one can simply overwrite $v_0, \ldots, v_{G-1}$ after running `ROMix`. However, this might be removed by a compiler due to optimization, since it is algorithmically ineffective.

Weak Garbage-Collector Attack on `scrypt`. In Line 12 of Algorithm 1, `scrypt` invokes PBKDF2 the second time, again using the password *pwd* as input. Thus, the value *pwd* has to be stored in memory during the entire invocation of `scrypt`, which implies that `scrypt` is trivially vulnerable to WGC attacks as defined in Definition 3.11.

Discussion. As already mentioned, the attacks above require an adversary to have physical access to the target's machine. A GCA requires an adversary to be provided with access to the internal memory, whereas the CTA requires to (1) run a spy process on the machine `ROMix` is running, (2) interrupt `ROMix` twice at the right points of time, and (3) precisely measure the timings of memory reads. Moreover, other processes that run on the same machine can add a huge amount of noise to the cache timings. That given, it is unclear if a real-world server could be exactly attacked as described above or requires a more sophisticated version of these attacks which, e.g., would be able to filter out noise. However, the applicability of CTAs to `ROMix` has been demonstrated [34] in an idealized setting, including execution right on the target system.

Remark 6.1. *Even without knowing the password hash at all,*

1. *the adversary is able to find out when the password has been changed,*
2. *and the adversary can mount a password-guessing attack*

just from knowing the memory-access pattern.

The second point of Remark 6.1 becomes highly interesting when comparing old password scramblers, e.g., `md5crypt` [149], with modern memory-demanding password scramblers, e.g., `scrypt`. When considering older password scramblers in terms of off-line attacks, it is not possible to find

out the password given that the adversary is not provided with the corresponding hash value. Even storing the password in plain would not be a problem as long as the adversary does not have access to the file it is saved in. However, memory-demanding password scramblers with a password-dependent memory-access pattern fall apart in this scenario.

6.6 Summary

This chapter covers the most noteworthy related work in terms of "classical" password hashing by providing a survey on its history. Moreover, we deduct all properties which we expect from a modern Password Scrambler (PS). As a motivation for providing resistance to Cache-Timing Attacks (CTAs), Garbage-Collector Attacks (GCAs), and Weak Garbage-Collector Attacks (WGCAs), we introduced particular attacks on the first (sequential) memory-hard PS `scrypt`.

CHAPTER 7

The Catena Password-Scrambling Framework

If you don't make mistakes, you're not working on hard enough problems. And that's a mistake.

FRANK ANTHONY WILCZEK

The previous chapter clearly defined which properties we expect from a modern Password Scrambler (PS) and which threats we should tackle when realizing a memory-demanding design. With that knowledge and those requirements in mind, we developed the Password-Scrambling Framework (PSF) CATENA, which, to the best of our knowledge, is the first that fulfills all the aforementioned properties. Since password hashing is desirable in a wide variety of applications, CATENA provides a framework rather than a dedicated design which may only be suitable for a single application. Therefore, one can easily replace the underlying primitives of CATENA to cover a large quantity of purposes (see Chapter 10). Therefore, in this chapter, we do not focus on particular instances but concentrate on the framework itself.

7.1 Design Decisions

Before we present our framework, we will elaborate on some decisions we had in mind when designing CATENA and give an informal overview over the main observation and ideas, starting with our understanding of the defender's machine, i.e., that of a legal user.

Defender's Machine. Our understanding of the defender's machine is straightforward: a typical CPU, as it would be running on a server, a PC, or a smartphone. While this still leaves a wide range of different choices open, we anticipate a limited number of cores and a limited amount of fast memory, i.e., cache. Although it thwarts GPU-based attacks, the usage of a large amount of memory may not always be acceptable, either because that amount of memory is unavailable, or because allocating too much memory would hinder other running processes on the same machine – further, a log-in process requiring too much memory may also ease denial-of-service attacks. Therefore, CATENA should be able to increase the running time using memory-independent parameters, e.g., increasing the bit size of the *pepper* (see Chapter 3).

Usage Scenarios. When thinking about when to apply a password scrambler P, we came up with the following three use cases:

Password-Based Authentication: Given a triple $(username, s, P(s, pwd))$, where $username$ is the alias provided to the user, s is the salt, and pwd the password, a user is authenticated by providing a password pwd', such that $P(s, pwd') = P(s, pwd)$. This can be used to log in at a server, providing a service for a multitude of different users, or at a machine serving a single user.

Password-Based Key Derivation: The PS is used to generate secure cryptographic keys.

Proof of Work / Proof of Space: The prover has to find a specific value x, such that $P(x)$ satisfies a statistically rare property determined by a boolean function p. This means, the prover searches for: *an input x such that $p(P(x)) = 1$ holds.* This is supposed to be a challenging task for the prover, while the verifier, given pwd, just needs to compute $y \leftarrow P(pwd)$ once and then check for the property. For example, $p(y) = 1$ could mean "the c Least Significant Bits (LSBs) of y are 0". For a well-designed password scrambler, the prover should have to call P about 2^c times, the verifier only once. Such schemes have been discussed as a defense against spam (the sender of an email would have to perform a proof of work or space in order to prevent them from sending emails to hundreds of thousands of receivers) [257] and they are also used for cryptocurrencies [178].

Regarding the first scenario, resistance to Cache-Timing Attacks (CTAs), Garbage-Collector Attacks (GCAs), and Weak Garbage-Collector Attacks (WGCAs) is highly important. Recently, it was shown that Cache-Timing Attacks (CTAs) can be used to perform attacks against virtual machines, even if the attack program is running on a different core than the defender's program [139]. For the remaining two scenarios, CTAs, GCAs, and WGCAs are less of an issue, and large amounts of memory should usually be available for the defender.

Capabilities of an Adversary. Making assumptions on the computational power of adversaries may seem like a futile exercise since they will actually use all computational power within their budget, like commodity hardware (CPUs and GPUs), reprogrammable hardware (FPGAs) and non-reprogrammable hardware (ASICs). Thus, we distinguish adversaries by the hardware they are using for their attacks:

Typical Password Crackers use cheap off-the-shelf hardware for their purpose, e.g., GPUs.

Low-Cost Hardware-Based Adversaries use low-cost reprogrammable hardware, e.g., FPGAs.

Clock-Cycle Thieves perform password cracking with hundreds of machines, e.g., botnet.

High-End Adversaries with a large budget who can afford dedicated hardware, e.g., ASICs.

Regarding the first two types, a modern memory-demanding PS is likely to cause trouble for adversaries who utilize massively parallel computations. For the third type of adversary, the capacities of a single machine from, e.g., a botnet, would be similar to that of the defender. Therefore, one should utilize a PS using the maximum amount of memory the defender can afford to reach a maximum slow-down for an adversary.

Algorithm 3 CATENA

Require:
pwd, t, s ▷ password, tweak, salt
$g_{\text{low}}, g_{\text{high}}, m$ ▷ min. garlic, garlic, output length
γ ▷ public input
Ensure: x ▷ hash of the password
procedure CATENA($pwd, t, s, g_{\text{low}}, g_{\text{high}}, m, \gamma$)
1: $x \leftarrow H(t \,||\, pwd \,||\, s)$
2: $x \leftarrow flap(\lceil g_{\text{low}}/2 \rceil, x, \gamma)$ ▷ provides resistance to WGCAs
3: $x \leftarrow H(x)$
4: **for** $g = g_{\text{low}}, \ldots, g_{\text{high}}$ **do**
5: $x \leftarrow flap(g, x \,||\, 0^*, \gamma)$ ▷ the core of CATENA
6: $x \leftarrow H(g \,||\, x)$ ▷ provides server relief
7: $x \leftarrow truncate(x, m)$
8: **return** x

Generic Design. We wanted CATENA to be a mode of operation for a cryptographic hash function rather than a primitive PS of its own right. The reasons are as follows:

- CATENA would be more flexibility and thus, more likely to be used in a wide range of applications.
- The underlying cryptographic primitive can be easily replaced by a new one in a case of, e.g., performance or security issues.
- CATENA inherits the security assurance and the cryptanalytic attempts from the underlying primitive, whereas a primitive PS would not. Therefore, the latter would require several years of cryptanalytical work to gain trust in the cryptographic community.
- It significantly eases the analysis if the underlying structure and cryptographic primitive is already well-analyzed.
- The diversity of possibilities for instantiating CATENA can frustrate adversaries which base their attacks on non-programmable hardware, e.g., ASICs.

7.2 Specification of Catena

Following the generic approach, we designed CATENA to be a mode of operation for a cryptographic hash function H (see Definition 3.3), an eventually reduced version H' of a cryptographic hash function, and a (λ-)memory-hard function $flap$ (see Definition 3.6). This fact is the reason for calling CATENA a Password-Scrambling Framework (PSF) rather than a Password Scrambler (PS), whereas the particular instances of that framework (see Chapter 10) would then be actual password scramblers. A formal definition of CATENA is shown in Algorithm 3. It processes the inputs $pwd, t, s, g_{\text{low}}, g_{\text{high}}, m$, and γ, whereas the functions Γ, F_λ, and Φ are given as implicit inputs since they define a particular instance of CATENA and have to be fixed before CATENA can actually be called.

pwd The (secret) password.

t A tweak which allows to customize the password hash, which is an additional multi-byte value given as follows:

$$t \leftarrow H(V) \,||\, d \,||\, \lambda \,||\, m \,||\, |s| \,||\, H(AD),$$

where the first n-bit value $H(V)$ denotes the hash value of a unique version identifier V (see Chapter 10 for details). The byte-value d denotes the domain (i.e., the mode) for which CATENA is used, where we set $d \leftarrow$ `0x00` when used as a PS, $d =$ `0x01` when used as a KDF, and $d =$ `0x02` when used in the proof-of-work/space scenario. The third byte λ determines the time-cost parameter (in contrast to the value g, which determines the memory cost). The 2-byte value m denotes the length of the final hash value in bits and the 2-byte value $|s|$ denotes the length of the salt in bits. Finally, the n-bit value $H(AD)$ denotes the hash of associated data AD, which can contain additional information like hostname, user-ID, name of the company, or the IP of the host. Note that the order of these values does not matter as long as they are fixed for a certain application.

s The salt value (usually provided by the system).

g_{low} The minimum value of g.

g_{high} The maximum value of g, where we usually, we set $g_{\mathrm{low}} \leftarrow g_{\mathrm{high}}$. Nevertheless, g_{high} determines the required memory cost for CATENA running in minimal computational time, i.e., optimizing the TMTO regarding to minimal running time and maximal available memory (see Paragraph **Memory Hardness** in Chapter 3).

m The length of the final hash value in bytes, which is guaranteed by a call to the function *truncate(x,m)* (see Line 6 of Algorithm 3) returning the m least significant bytes of x.

γ A *public* and *password-independent* value, with $\gamma \in \{0,1\}^*$, which can be used to customize the memory-access pattern during the invocation of CATENA (see later in this section for details).

The general idea of how CATENA works is depicted in Figure 7.1. It starts with processing the password, the tweak, and the salt, using the cryptographic hash function H (see Line 1 of Algorithm 3). The second step is given by a call to the core function of CATENA, i.e., $flap$ (see Algorithm 4), where we set the memory parameter to $\lceil g_{\mathrm{low}}/2 \rceil$ (determining the number of simultaneously stored memory units). The latter and the fact that the password pwd can (and should) be immediately removed after the first step provides resistance to Weak Garbage-Collector Attacks (WGCAs). In Line 3 of Algorithm 3, we call H again to provide an n-bit input to the core function $flap$. The core function $flap$ is called at least a second time as the first step of each invocation of the main loop (see Line 5 of Algorithm 3). Thereby, if for the password-dependent input x holds that $|x| < n$, it is padded with as many 0's as necessary so that $|x \,||\, 0^*| = n$. This padding is required since the truncation step at the end of CATENA allows $|x| < n$.

The function $flap$ consists of four phases: (1) an initialization phase (see Lines 1 to 3), where the memory of size $2^g \cdot k$ bits is written in a sequential order (k denotes the output size of the underlying hash function H' in bits), (2) a call to the function Γ (see Line 4) depending on the

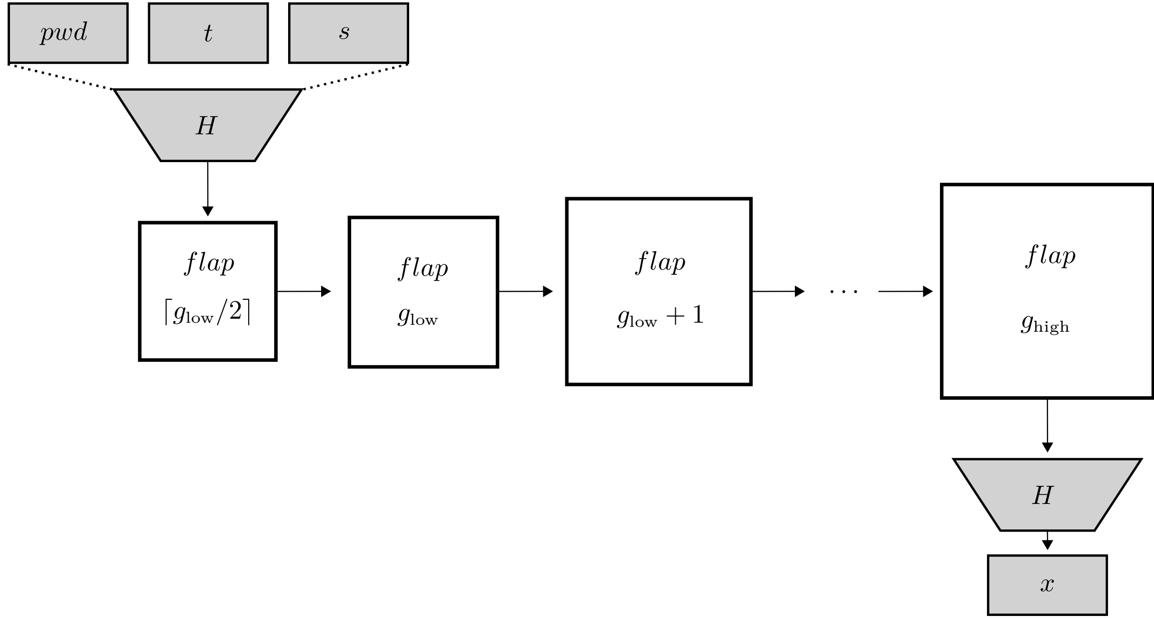

Figure 7.1: The general idea of Catena employing its core function *flap*.

Algorithm 4 Function *flap* of Catena using H_{init}

Require: g, x, γ ▷ garlic, value to hash, public input
Ensure: $x \in \{0,1\}^k$ ▷ intermediate hash value
procedure $flap(g, x, \gamma)$
1: $(v_{-2}, v_{-1}) \leftarrow H_{\text{init}}(x)$
2: **for** $i = 0, \ldots, 2^g - 1$ **do**
3: $v_i \leftarrow H'(v_{i-1} \,||\, v_{i-2})$ ▷ initialize the memory
4: $v \leftarrow \Gamma(g, v, \gamma)$ ▷ one layer with γ-based memory accesses
5: $v \leftarrow F_\lambda(v)$ ▷ memory-hard function
6: $x \leftarrow \Phi(g, v, \mu)$ ▷ one layer with μ-based memory accesses
7: **return** x

public input γ (e.g., the salt), (3) a call to a (λ-)memory-hard function F_λ (see Line 5), and (4) a call to the function Φ (see Line 6) depending on a secret (password-dependent) input $\mu \in \{0,1\}^*$, e.g., the last word of the state $v := v_0, \ldots, v_{2^g-1}$ obtained as output of F_λ, i.e., v_{2^g-1}. Note that we allow Γ to be the identity function and Φ to just return the last state word v_{2^g-1}. The reason that we do not state μ as explicit input to *flap* is straightforward: assume $\mu = pwd$, which satisfies the fact that μ must be a password-dependent value. Then, *pwd* must be in memory until the call to Φ, rendering Catena directly vulnerable to Weak Garbage-Collector Attacks (WGCAs). Both Γ and Φ are so-called *Options* to Catena which we discuss later in this section. The general idea of *flap* is depicted in Figure 7.2.

Note that the output size in bits k of H' does not necessarily have to be equal to that of H, i.e., n bits. This is also the reason for calling the initialization function H_{init} as first step in the function *flap* (see Line 1 of Algorithm 4). H_{init}, as defined in Algorithm 5, processes an n-bit input x and outputs two initial k-bit state words (v_{-2}, v_{-1}). To preserve at least one call to the underlying cryptographic

Algorithm 5 H_{init}

Require: x ▷ n-bit value to hash
Ensure: v_{-2}, v_{-1} ▷ two k-bit outputs
procedure $H_{\text{init}}(x)$
1: $\ell = 2 \cdot k/n$
2: **for** $i = 0, \ldots, \ell - 1$ **do**
3: $\quad w_i \leftarrow H(i \,||\, x)$
4: $v_{-2} \leftarrow (w_0 \,||\, \cdots \,||\, w_{\ell/2-1})$
5: $v_{-1} \leftarrow (w_{\ell/2} \,||\, \cdots \,||\, w_{\ell-1})$
6: **return** (v_{-2}, v_{-1})

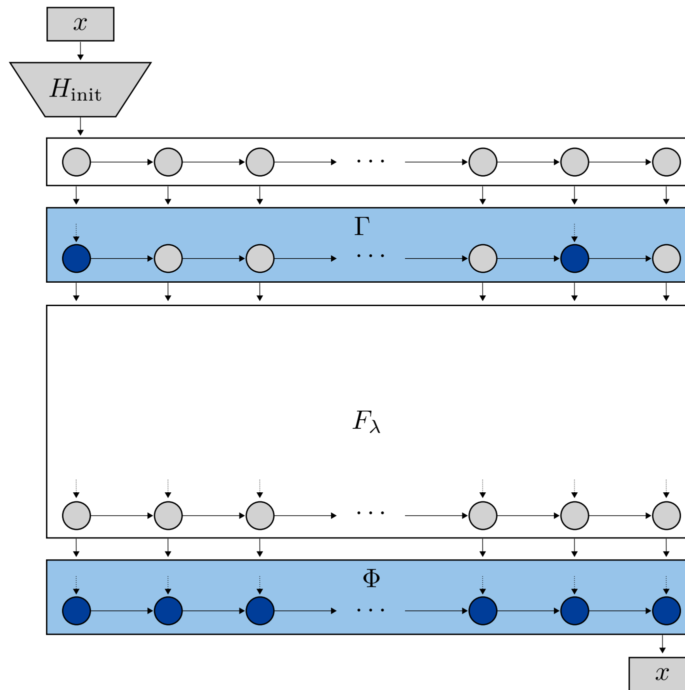

Figure 7.2: The general idea of *flap* with its core F_λ. The elements in light blue denote the optional layers Γ and Φ, where the state words which are overwritten (exemplary for Γ) are colored in dark blue.

hash function H per 2^g state-value computations, we use the function H in H_{init} to compute the first chaining values v_{-2} and v_{-1} as shown in Line 3 of Algorithm 5. The additional input $i \in \{0,1\}^n$ (loop counter) satisfies distinct inputs for each call to H to guarantee pseudorandom independent values $w_0, \ldots, w_{\ell-1}$. Moreover, we always assume k and n to be multiples of 2 and $n|k$, i.e., k is always a multiple of n even if $k > n$ holds.

The penultimate step of Catena consists of processing the output x of *flap* (see Line 5 of Algorithm 3) in a post-hashing step invoking H, allowing Catena to support Server Relief (SR) as explained in Chapter 3. Finally, the output of the post-hashing step is truncated to m bytes (see Line 7) providing outputs of length $\leq n$.

Algorithm 6 The functions $\Gamma(g, v, \gamma)$ and $\Phi(g, v, \mu)$.

Require:
g, v ▷ garlic, state
γ ▷ public input
Ensure: v ▷ updated state
procedure $\Gamma(g, v, \gamma)$
10: $r \leftarrow H(\gamma) \mid\mid H(H(\gamma))$
11: $p \leftarrow 0$
12: **for** $i \leftarrow 0, \ldots, 2^{\lceil 3g/4 \rceil} - 1$ **do**
13: $(j_1, r, p) \leftarrow R(r, p, g)$
14: $(j_2, r, p) \leftarrow R(r, p, g)$
15: $v_{j_1} \leftarrow H'(v_{j_1} \mid\mid v_{j_2})$
16: **return** v

Require:
g, v ▷ garlic, state
μ ▷ secret input
Ensure: v_{2^g-1} ▷ output of *flap*
procedure $\Phi(g, v, \mu)$
20: $j \leftarrow \pi(\mu, g)$
21: $v_0 \leftarrow H'(v_{2^g-1} \mid\mid v_j)$
22: **for** $i \leftarrow 1, \ldots, 2^g - 1$ **do**
23: $j \leftarrow \pi(v_{i-1}, g)$
24: $v_i \leftarrow H'(v_{i-1} \mid\mid v_j)$
25: **return** v_{2^g-1}

Options Γ and Φ. Next, we describe the two aforementioned options Γ and Φ, which can be integrated to the structure of Catena. The function Γ consists of a *password-independent* graph-based structure increasing the resistance to ASIC-based adversaries, whereas Φ is a *password-dependent* graph-based function providing Sequential Memory Hardness (SMH) (see Definition 3.7). We call a function *password-dependent* if its memory-access pattern, i.e., the order of access to the internal state words, depends on the password. Thus, it is different for each password and, in the best case pseudorandom, which thwarts, e.g., ASIC-based adversaries.

Password-Independent Random Layer Γ. The optional function Γ is defined in Algorithm 6 (left). It receives the garlic g, the internal state v, and a public value γ as inputs and updates the state v, where γ determines the memory-access pattern, i.e., the indidces of the internal state words are accessed in a γ-dependent pseudorandom manner during its invocation. In each iteration of the **for**-loop (see Lines 12-15), two random g-bit integers j_1 and j_2 are computed individually using a Random-Number Generator (RNG) $R : \{0,1\}^{2n} \times \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^g \times \{0,1\}^{2n} \times \{0,1\}^n$, where $r \in \{0,1\}^{2n}$ denotes its current state, $p \in \{0,1\}^n$ denotes the index of the currently computed output value, and $g \in \{0,1\}^n$ the garlic. The values j_1 determines the index for both the updated word and the first input to H', and j_2 for the second input to H' (see Line 15). Even though H may appear as a natural choice for a random-number generator, we later introduce an instance of R given a non-cryptographic but statistically sound function, that provides higher performance. Nevertheless, the stronger function H is used to produce the required seed for R, which is given by the $2n$-bit value $r \leftarrow H(\gamma) \mid\mid H(H(\gamma))$ as shown in Line 10. To increase the performance further, the main loop overwrites only $2^{\lceil 3g/4 \rceil}$ (out of 2^g) randomly chosen words of the internal state v.

Password-Dependent Random Layer Φ. This option introduces a further graph-based layer after the call to F_λ (see Line 6 of Algorithm 4). The function Φ is used to update the internal state v of size $2^g \cdot k$ bits by sequentially computing each state word v_i depending on two inputs (see Algorithm 6): first, the immediate predecessor v_{i-1} (or v_{2^g-1} if $i = 0$) and second, a value chosen uniformly at random from the state, where the index is determined by the function π. The function π can, for example, be given by the RNG R as used for the option Γ[1], or it can simply output the g LSBs of its input. A strict requirement to π is that it satisfies the properties of a RNG and for its call in Line 23, it must hold that its current output depends on the formerly computed value

[1]That requires the beforehand computation of the seed $r \leftarrow H(\mu) \mid\mid H(H(\mu))$.

v_{i-1}. When called the first time (see Line 20), its output depends on the secret input μ, which we set per default to the value v_{2^g-1} of the output state of F_λ. Thus, we basically follow a slightly more generic approach in comparison to the `ROMix` function used within `scrypt` (see Algorithm 1 in Section 6.3). It is easy to see that Φ conducts sequential writes, whereas Γ conducts writes to randomly determined words of the state. Thus, by calling Φ, Catena provides SMH

Remark 7.1. *Note that the usage of Φ implies Sequential Memory Hardness and thus, it renders an instance of* Catena *vulnerable to Cache-Timing Attacks (see Section 6.3). Therefore, when thinking about using Φ, one should carefully evaluate whether Sequential Memory Hardness is more important than resistance to Cache-Timing Attacks. Moreover, this is also the reason why invoke Φ only after F_λ, since else, a Cache-Timing Attack adversary could avoid the memory-demanding part of* Catena.

7.3 Functional Properties

Garlic. The function F_λ is defined by a graph-based structure (see Chapter 9 for our proposed instances), where the memory requirement of F_λ depends on the number of input vertices of the underlying permutation graph. Since the goal is to hinder an adversary making a reasonable number of parallel password checks using the same memory, we have to consider a minimal number of input vertices, which we denote by $G \leftarrow 2^g$, where we usually set $g_{\text{low}} \leftarrow g_{\text{high}} \leftarrow g$.

Client-Independent Update (CIU). Catena's sequential structure allows client-independent updates. Let $h \leftarrow \text{Catena}(pwd, t, s, g_{\text{low}}, g_{\text{high}}, m, \gamma)$ be the hash of a specific password pwd, where $t, s, g_{\text{low}}, g_{\text{high}}, m$, and γ denote tweak, the salt, the minimum garlic, the garlic, the output length, and the public input, respectively. After increasing the security parameter from g_{high} to $g'_{\text{high}} = g_{\text{high}} + 1$, we can update the hash value h without user interaction by computing

$$h' \leftarrow truncate(H(g'_{\text{high}} \,||\, flap(g'_{\text{high}}, h \,||\, 0^*, \gamma)), m).$$

It is easy to see that the equation $h' = \text{Catena}(pwd, t, s, g_{\text{low}}, g'_{\text{high}}, m, \gamma)$ holds.

Server Relief (SR). In the final iteration of the **for**-loop in Algorithm 3, the client has to omit the last invocation of the hash function H (see Line 6). The current output of Catena is then transmitted to the server. Next, the server computes the password hash by applying the hash function H and the function *truncate*. Thus, the vast majority of the effort (memory usage and computational time) for computing the password hash is handed over to the client, freeing the server. This enables someone to deploy Catena even under restricted environments or when using constrained devices – or when a single server has to handle a huge amount of authentication requests, e.g., in social networks.

Keyed Password Hashing. To thwart off-line attacks further, we introduce a technique to use Catena for keyed password hashing, where the password hash depends on both the password and a secret key K. To simplify the key management, the value K would be the same for all users, and therefore has to be stored on the side of the server. To still allow SR (see above), we encrypt the output of Catena by XORing it with $H(K \,||\, userID \,||\, g_{\text{high}} \,||\, K)$, which, under the reasonable

assumption that the value $(userID \,||\, g_{\text{high}})$ is a nonce[2], was proven to be IND-CPA-secure in [227]. Let $X := (pwd, t, s, g_{\text{low}}, g_{\text{high}}, m, \gamma)$, then, the output of Catena^K is computed as

$$y = \text{Catena}^K(userID, X) := \text{Catena}(X) \oplus H(K \,||\, userID \,||\, g_{\text{high}} \,||\, K),$$

where Catena is defined as in Algorithm 3 and the *userID* is a unique and user-specific identification number which is assigned by the server. Now, we show what happens during the client-independent update, i.e., when $g_{\text{high}} = g_{\text{high}} + r$ for arbitrary $r \in \mathbb{N}$. The process takes the following four steps:

1. Given K and $userID$, compute $z \leftarrow H(K \,||\, userID \,||\, g_{\text{high}} \,||\, K)$.
2. Compute $x \leftarrow y \oplus z$, where y denotes the current keyed hash value.
3. Update x, i.e., $x \leftarrow H(g \,||\, flap(g_{\text{high}}, x \,||\, 0^*, \gamma))$ for $g \in \{g_{\text{high}} + 1, \ldots, g_{\text{high}} + r\}$.
4. Compute the new hash value $y \leftarrow y \oplus H(K \,||\, userID \,||\, g_{\text{high}} + r \,||\, K)$.

Remark 7.2. *Obviously, it is a bad idea to store the secret key K together with the password hashes since it can leak in the same way as the password-hash database. One possibility to separate the key from the hashes is to securely store the secret key in Hardware Security Modules (HSMs), which provide a tamper-proof memory environment with verifiable security. Then, the protection of the secret key depends on the level provided by the HSM (see FIPS140-2 [65] for details). Another possibility is to derive K from a password during the bootstrapping phase. Afterwards, K will be kept in Random-Access Memory (RAM) and will never be on the hard drive. Thus, the key and the password-hash database should never be part of the same backup file.*

7.4 Security Properties

Memory Hardness. Catena inherits its memory hardness from the particular instance of F_λ. Therefore, we provide several possibilities ranging from λ-memory hardness to sequential memory hardness in Section 9.1, all resting upon a graph-based structure.

Preimage Security. One major requirement for password scramblers is described by the preimage security, i.e., given a password hash $h \leftarrow P(pwd)$, one cannot gain any information about *pwd* in practical time. This requirement becomes mostly crucial in the situation of a leaked password-hash database. In Section 8.1 in the next chapter, we show that the preimage security of Catena depends on 1) the assumption that the underlying hash functions H and H' are a one-way functions and 2) the entropy of the password (pwd).

[2] A **n**umber used **once**, i.e., a value which must never repeat.

PRF Security. For the application of Catena as a password scrambler, this property is noncritical. But, if Catena is used as a Key-Derivation Function (KDF), one wants the resulting secret key to be indistinguishable from a random string of the same length. In Section 8.2 we show that for a secret input (*pwd*), the output of Catena looks random. The presented proof is based on the assumption that the underlying hash function behaves like a random oracle.

Resistance to Cache-Timing Attacks. From Definition 3.9, it follows that an algorithm provides resistance to CTA if its control flow does not depend on the input. We state that Catena provides this property based on two requirements: (1) the option Φ is just returning the last state value v_{2^g-1}, i.e., we "neglect" the only function whose memory-access pattern is designed to depend on a secret input, e.g., an internal state value resulting from the computation of the password. The control flow of the function F_λ depends only on the security parameters g (garlic) and λ (depth) (see Section 9.1 for details). Given these two parameters, it provides a predetermined memory-access pattern, which is independent from the secret input (*pwd*); and (2) the particular instances of H and H' provide resistance to CTAs.

7.5 Usage

The discussion in this section is done under the reasonable assumption that the parameters λ, g_{low}, $flap$, γ, and m are fixed values.

7.5.1 Catena for Proof of Work

The concept of proofs of work was introduced by Dwork and Naor [85] in 1992. The basic design goal was to combat junk mail under the usage of CPU-bounded functions, i.e., to gain control over the access to shared resources. The main idea is "*to require a user to compute a moderately hard, but not intractable, function in order to gain access to the resource* " [85]. Therefore, they introduced so-called CPU-bound *pricing functions* based on certain mathematical problems which may be hard to solve (depending on the parameters), e.g., extracting square roots modulo a prime. Tromp recently proposed the "first trivially verifiable, scalable, memory-hard and TMTO-hard proof-of-work system" in [257].

As a further development to CPU-bound functions, Abadi et al. [1], and Dwork et al. [84] considered moderately hard memory-bound functions since memory-access speeds do not vary much on different machines as CPU accesses do. Therefore, memory-bound functions may behave more equitably than CPU-bound functions. The former base on a large table that is randomly accessed during the execution, causing a lot of cache misses. Dwork et al. presented in [86] a compact representation of such a table by using a Time-Memory Tradeoffs (TMTOs) for its generation. Dziembowski et al. [89] as well as Ateniese et al. [29] put forward the concept of proofs of space, i.e., they do not consider the number of accesses to the memory (as memory-bound function do), but the amount of disk space the prover has to use. In [89], the authors proposed a new scheme using "graphs with high pebbling complexity and Merkle hash-trees".

For Catena, there exist at least two possible approached to be used for proofs of work. We denote by C the client which has to fulfill the challenge to gain access to a server S. Furthermore, the methods explained below work for all possible instances of our framework.

Guessing Secret Bits (Pepper). Initially, S chooses fixed values for pwd, t, s and g_{high}, where s denotes a randomly chosen salt value, where, on the other hand, p bits of s are secret, i.e., pepper.

Afterwards, S computes $h \leftarrow \textsc{Catena}(pwd, t, s, g)$ and sends the tuple $(pwd, t, s_{[0,|s|-p-1]}, g_{\text{high}}, h, p)$ to C, where $s_{[0,|s|-p-1]}$ denote the $|s| - p$ least significant bits of s (the public part). Now, C has to guess the secret bits of the salt by computing $h' \leftarrow \textsc{Catena}(pwd, t, s', g_{\text{high}})$ about 2^p times and comparing if $h = h'$. If so, C sends the correct s to S and gains access to S. The effort of C is given by about 2^p computations of Catena (and about 2^p comparisons for $h = h'$). Hence, the effort of C is scalable by adapting p.

Guessing the Correct Password. In this scenario, S chooses a p-bit password pwd, a tweak t, a salt s, and the garlic g_{high}. Then, S computes $h \leftarrow \textsc{Catena}(pwd, t, s, g_{\text{high}})$ and sends the tuple $(t, s, g_{\text{high}}, p, h)$ to C. The client C then has to guess the password by computing about 2^p times $h' \leftarrow \textsc{Catena}(pwd', t, s, g_{\text{high}})$ for different values of pwd', and comparing if $h' = h$. If so, C send the correct pwd to S and gains access to S. The effort of C is given by about 2^p computations of Catena (and about 2^p comparisons for $h = h'$). Hence, in this case, the effort of C is scalable by adapting the length p of the password.

7.5.2 Catena in Different Environments

This section considers several applications of the generic Catena framework which would work independently from a particular instance. A discussion about where instances of Catena could be applied is presented in Chapter 10.

Backup of User Data. When maintaining a database of user data, e.g., password hashes, storage providers (servers) sometimes store a backup of their data on a third-party storage, e.g., a cloud. This implies that the owner looses control over its data, which can lead to unintentional publication (leakage). Therefore, we highly recommend to use Catena in the keyed password hashing mode (see Section 7.3) to encrypt/protect the outsources backups. Thus, the security of each password is given by the underlying secret key and does not longer solely depend on the strength of password itself. Note that the key must be kept secret, e.g., it must not be stored together with the backup.

Using Catena with Multiple Cores. Catena has been designed to run on a modern single-core machine. To use multiple cores during the legitimate login process, one can apply the pepper approach. Therefore, p bits of the salt are kept secret, i.e., when one is capable of using b cores, it would set $p \leftarrow \log_2(b)$. During the login process, the i-th core will then compute the value $h_i \leftarrow \textsc{Catena}(pwd, t, s_{0,\ldots,|s|-p-1} \mid\mid i, g_{\text{high}})$ for $i = 0, \ldots, b-1$. The login is successful if and only if one of the values h_i is valid. This approach is fully transparent for the user, since due to the parallelism, the login time is not effected. Nevertheless, the total memory usage and the computational effort are increased by a factor b. This also holds for an adversary, since it has to test 2^p possible values for the pepper to rule out a password candidate.

Low-Memory Environments. The application of the SR technique leads to significantly reduced effort on the side of the server for computing the output of Catena, i.e., Catena is split into two functions P (typically $flap$) and H, where P is time- and memory-demanding and H is efficient. Obviously, the application of this technique makes most sense when the server has to administrate a large amount of requests in little time. Then, each client has to compute an intermediate hash $y \leftarrow P(\cdot)$ and the server only has to compute $h \leftarrow H(y)$ for each user.

Algorithm 7 Catena-KG

Require:
pwd, t', s ▷ password, tweak, salt
$g_{\text{low}}, g_{\text{high}}, m$ ▷ min. garlic, garlic, output length
$\gamma, \ell, \mathcal{I}$ ▷ public input, key size, key identifier
Ensure: k ▷ ℓ-byte key derived from the password
procedure Catena-KG($pwd, t', s, g_{\text{low}}, g_{\text{high}}, m, \gamma, \ell, \mathcal{I}$)
1: $x \leftarrow \text{Catena}(pwd, t', s, g_{\text{low}}, g_{\text{high}}, m, \gamma)$
2: $k \leftarrow \varepsilon$
3: **for** $i = 1, \ldots, \lceil \ell/n \rceil$ **do**
4: $k \leftarrow k \mid\mid H(i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$ ▷ 2-byte counter i satisfies unique outputs
5: **return** $truncate(k, \ell)$

7.5.3 The Key-Derivation Function Catena-KG

In this section, we introduce Catena-KG – a mode of operation based on Catena, which can be used to generate different keys of different sizes (even larger than the natural output size of Catena, see Algorithm 7). To always guarantee distinct inputs even for equal values for *pwd*, the domain value d of the tweak for Catena is set to 1, i.e., t' is given by

$$t' \leftarrow H(V) \mid\mid \texttt{0x01} \mid\mid \lambda \mid\mid m \mid\mid |s| \mid\mid H(AD).$$

Note that for key derivation, it makes no sense to give the user control over the output length m of Catena. It controls only the output of Catena-KG by adapting ℓ. Thus, within Catena-KG, the value for m is set per default to the output size of the underlying hash function. The call to Catena is followed by an output transform that takes the output x of Catena, a byte array *key identifier* $\mathcal{I}$, and a 2-byte value ℓ for the key length as the input, and generates key material of the desired output size. Catena-KG is even able to handle the generation of extra-long keys (longer than the output size of H), by applying H in Counter Mode [87]. Note that longer keys do not imply improved security in that context.

The key identifier $\mathcal{I}$ is supposed to be used when different keys are generated from the same password. For example, when Alice and Bob set up a secure connection, they may need four keys: an encryption and a message authentication key for messages from Alice to Bob, and another two keys for the opposite direction. Further, one could argue that $\mathcal{I}$ should also become a part of the associated data. But actually, this would be a bad move, since setting up the connection would require legitimate users to run Catena several times. However, the adversary can search for the password for one key, and just derive the other keys, once that password has been found. For a given budget for key derivation, one should rather employ one single call to Catena with larger security parameters and then run the output transform for each key.

In contrast to the password-hashing scenario, where a user wants to perform a log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [258], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, one can increase the time- and memory-cost parameter λ and g for Catena-KG in contrast to Catena.

7.6 Summary

CATENA is a *framework* that allows users to choose their own instances that suits their specific needs best. Making the proper choice may be difficult for many users, though. Furthermore, cryptanalysts prefer a fixed target rather than a generic framework where any attack can be defended by changing the instances. For these reasons, we provide particular instances of CATENA in Chapter 10.

Catena is Flexible. An instantiation of CATENA is defined by: (1) a cryptographic primitive H, e.g., BLAKE2b [33], (2) a "reduced" primitive H' (e.g., a reduced-round version of H), though $H' = H$ is also possible, (3) a "memory-hard" function F_λ, that uses both H and H', (4) an option Γ that employs a password-independent memory-access pattern, and (5) an option Φ that employs a password-dependent memory-access pattern.

Catena has Tunable Parameters. The *garlic* g and the depth λ determine memory and time requirements for CATENA. Increasing the *pepper* allows to increase the time without affecting the memory, and the *salt* size impacts the resistance to so-called rainbow tables [203]. CATENA supports Server Relief, i.e., it allows to shift the effort (both time and memory) for computing the password hash from the server to the client, and it provides Client-Independent Updates, allowing the defender to increase the security parameters g and *pepper* at any time, even for inactive accounts.

Catena has a Sound Theoretical Foundation. All instances utilize a cryptographic hash function H as one of the underlying primitives and the CATENA framework itself is easy to analyze (see Chapter 8). The underlying graph-based structures (see Section 9.1) follow an elegant design and are understood well.

Catena is secure. We claim the following security properties of CATENA: *preimage security* (a strict requirement for password hashing), *indistinguishability from random bits (PRF security)* (important for key derivation), *lower bounds on the TMTOs*[3] (for high resilience to massively parallel attacks with constrained memory, e.g., when using GPUs), and *resistance to Side-Channel Attacks*, such as Cache-Timing Attacks, Garbage-Collector Attacks, and Weak Garbage-Collector Attacks. Furthermore, CATENA *supports keyed password hashing*, i.e., the output of the unkeyed version of CATENA is encrypted by XORing it with a hash value generated from the userID, the memory-cost parameter, and secret key.

[3]The lower bounds presented in Section 9.2 are achieved in the sequential pebbling game (see Section 11.2 for details and a discussion about a different model, i.e., the Parallel Random-Oracle Model [23].)

Chapter 8

Catena – Security Analysis of the Framework

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan M. Turing

During this chapter, we focus on the security properties which are provided by the general framework rather than immersing on particular instances of the underlying components of Catena. Moreover, the discussion about the two options Γ and Φ is also postponed to Chapter 9 since it requires knowledge about certain instances of the graph-based structure, which are presented earlier in the very same chapter. Thus, the focus lies on the preimage and PRF security of Catena. Further, in Section 8.3, we show that the KDF Catena-KG (see Section 7.5.3) provides PRF security for generating cryptographic keys. For all proofs shown in this chapter, we set $H = H'$ and model the underlying hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ as a random oracle (as our proofs are given in the random-oracle model). Additionally, for all values $x \in \{0,1\}^*$, we set $H_m(x) := truncate(H(x), m)$, i.e., H_m truncates the output of H to m bits. Finally, for simplicity, we set $g \leftarrow g_{\text{high}}$.

8.1 Password-Recovery Resistance

In this section, we provide an analysis of Catena that shows that for guessing a valid password and being provided with a hash value h, an adversary either has to try out all possible password candidates in likelihood order, it has to find a preimage for the underlying hash function, or a preimage for Catena itself.

Theorem 8.1 (Catena is Password-Recovery Resistant). *Let p denote the min-entropy of a password source $\mathcal{Q}$. Then, it holds that*

$$\mathbf{Adv}^{REC}_{\text{Catena},\mathcal{Q}}(q) \leq \frac{q}{2^p} + \frac{q}{2^m}.$$

Proof. Note that an adversary $\mathbf{A}$ can always guess a password by trying out about 2^p password candidates. For a maximum of q queries, it holds that the success probability is given by $q/2^p$. Instead of guessing 2^p password candidates, an adversary can also try to find a preimage for a given hash value h. It is easy to see from Algorithm 3 that an adversary has to find a preimage for H in Line 6, that is truncated to m bits in Line 7. More detailed, for a given value h with $h \leftarrow H_m(g_{\text{high}} \mid\mid x)$, $\mathbf{A}$ has to find a valid value for x. Since $\mathbf{A}$ is allowed to ask at most q queries, the success probability for that event can be upper bounded by $q/2^m$. Our claim follows by adding up the individual terms. □

8.2 PRF Security of Catena

In the following, we analyze the advantage of an adversary $\mathcal{A}$ in distinguishing the output of Catena from a random bitstring of the same length as the output of Catena. More detailed, as mentioned before, we consider the PRF security of Catena in the random-oracle model. Note that the time parameter λ and the memory parameter g_{low} (minimum garlic) are constant values which are set once when initializing a system the first time.

Theorem 8.2 (PRF Security of Catena). *Let q denote the number of queries made by an adversary and $s \in \{0,1\}^{|s|}$ a randomly chosen salt value. Furthermore, let H be modeled as a random oracle and $g \geq g_{low} \geq 1$. Then, it holds that*

$$\mathbf{Adv}^{PRF}_{\text{Catena}}(q) \leq \frac{(q \cdot g + q)^2}{2^m} + \mathbf{Adv}^{COLL}_{flap}(q \cdot g).$$

Proof. Let $a^i = (pwd^i \mid\mid u^i \mid\mid s^i \mid\mid g \mid\mid \gamma^i)$ represent the i-th query of $\mathbf{A}$, where pwd denotes the password, u denotes the tweak, s the salt, g the garlic, and γ the public input. For this proof, we impose the reasonable condition that all queries of an adversary are distinct, i.e., $a^i \neq a^j$ for $i \neq j$. Suppose that y^j denotes the output of $flap(g, x \mid\mid 0^*, \gamma^j)$ of the j-th query (see Line 5 of Algorithm 3). Then, $H_m(g \mid\mid y^j)$ is the output of $\text{Catena}(a^j)$. In the case that $y^1, \ldots, y^q$ are pairwise distinct and $q \ll 2^{m/2}$, an adversary $\mathbf{A}$ cannot distinguish $\text{Catena}(\cdot)$ from a random function $\$(\cdot)$ since in since both functions return a value chosen uniformly at random from $\{0,1\}^m$. Therefore, we have to upper bound the probability of the event $y^i = y^j$ with $i \neq j$. Due to the assumption that $\mathbf{A}$'s queries are pairwise distinct, there must be at least one collision for H_m or $flap$. For q queries, we have at most $q(g+1)$ invocations of H_m. Thus, we can upper bound the collision probability by

$$\frac{(q \cdot g + q)^2}{2^m}.$$

Furthermore, we have $q \cdot g$ invocations of the memory-consuming function $flap$ and thus, we can upper bound the probability of a collision for $flap$ by its collision advantage $\mathbf{Adv}^{\mathrm{COLL}}_{flap}(q \cdot g)$. Thus, our claim in Theorem 8.2 follows. □

8.3 PRF Security of Catena-KG

It is easy to see that Catena-KG inherits its memory hardness from Catena since it invokes Catena (Line 1 of Algorithm 7). Next, we show that Catena-KG is also a good PRF.

Theorem 8.3 (PRF Security of Catena-KG). *Let q denote the number of queries made by an adversary and $s \in \{0,1\}^{|s|}$ a randomly chosen salt value. Furthermore, let H be modeled as a random oracle and $g \geq g_{low} \geq 1$. Then, it holds that*

$$\mathbf{Adv}^{PRF}_{\text{Catena-KG}}(q) \;\leq\; \mathbf{Adv}^{PRF}_{\text{Catena}}(q) + \frac{\lceil \ell/n \rceil \cdot q}{2^n}.$$

Proof. For the sake of simplification, we omit the final truncation step (see Line 5 in Algorithm 7) and let the adversary always get access to the untruncated key k in Line 4. Suppose x^i denotes the output of Catena of the i-th query. In the case $x^i \neq x^j$ for all values with $1 \leq i < j \leq q$, the output k is always a random value, since H is modeled as a random oracle and is always invoked with a fresh input (see Line 4). The only chance for an adversary to distinguish Catena-KG$(\cdot)$ from the random function $\$(\cdot)$ is (1) a collision in Catena or (2) a preimage for H in Line 4. The advantage of a PRF adversary for the former event is upper bounded by $\mathbf{Adv}^{\mathrm{PRF}}_{\text{Catena}}(q)$ in Theorem 8.2. Since H is called at most $\lceil \ell/n \rceil \cdot q$ times in Line 4, the upper bond for the latter is given by

$$\frac{\lceil \ell/n \rceil \cdot q}{2^n}.$$

8.4 Summary

In this chapter, we have shown that Catena and Catena-KG provide preimage security as well as PRF security.

CHAPTER 9

Catena – Instances of its Components

Ideas do not always come in a flash but by diligent trial-and-error experiments that take time and thought.

CHARLES KUEN KAO

In this chapter, we elaborate on concrete instances of (1) the underlying graph-based structures F_λ, Γ, and Φ of CATENA (see Algorithm 3 in Section 7.2), and (2) the underlying cryptographic hash function H and the hash function H'. For the former, we also consider the impact induced by a different ordering of the graph-based layers, i.e., different placements of Γ within $flap$. Based on the insights gained during this chapter, we provide 14 instances of the CATENA framework in Chapter 10 that are suitable for a wide range of applications.

9.1 Instances of F_λ, Γ, and Φ

Generic Graph-Based Hashing Scheme and the Initialization Function H_{first}. Since four out of five considered instances (BRG^g_λ, SBRG^g_λ, GRG2^g_λ, and GRG3^g_λ) base on the same structure, we can provide the Generic Graph-Based Hashing Scheme GHS^g_λ (see Algorithm 8), where the difference between the four variants is given by an individual indexing function $\rho(i)$. The GHS^g_λ operation requires $\mathcal{O}(2^g)$ invocations of a given hash function H' per stack. It accepts the garlic g, the current state v, as well as the depth λ as input and returns the updated $2^g \cdot k$-bit state v, where k denotes the output length of H' in bits[1]. A call to H' requires two inputs r_{i-1} and $v_{\rho(i)}$ (see Line 13 of Algorithm 8), where r_{i-1} is the predecessor of the currently updated state word and $\rho(i)$ an index determined by the index function $\rho : \{0,1\}^g \to \{0,1\}^g$. A formal definition of the underlying graph-based structure, which we denote as a Generic (g, λ)-Graph Scheme (GGS^g_λ), is shown in Definition 9.1.

[1] Note that $n|k$ always holds, where n denotes the output size of H in bits.

Algorithm 8 Generic (g, λ)-Graph-Based Hashing (GHS^g_λ) and H_{first}

```
Require: g, v, λ          ▷ garlic, state, depth
Ensure: v                 ▷ updated state
procedure GHS^g_λ(g, v, λ)
10: for j = 1, ..., λ do
11:   r_0 ← H_first(v_{2^g−1}, v_{ρ(0)})
12:   for i = 1, ..., 2^g − 1 do
13:     r_i ← H'(r_{i−1} || v_{ρ(i)})
14:   v ← r
15: return v
```

```
Require: v_α, v_β         ▷ two k-bit state words
Ensure: r_0               ▷ k-bit state word
procedure H_first(v_α, v_β)
20: w_0 ← H(v_α || v_β)
21: ℓ ← k/n
22: for i = 1, ..., ℓ − 1 do
23:   w_i ← H(i || w_0)
24: r_0 ← (w_0, ..., w_{ℓ−1})
25: return r_0
```

Definition 9.1 (Generic (g, λ)-Graph Scheme). *Fix a natural number g, let $\mathcal{V}$ denote the set of vertices, and $\mathcal{E}$ the set of edges within this graph. Then, a $GGS^g_\lambda(\mathcal{V}, \mathcal{E})$ consists of $(\lambda+1)\cdot 2^g$ vertices*

$$\{v^j_i\},\ 0 \leq i \leq 2^g - 1,\ 0 \leq j \leq \lambda,$$

and $(2\lambda + 1) \cdot 2^g - 1$ edges as follows:

- $(\lambda + 1) \cdot (2^g - 1)$ *edges* $v^j_{i-1} \to v^j_i$ *for* $i \in \{1, \ldots, 2^g - 1\}$ *and* $j \in \{0, 1, \ldots, \lambda\}$.
- $\lambda \cdot 2^g$ *edges* $v^j_i \to v^{j+1}_{\rho(i)}$ *for* $i \in \{0, \ldots, 2^g - 1\}$ *and* $j \in \{0, 1, \ldots, \lambda - 1\}$.
- λ *additional edges* $v^j_{2^g-1} \to v^{j+1}_0$ *where* $j \in \{0, \ldots, \lambda - 1\}$.

Moreover, all considered instances of the function F_λ work on the same state v (see Line 5 of Algorithm 4) consisting of 2^g k-bit state words, which allows us to provide a generic function H_{first} (see Algorithm 8) computing the first k-bit value r_0 (or v_0 resp.) of the internal state[2]. This initialization function is required since Catena supports hash functions H' with arbitrary k-bit output sizes for which it may hold that $k \neq n$, where n denotes the output size of H in bits. All instances of F_λ consider exactly *two* inputs to the underlying hash function H (or H' resp.) when updating an internal state word. Therefore, the generic initialization function H_{first} takes exactly two k-bit state words v_α and v_β as input and computes the new first k-bit state word r_0 consisting of ℓ pseudorandom values $w_0, \ldots, w_{\ell-1}$, with $\ell \leftarrow k/n$ (see Lines 22-25 of Algorithm 8). The use of H and the loop counter i as its input is based on similar arguments as mentioned for the function H_{init} in Section 7.2.

For each instance of F_λ, we require a *password-independent* memory-access pattern, i.e., the resistance to Cache-Timing Attacks (CTAs) of an instance of Catena must not depend on the instance of F_λ. Nevertheless, if resistance to CTAs is no mandatory requirement and SMH is desirable, one can use the option Φ as introduced in Section 7.2 (see Algorithm 6).

[2]Note that in practice, r_0 is actually the same value as v_0. But, to increase the readability, we always use two variables r and v to represent the internal state in the pseudocode. Obviously, the actual implementation is working on a single variable v.

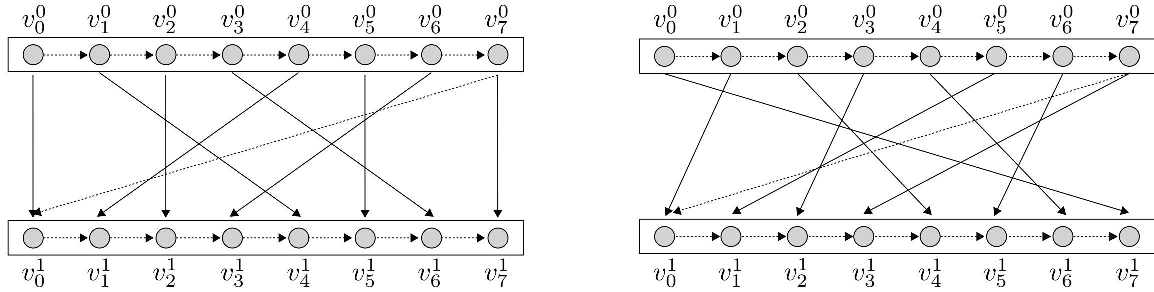

Figure 9.1: A $(3,1)$-Bit-Reversal Graph (BRG_1^3) (left) and a Shifted $(3,1)$-Bit-Reversal Graph (SBRG_1^3) with $c = 1$ (right).

9.1.1 (g, λ)-Bit-Reversal Graph

The graph-based structure of a (g, λ)-Bit-Reversal Graph (BRG_λ^g) is almost identical – except for one additional edge $e = (v_{2^g-1}^0, v_0^1)$ – to the Bit-Reversal Graph (BRG) presented by Lengauer and Tarjan in [170]. The (g, λ)-Bit-Reversal Hashing (BRH_λ^g), based on the GHS_λ^g operation (see Algorithm 8), defines the algorithm following the structure of a BRG_λ^g, where $\rho_{\text{BRG}}(i)$ with $i = (i_0, i_1, \ldots, i_{q-1})$ and $i_j \in \{0,1\}, 0 \le j \le q-1$, is defined by

$$\rho_{\text{BRG}}(i) = (i_{q-1}, \ldots, i_1, i_0).$$

Thus, the function $\rho_{\text{BRG}}(i)$ returns the bitwise reverse of an input i. For example, Figure 9.1 (left) illustrates a BRG_1^3. One structural property of an BRG_λ^g is its self-inverting structure, i.e., $\rho_{\text{BRG}}^2(i) = i,\ \forall i$. This was exploited by the TMTO attacks introduced by Biryukov and Khovratovich in [55], where they have shown that a BRG_λ^g does not satisfy λ-Memory Hardness (LMH). Therefore, we searched for an alternative with the objective of finding a graph structure with less dependencies between the single layers.

9.1.2 Shifted (g, λ)-Bit-Reversal Graph

A Shifted (g, λ)-Bit-Reversal Graph (SBRG_λ^g) can be adjusted by a constant c that avoids the self-inverse property of a BRG_λ^g, i.e., it leads to a less symmetric structure of the underlying permutation. The corresponding indexing function $\rho_{\text{SBRG}}(i)$ of a SBRG_λ^g is given by

$$\rho_{\text{SBRG}}(i) = (\rho_{\text{BRG}}(i) + c) \bmod 2^g.$$

We show an SBRG_1^3 with $c = 1$ in Figure 9.1 (right). Unfortunately, an SBRG_λ^g does not protect well against the TMTO attacks mentioned before, e.g., we have shown for $g = 12$, that even if we consider all possible shift constants c, the penalties (see Remark 9.2) do not exceed that of the (g, λ, ℓ)-Gray-Reverse Graph presented in the next section (see Table 9.1).[3] Based on the structure of an SBRG_λ^{12}, the penalties follow a cyclic property, i.e., it holds that $p_c = p_{c \bmod 16}$ for $0 \le c < 2^{12}$, where p_c denotes the penalty for the shift constant c. Thus, Table 9.1 contains all possible penalties which exist for an SBRG_λ^{12} with $\lambda \in \{2, 3\}$.

[3] The value $g = 12$ was chosen for time reasons.

Remark 9.2. *By **penalty**, we denote the relative additional costs of a TMTO adversary for computing the password hash with less than $2^g \cdot k$ bits of memory. For example, as shown in Table 9.1 for $c = 0$ and when considering the SBRG$_2^{12}$, an adversary with $2^{\lceil g/3 \rceil} \cdot k = 2^4 \cdot k$ bits of memory available requires 22 times the effort of an adversary with $2^g \cdot k$ bits of memory available.*

c	**Penalty**	c	**Penalty**
0	22.00	8	28.31
1	25.03	9	28.78
2	25.50	10	29.25
3	25.96	11	29.71
4	26.43	12	30.18
5	26.90	13	30.65
6	27.37	14	31.12
7	27.84	15	31.59
GRG2$_2^{12}$	54.56	GRG3$_2^{12}$	72.25

c	**Penalty**	c	**Penalty**
0	47.00	8	66.95
1	48.41	9	70.04
2	50.76	10	73.22
3	53.23	11	76.52
4	55.80	12	79.80
5	58.47	13	83.30
6	61.26	14	86.88
7	63.97	15	65.87
GRG2$_3^{12}$	254.93	GRG3$_3^{12}$	223.52

Table 9.1: Relative costs (penalties) for computing an SBRG$_2^{12}$ (left) and an SBRG$_3^{12}$ (right) depending on the shift constant c, where $c = 0$ corresponds to a BRG$_2^{12}$ and BRG$_3^{12}$, respectively. The penalties were computed for the case when an adversary has $\lambda \cdot 2^{g-\sigma} \cdot k$ bits of memory available (with $\sigma = g/3$). For comparison, we added the recomputation costs for a GRG2$_\lambda^{12}$ and a GRG3$_\lambda^{12}$ for $\lambda \in \{2, 3\}$.

It follows that instantiating F_λ with an SBRG$_\lambda^g$ does not significantly increase the resistance of Catena to the attacks presented by Biryukov and Khovratovich [55]. We assume these fact stems from the close relation between a BRG$_\lambda^g$ and an SBRG$_\lambda^g$. Following from this, we took another approach into consideration, called (g, λ, ℓ)-Gray-Reverse Graph.

9.1.3 (g, λ, ℓ)-Gray-Reverse Graph

The GRH$_\lambda^g$ operation (the algorithm defined by a GRGℓ_λ^g) is based on the combination of $\rho_{\mathrm{BRG}_\lambda^g}$ and the Gray Code [121].[4] It was initially suggested by Harris on the PHC mailing list [128] to increase the asymmetry between multiple layers of underlying graph-based permutation. It utilizes the following indexing function ρ_{GRG}:

$$\rho_{\mathrm{GRG}}(i) = \rho_{\mathrm{BRG}}(i) \oplus \left(\rho_{\mathrm{BRG}}\left(\bar{i}\right) \gg \lceil g/\ell \rceil\right),$$

where $\ell \in \{2, 3\}$ is fixed in advance and $\bar{i}$ denotes the bitwise inversion of i. In Figure 9.2, we show a GRG2$_1^3$ (left) and a GRG3$_1^3$ (right). In Table 9.1 and more detailed in Section 9.2, we observe that a GRG2$_\lambda^g$ is strong in comparison to the formerly introduced instances since it leads to a significantly increased penalty when considering the TMTO attacks in [55]. On the other hand, the choice of a GRG3$_\lambda^g$ does not have a significant influence on the penalty in comparison to a

[4] Even if the basic idea (hamming distance of 1 of adjacent indices) is hard to find (if at all), the author described it that way; and we have adopted his description.

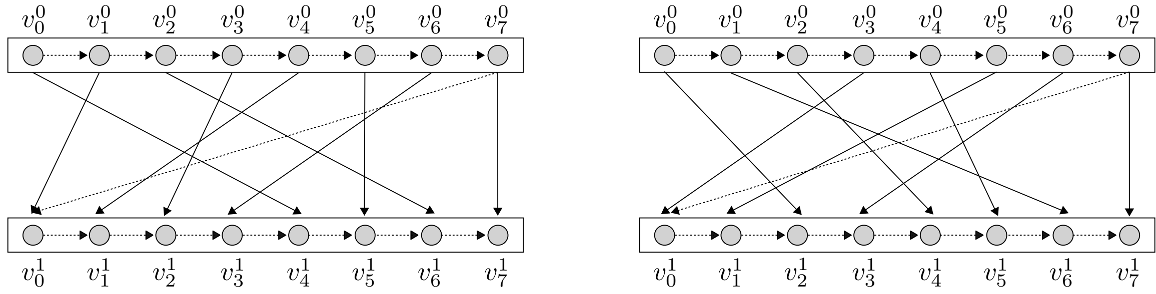

Figure 9.2: A $(3, 1, 2)$-Gray-Reverse Graph (GRG2_1^3) (left) and a $(3, 1, 3)$-Gray-Reverse Graph (GRG3_1^3) (right).

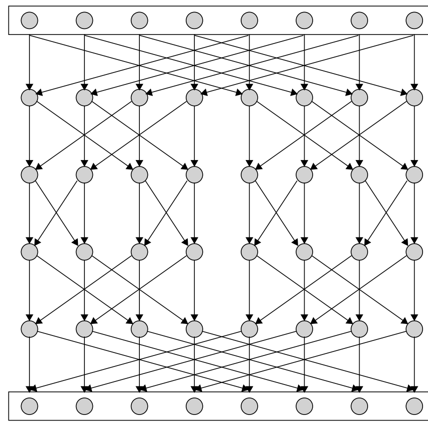

Figure 9.3: A Cooley-Tukey FFT graph with eight input and output vertices.

GRG2_λ^g. Nevertheless, it is an intuitive assumption to say that the increased resistance to TMTO attacks stems more from the general asymmetric structure of a (g, λ, ℓ)-Gray-Reverse Graph that also explains the weak resistance of an SBRG_λ^g to those attacks which were initially designed for a BRG_λ^g.

9.1.4 (g, λ)-Double-Butterfly Graph

The structure of a (g, λ)-Double-Butterfly Graph (DBG_λ^g) does not fit into the generic pattern of Algorithm 8 and Definition 9.1. The basic structure was discussed by Lengauer and Tarjan in [170]. It consists of a stack of λ stacked G-superconcentrators, where the following definition of a G-superconcentrator is a slightly adapted version of that introduced in [170].

Definition 9.3 (G-Superconcentrator). *A directed acyclic graph $\Psi(\mathcal{V}, \mathcal{E})$ with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$, a bounded indegree, G inputs, and G outputs is called a G-superconcentrator if for every ω such that $1 \leq \omega \leq G$ and for every pair of subsets $V_1 \subset \mathcal{V}$ of ω inputs and $V_2 \subset \mathcal{V}$ of ω outputs, there are ω vertex-disjoint paths that connect the vertices in V_1 to the vertices in V_2.*

A Double-Butterfly Graph (DBG) is a special form of a G-superconcentrator which is defined by the graph representation of two mirrored Fast Fourier Transformations [64]. More detailed, it is a representation of two invocations of the Cooley-Tukey FFT algorithm [71], omitting one row in the middle (see Figure 9.3 for an example where $g \leftarrow 3$ and thus, $G = 2^g = 2^3 = 8$). Therefore, a DBG consists of $2 \cdot g$ rows, each consisting of $2^g \cdot k$ state words. Based on the DBG, we define the sequential and stacked (g, λ)-Double-Butterfly Graph (see Definition 9.4) and the corresponding (g, λ)-Double-Butterfly Hashing operation in Algorithm 9, where the indexing function $\rho(g, j, i)$ (see Lines 3 and 5) is defined by

$$\rho(g, j, i) = \begin{cases} i \oplus 2^{g-1-j} & \text{if } 0 \leq j \leq g-1, \\ i \oplus 2^{j-(g-1)} & \text{otherwise.} \end{cases}$$

Thus, ρ determines the indices of the vertices of the diagonal edges (see Figure 9.4).

Definition 9.4 ((g, λ)-Double-Butterfly Graph). *Fix a natural number $g \geq 1$ and let $G = 2^g$. Then, a (g, λ)-Double-Butterfly Graph $DBG^g_\lambda(\mathcal{V}, \mathcal{E})$ consists of $2^g \cdot (\lambda \cdot (2g-1)+1)$ vertices*

- $v^\omega_{i,j},\ 0 \leq i \leq 2^g - 1,\ 0 \leq j \leq 2g-1,\ 1 \leq \omega \leq \lambda$
- $v^\lambda_{i,j},\ 0 \leq i \leq 2^g - 1,\ 0 \leq j \leq 2g-1,$

and $\lambda \cdot (2g-1) \cdot (3 \cdot 2^g) + 2^g - 1$ edges as follows:

vertical: $2^g \cdot (\lambda \cdot (2g-1))$ *edges*
$(v^\omega_{i,j}, v^\omega_{i,j+1})$ *for* $0 \leq j \leq 2g-2, 0 \leq i \leq 2^g - 1$ *and* $1 \leq \omega \leq \lambda,$

diagonal: $2^g \cdot \lambda \cdot g + 2^g \cdot \lambda \cdot (g-1)$ *edges*
$(v^\omega_{i,j}, v^\omega_{i \oplus 2^{g-1-j},j+1}),\ 0 \leq j \leq g-1,\ 0 \leq i \leq 2^g - 1, 1 \leq \omega \leq \lambda.$
$(v^\omega_{i,j}, v^\omega_{i \oplus 2^{j-(g-1)},j+1}),\ g \leq j \leq 2g-2,\ 0 \leq i \leq 2^g - 1,\ 1 \leq \omega \leq \lambda.$

sequential: $(2^g - 1) \cdot (\lambda \cdot (2g-1)+1)$ *edges*
$(v^\omega_{i,j}, v^\omega_{i+1,j}),\ 1 \leq j \leq 2g-1,\ 0 \leq i \leq 2g-2,\ 1 \leq \omega \leq \lambda.$
$(v^\lambda_{i,2g-1}, v^\lambda_{i+1,2g-1}),\ 0 \leq i \leq 2g-2.$

connecting layer: $\lambda \cdot (2g-1)$ *edges*
$(v^\omega_{2^g-1,j}, v^\omega_{0,i+1}),\ 1 \leq \omega \leq \lambda,\ 0 \leq j \leq 2g-2.$

Figure 9.4 illustrates the individual types of edges we used in Definition 9.4 and Figures 9.5 shows a DBG^3_1.

Since providing resistance to TMTO attacks is a crucial goal of a modern and memory-demanding PS, it is desired to implement it in the most efficient way possible to allow for increasing the

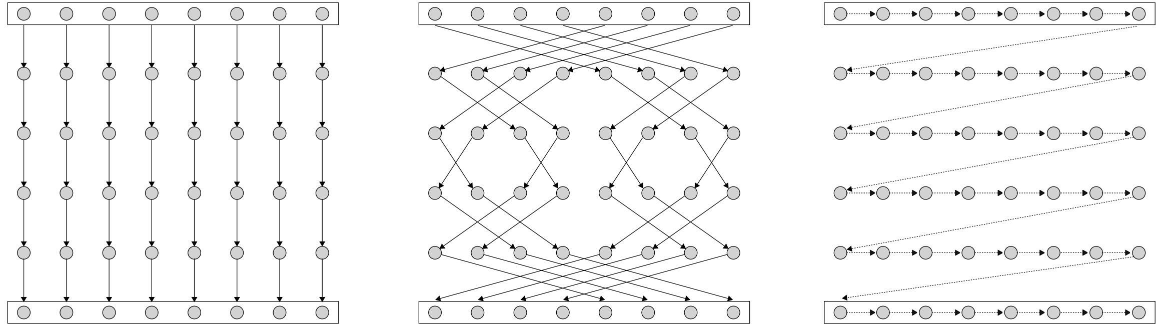

Figure 9.4: Types of edges as we use them in our definitions: **vertical** (left), **diagonal** (center), **sequential** and **connecting layer** (right).

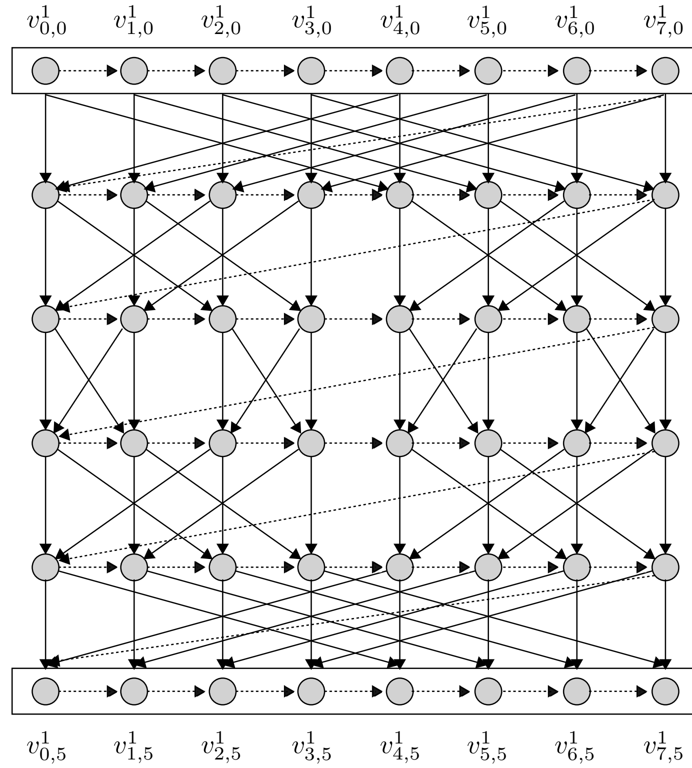

Figure 9.5: A $(3,1)$-double-butterfly graph (DBG^3_1.).

required memory. As mentioned before, all instances of H (and H') take exactly two input words when updating a word of the internal state. This is suitable for all formerly mentioned graph-based hashing operations since a BRG, a Shifted Bit-Reversal Graph (SBRG), as well as a Gray-Reverse Graph (GRG) satisfy a fixed indegree of at most 2. When considering the DBH^g_λ operation, we cannot simply concatenate the inputs to H (and H') while keeping the same performance per hash-function call, i.e., three inputs to H usually require two calls to the underlying compression function, which would be a strong slow-down in comparison to other instances. Therefore, we compute $H(X,Y,Z) = H(X \oplus Y \mid\mid Z)$ instead of $H(X,Y,Z) = H(X \mid\mid Y \mid\mid Z)$ (see Lines 3 and 5 of Algorithm 9) obtaining the same performance as for the BRH^g_λ, SBRH^g_λ, and GRH^g_λ operations per hash-function call.

Algorithm 9 (g, λ)-Double-Butterfly Hashing (DBH^g_λ)

Require: g, v, λ ▷ garlic, state, depth
Ensure: v ▷ updated state
procedure $\mathrm{DBH}^g_\lambda(v)$
1: **for** $k = 1, \ldots, \lambda$ **do**
2: **for** $j = 1, \ldots, 2g - 1$ **do**
3: $r_0 \leftarrow H_{\text{first}}(v_{2^g-1} \oplus v_0, v_{\rho(g,j-1,0)})$
4: **for** $i = 1, \ldots, 2^g - 1$ **do**
5: $r_i \leftarrow H'(r_{i-1} \oplus v_i \mid\mid v_{\rho(g,j-1,i)})$
6: $v \leftarrow r$
7: **return** v

Algorithm 10 The Random-Number Generator `xorshift1024star`

Require: r, p, g ▷ current state of the RNG, garlic
Ensure: idx, r, p ▷ current index, updated state of the RNG
procedure `xorshift1024star`(r, p, g)
1: $s_0 \leftarrow r$
2: $p \leftarrow (p + 1) \bmod 16$
3: $s_1 \leftarrow r$
4: $s_1 \leftarrow s_1 \oplus (s_1 \lll 31)$
5: $s_1 \leftarrow s_1 \oplus (s_1 \gg 11)$
6: $s_0 \leftarrow s_0 \oplus (s_0 \gg 30)$
7: $r \leftarrow s_0 \oplus s_1$
8: $idx \leftarrow r \cdot 1181783497276652981$
9: **return** $(idx \gg (64 - g), r, p)$

Unfortunately, this would then double the probability of an input collision. But, since our instances of H (and H') produce at least 512-bit outputs (see Section 9.3), the success probability for a collision of an adversary is still negligible.

> **Remark 9.5.** *Note that the performance optimization discussed above has no influence on the memory hardness of the DBH^g_λ operation since the first input $X \oplus Y$ is given by XORing vertices from the* sequential *or* connecting layer *and the* vertical *layer. In Section 9.2, we discuss the results of [170] who have shown that even without the sequential inputs, the DBH^g_λ operation provides LMH. Adding additional inputs to that operation does not invalidate their results. The objective of the sequential layer is to harden attacks in a parallel setting (see Section 11.2 for a discussion).*

9.1.5 Instance of Γ and Φ

The TMTO attacks by Biryukov and Khovratovich [55] are only possible due to the predictable memory-access pattern of a BRG^g_λ. Therefore, we introduced the functions Γ and Φ (see Algorithm 6 in Chapter 7.2), which update the state v by accessing its elements dependent on a public input γ (e.g., the salt s), or secret input μ, e.g., the last state word v_{2^g-1} of F_λ, respectively (see Section 7.2). Both functions come with the hope to strengthen an instance of Catena against the implementation

of TMTOs attacks. Furthermore, we claim that such a construction would also increase the costs of attacks with expensive non-reprogrammable hardware, e.g., ASICs: either, an adversary would have to design a new computational circuit for each public/secret input; It would require almost "reprogrammable" hardware to support multiple distinct values of the former.

Instance of both Γ as well as Φ are defined by the functions H, H', and R. The concrete instances for H and H' are discussed in Section 9.3, whereas now, we present our proposed instance of the RNG R (see Algorithm 10). Even if the cryptographic hash function H may appear as a natural choice for an RNG, we preferred the non-cryptographic but statistically sound RNG `xorshift1024star` [259] due to its better performance. Another possible approach was presented by the designers of Argon2 [52], which uses the g LSBs of the previous computed value to determine the new index.

9.2 Security Analysis of the Instances of F_λ

In this section, we first briefly discuss a well-known proof method from theoretical computer science called the *Pebble Game*, which is used to analyze Time-Memory Tradeoffs (TMTOs) for a restricted set of programs. This means, it can be used for determining the memory hardness of a particular algorithm. Even if we do not apply that technique by ourselves, the memory hardness results of the aforementioned instances of F_λ depend on that technique and knowing it helps to understand the underlying proofs given by Lengauer and Tarjan in [170]. Second, we present the memory hardness results of named instances and show the behavior of the graph-based structures in terms of the TMTO attacks presented by Biryukov and Khovratovich [53, 55]. Finally, we complete our security analysis by considering (1) the resistance to CTAs, GCAs, and WGCAs, and (2) the collision security and pseudorandomness of the given instances.

The Pebble Game. In 1970, Hewitt and Paterson [209] introduced the pebble game as a method for analyzing TMTOs on Directed Acyclic Graphs (DAGs), which became an important tool for that purpose (see [237, 238, 241, 252, 255]). Additionally, the pebble game has been occasionally used in cryptographic context (see [90] for a recent example).

The pebble game is meant to analyze TMTOs for a restricted set of programs, which are required to fulfill two conditions. First, the programs must be "straight-line programs", i.e., *without any data-dependent branches*. Thus, neither conditional statements (if-then-else) nor loops are allowed, except when the number of loop-iterations is a fixed number since one can remove such loops by unrolling. Second, reading to or writing from a certain element v_i of an array $v_0, \ldots, v_{n-1}$ in memory is only allowed if the index i is statically determined a priori – and thus, independent from the input. Programs following these two restrictions can be represented by a DAG.

The goal of the pebble game is to determine a TMTO for a given algorithm by pebbling a predetermined vertex within the corresponding DAG, considering a certain amount of available memory, i.e., number of available pebbles. Initially, there is a heap of free pebbles, and no pebbles on the DAG. The player performs a number of certain actions until a predefined output vertex has been pebbled, where the following two actions are possible:

Move: If a vertex v_i is unpebbled and all vertices w_i with edges $w_i \rightarrow v_i$ are pebbled, perform either one of the following two operations:

1. Put a pebble from the heap onto v_i (all w_i remain pebbled).
2. Move a pebble from one of the w_i to v_i (all w_j with $j \neq i$ remain pebbled).

Collect: Remove one pebble from any vertex. The pebble goes back into the heap.

Note that a "move" is either a "read input" operation (if it applies to an input vertex, i.e., one without any edges $w_i \rightarrow v$) or the actual computation of a value. The computational time for a straight-line program is then given by counting the number of moves, whereas the required memory is given by the maximum number of pebbles simultaneously placed on the DAG.

The game explained above is called the *black pebble game*. A variation of the former, studied in [70, 186, 114], is called the *black and white pebble game*, where two types of pebbles exist. As mentioned by Lengauer and Tarjan [170], a few additional rules for white pebbles have to be considered:

- A white pebble can be placed on an empty vertex at any time.
- A white pebble can be removed from a vertex v_i if all its immediate predecessors are pebbled.
- If all but one of the immediate predecessors of a vertex v_i having a white pebble are pebbled, then, the white pebble can be moved from v_i to its unpebbled immediate predecessor.

That very game is used for the analysis of a stack of λ G-Superconcentrators [170] (see Theorem 9.9), whereas the black pebble game is used for the analysis of the BRG (see Theorem 9.6).

9.2.1 Memory Hardness

In [170], Lengauer and Tarjan have proven the lower bound of pebble movements for a $(g, 1)$-Bit-Reversal Graph.

Theorem 9.6 (Lower Bound for a BRG$_1^g$ [170]). *If $S \geq 2$, then, pebbling the Bit-Reversal Graph BRG$_1^g(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input nodes with S pebbles takes time*

$$T > \frac{G^2}{16S}.$$

Since the SBRG$_\lambda^g$ is directly derived from a BRG$_\lambda^g$, we state the following corollary.

Corollary 9.7 (Lower Bound for an SBRG$_1^g$). *Let $S \geq 2$ and $G = 2^g$ denote the number of input nodes. Then, it holds that pebbling an Shifted Bit-Reversal Graph SBRG$_1^g(\mathcal{V}, \mathcal{E})$ with S pebbles takes time*

$$T > \frac{G^2}{16S}.$$

Biryukov and Khovratovich have shown in [55] that stacking more than one BRG only adds some linear factor to the quadratic TMTO. Hence, a BRG$_\lambda^g$ with $\lambda > 1$ does not achieve the properties of a λ-memory-hard function.

In contrast to a BRG$_\lambda^g$ and an SBRG$_\lambda^g$, a GRGℓ_λ^g does not follow such a symmetric structure. Therefore, we state the following conjecture.

Conjecture 9.8 (Lower Bound for an GRGℓ_1^g). *Let $S \geq 2$ and $G = 2^g$ denote the number of input nodes. Then, it holds that pebbling a Gray-Reverse Graph GRG$\ell_1^g(\mathcal{V}, \mathcal{E})$ with S pebbles takes time*

$$T \; > \; \frac{G^2}{16S}.$$

Also by utilizing the pebble game, the authors of [170] analyzed the TMTO for a stack of λ G-superconcentrators. Since the DBG is a special form of a G-superconcentrator, their bound also holds for DBG$_\lambda^g$.

Theorem 9.9 (Lower Bound for a Stack of λ G-Superconcentrators [170]). *Pebbling a stack of λ G-superconcentrators using $S \leq G/20$ black and white pebbles requires T placements such that*

$$T \; \geq \; G\left(\frac{\lambda G}{64S}\right)^{\lambda}.$$

From Theorem 9.9, we can derive Corollary 9.10.

Corollary 9.10 (Lower Bound for a DBG$_\lambda^g$). *If $S \leq G/20$, then, pebbling the Double-Butterfly Graph DBG$_\lambda^g(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input nodes with S pebbles takes time*

$$T \; \geq \; G\left(\frac{\lambda G}{64S}\right)^{\lambda}.$$

Thus, a DBG$_\lambda^g$ fulfills requirement of a λ-memory-hard function as defined in Definition 3.6.

Note that the computational effort for computing the DBH$_\lambda^g$ operation with reasonable values for g, e.g., $g \in [19, 21]$, may stress the patience of many users since the number of vertices and edges grows logarithmically with $G = 2^g$. Thus, it remains an open research problem to find a stack of λ G-superconcentrators – or any other λ-memory-hard function– that can be computed more efficiently than DBH$_\lambda^g$.

Remark 9.11. *All results presented in this section only hold in the so-called* sequential *pebble game as introduced by [170]. A different model considering parallel executions of the underlying graph-based structure was presented by Alwen and Serbinenko [23] and is called the Parallel Random-Oracle Model (pROM). We discuss the results following the pROM in Section 11.2.*

Next, we have a deeper look at the resistance of the instances GRG2$_\lambda^g$ and GRG3$_\lambda^g$ to the precomputation attack described in [55] (see the paper for details on the attack) and we consider both in

comparison to the BRG^g_λ.[5] The reason for disregarding DBG^g_λ from our analysis is given by the fact that the aforementioned attacks were focused on the BRG^g_λ and we were searching for an alternative for that graph structure[6] while preserving a decent performance, where the latter cannot be guaranteed by a DBG^g_λ. Moreover, we disregarded the SBRG^g_λ since it does not defend well against those attacks as already shown in Section 9.1.1.

For our analysis, we first look for the sweet spot (regarding the required memory) for the precomputation attack [55]. Assume that an adversary has $2^{g-\sigma} \cdot k$ bits of memory available (as always, k denotes the size of a state words in bits) that are used to permanently store $2^{g-\sigma}$ words of the internal state v in an equidistant manner. Moreover, we say that an adversary can afford $2^{2\sigma} \cdot k$ bits of memory that are used to store additional state words, temporarily. Then, it is shown in [55] that the memory effort for computing a BRG^g_λ is given by

$$M(\lambda, g, \sigma) \leq \lambda \cdot 2^{g-\sigma} + \lambda \cdot 2^{2\sigma}.$$

We show the required memory M, depending on $\lambda \in \{2, 3\}$ and $g \in \{18, 21\}$, in Figure 9.6. Note that we disregard values of $\sigma/g > 0.5$ since for those values, the precomputation method performs worse than a naive recomputation and moreover, from $\sigma/g = 0.5$ follows $2^{2\sigma} = 2^g$ and thus, the adversary would already have the same amount of memory as the defender. From Figure 9.6, we can observe that the minimal required memory of an adversary is given when $\sigma = g/3$. Therefore, we assume $\sigma = g/3$ for all considered penalty computations independently from the instance of F_λ. Based on the available memory, we compute the relative cost (penalty) an adversary would require in comparison to the case when it has $2^g \cdot k$ memory available (which would be the case for a defender using Catena). First, we will consider graphs with a depth of 2 or 3, i.e., $\lambda \in \{2, 3\}$. Then, we add the optional random layer Γ (see Algorithm 6) and recompute the penalties to determine the impact of the random layer on the precomputation method.

Shifting Sampling Points. The precomputation method of Biryukov and Khovratovich with an optimal tradeoff considers $2^{g-\sigma}, \sigma = g/3$, sampling points stored in each level of the underlying graph-based structure. Thus, an adversary is allowed to store $2^{2g/3}$ memory units per level, where one memory unit is given by one word of internal state v. The sampling points are placed on the internal state so that it consists of $2^{2g/3}$ segments of $2^{g/3}$ state words each (beginning with placing the first sampling point on the first state value v_0). First, we were interested in the fact whether shifting the sampling points by a constant amount (which could differ for each layer) would strengthen the resistance of an instance of F_λ to the precomputation attack method. Thus, for all possible $(2^\sigma)^\lambda = 2^{\sigma \cdot \lambda}$ shift configurations of the sampling points (where the sampling points were still ordered with a distance of $2^{g/3}$ state words), we recomputed the penalty of an adversary. The results can be seen in Table 9.2, where we only considered the minimum and maximum values for the penalty. The values in brackets denote the shift constants which are applied to the particular layers, e.g., 67.61 $(4, 7)$ denotes that all sampling points in the first layer are shifted by four positions to the right, whereas the sampling points in the second layer are shifted by seven positions to the right.

From Table 9.2, we conclude that shifting the sampling points on the internal state does not significantly help an adversary. Therefore, all following results are based on the same configuration of sampling points as used in the original attack [55]. Nevertheless, we leave the question open if an

[5] Most of the upcoming discussion is an only slightly adopted version of that presented in [174].

[6] The precomputation attack was the actual reason for introducing the SBRG^g_λ and $\mathrm{GRG}\ell^g_\lambda$ with the hope that those graphs provide a better resistance.

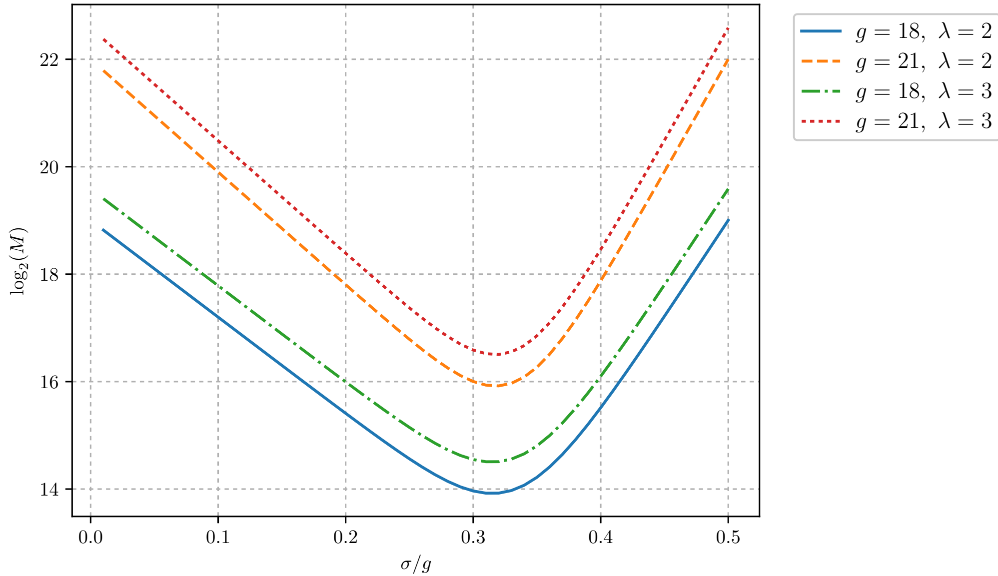

Figure 9.6: Required memory for the recomputation method shown in [55], depending on the garlic g and the depth λ.

Graph	No Shift	Min. Penalty		Max. Penalty	
BRG_2^{12}	25.03	25.00	(0,7)	31.59	(1,0)
GRG2_2^{12}	54.56	54.35	(14,12)	67.61	(4,7)
GRG3_2^{12}	74.25	72.88	(5,8)	74.25	(0,0)
BRG_3^{12}	47.59	46.70	(0,0,15)	90.13	(1,1,14)
GRG2_3^{12}	254.93	246.72	(12,13,4)	296.54	(6,4,11)
GRG3_3^{12}	223.52	217.64	(8,6,15)	262.36	(11,4,10)

Table 9.2: Relative costs (penalty) depending on the shift of the sampling points and the depth λ.

adversary could obtain a significant lower penalty when considering an optimal distribution of all $\lambda \cdot 2^{2g/3}$ sampling points over the whole graph.

Naive Recomputation vs. Precomputation Method. First, we briefly discuss the difference between the naive recomputation approach and the precomputation method, where the former serves as a base for the latter. The configuration of the sampling points is the same for both attack methods, i.e., an adversary stores $\lambda \cdot 2^{g-\sigma}$ sampling points over the whole graph with a constant distance of 2^σ state words between each other. For the naive approach, the sampling points already determine the required memory for an adversary, whereas the precomputation method allows to use additional memory in each layer to speed-up the recomputation. In Table 9.3, we show the memory requirement for both attacks depending on the garlic g and depth λ. Note that the values shown for the precomputation method are given by the maximum additional memory which is required within one layer. For comparison, we also include the memory requirement for one layer

when conducting the naive recomputation approach which solely consists of the sampling points. For simplicity, we assume $H = H'$ so that all state words are of n bits in size. Therefore, the values for the precomputation method given in Table 9.3 denote the number of state words which have to be stored (excluding the fixed sampling points).

		$\lambda = 2$		$\lambda = 3$	
Attack	**Graph**	**g = 18**	**g = 21**	**g = 18**	**g = 21**
Precomp.	BRG_λ^g	$2^{11.98}$	$2^{13.98}$	$2^{11.98}$	$2^{11.98}$
	$\mathrm{GRG2}_\lambda^g$	$2^{14.95}$	$2^{16.98}$	$2^{17.95}$	$2^{19.98}$
	$\mathrm{GRG3}_\lambda^g$	$2^{17.95}$	$2^{20.98}$	$2^{17.95}$	$2^{20.98}$
Naive	**all**	2^{13}	2^{15}	$2^{13.58}$	$2^{15.58}$

Table 9.3: Memory requirement depending on the garlic g, the attack method, the particular graph instance, and the depth λ. All values refer to the maximum memory required within one layer.

From Table 9.3, we can deduce that the precomputation method is highly optimized for the application to a BRG_λ^g. We observed that for $\lambda \in \{2, 3\}$, the instance given by a $\mathrm{GRG3}_\lambda^g$ massively thwarts an adversary since its memory savings are negligible. For example, for $g = 18$ and $\lambda = 2$, it would have to store 2^{12} state words for the fixed sampling points plus $2^{17.95}$ additional state words, leading to a total amount of about $2^{17.97}$ state words. The same holds for a $\mathrm{GRG2}_3^g$, whereas at least some memory is saved when $\lambda = 2$. For the sake of completeness, the relative costs (penalties) of a naive recomputation and a precomputation attack on the instances given above are shown in Table 9.4. Considering the naive approach, the penalties do not differ between the graph instances since the underlying structure is given by a permutation, i.e., all values from the previous layer are used to compute the current layer.

		$\lambda = 2$		$\lambda = 3$	
Attack	**Graph**	**g = 18**	**g = 21**	**g = 18**	**g = 21**
Precomp.	BRG_λ^g	32.33	64.33	47.81	95.87
	$\mathrm{GRG2}_\lambda^g$	150.11	334.08	1051.77	2344.44
	$\mathrm{GRG3}_\lambda^g$	352.75	1387.33	912.11	3666.58
Naive	**all**	352.75	1387.41	5895.25	45411.00

Table 9.4: Penalties depending on the garlic g, the attack method, the particular graph instance, and the depth λ.

Even with the enormous extra costs in terms of memory, the application of the precomputation method to a $\mathrm{GRG}\ell_\lambda^g$ leads to a significantly higher penalty in comparison to the same attack on a BRG_λ^g. Thus, we can conclude that the instances of F_λ given by a $\mathrm{GRG}\ell_\lambda^g$ provide a strong resistance to the TMTO attacks introduced by Biryukov et al. Nevertheless, there might exist attacks which are specifically focused on such instances, hence, reducing the penalty of an adversary. At the moment, we are not aware of such attacks and leave it as an open research question.

Impact of the Random Layer Γ. As shown in Algorithms 6 and 10, the function Γ overwrites $2^{\lceil 3g/4 \rceil}$ randomly chosen state words. With respect to the memory requirement in Table 9.3, it is intuitive to say that the penalty for a $\mathrm{GRG}\ell_\lambda^g$ will not significantly increase since almost the whole state is already stored. On the other hand, when considering a BRG_λ^g, one would expect an increased penalty.

First, we wanted to know if the currently designated position of Γ (invoked before F_λ, see Algorithm 4) provides the best possible resistance (in comparison to other chosen positions) to the precomputation attack. Therefore, we set $g = 12$ (for time reasons), and recomputed the penalties for different positions of Γ within F_λ, for $\lambda \in \{2,3\}$, and the graph instances $\mathrm{GRG2}_2^{12}$ and $\mathrm{GRG3}_2^{12}$ as before. Further, we set the public input γ to the salt and fixed it to a constant value ($\gamma = 0$). This as a valid approach since if R is instantiated with a good RNG (e.g., the function `xorshift1024star` given in Algorithm 10), all indices computed during the invocation of Γ are pseudorandom. The penalties depending on the positions of Γ are shown in Table 9.5, where we can observer that placing the random layer Γ at the third-last position leads to the highest penalty.

	Layer Structure				
	$\lambda = 2$		$\lambda = 3$		
Graph	$\Gamma F_{\lambda 0} F_{\lambda 1}$	$F_{\lambda 0} \Gamma F_{\lambda 1}$	$\Gamma F_{\lambda 0} F_{\lambda 1} F_{\lambda 2}$	$F_{\lambda 0} \Gamma F_{\lambda 1} F_{\lambda 2}$	$F_{\lambda 0} F_{\lambda 1} \Gamma F_{\lambda 2}$
$\mathrm{BRG}_\lambda^{12}$	33.34	31.84	62.44	103.81	81.27
$\mathrm{GRG2}_\lambda^{12}$	62.02	61.39	270.68	306.48	288.73
$\mathrm{GRG3}_\lambda^{12}$	79.68	81.06	248.48	264.10	257.30

Table 9.5: Penalties depending on the position of Γ within F_λ, the particular graph instance, where $F_{\lambda i}$ denotes the i-th layer of F_λ and $g = 12$, and the depth $\lambda \in \{2,3\}$.

Thus, we assumed the configurations $\Gamma F_{\lambda 0} F_{\lambda 1}$ and $F_{\lambda 0} \Gamma F_{\lambda 1} F_{\lambda 2}$ when computing the impact of the random layer, where we choose $g \in \{18, 21\}$ for the garlic as before. Our first intuition was confirmed by the results shown in Table 9.6. Note that due to time issues, the values for the precomputation method with $\lambda = 3$ and $g = 21$ are rough estimations based on the values for $\lambda = 3$ in Table 9.4.

		$\lambda = 2$		$\lambda = 3$	
Attack	**Graph**	**g = 18**	**g = 21**	**g = 18**	**g = 21**
	BRG_λ^g	54.11	107.36	417.03	836.24
Precomp.	$\mathrm{GRG2}_\lambda^g$	169.18	374.66	1386.31	3090.14
	$\mathrm{GRG3}_\lambda^g$	365.40	1411.90	1163.19	4675.89
Naive	**all**	376.59	1432.87	6456.97	47581.00

Table 9.6: Penalties depending on the garlic g, the attack method, the particular graph instance extended by a random layer Γ, and the depth $\lambda \in \{2,3\}$.

By comparing the values presented in Table 9.4 and Table 9.6, we observed that the penalties increase monotonically only for a BRG_λ^g. Thus, we can conclude that the additional invocation of Γ helps substantially against precomputation attacks since the penalties given for a BRG_λ^g are

still much smaller than that of a $\mathrm{GRG}\ell_\lambda^g$. Nevertheless, the main aspect of including Γ in our core function *flap* was the increased resistance to ASIC-based adversaries, and not to thwart TMTO attacks. Furthermore, the penalties presented in Table 9.6 hold only if an adversary does not have the additional $2^{\lceil 3g/4 \rceil} \cdot k$ bits of memory available.

9.2.2 Resistance to Side-Channel Attacks

Straightforward implementations of all presented instances of F_λ follow neither a password-dependent memory-access pattern nor password-dependent branches. Therefore, those instances are resistant to CTAs (see Definition 3.9) as long as H and H' provide resistance to CTAs.

When considering GCAs (see Definition 3.10), we have to consider only the functions GHS_λ^g (see Algorithm 8) and DBH_λ^g (see Algorithm 9) since all instances of GHS_λ^g differ only in their indexing function but none of them requires a password-dependent input. Both hashing operations maintain the arrays v and r, which are overwritten multiple times (depending on the choice of λ). Assume $g = 3$ and $\lambda = 1$. Then, for GHS_2^3, the array v is overwritten twice and the array r once, whereas for DBH_2^3, v is overwritten ten times and r nine times. Thus, even for $\lambda = 1$, both algorithms provide resistance to GCAs. Furthermore, it follows that *any variant of* Catena *with some fixed* $\lambda \geq 2$ *is at least as resistant to GCAs as the same variant with* $\lambda - 1$ *layers in the absence of a malicious garbage collector.*

The resistance to WGCAs (see Definition 3.11) is provided by two facts. First, the password *pwd* is overwritten immediately after it is processed (see Line 1 of Algorithm 3). Second, the value $x \leftarrow H(t \,||\, pwd \,||\, s)$ (which also could be fetched from memory to launch a WGCA), is overwritten by the call to *flap* in Line 2 of Algorithm 3 using only $\lceil g_{\mathrm{low}}/2 \rceil$.

Remark 9.12. *Note that CTAs, GCAs, and WGCAs have even more severe consequences. Not only do they speed-up regular password-recovery attacks where the password hash is already in possession of the adversary, but also enable an adversary* **A** *to recover a password without knowing the password hash at all, e.g., by verifying the memory-access pattern (CTA) or reading valuable information from the memory (GCA and WGCA).*

9.2.3 Pseudorandomness

For proving the pseudorandomness of the underlying graph-based hashing operations, we refer to the PRF security (see Definition 3.2). Therefore, we set $H = H'$ and model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle. The upper bounds presented in that very section depend on the number of calls to H. We start by considering the function *flap*, consisting of its initialization step (including the call to H_{init}; see Lines 1-3 of Algorithm 4), a call to the option Γ (see Line 4), the memory-hard function F_λ (see Line 5), and the option Φ (see Line 6).

Theorem 9.13 (PRF Security of *flap*). *Let q denote the number of queries made by an adversary and $s \twoheadleftarrow \{0,1\}^{|s|}$. Furthermore, let H be modeled as a random oracle. Then,*

$$\mathbf{Adv}_{flap}^{PRF}(q) \;\leq\; \frac{4q^2}{2^{n-2g}} + \mathbf{Adv}_{\Gamma}^{COLL}(q \cdot g) + \mathbf{Adv}_{F_\lambda}^{COLL}(q \cdot g) + \mathbf{Adv}_{\Phi}^{COLL}(q \cdot g).$$

Proof. By looking at Algorithm 4, we can observe that the PRF security of $flap$ is determined by the probability of a collision of the form $flap(g, x, \gamma) = flap(g, x', \gamma)$ for $x \neq x'$. On the other hand, that implies a collision for H. We upper bound the collision probability for H by deducing the total amount of invocations of H per query, where the respective bounds for Γ, Φ, and F_λ are given in Theorems 9.16, 9.17, and 9.14 or 9.15, depending on the particular instance of F_λ. Thus, it remains to upper bound the collision probability of H for the initialization step of $flap$. Since we set $H = H'$, the function H_{init} (see Algorithm 5) requires only two calls to H since for $n = k$ it holds that $\ell = 2k/n = 2$. In addition, there are 2^g invocations of H in Lines 2-3 of Algorithm 4. Since we model H as a random oracle, we can upper bound the collision probability for the initialization step for q queries by

$$\frac{(q \cdot (2^g + 2))^2}{2^n} \leq \frac{4q^2}{2^{n-2g}}.$$

Thus, we have

$$\mathbf{Adv}^{\mathrm{PRF}}_{flap}(q) \leq \frac{4q^2}{2^{n-2g}} + \mathbf{Adv}^{\mathrm{COLL}}_{\Gamma}(q \cdot g) + \mathbf{Adv}^{\mathrm{COLL}}_{F_\lambda}(q \cdot g) + \mathbf{Adv}^{\mathrm{COLL}}_{\Phi}(q \cdot g).$$

□

Next, we consider the generic hashing scheme GHS^g_λ and the hashing operation DBH^g_λ as they cover all instances of F_λ which we have presented earlier in this chapter.

Theorem 9.14 (Collision Security of GHS^g_λ). *Let q denote the number of queries made by an adversary and $s \twoheadleftarrow \{0,1\}^{|s|}$. Furthermore, let H be modeled as a random oracle. Then,*

$$\mathbf{Adv}^{COLL}_{GHS^g_\lambda}(q) \leq \frac{(q \cdot \lambda)^2}{2^{n-2g}}.$$

Proof. It is easy to see from Algorithm 8 that a collision $\mathrm{GHS}^g_\lambda\lambda(x) = \mathrm{GHS}^g_\lambda(x')$ for $x \neq x'$ implies a collision for H. We upper bound the collision probability for H by deducing the total amount of invocations of H per query. There are $\lambda \cdot 2^g$ invocations in Lines 10-14 of Algorithm 8. Since H is modeled as a random oracle, we can upper bound the collision probability for q queries by

$$\frac{(q \cdot \lambda \cdot 2^g)^2}{2^n} \leq \frac{(q \cdot \lambda)^2}{2^{n-2g}}.$$

Thus, our claim follows. □

Now, we analyze the collision resistance of DBH^g_λ. Again, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle.

Theorem 9.15 (Collision Security of DBH^g_λ). *Let q denote the number of queries made by an adversary and $s \twoheadleftarrow \{0,1\}^{|s|}$. Furthermore, let H be modeled as a random oracle. Then,*

$$\mathbf{Adv}^{COLL}_{DBH^g_\lambda}(q) \leq \frac{(q \cdot \lambda \cdot g)^2}{2^{n-2g-3}}.$$

Proof. It is easy to see from Algorithm 9 that a collision $\mathrm{DBH}^g_\lambda(x) = \mathrm{DBH}^g_\lambda(x')$ for $x \neq x'$ implies either an input or an output collision for H. For our analysis, we replace the random oracle H by $H^t(x) := H(\mathrm{truncate}_n(x))$ that truncates any input to n bits before hashing. Thus, any collision in the first n bits of the input of H in Lines 3 and 5 of Algorithm 9 leads to a collision of the output of H^t, regardless of the remaining inputs.

Output Collision. In this case, we upper bound the collision probability for H by deducing the total amount of invocations of H^t per query. There are 2^g invocations of H^t in Lines 2-3 (initialization) of Algorithm 4. In addition, there are $\lambda \cdot (2g-1) \cdot 2^g$ invocations in Lines 3-5 of Algorithm 9 leading to a total of $\lambda \cdot 2g \cdot 2^g$ invocations of H^t. Since H (and thus H^t) is modeled as a random oracle, we can upper bound the collision probability for q queries by

$$\frac{(q \cdot \lambda \cdot 2g \cdot 2^g)^2}{2^n} \leq \frac{(q \cdot \lambda \cdot g)^2}{2^{n-2g-2}}.$$

Input Collision. In this case we have to take into account that an input collision for distinct queries a and b in Lines 3 and 5 of Algorithm 9 can occur:

$$v^a_{2^g-1} \oplus v^a_0 = v^b_{2^g-1} \oplus v^b_0 \qquad \text{(Algorithm 9, Line 3)}$$

or

$$r^a_{i-1} \oplus v^a_i = r^b_{i-1} \oplus v^b_i \qquad \text{(Algorithm 9, Line 5).}$$

For each query, this can happen $\lambda \cdot (2g-1) \cdot 2^g$ times. Note that all values v_i and r_i are outputs from the random oracle H^t, except the initial value v_0. Hence, we can upper bound the collision probability for this event by

$$\frac{(q \cdot \lambda \cdot (2g-1) \cdot 2^g)^2}{2^n} \leq \frac{(q \cdot \lambda \cdot g)^2}{2^{n-2g-2}}.$$

Our claim follows from the union bound. □

Finally, we present the collision security of Γ and Φ.

Theorem 9.16 (Collision Security of Γ). *Let q denote the number of queries made by an adversary and $s \twoheadleftarrow \{0,1\}^{|s|}$. Furthermore, let H be modeled as a random oracle. Then,*

$$\mathbf{Adv}^{COLL}_{\Gamma}(q) \leq \frac{q^2}{2^{n-\lceil 3g/4 \rceil}}.$$

Proof. The function Γ (see Algorithm 6) overwrites $2^{\lceil 3g/4 \rceil}$ randomly chosen state words v_i. By similar arguments as in the proof for Theorem 9.14, we can upper bound the collision probability for q queries by

$$\frac{\left(q \cdot 2^{\lceil 3g/4 \rceil}\right)^2}{2^n} \leq \frac{q^2}{2^{n-\lceil 3g/4 \rceil}}.$$

□

Theorem 9.17 (Collision Security of Φ). *Let q denote the number of queries made by an adversary and $s \twoheadleftarrow \{0,1\}^{|s|}$. Furthermore, let H be modeled as a random oracle. Then,*

$$\mathbf{Adv}_{\Phi}^{COLL}(q) \leq \frac{q^2}{2^{n-2g}}.$$

Proof. The function Φ (see Algorithm 6) overwrites 2^g randomly chosen state words v_i. By similar arguments as in the proof for Theorem 9.14, we can upper bound the collision probability for q queries by

$$\frac{(q \cdot 2^g)^2}{2^n} \leq \frac{q^2}{2^{n-2g}}.$$

□

Remark 9.18. *The proofs in this section clearly hold when $H = H'$, which is our recommendation for* Catena-KG*. For the case $H \neq H'$, the PRF security of* Catena *may be weakened due to a poor choice for H', e.g., the choice $H'(x) = c$ for all possible values x and a fixed constant c would lead to the same password hash independently from any input parameter. However, the instances of H' used in our recommended instances of* Catena *(see Chapter 10) are well-analyzed and should have only a negligible effect (if at all) on the PRF security of* Catena*.*

9.3 Instances of H and H'

Since the choice of H has a significant impact on the security of an instance of Catena (see Theorems 8.1, 8.2, 8.3, 9.14, and 9.15), the criteria are straightforward: H has to be a cryptographic hash function (see Definition 3.3). On the other hand, the function H' can be a reduced variant of H or any other reasonable secure[7] function which is able to compress two k-bit input values to one k-bit output value.[8] In total, we discuss eight instances[9] and compare them regarding to their throughput in gigabyte per second (GB/s) and their running time when employed in one of the recommended instances of Catena, i.e., Catena-Dragonfly (see Section 10 for details about Catena-Dragonfly and Section 9.3.4 for the comparison).

For the practical application of Catena, we looked first for a hash function with a 512-bit (64 byte) output since this often complies with the size of a cache line on common CPUs (e.g., [73]). In any case, we assume that both output sizes of H and H' and the cache-line size are powers of two. So, if they differ, the bigger number is a multiple of the smaller one. Inspired by Argon2 [52], we also provide instances of H' with a larger output size, i.e., $k > n$ with e.g., $n = 512$ and $k = 8192$ bits. This allows an instance of Catena to fill the memory significantly faster.

[7] For example, H' must provide resistance to side-channel attacks if that is a requirement on Catena. Moreover, H' must not be a constant function or the identity. In general, H' should be an entropy-preserving one-way function.

[8] Note that the restriction $H = H'$ is only given for the usage of Catena as a KDF.

[9] Note that we do not provide an extra paragraph for SHA-512 since it is an already well-known standard. Moreover, it is well-analyzed [27, 124, 154], standardized, and widely used, e.g., in `sha512crypt`, the common PS in several Linux distributions [82].

Remark 9.19. *Note that choosing $H \neq H'$ has the additional side effect of frustrating well-funded adversaries who use expensive non-reprogrammable hardware. Thus, for every instance of* Catena *using different instances of H and H', the adversaries would have to buy new hardware, significantly increasing their costs.*

Remark 9.20. *The security of* Catena *relies not only on the performance of a specific hash function, but also on the size of the underlying graph (GGS^g_λ or DBG^g_λ), i.e., the depth λ and the width g. Thus, even in the case of a secure but very fast cryptographic hash function, which may be counter-intuitive for a PS, one can adapt the security parameter to reach the desired computational effort.*

For two instances of H', we also discuss the functions G_B of BlaMka [244] as a possible variant of the underlying permutation. For simplicity, we denote by G_O the underlying permutation of the original BLAKE2b and by G_L the underlying permutation of Lyra2 [244]. Note that G_B is a variant of G_L extended by 32-bit multiplications.

9.3.1 BLAKE2b and BLAKE2b-1

Our recommended instance of H is BLAKE2b [33] (see Algorithm 11), which possesses a block size of 1024 bits with 512 bits of output. Thus, it can process two input blocks within one compression function call. Its high performance in software allows to use a large value for the garlic g, resulting in higher memory effort than for, e.g., SHA3-512 [48]. We used BLAKE2b (consisting of twelve rounds) also as a basis of H' by introducing BLAKE2b-1, which is actually a single round of BLAKE2b including its finalization (see Algorithm 12). The difference between the function BLAKE2b-1 and the original BLAKE2b can be seen in Algorithms 11 and 12, where the lengths are given in bytes.

Following from Line 1 in Algorithm 11, BLAKE2b initializes the internal state S in every invocation, whereas BLAKE2b-1 does not. Thus, for Catena, S is only (re)initialized when computing the first value of each layer of the underlying graph structure using H as specified for example in Algorithms 8 and 9. This assures that twelve invocations of BLAKE2b-1 are as close as possible to the original BLAKE2b and also saves computations as shown in Table 9.7. To further ensure this similarity, we compute the round index as shown in Line 5 of Algorithm 12.

Remark 9.21. *Note that we iterate H' = BLAKE2b-1 thousands of times and have implemented it in a compatible way to H, i.e., twelve times the application of H' (excluding finalization except for the last step) is similar to one times the application of H. Thus, we (informally) assume that twelve times the application of H' provides similar security as one invocation of H.*

The functions blake2b_init, increment_counter, and set_last_block are used as specified in the reference implementation of BLAKE2b [245]. For BLAKE2b-1, we fixed the input length to

Algorithm 11 BLAKE2b

Require: I, ℓ, m ▷ input array, input length, output length
Ensure: h ▷ hash value
procedure BLAKE2b(I, ℓ, m)
1: BLAKE2B_INIT(S, m) ▷ init state
2: **for** $i = 0, \ldots, \lfloor \ell/128 \rfloor - 1$ **do**
3: S.BUF ← $I[i]$
4: S.BUFLEN ← 128
5: INCREMENT_COUNTER(S, S.BUFLEN)
6: COMPRESS(S)
7: S.BUF ← $I[\lfloor l/128 \rfloor] \,||\, 0^*$
8: S.BUFLEN ← ℓ mod 128
9: INCREMENT_COUNTER(S, S.BUFLEN)
10: SET_LAST_BLOCK(S)
11: COMPRESS(S)
12: $h \leftarrow S.h$
13: **return** h

Algorithm 12 BLAKE2b-1

Require: i_1, i_2, j ▷ first input, second input, vertex index
Ensure: h ▷ hash value
procedure BLAKE2b-1(i_1, i_2, j)
1: S.BUF ← $i_1 \,||\, i_2$
2: S.BUFLEN ← 128
3: INCREMENT_COUNTER(S, S.BUFLEN)
4: SET_LAST_BLOCK(S)
5: $r \leftarrow j$ mod 12
6: COMPRESS(S, r)
7: $h \leftarrow S.h$
8: **return** h

Algorithm	Cycles per Byte
BLAKE2b	9.81
BLAKE2b-1	2.44
BLAKE2b-1 w/o initialization	0.86

Table 9.7: Benchmark comparison of BLAKE2b-1, BLAKE2b-1 without initialization, and BLAKE2b. Timings are measured on an Intel(R) Core(TM) i7-3930M CPU @ 3.20GHz with Turbo Boost and Hyper-Threading enabled. The values denote the median of 1001 runs processing a single message block, each.

128 bytes and neglect the padding since the size of the inputs never changes when using BLAKE2b-1 within Catena. For the sake of completeness, we also show the core functions of BLAKE2b and BLAKE2b-1, i.e., the functions COMPRESS from Algorithms 13 and 14, where σ denotes the message schedule and G_O [33] the underlying permutation. There are mainly two differences for the function COMPRESS of BLAKE2b-1 in comparison to that of BLAKE2b: (1) it requires the round index r as input and (2) the **for** loop of the original BLAKE2b (see Lines 5-14 of Algorithm 11) is invoked only once to behave like a single round.

Algorithm 13 Function *compress* of BLAKE2b

Require: S ▷ BLAKE2b state
Ensure: S ▷ updated BLAKE2b state
procedure COMPRESS(S)
1: $v[0\dots 7] \leftarrow S.h$
2: $v[8\dots 15] \leftarrow \mathrm{IV}$
3: $v[12,13] \leftarrow v[12,13] \oplus S.t$
4: $v[14,15] \leftarrow v[14,15] \oplus S.f$
5: **for** $r = 0,\dots,11$ **do**
6: $s[0\dots 15] \leftarrow \sigma[r \mod 10][0\dots 15]$
7: $v \leftarrow G_O(v, 0, 4, 8, 12, S.\textsc{buf}[s[0]], S.\textsc{buf}[s[1]])$
8: $v \leftarrow G_O(v, 1, 5, 9, 13, S.\textsc{buf}[s[2]], S.\textsc{buf}[s[3]])$
9: $v \leftarrow G_O(v, 2, 6, 10, 14, S.\textsc{buf}[s[4]], S.\textsc{buf}[s[5]])$
10: $v \leftarrow G_O(v, 3, 7, 11, 15, S.\textsc{buf}[s[6]], S.\textsc{buf}[s[7]])$
11: $v \leftarrow G_O(v, 0, 5, 10, 15, S.\textsc{buf}[s[8]], S.\textsc{buf}[s[9]])$
12: $v \leftarrow G_O(v, 1, 6, 11, 12, S.\textsc{buf}[s[10]], S.\textsc{buf}[s[11]])$
13: $v \leftarrow G_O(v, 2, 7, 8, 13, S.\textsc{buf}[s[12]], S.\textsc{buf}[s[13]])$
14: $v \leftarrow G_O(v, 3, 4, 9, 14, S.\textsc{buf}[s[14]], S.\textsc{buf}[s[15]])$
15: $S.h \leftarrow S.h \oplus v[0\dots 7] \oplus v[8\dots 15]$

Algorithm 14 Function *compress* of BLAKE2b-1

Require: S, r ▷ BLAKE2b state, round index
Ensure: S ▷ updated BLAKE2b state
procedure COMPRESS(S, r)
1: $v[0\dots 7] \leftarrow S.h$
2: $v[8\dots 15] \leftarrow \mathrm{IV}$
3: $v[12,13] \leftarrow v[12,13] \oplus S.t$
4: $v[14,15] \leftarrow v[14,15] \oplus S.f$
5: $s[0\dots 15] \leftarrow \sigma[r \mod 10][0\dots 15]$
6: $v \leftarrow G_O(v, 0, 4, 8, 12, S.\textsc{buf}[s[0]], S.\textsc{buf}[s[1]])$
7: $v \leftarrow G_O(v, 1, 5, 9, 13, S.\textsc{buf}[s[2]], S.\textsc{buf}[s[3]])$
8: $v \leftarrow G_O(v, 2, 6, 10, 14, S.\textsc{buf}[s[4]], S.\textsc{buf}[s[5]])$
9: $v \leftarrow G_O(v, 3, 7, 11, 15, S.\textsc{buf}[s[6]], S.\textsc{buf}[s[7]])$
10: $v \leftarrow G_O(v, 0, 5, 10, 15, S.\textsc{buf}[s[8]], S.\textsc{buf}[s[9]])$
11: $v \leftarrow G_O(v, 1, 6, 11, 12, S.\textsc{buf}[s[10]], S.\textsc{buf}[s[11]])$
12: $v \leftarrow G_O(v, 2, 7, 8, 13, S.\textsc{buf}[s[12]], S.\textsc{buf}[s[13]])$
13: $v \leftarrow G_O(v, 3, 4, 9, 14, S.\textsc{buf}[s[14]], S.\textsc{buf}[s[15]])$
14: $S.h \leftarrow S.h \oplus v[0\dots 7] \oplus v[8\dots 15]$

9.3.2 Compression Function of Argon2

In this section, we consider the underlying compression function of the PHC winner Argon2 [52]. The function CF[10] is built upon a permutation P using the round function G_L as defined in Lyra2 [244] and is formally defined in Algorithm 15. Note that the function G_L, in comparison to the original round function G_O of BLAKE2b, considers neither the message schedule σ nor processing the message input. The input state words are written directly to the internal 256-bit state (a, b, c, d)

[10] Within the specification of Argon2, this function is called G, but since we use G already for the variants of the round function of BLAKE2b, we rename it here to CF.

Algorithm 15 Compression Function CF of Argon2 [52]

Require: X, Y ▷ two 8192-bit values fetched from memory
Ensure: Z ▷ intermediate hash (8192 bits)
procedure CF(X, Y)
1: $R \leftarrow X \oplus Y$
2: **for** $i = 0, \ldots, 7$ **do**
3: $(Q_{8i}, \ldots, Q_{8i+7}) \leftarrow P(R_{8i}, \ldots, R_{8i+7})$ ▷ updating the rows
4: **for** $i = 0, \ldots, 7$ **do**
5: $(Q_i, Q_{i+8}, \ldots, Q_{i+56}) \leftarrow P(R_i, R_{i+8} \ldots, R_{i+56})$ ▷ updating the columns
6: $Z \leftarrow R \oplus Q$
7: **return** Z

Algorithm 16 Functions G_L (left) and G_B (right)

Require: a, b, c, d ▷ 256-bit state
Ensure: a, b, c, d ▷ updated state
procedure $G_L(a, b, c, d)$
1: $a \leftarrow a + b$
2: $d \leftarrow (d \oplus a) \ggg 32$
3: $c \leftarrow c + d$
4: $b \leftarrow (b \oplus c) \ggg 24$
5: $a \leftarrow a + b$
6: $d \leftarrow (d \oplus a) \ggg 16$
7: $c \leftarrow c + d$
8: $b \leftarrow (b \oplus c) \ggg 63$

Require: a, b, c, d ▷ 256-bit state
Ensure: a, b, c, d ▷ updated state
procedure $G_B(a, b, c, d)$
1: $a \leftarrow a + b + 2 \cdot lsw(a) \cdot lsw(b)$
2: $d \leftarrow (d \oplus a) \ggg 32$
3: $c \leftarrow c + d + 2 \cdot lsw(c) \cdot lsw(d)$
4: $b \leftarrow (b \oplus c) \ggg 24$
5: $a \leftarrow a + b + 2 \cdot lsw(a) \cdot lsw(b)$
6: $d \leftarrow (d \oplus a) \ggg 16$
7: $c \leftarrow c + d + 2 \cdot lsw(c) \cdot lsw(d)$
8: $b \leftarrow (b \oplus c) \ggg 63$

and then processed as shown in Algorithm 16.

The function CF takes two 8192-bit values X, Y as input and outputs a 8192-bit value Z. The value $R = X \oplus Y$ (see Line 1 of Algorithm 15) is represented as an 8×8 matrix consisting of 64 128-bit values $R_0, \ldots, R_{63}$ ordered from left to right and top to down. First, all rows are updated using the permutation P generating an intermediate matrix Q (Lines 2 and 3). Then, all columns of Q are updated, again using P (Lines 4 and 5). The output of CF is then given by the XOR operation of the input matrix R and the intermediate matrix Q. Based on the large state of CF, it is possible to fill memory fast, e.g., about 1 GB memory is filled in less than one second [52]. For a formal definition of P, we refer to the specification of Argon2 [52].

Even if the function CF requires two calls to the underlying round function of BLAKE2b to process 1024 bits, it is still significantly faster than BLAKE2b-1 (see Table 9.8 in Section 9.3.4). The speed-up stems from the fact that CF processes 16384 bits per invocation, whereas BLAKE2b-1 processes only 1024 bits. Thus, the usage of BLAKE2b-1 requires significantly more loading operations in comparison to the usage of CF when processing inputs of equal size.

Remark 9.22. *Note that we evaluate another instance of H' in Section 9.3.4, called $P_{compress}$. This is a modification of the permutation P, where we split the output of P into two 512-bit halves and combine them using the XOR operation. This is a simple way to build a compression function out of the permutation P. We introduce $P_{compress}$ for comparison with BLAKE2b-1,*

H'	Garlic (g)	Time (s)	Throughput (GB/s)
BLAKE2b	21	1.54	0.08
BLAKE2b-1 (G_O)	21	0.37	0.34
CF (G_B)	17	0.26	0.49
CF (G_L)	17	0.18	0.69
$P_{compress}$ (G_B)	21	0.43	0.30
$P_{compress}$ (G_L)	21	0.33	0.39
SHA-512	21	4.04	0.03

Table 9.8: Measurements of Catena-Dragonfly for different hash functions H', where we fix the memory usage to 128 MB, $\lambda = 2$, and both options Γ and Φ are disabled.

which includes the regular message schedule, the initialization, and finalization of the round function of BLAKE2b (as shown in Algorithm 12), whereas $P_{compress}$ does not. Nevertheless, we do not consider it as a recommended instance of H' since it will never achieve the same throughput as the function CF due to the smaller size of the state words.

9.3.3 BlaMka

The function G_B was introduced by the authors of Lyra2 [244]. It described a slightly adjusted variant of G_L (see Algorithm 16 for a comparison) and is used in Lyra2 as the underlying permutation of a sponge-based structure. Due to the missing cryptanalysis of BlaMka, the authors do not recommend its usage. Nevertheless, since it consists of a neat combination of ARX (Addition, Rotation, XOR) and integer multiplication, we also consider it as an alternative to G_L.

The only difference between the functions G_L and G_B is that each addition $a + b$ is replaced by $a + b + 2 \cdot lsw(a) \cdot lsw(b)$, where $lsw(x)$ denotes the least significant 32-bit word of x. The idea behind the so-called *multiplication hardening* is to provide a similar performance for hardware- and software-based adversaries [243, 246].

9.3.4 Comparison

We provide performance values based on the instance of the underlying hash function H'. All measurements shown in Table 9.8 are done using the tool Catena-Variants written by Schmidt [113] and its extended version by Schilling [138]. Due to the fact that the compression function of Argon2 (CF) supports state words of $k > n$ bits, we fix the memory usage to 128 MB and let the garlic g to be variable. Moreover, we fixed $\lambda = 2$ and disabled both options Γ and Φ. The measurements reflect the median of 101 runs using an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz with Turbo Boost and Hyper-Threading enabled. For SHA-512, we used the implementation from the OpenSSL version 1.0.2d (9th July 2015).

9.4 Summary

We have shown and discussed several instantiations of the underlying hash function H' as well as of the graph-based structure F_λ. Furthermore, we considered instances of the two options Γ and Φ. Based on the presented instances and their variety of properties, we are now able to combine them with the goal of constructing suitable instances of Catena.

Catena – Instances of the Framework

A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.

Alan M. Turing

Catena is a highly flexible Password-Scrambling Framework (PSF). More detailed, an instance of Catena is defined by the following parameters, which can be freely adapted to form a variant of Catena suitable for a particular application:

- a memory-cost parameter g (minimum: g_{low}, maximum: g_{high}),
- time-cost parameter λ (depth of the underlying graph used as instance of F_λ),
- a cryptographic hash function H,
- a hash function H',
- a memory-hard function F_λ,
- a password-independent option Γ (and its public input γ), and
- a password-dependent option Φ (and its secret input μ).

This enables Catena to maximize the defense against specific adversaries, their capabilities and goals, and to cope with a high variation of hardware and constraints on the side of the defender. Before we present our recommendations, we briefly discuss the common parameter choices, the naming when using an instance as a KDF, and certain security-related facts.

Common Parameter Choices. For all instances, we set $g_{\text{low}} = g_{\text{high}}$, $\gamma = s$, and μ to the output of F_λ. The particular choices of g and λ are done while focusing on COTS systems. The recommended parameters for keyed password hashing are exactly the same as they are for password hashing, plus an additional 128-bit key. The unique identifier V (used as input of the tweak computation, see Section 7.2) is always given by the string representation of the suffix of a particular

instance, e.g., we set V ="Dragonfly-Full" for Catena-Dragonfly-Full and V ="Butterfly" for Catena-Butterfly. For all but the default instances of Catena, the indexing function R used within Φ simply returns the g LSBs of its input, whereas for the default instances, R is given by `xorshift1024star`.[1] Finally, we set the maximal running time for password hashing to 0.5 s and for the usage as a KDF to 5 s.

Key-Derivation Function. An instance of Catena can be used as a KDF if (1) the cost parameters are adapted accordingly and (2) if $H = H'$ holds. An instance that uses $H = H'$ contains the suffix '-FULL' in its name.

Security. All variants presented and recommended in this section provide preimage resistance as well as resistance to Weak Garbage-Collector Attacks (WGCAs). Resistance to Garbage-Collector Attacks (GCAs) is only a mandatory requirement for instances of Catena for password hashing since an adversary launching a GCA on an instance of Catena must already have access to the RAM used by this particular instance. Thus, reading the internal state of an instance of Catena (required for launching a GCA) is usually as easy as obtaining the key generated by this instance. Nevertheless, if a variant uses the option Φ, it directly becomes resistant to GCAs since the internal state is overwritten at least twice: first, during the call to F_λ and second, during the call to Φ. To obtain memory hardness of an instance of Catena, it is sufficient to choose a memory-hard instance of F_λ. If one would like an instance of Catena to thwart parallel-working adversaries by providing Sequential Memory Hardness (SMH), we recommend to use the option Φ. If an instance of Catena should be used as a KDF, it must provide PRF security. Fortunately, as long as H is a cryptographic hash function and it holds that $H = H'$, PRF security is already provided by the framework itself, as shown in Theorems 8.2 and 8.3. The resistance to adversaries able to utilize ASICs or FPGAs mainly depends on whether Γ or Φ is enabled or not. Finally, the decision for a particular instance should always considering the most probably occurring type of adversary in mind.

Remark 10.1. *All instances of* Catena *set $\gamma = s$, which may leak as a result of a Cache-Timing Attack (CTA). Usually, the salt is not secret and this is not an issue. But, in some special cases, an application may require to keep the salt secret. In that case, we suggest to choose any fixed random value as γ. Consider, e.g., an encrypted file system with different partitions. Naturally, the salt used to generate the secret key is different for each partition. If the security requirement is to hide which partition has been mounted, the salt must not be used for γ.*

In the following, we first provide the default instances of Catena as they were submitted to the Password Hashing Competition (PHC), and second, introduce four more instances suitable for different applications.

10.1 Default Instances (Catena-Dragonfly, Catena-Butterfly)

For Catena-Dragonfly and Catena-Butterfly, we instantiate the function F_λ with BRH_λ^g (see Section 9.1.1) and DBH_λ^g (see Section 9.1.4), respectively. Therefore, all variants of Catena-

[1] The usage of `xorshift1024star` is due to historical reasons since the choice was made at a time when we were not aware of the second approach.

Dragonfly provide only memory hardness (see Definition 3.5), whereas variants of Catena-Butterfly satisfy λ-memory hardness (see Definition 3.6), i.e., the latter provide higher resistance to TMTO attacks. Since the number of vertices is given by $2 \cdot 2^g$ for a BRG_λ^g and by $2g \cdot 2^g$ for a DBG_λ^g, a variant of Catena-Dragonfly runs significantly faster than a variant of Catena-Butterfly, allowing to allocate more memory, i.e., a larger number of state words. Therefore, Catena-Dragonfly should be used in settings, where the defender can afford to allocate much memory without any problems and as a technique to maximize the cost of password cracking even for high-budget adversaries. On the other hand, Catena-Butterfly should be used in settings, where the defender has limited memory available, or as a defense against typical “memory-constrained” adversaries that cannot afford 2^g units of memory for many parallel cores, and thus, would suffer from the resistance to TMTO attacks.

In total, six default instances were submitted to the PHC (see Table 10.1). The versions Catena-Dragonfly-Full and Catena-Butterfly-Full instantiated with the operations BRH_2^{18} and DBH_4^{14}, respectively, show the impact of the choice $H' = H =$ BLAKE2b in comparison to $H =$ BLAKE2b and $H' =$ BLAKE2b-1. We can observe that, while adhering to the maximal running time for password hashing, i.e., 0.5 s, the choice of $H' =$ BLAKE2b-1 allows to choose four to eight times the memory compared to the full variants. For completeness, we also provide two instances for the usage as a KDF, i.e., Catena-Dragonfly-Full and Catena-Butterfly-Full which use the operations BRH_2^{22} and DBH_4^{17}, respectively. None of the default instances enable the option Φ since this would render them vulnerable to CTAs, where avoiding those attacks was one of the main goals when submitting Catena to the PHC. Resistance to the precomputation attacks by Biryukov and Khovratovich [55] is only provided for instances using the hashing operation DBH_λ^g, whereas the resistance to GCAs is provided by all default instances.

10.2 Catena-Stonefly – An ASIC-Resistant Instance of Catena

For password hashing, strong resistance to ASIC-based adversaries can be achieved (among other things) by a memory-access pattern that depends on the password (providing SMH), a public input, and multiplication hardening. The password-dependent memory-access pattern is realized by the option Φ. We highly recommend to place the Φ at the end of $flap$ since then, the advantage of an adversary launching a CTA against Catena is negligible. We further recommend to use a function conducting a public-input-dependent memory-access at the start of $flap$. Then, an ASIC-based adversary would have to copy the whole state to and from the hardware if it aims to compute the underlying memory-hard function F_λ on the ASIC itself (which would be intuitive since it follows a password-independent memory-access pattern and thus, can be computed efficiently there). For that purpose, we recommend the option Γ, where the public input is given by the salt as mentioned above. Finally, multiplication hardening is provided by using the function G_B (see Section 9.3.3) as the underlying permutation of H'. For performance reasons, we instantiate F_λ with a BRG_2^{18} since we do not aim to provide full resistance to TMTO attacks for this instance. Catena-Stonefly and Catena-Stonefly-Full can be seen as the Catena complements to the data-dependent variant of the PHC winner Argon2, i.e., Argon2d [52].

10.3 Catena-Horsefly – A High-Throughput Instance of Catena

Even if we aim for maximal performance and memory usage, each instance of Catena should still provide a certain level of security. For H', we opted for the compression function CF of Argon2 using G_L (see Section 9.3.2). The underlying graph-based function F_λ is instantiated by BRH_1^{19},

which is faster than $\mathrm{GRH3}_1^g$ and DBH_1^g. Be aware that $\lambda = 1$ provides high performance but, on the other hand, allows for GCAs. Therefore, we refer to $\lambda = 2$ if GCAs are likely to happen. The instance presented here would be also suitable for the application in a proof-of-space scenario based on a challenge-response protocol.

10.4 Catena-Mydasfly – A TMTO-Resistant Instance of Catena

For a tradeoff-resistant instance of CATENA, we decided to use a DBG_2^{14} for F_λ. This decision is based on the fact that we want to provide strong resistance to TMTO attacks without enabling an adversary to launch a CTA on the "first part" of *flap*. To further increase the resistance to TMTO attacks, we use the option Φ, providing SMH. Since the focus of this instance does not lie in ASIC resistance, we favor CF with G_L over G_B as the underlying hash function H'. CATENA-MYDASFLY and CATENA-MYDASFLY-FULL can be seen as the CATENA complements to the data-independent variant of the PHC winner Argon2, i.e., Argon2i [52].

10.5 Catena-Lanternfly – A Hybrid Instance of Catena

Here, we wanted to aim for the best performance while retain suitable security against ASIC-based adversaries, tradeoff attacks, as well as resistance to CTAs. Therefore, in comparison to CATENA-STONEFLY, we disregard the option Φ, but keep the invocation of Γ to provide sufficient resistance to ASICs. We instantiate F_λ by a $\mathrm{GRG3}_2^{17}$, which maintains reasonable resistance to precomputation attacks while still providing acceptable performance (in comparison to a DBG_λ^g). The hash function H' is instantiated with CF of Argon2 using G_B internally, which leads to a good throughput in terms of memory while providing multiplication-hardening. CATENA-LANTERNFLY and CATENA-LANTERNFLY-FULL can be seen as the CATENA complements to the hybrid variant of the PHC winner Argon2, i.e., Argon2id [52].

In Table 10.1, we list and compare all variants of CATENA which we recommend.

Implementation. The latest reference implementation of CATENA can be found on

`https://github.com/medsec/catena`.

Furthermore, we provide CATENA-VARIANTS, which allows to construct arbitrary instances of CATENA by choosing the parameters g, λ, H, H', F_λ, Γ, and Φ. CATENA-VARIANTS can be found on:

`https://github.com/medsec/catena-variants`.

It supports also the possibility to change the position of Γ within *flap* (see Section 9.2.1 for a discussion about the impact of changing the order).

Finally, a search tool for optimal parameters under given constraints is given by CATENA-AXUNGIA, which can be found on:

`https://github.com/medsec/catena-axungia`.

Name	F_λ	H'	*Memory*	*Time (s)*	*MH*	Γ	Φ	*CTA-Res.*	*PCA-Res.*	*GCA-Res.*	*KDF*
Default Instances											
Catena-Dragonfly	BRH_2^{21}	BLAKE2b-1	128 MB	0.38	•	•	-	•	-	•	-
Catena-Dragonfly-Full	BRH_2^{18}	BLAKE2b	16 MB	0.18	•	•	-	•	-	•	-
Catena-Dragonfly-Full	BRH_2^{22}	BLAKE2b	256 MB	3.15	•	•	-	•	-	•	•
Catena-Butterfly	DBH_4^{16}	BLAKE2b-1	4 MB	0.33	LMH	•	-	•	•	•	-
Catena-Butterfly-Full	DBH_4^{14}	BLAKE2b	1 MB	0.41	LMH	•	-	•	•	•	-
Catena-Butterfly-Full	DBH_4^{17}	BLAKE2b	8 MB	3.97	LMH	•	-	•	•	•	•
ASIC-Resistant											
Catena-Stonefly	BRH_2^{18}	CF (G_B)	256 MB	0.69*	SMH	•	•	-	-	•	-
Catena-Stonefly-Full	BRH_2^{22}	BLAKE2b	256 MB	4.41	SMH	•	•	-	-	•	•
High Throughput											
Catena-Horsefly	BRH_1^{19}	CF (G_L)	512 MB	0.45	•	-	-	•	-	-	-
Catena-Horsefly-Full	BRH_1^{23}	BLAKE2b	512 MB	3.64	•	-	-	•	-	-	•
TMTO-Resistant											
Catena-Mydasfly	DBH_2^{14}	CF (G_L)	128 MB	0.40	SMH, LMH	-	•	-	•	•	-
Catena-Mydasfly-Full	DBH_2^{18}	BLAKE2b	256 MB	4.40	SMH, LMH	-	•	-	•	•	•
Hybrid											
Catena-Lanternfly	$\mathrm{GRH3}_2^{17}$	CF (G_B)	128 MB	0.35	•	•	-	•	•	•	-
Catena-Lanternfly-Full	$\mathrm{GRH3}_2^{22}$	BLAKE2b	256 MB	3.66	•	•	-	•	•	•	•

Table 10.1: Comparison of instances of Catena regarding their instances of the underlying primitives, memory usage, time, memory hardness (SQ = sequential), options, resistance to Cache-Timing Attacks (CTAs), Precomputation Attacks (PCAs), and Garbage-Collector Attacks (GCAs), and their applicability as a Key-Derivation Function (KDF). By '•', we denote that a property is fulfilled and by '-' that it is not fulfilled. All instances use H = BLAKE2b. The function Γ for the here-mentioned instances is always called right before F_λ due to the given layer structure. The measurements reflect the median of 101 runs in each case, using an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz with Turbo Boost and Hyper-Threading enabled. The recommendations were made in 2015 [174], using an older version of `clang`. Therefore, some of the currently computed values (marked by '*') exceed the maximum time for password hashing. SMH – sequential memory hardness, LMH – λ-memory hardness.

10.6 Summary

In this chapter, we provided a comprehensive Catena portfolio including six instances for a wide variety of applications. For each of these instances, we further provide versions suitable for the usage as a Key-Derivation Function (KDF). Note that Catena has obtained special recognition of the Password Hashing Competition (PHC) based on the two instances Catena-Dragonfly and Catena-Butterfly.[2]

[2]The other instances of Catena were published after the PHC.

Chapter 11

Comparison of Modern Password Scramblers

When it is obvious that the goals cannot be reached, don't adjust the goals, adjust the action steps.

Confucius

Overview of Modern Password Scramblers
and their Resistance to Garbage-Collector Attacks

After introducing the Password-Scrambling Framework (PSF) Catena and certain instances, we obviously wanted to see how those instances perform in comparison to other modern password scramblers, i.e., the (non-withdrawn) submissions to the PHC and `scrypt`. In this chapter, the considered algorithms are analyzed with regards to their functional properties (e.g., Client-Independent Update (CIU) and Server Relief (SR)), their security properties (e.g., memory hardness and resistance to Cache-Timing Attacks (CTAs)), their general properties (e.g., amount of memory required), and their flexibility (i.e., is the PS designed to replace the underlying algorithms?) Next, we provide a brief discussion for each candidate of the PHC regarding to their resistance to Garbage-Collector Attacks (GCAs) and Weak Garbage-Collector Attacks (WGCAs). We close this chapter with a discussion on the Parallel Random-Oracle Model (pROM) introduced by Alwen and Serbinenko [23]. Before we go on with the comparison, we briefly introduce two types of memory-hard password scramblers which are distinguished by their corresponding memory-handling.

Type-A: Allocating a huge amount of memory which is rarely overwritten.

Type-B: Allocating a reasonable amount of memory which is overwritten multiple times.

Algorithms following Type-A usually aim at providing decent resistance to dedicated password-cracking hardware, i.e., ASICs and FPGAs. However, the fact that the memory locations are overwritten rarely can render those algorithms vulnerable to GCAs. On the other hand, algorithms following Type-B have the goal to thwart GPU-based attacks by forcing a high amount of cache

misses during the computation of the password hash. Naturally, those algorithms provide some built-in protection against GCAs. In general, while implementing cryptographic algorithms, it is advised to remove any critical data from memory as soon as possible (latest after the termination of the algorithm)[1].

> **Remark 11.1.** *For our theoretical consideration of the proposed attacks, we assume a natural implementation of the algorithms, e.g., we assume that overwriting the internal state of an algorithm* ***after*** *its invocation is neglected due to optimization,*

In Tables 11.1 through 11.4, we provide an overview of modern password scramblers in comparison to the proposed instances of Catena. Note that we disregard Catena-Dragonfly-Full and Catena-Butterfly-Full using a BRH_2^{18} and a DBH_4^{14}, respectively (see Table 10.1 in Chapter 10) since they were only added for showing the impact of setting $H = H'$.

Tables 11.1 and 11.2 focus on general properties, i.e., the basis of the algorithm, the range of required memory (RAM and ROM) which can be utilized if the algorithm can be parallelized, and which underlying cryptographic primitive(s) they utilize. For that comparison, we use our recommendations of Catena for password hashing. Our full versions (recommendations as KDFs) only differ in setting $H' = H$ instead of $H \neq H'$ and the particular instance of F_λ.

Tables 11.3 and 11.4 compare the functional and security properties of the password scramblers. Since we see the possibility of being used as a KDF an important feature of a modern PS, we use the full versions of the proposed instances of Catena for comparison.

> **Remark 11.2.** *Note that we do not claim completeness for Tables 11.3 and 11.4. For example, we defined a scheme not to be resistant to Side-Channel Attacks (SCAs) if it maintains a password-dependent memory-access pattern. Nevertheless, there exist several other types of SCAs such as those based on power consumption or acoustic analysis.*

A further comparison of modern password scramblers (including `scrypt`, `bcrypt`, PBKDF2, and the PHC finalists) is presented by Hatzivasilis et al. [130]. Next, we briefly discuss the results stated in Tables 11.3 and 11.4 regarding the resistance of a scheme to GCAs and WGCAs.

[1] `https://cryptocoding.net/index.php/Coding_rules`

Algorithm	Based On	Memory Usage		Parallel	Primitive	
		RAM	ROM		BC/SC/PERM	HF
Argon	AES	1 kB - 1 GB	-	-	AES (5R)	BLAKE2b
Argon2d		250 MB - 4 GB	-	•	BLAKE2b (CF, 2R)	BLAKE2b
Argon2i		1 GB - 6 GB	-	•	BLAKE2b (CF, 2R)	BLAKE2b
battcrypt	Blowfish/`bcrypt`	128 kB - 128 MB	-	○	Blowfish-CBC	SHA-512
Lyra2	Sponge	400 MB - 1 GB	-	•	BLAKE2b (CF)/(BlaMka)	-
Makwa	Squarings	negl.	-	•	-	SHA-256
Parallel		negl.	-	•	-	SHA-512
POMELO		(8 KB, 256 GB)	-	•	-	-
Pufferfish	mod. Blowfish/`bcrypt`	4 kB - 16 kB	-	-	mod. Blowfish	SHA-512
yescrypt	`scrypt`	3 MB	3 GB	○	Salsa20/8	SHA-256
Catena-Dragonfly	BRH_2^{21}	128 MB	-	○	-	BLAKE2b/BLAKE2b-1
Catena-Butterfly	DBH_4^{16}	4 MB	-	○	-	BLAKE2b/BLAKE2b-1
Catena-Stonefly	BRH_1^{18}	256 MB	-	○	-	BLAKE2b/CF(G_B)
Catena-Horsefly	BRH_1^{19}	512 MB	-	○	-	BLAKE2b/CF(G_L)
Catena-Mydasfly	DBH_2^{14}	128 MB	-	○	-	BLAKE2b/CF(G_L)
Catena-Lanternfly	$GRH3_2^{17}$	128 MB	-	○	-	BLAKE2b/CF(G_B)

Table 11.1: Overview of the finalists of the PHC and all noteworthy instances of Catena regarding to their general properties. The values in the column "Memory Usage" are taken from the authors recommendation for password hashing. The entry 'A(CF)' denotes that only the compression function of algorithm A is used. An entry A(xR) denotes that an algorithm A is reduced to x rounds. If an algorithm can be only be partially computed in parallel, we marked the corresponding entry with '○'. All algorithms shown here are iteration-based. BC – block cipher, SC – stream cipher, PERM – keyless permutation, HF – hash function.

		Memory Usage			Primitive	
Algorithm	**Based On**	**RAM**	**ROM**	**Parallel**	**BC/SC/PERM**	**HF**
AntCrypt		32 kB	-	○	-	SHA-512
CENTRIFUGE		2 MB	-	-	AES-256	SHA-512
EARWORM		-	2 GB	●	AES (1R)	SHA-256
Gambit	Sponge	50 MB	-	-	Keccak_f	-
Lanarea DF		256 B	-	-	-	BLAKE2b
MCS_PHS		negl.	-	-	-	MCSSHA-8
ocrypt	`scrypt`	1 MB - 1 GB	-	-	ChaCha	CubeHash
PolyPassHash	Shamir Sec. Sharing	negl.	-	-	AES	SHA-256
Rig	BRG	15 MB	-	○	-	BLAKE2b
schvrch		8 MB	-	○	-	-
`scrypt`		1 GB	-	-	Salsa20/8	-
`Tortuga`	Sponge & recursive Feistel	○	-	-	Turtle	-
SkinnyCat	BRG	○	-	-	-	SHA-*/BLAKE2*
TwoCats	BRG	○	-	●	-	SHA-*/BLAKE2*
Yarn		○	-	○	BLAKE2b (CF), AES	-

Table 11.2: Overview of the first- and second-round PHC candidates and `scrypt` regarding to their general properties. The values in the column "Memory Usage" are taken from the authors recommendation for password hashing or are marked as '○' if no recommendation exists. The entry 'A(CF)' denotes that only the compression function of algorithm A is used. An entry A(xR) denotes that an algorithm A is reduced to x rounds. If an algorithm can only be partially computed in parallel, we marked the corresponding entry with '○'. Except for PolyPassHash, all other algorithms are iteration-based. BC – block cipher, SC – stream cipher, PERM – keyless permutation, HF – hash function.

11.1 Resistance of PHC Candidates against (W)GC Attacks

Note that for the upcoming discussion, we assume, the reader is familiar with the internals of the particular PHC candidates since we concentrate only on those parts of the candidates that are relevant regarding GCAs and WGCAs.

AntCrypt [83]. The internal state of AntCrypt is initialized with the secret *pwd*. During the hashing process, the state is overwritten multiple times (based on the parameter `outer_rounds` and `inner_rounds`), which thwarts GCA. Moreover, since the password *pwd* is used only to initialize the internal state, WGCAs are not applicable.

Argon/Argon2d/Argon2i [54]. First, the internal state, derived from the password *pwd*, is the input to the padding phase. After the padding phase, the internal state is overwritten by applying the functions `ShuffleSlices` and `SubGroups` at least R times. Based on this structure, and since *pwd* is used only to initialize the state, Argon is not vulnerable to GCAs/WGCAs. Within Argon2d and Argon2i, after hashing the password and salt among other inputs, the internal state is overwritten t times using the compression function G. Thus, Argon2d and Argon2i provide a similar resistance to GCAs/WGCAs as Argon.

battcrypt [253]. Within battcrypt, the plain password is used only once, namely, to generate a value $key \leftarrow$ SHA-512(SHA-512($salt \mid\mid pwd$)). The value *key* is then used to initialize the internal state, which is expanded afterwards. In the *Work* phase, the internal state is overwritten `t_cost`×`m_size` times using password-dependent indices. Thus, GCAs are not applicable. Note that the value *key* is used in the three phases *Initialize blowfish*, *Initialize data*, and *Finish*, whereas it is overwritten in the phase *Finish* the first time. Further, the main effort for battcrypt is given by the *Work* phase. Thus, one can assume that one iteration of the outer loop (iterating over *t_cost_upgrade*) lasts enough time so that a WGCA adversary can launch the following attack: For each password candidate x and the public salt value s, compute $key' \leftarrow \text{SHA512}(\text{SHA512}(s \mid\mid x))$ and check whether $key' = key$. If so, mark x as a valid password candidate.

Catena [109]. The resistance of the instances of Catena was already discussed before in the respecting parts of this thesis. Summarizing, due to the fact that the password *pwd* is only used once and should be overwritten right after its usage by the call to $flap$ (see Line 2 of Algorithm 3), all instances provide resistance to WGCAs. Furthermore, since we assume an instance of Catena to provide resistance to GCAs if the internal (full) state (consisting of $2^g \cdot k$ bits[2]) is at least overwritten twice, only Catena-Horsefly and Catena-Horsefly-Full are vulnerable since they instantiate F_λ by BRH_1^{19} and a BRH_1^{23}, respectively, and do not make use of the option Φ. For all other instances, the state is overwritten at least λ (BRG_λ^g, SBRG_λ^g, $\text{GRG}\ell_\lambda^g$) or $\lambda(2g-1)$ times (DBG_λ^g). Thus, the state is overwritten at least twice.

CENTRIFUGE [18]. The internal state M of size `p_mem`×`outlen` byte is initialized with a seed S derived from the password and the salt as follows: $S \leftarrow H(s_L \mid\mid s_R)$, where $s_L \leftarrow H(pwd \mid\mid len(pwd))$ and $s_R \leftarrow H(\text{salt} \mid\mid len(\text{salt}))$. Furthermore, S is used as the initialization vector (IV) and the key

[2] For GCAs, we neglect the call to $flap$ using only $2^{\lceil g_{\text{low}}/2 \rceil}$ state words (see Line 2 of Algorithm 3) since that would not initialize the full state size which could be leaked to an adversary, i.e., the (full) state, consisting of $2^g \cdot k$ state words, is written first in Line 5 of Algorithm 3. From that point on, we count the times the state is overwritten.

	Features					**Security**						
Algorithm	*CIU*	*SR*	*FPO*	*Flexible*	*Type*	*MH*	*KDF*	*GCA Res.*	*WGCA Res.*	*SCA Res.*	*Sec. Analysis*	*Shortcut*
Argon	•	•	-	•	B	•	•	•	•	-	•	-
Argon2d	•	•	-	•	B	•	•	•	•	-	•	-
Argon2i	•	•	-	•	B	•	•	•	•	•	•	-
battcrypt	•	•	-	○	B	•	•	•	-	-	○	-
Lyra2	•	•	-	○	B	•	•	•	•	-	•	-
Makwa	○	•	-	○	-	-	•	•	•	○	•	•
Parallel	•	•	-	•	-	-	•	•	-	•	○	-
POMELO	-	-	-	-	B	•	•	•	•	○	○	-
Pufferfish	-	•	-	○	B	○	•	•	•	-	○	-
yescrypt	•	•	-	•	A	RPH, SMH	•	○	○	-	○	-
Catena-Dragonfly-Full	•	•	-	•	B	•	•	•	•	•	•	-
Catena-Butterfly-Full	•	•	-	•	B	LMH	•	•	•	•	•	-
Catena-Stonefly-Full	•	•	-	•	B	SMH	•	•	•	-	•	-
Catena-Horsefly-Full	•	•	-	•	A	•	•	-	•	•	•	-
Catena-Mydasfly-Full	•	•	-	•	B	SMH, LMH	•	•	•	•	•	-
Catena-Lanternfly-Full	•	•	-	•	B	•	•	•	•	-	•	-

Table 11.3: Overview of the finalists of the PHC and all noteworthy instances of Catena regarding their security properties. By '•', we denote that a property is fulfilled, by '○' that it is partially fulfilled or only under certain requirements, and by '-' that it is not fulfilled. Note that we say that an algorithm does not support SR when it requires the whole state to be transmitted to the server. Moreover, we say that an algorithm does not support CIU if any additional information to the password hash itself is required. SMH – sequential memory hardness, LMH – λ-memory hardness, RPH – ROM-port hardness as defined in [80].

for the CFB encryption. The internal state M is written once and later accessed only in a password-dependent manner. Thus, an adversary can launch the GCA in Algorithm 17.
The final step of CENTRIFUGE is to encrypt the internal state, requiring the key and the IV, which therefore must remain in memory during the invocation of CENTRIFUGE. Thus, an adversary can launch the WGCA in Algorithm 18.

EARWORM [112]. EARWORM maintains an array called *arena* which consists of $2^{m_cost} \times L \times W$ 128-bit blocks, where $W \leftarrow 4$ and $L \leftarrow 64$ are recommended by the authors. This read-only array is randomly initialized (using an additional secret input which has to be constant within a given system) and is used blockwise as AES round keys. Since the values within this array do not depend on the secret *pwd*, knowledge about *arena* does not help any malicious garbage collector. Within the main function of EARWORM (WORKUNIT), an internal state *scratchpad* is updated multiple times using password-dependent accesses to *arena*. Thus, a GCA adversary cannot profit

	Features					Security						
Algorithm	*CIU*	*SR*	*FPO*	*Flexible*	*Type*	*MH*	*KDF*	*GCA Res.*	*WGCA Res.*	*SCA Res.*	*Sec. Analysis*	*Shortcut*
AntCrypt	•	-	•	○	B	•	•	•	•	•	○	-
CENTRIFUGE	-	-	-	•	A	○	-	-	-	•	○	-
EARWORM	-	•	-	-	B	RPH	-	•	-	•	•	-
Gambit	-	•	opt.	○	B	○	•	•	•	•	○	-
Lanarea DF	-	•	-	•	B	○	•	•	•	○	○	-
MCS_PHS	-	•	-	○	-	-	•	•	•	•	-	-
ocrypt	-	-	-	•	B	○	•	•	•	-	○	-
PolyPassHash	•	-	-	•	-	-	-	-	-	-	•	•
Rig	•	•	-	•	B	LMH	•	•	•	•	•	-
schvrch	-	-	-	-	B	-	-	•	•	•	○	-
`scrypt`	-	-	-	•	A	SMH	•	-	-	-	•	-
`Tortuga`	-	-	-	-	B	○	•	•	•	•	○	-
SkinnyCat	-	•	-	•	A	SMH	•	-	-	○	•	-
TwoCats	•	•	-	•	B	SMH	•	•	•	○	•	-
Yarn	-	•	-	-	B	○	-	•	-	-	○	-

Table 11.4: Overview of first-round and second-round PHC candidates and `scrypt` regarding their security properties. By '•', we denote that a property is fulfilled, by '○' that it is partially fulfilled or only under certain requirements, and by '-' that it is not fulfilled. Note that we say that an algorithm does not support SR when it requires the whole state to be transmitted to the server. Moreover, we say that an algorithm does not support CIU if any additional information to the password hash itself is required. SMH – sequential memory hardness, LMH – λ-memory hardness, RPH – ROM-port hardness as defined in [80].

from knowledge about *scratchpad*, rendering GCAs not applicable.

Within the function WORKUNIT, the value *scratchpad_tmpbuf* is derived directly from the password as follows:

$$scratchpad_tmpbuf \leftarrow \text{EWPRF}(pwd, 01 \mid\mid salt, 16W),$$

where EWPRF denotes $\text{PBKDF2}_{\text{HMAC-SHA256}}$ with the first input denoting the secret key. This value is updated only at the end of WORKUNIT using the internal state. Thus, it has to be in memory during almost the whole invocation of EARWORM, rendering the following WGC attack possible: For each password candidate x and the known value $salt$, compute $y \leftarrow \text{EWPRF}(x, 01 \mid\mid salt, 16W)$ and check whether $scratchpad_tmpbuf = y$. If so, mark x as a valid password candidate.

Gambit [214]. Gambit bases on a duplex-sponge construction [49] maintaining two internal states S and Mem, where S is used to subsequently update Mem. First, password and salt are absorbed into the sponge. After one call to the underlying permutation, the value obtained from the internal

Algorithm 17 Garbage-Collector Attack on CENTRIFUGE

1: receive the internal state M (or at least $M[1]$) from memory
2: **for** each password candidate x **do**
3: initialization (seeding and S-box)
4: compute the first table entry $M'[1]$ (during the *build table* step)
5: **if** $M'[1] = M[1]$ **then**
6: mark x as a valid password candidate
7: **else**
8: go to Line 2

Algorithm 18 Weak Garbage-Collector Attack on CENTRIFUGE

1: $s_R \leftarrow H(salt \mid\mid len(salt))$
2: **for** every password candidate x **do**
3: $s'_L \leftarrow H(x \mid\mid len(x))$
4: $S' \leftarrow H(s'_L \mid\mid s_R)$
5: **if** $S' = IV$ **then**
6: mark x as a valid password candidate
7: **else**
8: go to Line 2

state is written to the internal state Mem and Mem is updated r times (number of words in the ratio of S). The output after the r steps is optionally XORed with an array lying in ROM. Subsequently, Mem is absorbed into S again. This step is executed t times, where t denotes the time-cost parameter. The size of Mem is given by m, the memory-cost parameter. Continuously updating the states Mem and S thwarts GCAs. Moreover, since *pwd* is used only to initialize the state within the sponge construction, WGCAs are not applicable.

Lanarea DF [192]. Lanarea DF maintains a matrix (internal state) consisting of $16 \times 16 \times m_cost$ byte values, where m_cost denotes the memory-cost parameter. After the password-independent setup phase, the password is processed by the internal pseudorandom function producing the array $(h_0, \ldots, h_{31})$, which determines the positions on which the internal state is accessed during the core phase (thus, allowing CTAs). In the core phase, the internal state is overwritten $t_cost \times m_cost \times 16$ times, rendering GCAs impossible. Moreover, the array $(h_0, \ldots, h_{31})$ is overwritten $t_cost \times m_cost$ times which thwarts WGCAs.

Lyra2 [146]. The Lyra2 password scrambler (and KDF) is based on a duplex sponge construction that maintains a state H, which is initialized with the password, the salt, and some tweak in the first step of its algorithm. The authors indicate that the password can be overwritten from this point on, rendering WGCAs impossible. Moreover, Lyra2 maintains an internal state M, which is overwritten (updated using values from the sponge state H) multiple times. Thus, GCAs are not applicable for Lyra2.

Makwa [215]. Makwa has not been designed to be a memory-demanding password scrambler. Its strength is based on a high number of squarings modulo a composite (Blum) integer n. The plain (or hashed) password is used twice to initialize the internal state, which is then processed by squarings modulo n. Thus, neither GCAs nor WGCAs are applicable for Makwa.

MCS_PHS [179]. Depending on the size of the output, MCS_PHS applies iterated hashing operations, reducing the output size of the hash function by one byte in each iteration – starting from 64 bytes. Note that the memory-cost parameter m_cost is used only to increase the size of the initial chaining value T_0. The secret input *pwd* is used once, namely when computing the value T_0 and can be deleted afterwards, rendering WGCAs not applicable. Furthermore, since the output of MCS_PHS is computed by iteratively applying the underlying hash function (without handling an internal state which has to be placed in memory), GCAs are not possible.

ocrypt [91]. The basic idea of ocrypt is similar to that of `scrypt`, besides the fact that the random memory accesses are determined by the output of a stream cipher (ChaCha) instead of a hash-function cascade. The output of the stream cipher determines which element of the internal state is updated, which consists of $2^{17+m_{cost}}$ 64-bit words. During the invocation of ocrypt, the password is used only twice: (1) as input to CubeHash, generating the key for the stream cipher and (2) to initialize the internal state. Neither the password nor the output of CubeHash are used again after the initialization. Thus, ocrypt is not vulnerable to WGCAs.

The internal state is processed $2^{17+t_{cost}}$ times, where in each step, one state word is updated. Since the indices of the array elements accessed depend only on the password, GCAs are not possible by observing the internal state after the invocation of ocrypt.

Remark 11.3. *Note that the authors of ocrypt claim resistance to SCAs since the indices of the array elements are chosen in a password-independent way. But, since the password (beyond other inputs) is used to derive the key of the underlying stream cipher, this assumption does not hold, i.e., the output of the stream cipher depends on the password, rendering (theoretical) CTAs possible.*

Parallel [254]. Parallel has not been designed to be a memory-demanding password scrambler. Instead, it is highly optimized for parallelization. First, a value *key* is derived from the secret input *pwd* and the salt s by

$$key \leftarrow \text{SHA-512}(\text{SHA-512}(s) \mid\mid pwd).$$

The value *key* is used (without being changed) during the CLEAR WORK phase of Parallel. Since this phase defines the main effort for computing the password hash, it is highly likely that a WGCA adversary can gain knowledge about *key*. Then, the following WGC attack is possible: For each password candidate x and the known value s, compute $y \leftarrow \text{SHA-512}(\text{SHA-512}(s) \mid\mid x)$ and check whether $key = y$. If so, mark x as a valid password candidate. Since the internal state is given by only the subsequently updated output of SHA-512, GCAs are not applicable for Parallel.

PolyPassHash [66]. PolyPassHash denotes a threshold system with the goal to protect an individual password (hash) until a certain number of correct passwords (and their corresponding hashes) are known. Thus, it aims at protecting an individual password hash within a file containing a lot of password hashes, rendering PolyPassHash not to be a password scrambler itself. The protection lies in the fact that one cannot easily verify a target hash without knowing a minimum number of hashes (this technical approach is referred to as PolyHashing). In the PolyHashing construction, one maintains a (k, n)-threshold cryptosystem, e.g., Shamir Secret Sharing [242]. Each password hash $h(pwd_i)$ is blinded by a share $s(i)$ for $1 \leq i \leq k \leq n$. The value $z_i \leftarrow h(pwd_i) \oplus s(i)$ is

stored in a so-called PolyHashing store at index i. The shares $s(i)$ are not stored on disk. For efficiency reasons, a legal party, e.g., a server of a social networking system, has to store at least k shares in the RAM to compare incoming requests on-the-fly. Thus, this system provides security only against adversaries that are only able to read the hard disk but not the volatile memory (RAM).

Since the secret (of the threshold cryptosystem) or at least the k shares have to be in memory, GCAs are possible by just reading the corresponding memory. The password itself is only hashed and blinded by $s(i)$. Thus, if an adversary is able to read the shares or the secret from memory, it can easily filter wrong password candidates, i.e., rendering PolyPassHash vulnerable to WGCAs.

POMELO [263]. POMELO contains three update functions $F(S,i), G(S,i,j)$, and $H(S,i)$, where S denotes the internal state and i and j the indices at which the state is accessed. Those functions update at most two state words per invocation. The functions F and G provide deterministic random memory accesses (determined by the cost parameter t_cost and m_cost), whereas the function H provides random memory accesses determined by the password, rendering POMELO at least vulnerable to CTAs. Since the password is used only to initialize the state, which itself is overwritten $3 \cdot 2^{t_cost} + 2$ times on average, POMELO provides resistance to GCAs and WGCAs.

Pufferfish [120]. The main memory used within Pufferfish is given by a two-dimensional array consisting of 2^{5+m_cost} 512-bit values, which is regularly accessed during the password-hash generation. The first steps of Pufferfish hash the password. The result is then overwritten $2^{5+m_cost} + 3$ times, rendering WGCAs not possible. The state word containing the hash of the password ($S[0][0]$) is overwritten 2^{t_cost} times. Thus, there exists no shortcut for an adversary, rendering GCAs impossible.

Rig [67]. Rig maintains two arrays a (sequential access) and k (bit-reversal access). Both arrays are iteratively overwritten $r \cdot n$ times, where r denotes the round parameter and n the iteration parameter, rendering Rig resistant to GCAs. Note that within the setup phase, a value α is computed by

$$\alpha = H_1(x) \quad \text{with} \quad x = pwd \mid\mid len(pwd) \mid\mid \ldots,$$

Since the first α (which is directly derived from the password) is only used during the initialization phase, WGCAs are not applicable.

***schvrch* [260].** The password scrambler *schvrch* maintains an internal state of $256 \cdot 64$-bit words (2 kB), which is initialized with the password, salt and their corresponding lengths, and the final output length. After this step, the password can be overwritten. This state is processed t_cost times by a function $revolve()$, which affects in each invocation all state words. Next, after applying a function $stir()$ (again, changing all state entries), it expands the state to m_cost times the state length. Each part (of size state length) is then processed to update the internal state, producing the hash after each part was processed. Thus, the state word initially containing the password is overwritten $t_cost \cdot m_cost$ times, rendering GCAs impossible. Further, neither the password nor a value directly derived from it is required during the invocation of *schvrch*, which thwarts WGCAs.

`Tortuga` [239]. GCAs and WGCAs are not possible for `Tortuga` since the password is absorbed into the underlying sponge structure, which is then processed at least twice by the underlying keyed permutation (Turtle block cipher [61]), and neither the password nor a value derived from it has to be in memory.

Algorithm 19 Garbage-Collector Attack on SkinnyCat

1: Obtain $mem[0, \ldots, memlen/(2 \cdot blocklen) - 1]$ and PRK from memory
2: Create a state $state'$ and an array mem' of the same size as $state$ and mem
3: $fromAddr \leftarrow slidingReverse(1) \cdot blocklen$
4: $prevAddr \leftarrow 0$
5: $toAddr \leftarrow blocklen$
6: **for** each password candidate x **do**
7: Compute PRK' as described using the password candidate x
8: Initialize $state'$ and mem' as prescribed using PRK'
9: $state'[0] \leftarrow (state'[0] + mem'[1]) \oplus mem'[fromAddr++]$
10: $state'[0] \leftarrow \text{ROTATE_LEFT}(state'[0], 8)$
11: $mem'[blocklen + 1] \leftarrow state'[0]$
12: **if** $mem'[blocklen + 1] = mem[blocklen + 1]$ **then**
13: mark x as a valid password candidate
14: **else**
15: go to Line 6.

SkinnyCat and TwoCats [74]. SkinnyCat is a subset of the TwoCats scheme optimized for implementation. Both algorithms maintain a 256-bit state *state* and an array of 2^{m_cost+8} 32-bit values (*mem*). During the initialization, a value PRK is computed as follows:

$$PRK \leftarrow Hash(len(pwd), len(s), \ldots, pwd, s),$$

where s denotes the salt. The value PRK is used in the initialization phase and is overwritten only in the penultimate step of SkinnyCat (in the function *addIntoHash()*). Thus, an adversary that gains knowledge about the value PRK is able to launch the following WGCA: For each password candidates x and the known value s, compute $PRK' \leftarrow Hash(len(x), len(s), \ldots, x, s)$ and check whether $PRK = PRK'$. If so, mark x as a valid password candidate.

Within TwoCats, the value PRK is overwritten at an early state of the hash value generation. TwoCats maintains consists of a garlic application loop from $startMemCost = 0$ to $stopMemCost$, where $stopMemCost$ is a user-defined value. In each iteration, the value PRK is overwritten, rendering WGCAs for TwoCats not possible.

Both SkinnyCat and TwoCats consist of two phases each. The first phase updates the first half of the memory (early memory) $mem[0, \ldots, memlen/(2 \cdot blocklen) - 1]$, where the memory is accessed in a password-independent manner. The second phase updates the second half of the memory $mem[memlen/(2 \cdot blocklen), \ldots, memlen/blocklen - 1]$, where the memory is accessed in a password-dependent manner. Thus, both schemes provide only partial resistance to CTAs. For SkinnyCat, the early memory is never overwritten, allowing an adversary to launch the GCA in Algorithm 19. Note that this attack does not work for TwoCats because the memory is overwritten early.

Yarn [150]. Yarn maintains two arrays *state* and *memory*, consisting of *par* and 2^{m_cost} 16-byte blocks, respectively. The array *state* is initialized using the salt. Afterwards, *state* is processed using the BLAKE2b compression function with the password *pwd* as message, resulting in an updated array *state1*. This array has to be stored in memory since it is used as input to the final phase of Yarn. The array *state* is expanded afterwards and further, it is used to initialize the array *memory*. Next, *memory* is updated continuously and both *memory* and *state* are overwritten continually. The array *state1* is overwritten lastest in the final phase of Yarn. Thus, GCAs are not possible for

Yarn. Nevertheless, the array *state1* is directly derived from *pwd* and is stored until the final phase occurs. Thus, an adversary can launch the WGCA in Algorithm 20, where s denotes the salt.

Algorithm 20 Garbage-Collector Attack on Yarn

1: $h \leftarrow$ Blake2b_GenerateInitialState($outlen, s, pers$)
2: **for** each password candidate x **do**
3:   $h' \leftarrow$ Blake2b_ConsumeInput(h, x)
4:   *state1'* $\leftarrow$ Truncate($h', outlen$)
5:   **if** *state1'* = *state1* **then**
6:     mark x as a valid password candidate
7:   **else**
8:     go to Line 2

yescrypt [212]. The yescrypt password scrambler maintains two lookup tables V and $VROM$, where V is located in the RAM and $VROM$ in the ROM. Since our attacks only target the RAM, we neglect the lookup table $VROM$ for our analysis. Depending on the flag `YESCRYPT_RW`, the behavior of the memory management in the RAM can be switched from “write once, read many” to “read-write”, which leads to (at least) partial overwriting of V using random memory accesses. Furthermore, yescrypt provides (among others) a flag `YESCRYPT_WORM`, which is used to activate the `scrypt` compatibility mode by enabling a parameter t (controlling the computational time of yescrypt) and pre- and post-hashing (whereas pre-hashing is used to overwrite the password before any time- and memory-consuming action is performed). In the following, we briefly analyze under which requirements (parameter sets) yescrypt provides resistance to GCAs and WGCAs.

No flags are set and $g = 0$: Then, yescrypt runs in `scrypt` compatibilitry mode (when used without Read-Only Memory (ROM)) and thus, the same attacks as described in Section 6.3 are applicable.

No flags are set and $g >= 0$: Then, yescrypt is vulnerable to WGCAs. Thus, even if $g > 0$, the password remains in memory for one full invocation of the time- and memory-consuming core of yescrypt since pre- and post-hashing do not overwrite the password.

`YESCRYPT_RW` is set and $g = 0$: Then, the second loop of `ROMix` (Lines 23-25 of Algorithm 1) performs less than N writes to V if $t = 0$ or if $t = 1$ and $N \geq 8$. Since V is not fully overwritten, this allows GCAs similar to the ones explained for `scrypt` (but with higher effort since V is at least partially overwritten). For $t > 1$, it is most likely that the whole internal state V is overwritten; hence, we say that yescrypt provides resistance to GCAs in this case.

$g > 0$: Then, yescrypt provides resistance to GCAs since V is overwritten at least once in the second invocation of the first loop of `ROMix` (Lines 20-22 of Algorithm 1). This holds independently from any flags or the parameter t.

Smaller instance called before: Under the following requirements, a 64-times smaller instance of yescrypt is invoked before the full yescrypt:

- `YESCRYPT_RW` is set.
- $p \geq 1$, where p denotes the number of threads running in parallel.
- $N/p \geq 256$, where N denotes the memory size of the state in the RAM, i.e., size of V.

Algorithm	Complexity in pROM
Catena-Dragonfly	$\mathcal{O}(G^{1.5})$ (BRG) [23], $\widetilde{\Omega}(G^{1.5}), \widetilde{\mathcal{O}}(G^{1.625})$ [21]
Catena-Butterfly	$\mathcal{O}(G\log^2(G))$ (DBG) [23], $\widetilde{\Omega}(G^{1.5}), o(G^{1.625})$ [21]
Argon2i-A	$\widetilde{\Omega}(G^{1.\bar{6}}), \widetilde{\mathcal{O}}(G^{1.708})$ [21]
Argon2i-B	$\widetilde{\Omega}(G^{1.\bar{6}})$ [21], $\mathcal{O}(G^{1.8})$ [20]
Balloon Hashing (SB)	$\widetilde{\Omega}(G^{1.\bar{6}}), \widetilde{\mathcal{O}}(G^{1.708})$ [21]
Balloon Hashing (DB)	$\widetilde{\Omega}(G^{1.5}), \widetilde{\mathcal{O}}(G^{1.625})$ [21]
Rig, TwoCats, Gambit	$\mathcal{O}(G^{1.75})$ [24]
POMELO	$\mathcal{O}(G^{1.83})$ [24]
Lyra2	$\mathcal{O}(G^{1.67})$ [24]
MHF example [23]	$\mathcal{O}(G^2)$

Table 11.5: Overview of the pebbling complexities in the pROM. All candidates provide a pebbling complexity of $\mathcal{O}(G^2)$ in the sequential setting. The Balloon Hashing family consists of three members, where SB refers to the Single-Buffer and DB to the Double-Buffer variant. The family Argon2i provides two members, Argon2i-A and Argon2i-B. The '$\widetilde{\ }$' symbol denotes that the given bounds disregard logarithmic factors.

- $N/p * r \geq 2^{17}$, where r denotes the memory per thread.

If these conditions hold, yescrypt overwrites the password very fast; hence, providing resistance to WGCAs.

11.2 Sequential vs. Parallel Model

In the last section of this chapter, we will briefly discuss the so-called Parallel Random-Oracle Model (pROM) that was introduced in 2015 by Alwen and Serbinenko [23]. This model contrasts the sequential pebbling game introduced by Lengauer and Tarjan [170] since it considers multiple invocations of the considered memory-hard function in parallel for distinct inputs. Lengauer and Tarjan have shown that Directed Acyclic Graphs (DAGs) with constant in-degree, e.g., DBG_λ^g and BRG_λ^g, do not scale well in that setting. In Table 11.5, we have summarized the best results obtained in the parallel setting published so far (at the time when writing this thesis), where all algorithms employ a data-independent graph-based structure which can be exploited in the pROM, except for the memory-hard function presented in [23].
In 2016, Alwen and Blocki introduced a new measure for memory-hard functions considering the required amount of energy (i.e., electricity) for their introduced attack [19]. Further, they introduced the security notion of *depth robustness*, which they describe as follows:

> Informally, 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 [...] has low depth (i.e. contains only short paths). [19]

Based on this new notion, the authors provide a generic attack on graphs which do not satisfy depth robustness, i.e.,

> [...] no DAG with constant indegree is sufficiently depth-robust to completely resist the attack *(referring to the newly introduced attack in the pROM)*. More precisely, we

> show that any iMHF *(data-independent memory-hard function)* is at best c-ideal[3] for $c = \Omega(\log^{1-\epsilon} n)$ and any $\epsilon > 0$. [19]
> *(Annotation added by the author of this thesis.)*

Assume an algorithm designed to require $\mathcal{O}(G)$ memory units. Then, the generic attack considers two phases:

Balloon Phase: Requires about $\mathcal{O}(G)$ memory; can be computed fast based on the stored values in S. After that phase, all but the S pebbles can be removed from the graph.

Light Phase: Requires negligible $\ell \ll G$ (much less than G) memory units and can be computed in parallel for many distinct inputs.

The main idea behind the two phases is that an adversary in the pROM model requires only $\mathcal{O}(G+\ell)$ memory units while computing many password hashes in parallel. The authors also presented a hypothetical machine which could run using $G^{1/4}$ light-phase chips while requiring only a single balloon-phase chip. Thereby, the light phase requires $\mathcal{O}(G^{3/4} \ln G)$ memory and a single instance of H while the balloon phase requires $\mathcal{O}(G \ln G)$ memory and $\sqrt{G}$ instances of H. However, they concluded their paper with the following statement:

> Because pROM is a theoretical model of computation it is not obvious a priori that our attacks translate to practically efficient attacks that could be implemented in real hardware because it can be difficult to dynamically reallocate memory between processes in an ASIC (the amount of memory used during each round of a balloon phase is significantly greater than the amount of memory used during each round of a light phase). [19]

A further extension of their attacks and its application to Argon2-B is given by Alwen and Blocki in [20].

Remark 11.4. *All results discussed in this section do only hold for memory-hard functions which employ a secret-independent memory-access patter. When considering* Catena*, one would enable the options Γ and Φ as discussed in Sections 7.2 and 9.1. Then, an adversary in the Parallel Random-Oracle Model (pROM) would suffer from the following two facts: (1) it requires a new balloon-phase chip for every fresh public input given to Γ, and (2) the graph layer within Φ could not be computed in parallel due to its property of being sequential memory-hard. Nevertheless, the adversary could still reduce the memory effort for computing F_λ between Γ and Φ. However, that would require an even higher number of synchronization steps between the light-phase and balloon-phase chips.*

The application of the proposed attack [19] makes most sense as long as the memory-access pattern depends on neither a public or a secret input. Thus, one must make a decision whether to provide resistance in the pROM or to CTAs.

In a second work [22], Alwen et al. introduced two new classes of the pebble game: the *randomized* and the *entangled* pebbling game:

[3] See [19] for a definition of c-ideal.

Randomized: In the randomized pebble game, a single node of the underlying graph structure is chosen at random and the complexity is given by measuring the number of required pebble movements for pebbling that very node. In the parallel setting, multiple challenges (nodes) are chosen at random. Alwen et al. analyzed `scrypt`-like functions in this model providing a lower bound of $\Theta(G^2/\log^2(G))$. The complexity can be further reduced to $\mathcal{O}(G^{1.5})$, if the challenges are known in advance.

Entangled: In this model, Alwen et al. improved the pebble game by storing the XOR value of hash values instead of the values itself, reducing the total amount of memory required. They have shown, that the complexity of an entangled pebbling adversary is given by $\Theta(G^2/\log^2(G))$ for `scrypt`-like functions.

The authors further provided a lower bound for *arbitrary* adversaries for pebbling `scrypt`-like functions: $\Theta(G^2/\log^2(G) \cdot \gamma_G)$, where $\gamma_G > 3/2$ is the best lower bound the authors can provide.

Summarising, the parallel attack model gives a new opportunity to analyze password-hashing schemes. And, even if there does not exist a practical instance of the depth-robust example introduced by Alwen and Serbinenko [23], we recommend depth robustness to be considered as a desirable goal of modern password scramblers.

11.3 Summary

In this chapter, we provided an overview (functionality, security, and general properties) of the non-withdrawn candidates of the Password Hashing Competition (PHC), the instances of Catena introduced in Chapter 10, and `scrypt`. Furthermore, we analyzed each algorithm regarding to its vulnerability to Garbage-Collector Attacks (GCAs) and Weak Garbage-Collector Attacks (WGCAs) – two attack types introduced in this thesis (see Section 3). Even if both attacks require access to the memory on the target's machine, they show potential weaknesses that should be taken into consideration. As a results, we have shown GCAs on CENTRIFUGE, PolyPassHash, `scrypt` (see Section 6.3), SkinnyCat, and yescrypt. Additionally, we have shown that WGCAs are applicable to battcrypt, CENTRIFUGE, EARWORM, Parallel, PolyPassHash, `scrypt`, SkinnyCat, Yarn, and yescrypt. Moreover, we discussed the Parallel Random-Oracle Model (pROM) introduced by Alwen and Serbinenko [23] and further developments and enhancements of the generic attacks in that model.

Part IV

Authenticated Encryption

CHAPTER 12

Reforgeability of Authenticated Encryption Schemes

The question of integrity will get finer and finer and more delicate and more beautiful.

RICHARD BUCKMINSTER FULLER

The goal of Authenticated Encryption (AE) schemes is to protect simultaneously privacy and authenticity of messages. Authenticated Encryption with Associated Data (AEAD) schemes provide additional authentication for associated data. The standard security requirement for AE schemes is to prevent leakage of any information about protected messages except for their respective lengths. However, stateless encryption schemes would enable adversaries to detect if the same associated data and message has been encrypted before under the current key. Thus, Rogaway proposed nonce-based encryption [228], where the user must provide an additional nonce for every message it wants to process – a number used once (nonce). An AE scheme requiring a nonce as input is called a Nonce-Based Authenticated Encryption (NAE) scheme.

In this chapter, we focus on the integrity of NAE schemes. More detailed, we elaborate on the possibility for finding multiple forgeries once a single forgery against the considered scheme is found. The first attack that headed in that direction was introduced in 2002 by Ferguson, who showed collision attacks on OCB (version 1) [230] and a Ctr-CBC-like MAC [92]. He demonstrated that finding a collision within the message processing of OCB "*leads to complete loss of an essential function*" (referring to the loss of authenticity/integrity).

Later on, in 2005, the term *multiple forgery attacks* was defined by McGrew and Fluhrer [181]. They introduced the measure of expected number of forgeries and conducted a thorough analysis of GCM [180], HMAC [36], and CBC-MAC [38]. In 2008, Handschuh and Preneel [127] introduced key-recovery and universal forgery attacks against several Message Authentication Code (MAC) algorithms. The term *Reforgeability* was first formally defined by Black and Cochran in 2009, where they examined common MACs regarding their security to this new measurement [57]. Further, they introduced WMAC, which they argued to be the "*best fit for resource-limited devices*".

Here, we recoin the term *Reforgeability* so that refers to the complexity of finding subsequent forgeries once a first forgery has been found. Thus, it defines the hardness of forging a ciphertext after the first forgery succeeded. For a reforgeability attack to work, an adversary must be provided

with a verification oracle in addition to its authentication (and encryption) oracle. In practice, such a setting can, for example, be found when a client tries to authenticate itself to a server and has multiple tries to log in to a system. Thus, the server would be the verification oracle for the client. Obviously, the same argument holds for the case when the data to be send is of sensitive nature, i.e., the data itself has to be encrypted. Thus, besides the resistance of MACs to reforgeability, also the resistance of AE schemes is of high practical relevance.

Since modern and cryptographically secure AE schemes should provide at least INT-CTXT security when considering integrity, the first forgery is usually not trivially found and depends on the size of the tag or the internal state. For that reason, reforgeability becomes especially essential when considering resource-constrained devices limited by, e.g., radio power, bandwidth, area, or throughput. This is not uncommon when considering low-end applications such as sensor networks, Voice over IP (VoIP), streaming interfaces, or, for example, devices connected to the Internet of Things (IoT). In these domains, the tag size τ of MACs and AE schemes can be quite small, e.g., $\tau = 64$ or $\tau = 32$ bits [137, 183], or even drop to $\tau = 8$ bits as mentioned by Ferguson for voice systems [93]. Therefore, even if the AE scheme is secure in the INT-CTXT setting up to τ bits, it is not unreasonable for an adversary to find a forgery for such a scheme in general. Nevertheless, even if finding the first forgery requires a large amount of work, a rising question is, whether it can be exploited to find more forgeries with significantly less effort than 2^τ operations per forgery.

For our analysis, we derive a new security notion j-INT-CTXT: an adversary, finding the first forgery with an effort of t_1, can generate j additional forgeries in polynomial time depending on j. In general, the best case for an adversary would be to find j additional forgeries in $t_1 + j$. Nevertheless, for three schemes (AES-OTR [188], COLM [25], and OCB [162]), we recall and show that finding j forgeries can also be done in time t_1 (thus, the j additional authentication queries are not even required).

Due to the vast number of submissions to the CAESAR competition [44], cryptanalysis proceeds slowly for each individual scheme, e.g., the only forgery attacks published for third-round CAESAR candidates are on AES-COPA [26, 172, 194], which even might become obsolete since AES-COPA and ELmD [77] have been merged to COLM [25]. Besides looking at third-round CAESAR candidates, we also analyzed further widely-used AE schemes, e.g., GCM [180], EAX [42], CCM [88], and CWC [158]. Naturally, due to their longer existence, there exist a lot more cryptanalysis on those schemes in comparison to the CAESAR candidates (see [111, 144, 145, 182, 225, 233] for some examples. The hope is that an INT-CTXT-secure AE scheme does not lose its security when considering reforgeability, i.e., j-INT-CTXT.

We briefly introduce what we mean by *resistant to j*-IV-Collision Attacks (j-IV-CAs), whereby we assume the first forgery to be the result from an internal collision of the processing of the associated data and or the nonce. Hereby, we consider nonce-ignoring as well as nonce-respecting adversaries.

Nonce-Ignoring Adversaries: We call an NAE scheme resistant to j-IV-CAs if the required effort of a *nonce-ignoring* j-IV-CA adversary for finding $j + 1$ forgeries (including the first one) is greater than $t_1 + j$, where t_1 denotes the effort for finding the first forgery.

Nonce-Respecting Adversaries: We call an NAE scheme resistant to j-IV-CAs if the required effort of a *nonce-respecting* j-IV-CA adversary for finding $j + 1$ forgeries (including the first one) is greater than $t_1 \cdot j/2$, where t_1 denotes the effort for finding the first forgery.

Further, we say that an NAE scheme is *semi-resistant* to j-IV-CAs if the internal state is wide, e.g., greater or equal to $2n$ bits (e.g., sponge-based NAE schemes), where n denotes the block size of

Scheme		NI	NR	Scheme		NI	NR
Third-Round CAESAR Candidates							
ACORN	[264]	$t_1 + j$	$t_1 \cdot j/2$	KETJE	[50]	$t_2 + j$	$t_2 \cdot j/2$
AEGIS	[267]	$t_2 + j$	$t_2 \cdot j/2$	KEYAK	[122]	$t_2 + j$	$t_2 \cdot j/2$
AES-OTR	[188]	t_1	t_1	MORUS	[265]	$t_2 + j$	$t_2 \cdot j/2$
AEZv4	[135]	$t_1 + j$	$t_1 \cdot j/2$	NORX	[32]	$t_2 + j$	$t_2 \cdot j/2$
ASCON	[81]	$t_2 + j$	$t_2 \cdot j/2$	NR-NORX	[32]	$t_2 + j$	$t_2 \cdot j$
CLOC	[141]	$t_1 + j$	$t_1 \cdot j$	OCB	[162]	t_1	t_1
COLM	[25]	t_1	$t_1 + j$	SILC	[141]	$t_1 + j$	$t_1 \cdot j$
DEOXYS	[143]	$t_1 + j$	$t_1 \cdot j$	TIAOXIN	[201]	$t_2 + j$	$t_2 \cdot j/2$
JAMBU	[266]	$t_1 + j$	$t_1 \cdot j/2$				
Classical AE Schemes							
CWC	[158]	$t_1 + j$	$t_1 \cdot j$	CCM	[88]	$t_1 + j$	$t_1 + j$
EAX	[42]	$t_1 + j$	$t_1 \cdot j$	GCM	[180]	$t_1 + j$	$t_1 + j$

Table 12.1: Expected #oracle queries required for j forgeries for IV/nonce-based classical schemes and third-round CAESAR candidates. By t_1 and t_2, we denote the computational cost for obtaining the first forgery, where t_2 relates to wide-state designs. **NI/NR** = nonce-ignoring/nonce-respecting setting. Since we obtained the same results for DEOXYS-I and DEOXYS-II, we combine them to DEOXYS in this table. NR-NORX (draft) means the nonce-misuse-resistant version of NORX.

message and ciphertext blocks. That would make the search for a generic collision significantly more demanding than the search for multiple forgeries. We denote the effort for finding a collision within a wide internal state by t_2. Finally, we call an NAE scheme *vulnerable* to j-IV-CAs if it is neither resistant nor semi-resistant to j-IV-CAs.

This chapter classifies NAE schemes depending on the usage of their inputs to the initialization, encryption, and authentication process, and categorize the considered AE schemes regarding to that classification.

For a systematic analysis, we introduce the j-IV-CA based on the introduced security definition j-INT-CTXT where the former allows us to compute expected upper bounds on the hardness of further forgeries (a summary of our results can be found in Table 12.1). For our attack, we pursue the idea of the message-block-collision attacks presented in [92, 230]. However, in contrast to those works, we focus on an internal collision within the processing of the associated data and the nonce. In the last section, we provide two alternative approaches to provide resistance in the sense of reforgeability and j-IV-CAs. Moreover, for AES-OTR, COLM, and OCB, we recall and describe further attacks that render multi-forgery attacks more efficient than our generic approach.

12.1 Classification of AE Schemes

In our work, we consider AE schemes from a general point of view. Therefore, in comparison to the classification of Namprempre, Rogaway, and Shrimpton [193], we introduce one additional optional input to the tag-generation step (a key-dependent chaining value) and further, we distinguish between the message and the ciphertext being input to the tag generation.

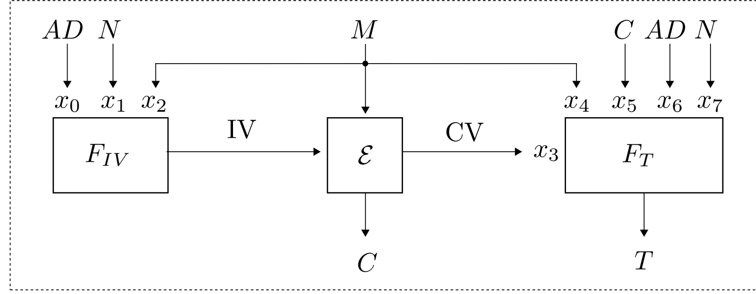

Figure 12.1: Generic AE scheme as considered in our analysis.

We classify AE schemes according to their inputs to an initialization function F_{IV} and a tag-generation function F_T. Let $\mathcal{K}, \mathcal{AD}, N, \mathcal{IV}, \mathcal{T}, \mathcal{M}, \mathcal{CV}$, and $\mathcal{C}$ define the key, associated data, nonce, IV, tag, message, chaining-value, and ciphertext space, respectively. We define three functions F_{IV}, $\mathcal{E}$, and F_T as follows:

$$\begin{aligned} F_{IV}: &\quad \mathcal{K}[\times\mathcal{AD}][\times N][\times\mathcal{M}] && \to \mathcal{IV}, \\ \mathcal{E}: &\quad \mathcal{K}\times\mathcal{IV}\times\mathcal{M} && \to \mathcal{C}[\times\mathcal{CV}], \\ F_T: &\quad \mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}][\times\mathcal{AD}][\times N] && \to \mathcal{T}, \end{aligned}$$

where $\mathcal{AD}, N, \mathcal{M}, \mathcal{CV}, \mathcal{C} \subseteq \{0,1\}^*$, $\mathcal{T} \subseteq \{0,1\}^\tau$, and $\mathcal{IV} \subseteq \{0,1\}^*$. The expressions (sets) given in brackets are *optional* inputs to the corresponding function, e.g., the function F_{IV} must be provided with at least one input (the key $K \in \mathcal{K}$), but is able to process up to four inputs (including associated data $AD \in \mathcal{AD}$, nonce $N \in N$, and message $M \in \mathcal{M}$).

From this, we introduce a generic classification based on which input is used in F_{IV} and F_T. Note that the signature of the encryption algorithm $\mathcal{E}$ is equal for all classes described, i.e., it encrypts a message M under a key K and an $IV \in \mathcal{IV}$, and outputs a ciphertext $C \in \mathcal{C}$.

In the following, we encode the combination of inputs as a sequence of eight bits $x_0, \ldots, x_7$, where each bit denotes whether an input is used (1) or not (0), resulting in a total of $2^8 = 256$ possible classes. More detailed, the first three bits x_0, x_1, x_2 denote whether the associated data AD, the nonce N, or the message M are used as input to F_{IV}, respectively. The bits $x_3, \ldots, x_7$ denote whether a key-dependent chaining value CV, M, C, AD, or N are used as input to F_T, respectively. Figure 12.1 depicts our generic AE scheme. For example, the string (11010011) represents F_{IV} : $\mathcal{K} \times \mathcal{AD} \times N \to \mathcal{IV}$ and $F_T : \mathcal{K} \times \mathcal{CV} \times \mathcal{AD} \times N \to \mathcal{T}$ as it would be the case for, e.g., POET [3], CLOC, and SILC [141]. Further, we mark a bit position by '*' if we do not care about whether the specific input is available or not.

The authors of [193] distinguished between IV-based (ivE) and nonce-based (nE) encryption schemes. Such a distinction is covered by our generalized approach since one can simply assume the only input to F_{IV} to be the nonce (and the key) and making F_{IV} itself the identity function, i.e., it forwards the nonce N to the encryption function $\mathcal{E}$. Moreover, AE schemes built from generic composition can be modeled by setting $x_3 = 0$ and assuming F_T to be a PRF-secure MAC.

Our next step is to significantly reduce the number of possible classes by disregarding those that are trivially insecure. First, we can simply discard $2^4 = 16$ classes of the form $(00{*}{*}{*}{*}00)$, where neither the nonce N nor the associated data AD is considered as input. Similarly, we can exclude $6 \cdot 2^4 = 96$ classes which lack the use of either the nonce *or* the associated data, i.e., $\{(01{*}{*}{*}{*}00), (01{*}{*}{*}{*}01), (10{*}{*}{*}{*}00), (10{*}{*}{*}{*}10), (00{*}{*}{*}{*}01), (00{*}{*}{*}{*}10)\}$. Finally, since a secure nonce-based AE scheme requires the nonce to influence at least the encryption step, we can

Set of Classes	Input to F_{IV}	Input to F_T
$(01****10)$	$\mathcal{K} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}] \times \mathcal{AD}$
$(01****11)$	$\mathcal{K} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}] \times \mathcal{AD} \times N$
$(11****00)$	$\mathcal{K} \times \mathcal{AD} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}]$
$(11****01)$	$\mathcal{K} \times \mathcal{AD} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}] \times N$
$(11****10)$	$\mathcal{K} \times \mathcal{AD} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}] \times \mathcal{AD}$
$(11****11)$	$\mathcal{K} \times \mathcal{AD} \times N[\times\mathcal{M}]$	$\mathcal{K}[\times\mathcal{CV}][\times\mathcal{M}][\times\mathcal{C}] \times \mathcal{AD} \times N$

Table 12.2: Overview of accepted classes. All excluded classes are trivially insecure.

Name & Class [193]	Class	Name & Class [193]	Class
A1, A1.100111	(01001011)	A7, A3.100111	(01001011)
A2, A1.110111	(11001011)	A8, A3.110111	(11001011)
A3, A1.101111	(01101011)	N1, N1.111	(11100000)
A4, A1.111111	(11101011)	N2, N2.111	(01000111)
A5, A2.100111	(01000111)	N3, N3.111	(01001011)
A6, A2.110111	(11000111)		

Table 12.3: The eleven "favored" NAE schemes considered by [193] and how we map them according to our classification. The classes (A1, A7) and (A2, A8) have pairwise the same class according to our generic NAE scheme. That stems from the fact that we do not follow the distinction of NAE schemes from [193] regarding to whether the message/ciphertext can be processed in parallel or if the tag can be truncated. For the scheme N3, it holds that $\mathcal{E}$ gets the two separate inputs $F_L(A, N, M)$ and the nonce N. Since there is no segregated tag generation for N3 (the tag is part of the ciphertext), we interpreted F_L as F_{IV} and consider F_{IV} to additionally hand over the nonce N to the encryption $\mathcal{E}$ internally in plain.

further disregard the $3 \cdot 2^4 = 48$ classes $\{(00****11), (10****01), (10****11)\}$ which omit the nonce in the initialization function F_{IV}. As a result, we reduced the number of relevant classes to 96. An overview can be found in Table 12.2.

> **Remark 12.1.** *Namprempre et al. [193] reduced 160 to eight IV-based and three nonce-based "favored" schemes that can be mapped according to our classification (see Table 12.3, and Section 12.1 for our classification). Moreover, our generalized AE scheme, as introduced in Section 12.1, allows to model any possible instantiation, including GC-based schemes.*

12.2 j-INT-CTXT Analysis of NAE Schemes

In this section, we introduce a new attack type called j-IV-Collision Attack (j-IV-CA) as one possible way to analyze the security of an NAE scheme regarding to reforgeability. We provide two variants (1) for the nonce-ignoring (NI; also known as nonce-misuse) and (2) the nonce-respecting (NR) setting.

Algorithm 21 j-IV-Collision Attack for nonce-ignoring adversaries.

1: Choose an arbitrary fixed message M
2: $\mathcal{Q} \leftarrow \emptyset$
3: **for** $i \leftarrow 1$ to t_1 **do**
4: Choose (AD_i, N_i) with $(AD_i, N_i) \notin \mathcal{Q}_{|AD,N}$
5: Query (AD_i, N_i, M) and receive (C_i, T_i).
6: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(AD_i, N_i, M, C_i, T_i)\}$
7: **if** $T_i \in \mathcal{Q}_{|T}$ **then**
8: Store the tuples (AD_i, N_i, M, C_i, T_i) and (AD_k, N_k, M, C_k, T_k) for which $T_i = T_k$
9: **break**
10: **for** $\ell \leftarrow 1$ to j **do**
11: Choose $M_\ell \notin \mathcal{Q}_{|M}$
12: Query (AD_i, N_i, M_ℓ) and receive (C'_ℓ, T'_ℓ)
13: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(*, *, M_\ell, *, *)\}$
14: Output the forgery $(AD_k, N_k, C'_\ell, T'_\ell)$

12.2.1 j-IV-Collision Attack

The core idea of a j-IV-CA is to (1) assume a first forgery can be found caused by an internal collision within the processing of the associated data AD and/or the nonce N and (2) to exploit this collision for efficiently constructing j further forgeries. Depending on the class of an AE scheme, such a collision can occur during the invocation of F_{IV}, F_T, or both.

Due to the character of the attacks presented in this section, we can derive a set of classes $\mathcal{C}_0$ of NAE schemes for which those attacks are trivially applicable. For all schemes belonging to this class, neither the message M, nor a message/ciphertext-depending chaining value CV, nor the ciphertext C influence the first collision found by our adversary. For instance, if an adversary tries to construct a collision for the outputs of F_{IV}, the only possible inputs to F_{IV} are either the nonce N, the associated data AD, or both. Therefore, the set $\mathcal{C}_0$ contains the following 22 classes of AE schemes:

$$\mathcal{C}_0 = \{(110***0*), (01*0001*), (1100001*)\}.$$

Nonce-Ignoring Setting. The attack for the nonce-ignoring setting is described in Algorithm 21. An adversary $\mathbf{A}$ starts by choosing a fixed arbitrary message M and pairs (AD_i, N_i) not queried before ($(AD_i, N_i) \notin \mathcal{Q}_{|AD,N}$, see Line 4). That builds up a query (AD_i, N_i, M) resulting in an oracle answer (C_i, T_i) which is stored by $\mathbf{A}$ in the query history $\mathcal{Q}$. Once a collision of two tag values T_i and T_k (implying a collision of two pairs $(AD_i, N_i) \neq (AD_k, N_k)$)[1] was found (Line 7 of Algorithm 21), $\mathbf{A}$ starts to generate j additionally queries with an effort of $\mathcal{O}(j)$ (Lines 10-14). In Lines 6 and 13, the adversary is collecting all tuples queried so far, where in Line 13 we are only interested in the values of M_ℓ, since these are not allowed to repeat (see Line 11) by the definition of $\mathbf{A}$.

It is easy to observe that $\mathbf{A}$ has to use the same nonce twice, i.e., N_i is chosen in Line 4 and reused in Line 12 of Algorithm 21. Independent from the number of queries of finding the j additional forgeries, $\mathbf{A}$ always (in the nonce-ignoring as well as in the nonce-respecting setting) has to find a collision for two pairs $(AD_i, N_i) \neq (AD_k, N_k)$. That number of queries (denoted by t_1 in general, or by t_2 if the scheme employs a wide state of $\geq 2n$ bits (or $\geq 2\tau$ bits, when referring to the size of the

[1] Based on our assumption, the case $T_i = T_k$ can be caused by an internal collision of the processing of two pairs $(AD_i, N_i) \neq (AD_k, N_k)$. Moreover, since we are considering the nonce-ignoring setting allowing an adversary for repeating the values N_i, we can say WLOG that we must have found two associated data values $A_i \neq A_k$ leading to an equal output of the processing of the associated data, e.g., the initialization vector IV (see Figure 12.1).

Algorithm 22 j-IV-Collision Attack for nonce-respecting adversaries.

1: Choose an arbitrary fixed message block M
2: $\mathcal{Q} \leftarrow \emptyset$
3: **for** 1 to j **do**
4: **for** $i \leftarrow 1$ to t_1 **do**
5: Choose (AD_i, N_i) with $(AD_i, N_i) \notin \mathcal{Q}_{|AD,N}$
6: Choose P_i with $P_i \notin \mathcal{Q}_{|P}$
7: Query $(AD_i, N_i, M \parallel P_i)$ and receive $(C_i^1 \parallel C_i^{P_i}, T_i)$.
8: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(AD_i, N_i, C_i^1 \parallel C_i^{P_i}, T_i)\}$
9: **if** $C_i^1 \in \mathcal{Q}_{|C^1}$ **then**
10: **A** outputs the tuples $(AD_i, N_i, C_i^1 \parallel C_k^{P_k}, T_k)$ and $(AD_k, N_k, C_k^1 \parallel C_i^{P_i}, T_i)$
11: for which $C_i^1 = C_k^1$ holds
12: **goto** Step 4

Scheme	NI	NR	Scheme	NI	NR
Third-Round CAESAR Candidates					
ACORN	–	2^{τ}	JAMBU	$2^{2n/2}$	$2^{2n/2}$
AEGIS	–	2^{τ}	Ketje	–	$2^{\min\{\tau,s\}}$
AES-OTR	–	$2^{\tau/2}$	Keyak	$2^{\min\{c/2,\tau\}}$	$2^{\min\{c/2,\tau\}}$
AEZv4	2^{55}	2^{55}	MORUS	–	2^{128}
Ascon	–	2^{τ}	OCB	–	2^{τ}
CLOC	$2^{n/2}$	$2^{n/2}$	SILC	–	$2^{\tau/2}$
COLM	2^{64}	2^{64}	NORX	–	$2^{\|\tau\|}$
Deoxys-I	–	2^{τ}	Tiaoxin	–	2^{128}
Deoxys-II	$2^{\tau/2}$	$2^{\tau-1}$			
Classical AE Schemes					
CCM	–	$2^{n/2}$	CWC	–	$2^{n/2}$
EAX	–	$2^{n/2}$	GCM	–	$2^{n/2}$

Table 12.4: Claimed INT-CTXT bounds of the considered AE schemes; **NR/NI** = nonce-respecting/nonce-ignoring adversary, where τ denotes the length of the tag, n the size of the internal state (usually the block size of the internally used block cipher), and c the capacity for sponge-based designs.

tag value), see Table 12.1) always depends on the concrete instance of our generic AE scheme and is usually bounded by at least $\mathcal{O}(q^2/2^n)$ (birthday bound), where q denotes the number of queries and n the state size in bits. In Table 12.4, the reader can find the security claims of the considered AE schemes provided by their respective designers.

Nonce-Respecting Setting. The second setting prohibits an adversary from repeating any value N_i during its encryption queries. Therefore, we introduce a modified version of the j-IV-CA as proposed above. Such an attack works for all schemes that allow to observe a collision of the outputs of the IV-generation step by just looking at the ciphertext blocks. Thus, during the first step, we

do not care about finding the first forgery but only about the collision during F_{IV} as shown in Algorithm 22. This attacks works also for NAE schemes that consider the associated data AD_i only as input to F_T. In such a situation, **A** would leave AD_i constant (or empty when considering F_{IV}) and would vary only N_i to find a collision within F_{IV}. If the number of queries for finding a collision during the processing of the associated data is given by t_1, an adversary requires $j \cdot t_1$ queries in average to obtain $2 \cdot j$ forgeries. Clearly, this attack is weaker than that in the nonce-misuse setting above, but still reduces the number of queries for finding j forgeries from $j \cdot t_1$ to $1/2 \cdot (j \cdot t_1)$.

12.2.2 Security Analysis

For all NAE schemes which belong to $\mathcal{C}_0$, there exists a straight-forward argument that they are insecure in the nonce-ignoring setting. A j-IV-CA, as defined in Algorithm 21, requires an adversary **A** to choose j pair-wise distinct messages $M_1, \ldots, M_j$. Beforehand, we assume **A** has found the first forgery for two distinct pairs (AD_i, N_i) and (AD_k, N_k) (Lines 3-9 of Algorithm 21) using t_1 queries.

Therefore, the j-IV-CA adversary **A** queries t_1 distinct pairs $(AD_i, N_i) \neq (AD_k, N_k)$, together with a fixed message M, until an internal collision leads to the case $T_i = T_k$. Since the event of that very first collision is independent from the message, a chaining value, and/or the ciphertext (requirement for an NAE scheme to be placed in $\mathcal{C}_0$), we can always choose a new message and still can ensure the internal collision for the pairs (AD_i, N_i) and (AD_k, N_k). Then, **A** only has to query (AD_i, N_i, M_ℓ) for a fresh message M_ℓ to the encryption oracle and receives (C'_ℓ, T'_ℓ), where it is trivial to see that the pair (C'_ℓ, T'_ℓ) will also be valid for (AD_k, N_k, M_ℓ). **A** then only has to repeat this process for j pairwise distinct messages M_ℓ.

In the case of a nonce-respecting adversary (see Algorithm 22), an internal collision of the processing of (AD_i and) N_i is detected by observing colliding ciphertext blocks (see Line 9). Since the attack requires an internal collision within the IV-generation step and the nonce N_i must not directly influence the tag-generation step F_T, the nonce N_i must be given as input to F_{IV}, but not to F_T. The associated data AD_i can be given as input to F_{IV}, F_T, or both. Therefore, the attack described in Algorithm 22 is applicable to all schemes belonging to the subset $\{(11{*}{*}{*}{*}00), (11{*}{*}{*}{*}00), (01{*}{*}{*}{*}10)\}$ of $\mathcal{C}_0$.

All remaining 74 classes in the set $\mathcal{C}_1$ provide resistance to j-IV-CAs from a theoretical point of view, i.e., with regard to our generalized AE scheme as shown in Figure 12.1.

$$\begin{aligned}\mathcal{C}_1 = \{&(01*0011*), (01*0101*), (01*0111*), (01*1001*), (01*1011*),\\&(01*1101*), (01*1111*), (1100011*), (1100101*).(1100111*),\\&(1101001*), (1101011*), (1101101*), (1101111*), (111*****)\}\end{aligned}$$

However, in practice, their security depends on the specific instance of F_{IV} and/or F_T. In the next section, we look at concrete instances from the class $\mathcal{C}_1$ as well as from $\mathcal{C}_0$ when we consider classical NAE schemes and third-round CAESAR candidates.

12.2.3 Concrete Instances of $\mathcal{C}_1$ and $\mathcal{C}_0$

The resistance to j-IV-CA of AE schemes from the class $\mathcal{C}_1$ is based on the fact that the message, and/or a chaining value, and/or the ciphertext affect the generation of the IV or the tag, i.e., is input to F_{IV} and/or F_T. However, if we move from our generalized approach to concrete instances of these classes, i.e., to existing AE schemes whose structure is defined by a class in $\mathcal{C}_1$, we will see that some of those classes do not provide resistance to j-IV-CAs. However, AE schemes whose

Scheme	Class	NI	NR	Scheme	Class	NI	NR
Third-Round CAESAR Candidates ($\mathcal{C}_0$)				Third-Round CAESAR Candidates ($\mathcal{C}_1$)			
ACORN	(11011000)	–	–	AEGIS	(11011010)	–	∘
AES-OTR (ser.)	(11001100)	–	–	AES-OTR (par.)	(01001110)	–	–
Ascon	(11010100)	–	∘	AEZv4	(11011011)	–	–
COLM	(11011000)	–	–	CLOC	(11010101)	–	•
JAMBU	(11011000)	–	–	Deoxys-I	(01011001)	–	•
Ketje	(11010000)	–	∘	Deoxys-II	(01011001)	–	•
NORX	(11010100)	–	∘	Keyak	(01011010)	–	∘
Classical AE Schemes ($\mathcal{C}_1$)				MORUS	(11011010)	–	∘
CCM	(01011011)	–	•	NR-NORX	(11110100)	∘	•
CWC	(01010110)	–	–	OCB	(01001010)	–	–
EAX	(01000111)	–	•	SILC	(11010101)	–	•
GCM	(01000111)	–	–	Tiaoxin	(11011010)	–	∘

Table 12.5: j-IV-CA-Resistance of the third-round CAESAR candidates and considered classical AE schemes, in the nonce-ignoring (**NI**) and the nonce-respecting (**NR**) setting. '•' indicates resistance, '∘' vulnerability under certain requirements, and '–' vulnerability. AES-OTR (ser.) means the serial and (par.) the parallel mode.

classes belong to $\mathcal{C}_0$ are vulnerable or at most semi-resistant to j-IV-CAs in both the NI and the NR setting. In Table 12.5, we give an overview of the resistance the considered AE schemes to j-IV-CAs. We also provide a brief discussion for those cases that are not trivially observable in the following. In addition to the generic j-IV-CAs in this section, we recall stronger multi-forgery attacks on OCB, AES-OTR, and COLM in Section 12.3.

AEGIS, MORUS, and Tiaoxin. These schemes are semi-resistant to j-IV-CAs in the nonce-respecting setting.[2] This stems from the fact that they employ very wide states, which are initialized by nonce and associated data, and which are more than twice as large as the final ciphertext stretch; therefore, the search for state collisions is at best a task of sophisticated cryptanalysis, and at worst by magnitudes less efficient than the trivial search by querying many forgery attempts. As a side effect, the search for state collisions is restricted to associated data and messages of equal lengths since their lengths are used in F_T (for that reason, we set $x_6 = 1$).

CWC and GCM. In the nonce-ignoring setting, forgeries for CWC and GCM can be obtained with a few queries. The tag-generation procedures of both modes employ a Carter-Wegman MAC consisting of XORing the encrypted nonce with an encrypted hash of associated data and ciphertext. The employed hash are polynomial hashes in both cases, which is well-known to lead to a variety of forgeries after a few queries when nonces are repeated.

In the nonce-respecting setting, both CWC and GCM possess security proofs that show that they provide forgery resistance up to the birthday bound (Iwata et al. [142] invalidated those for GCM and presented revised bounds which still are bound by the birthday paradox). However, a series of works from the past five years [233, 218, 2] illustrated that the algebraic structure of polynomial hashing may allow to retrieve the hashing key from forgery polynomials with many

[2]None of the designers of the corresponding schemes claim any security in the nonce-ignoring setting.

roots. The most recent work to date by Abdelraheem et al. [2] proposes universal forgery attacks that work on a weak key set. Thus, a nonce-respecting adversary could find the hash key and possess the power to derive universal forgeries for those schemes, even with significantly less time than our nonce-respecting attack.

AES-OTR and OCB. In the nonce-ignoring setting, these schemes are trivially insecure, as has been clearly stated by their respective authors. We consider OCB as an example, a similar attack can be performed on AES-OTR if nonces are reused. A nonce-ignoring adversary simply performs the following steps:

1. Choose (A, N, M) such that M consists of at least three blocks: $M = (M_1, M_2, \ldots)$, and ask for their authenticated ciphertext $(C_1, C_2, \ldots, T)$.
2. Choose $\Delta \neq 0^n$, and derive $M'_1 = M_1 \oplus \Delta$ and $M'_2 = M_2 \oplus \Delta$. For $M' = M'_1, M'_2$ and $M'_i = M_i$, for $i \geq 3$, ask for the authenticated ciphertext $(C'_1, C'_2, \ldots, T)$ that corresponds to (A, N, M').
3. Given the authenticated ciphertext (C'', T'') for any further message $(A,\ N,\ M'')$ with $M'' = (M_1, M_2, \ldots)$, the adversary can forge the ciphertext by replacing $(C''_1, C''_2) = (C_1, C_2)$ with (C'_1, C'_2).

Therefore, the complexities for j forgeries under nonce-ignoring adversaries are only t_1 (and not $t_1 + j$, see Table 12.1). Because of their structure, there exist nonce-respecting forgery attacks on AES-OTR and OCB that are stronger than our generic j-IV-CA (see Section 12.3).

AEZv4. Since AEZv4 does not separate the domains of (AD_i, N_i) for IV and tag generation, our j-IV-CAs work out-of-the box here. More detailed, nonce and associated data are parsed into a string $T_1, \ldots, T_t$ of n-bit strings T_i, and simply hashed in a PHASH-like manner inside AEZ-hash: $\Delta \leftarrow \bigoplus_{i=1}^{t} E_K^{i+2,1}(T_i)$, where E denotes a variant of four-round AES. The adversary can simply ask for the encryption of approximately 2^{64} tuples (A_i, N_i, M) for fixed M. Obtaining a collision for this hash (requiring birthday-bound complexity) can be easily detected when the message is kept constant over all queries. Given such a hash collision for (A_i, N_i) and (A_k, N_k), the adversary can directly construct subsequent forgeries by asking for the encryption of (A_i, N_i, M') and the same ciphertext will be valid for (A_k, N_k, M') for arbitrary M'.

Deoxys. Deoxys-I, i.e., the nonce-requiring variant, possesses a similar structure as OCB. Hence, there are trivial multi-forgery attacks with few queries if nonces repeat:

1. Choose (A, N, M) arbitrarily and ask for (C, T).
2. Choose $A' \neq A$, leave N and M constant, and ask for $C' = (C, T')$. Since the tag is computed by the XOR of Hash(A) with the encrypted checksum under the nonce as tweak, the adversary sees the difference in the hash outputs in the tags: $\text{Hash}(A) \oplus \text{Hash}(A') = T \oplus T'$.
3. Choose (A, N', M') and ask for (C'', T''). It instantly follows that for (A', N', M'), $(C'', T''' = T \oplus T' \oplus T'')$ will be valid.

However, in the nonce-respecting setting, the use of a real tweakable block cipher that employs the nonce in tweak (instead of the XEX construction as in AES-OTR and OCB) prevents the attacks in Section 12.3; the tag generation seems surprisingly strong in the sense that an adversary can not detect collisions between two associated data since the hash is XORed with an output of a fresh

Scheme	Type	Attack	Time	Data	Memory	Succ.
AES-OTRv3.1 par. ADP	NR	AUF	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$	1.0
AES-OTRv3.1 ser. ADP	NR	AUF	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$	1.0
AES-OTRv3.1 ser. ADP	NI	AUF	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$	1.0
COLMv1	NI	AUF	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$	1.0
OCBv3	NR	AUF	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$	1.0

Table 12.6: Stronger forgery attacks on NAE schemes. **NR/NI** = nonce-respecting/nonce-ignoring; **AUF** = almost universal forgery; **Succ.** = success probability; **ADP** = Associated Data Processing.

block cipher (because of the nonce is used as tweak) for every query. Therefore, we indicate that Deoxys-I provides resistance in the nonce-respecting setting.

Deoxys-II is a two-pass mode, i.e., the message is processed twice (1) once for the encryption process and (2) for the authentication process. In the nonce-ignoring setting, an adversary can simply fix N_i and vary AD_i for finding a collision for `Auth`, which renders the scheme vulnerable to j-IV-CAs. Therefore, that kind of two-pass scheme (in comparison to SIV, where the message is used as input to F_{IV}), does not implicitly provide resistance to j-IV-CAs.

NORX. The authors of NORX presented a nonce-misuse-resistant version of their scheme in Appendix D of [32]. NR-NORX follows the MAC-then-Encrypt paradigm, which yields a two-pass scheme similar to SIV. Therefore, NR-NORX provides at the least resistance to j-IV-CAs in the NR setting, which renders it stronger than NORX. However, this security comes at the cost of being off-line and two-pass.

CCM, EAX, CLOC, and SILC. The resistance to j-IV-CAs in the nonce-respecting setting provided by CCM, EAX, CLOC, and SILC stems from similar reasons as for Deoxys-II; the tag is generated by the XOR of the MAC of the nonce with the MAC of the ciphertext and the MAC of the associated data. Hence, collisions in ciphertext or header can not be easily detected since the MAC of a fresh nonce is XORed to it.

12.3 Stronger Forgery Attacks

This section summarizes existing attacks on third-round CAESAR candidates and classical AE schemes that yield multiple forgeries. This can be induced by the recovery of, e.g., a masking key or authentication key after a collision. While the complexity of the attacks are beyond the proved security bounds and therefore do not invalidate the according proofs, they can be considered as undesirable properties that should be avoided in recommendations by the community or even in future standards. See Table 12.6 for the results that are discussed in this section.

12.3.1 OCB

Ferguson [92] showed collision attacks on OCBv1, which allowed to recover the masking key L from a collision of the sums of input and output of two blocks $M_i \oplus C_i = M_j \oplus C_j$. Thereupon, the recovered L allows the construction of many selective forgeries out of a single long message. To address the length restriction of messages in OCBv1, Ferguson also derived attacks from collisions

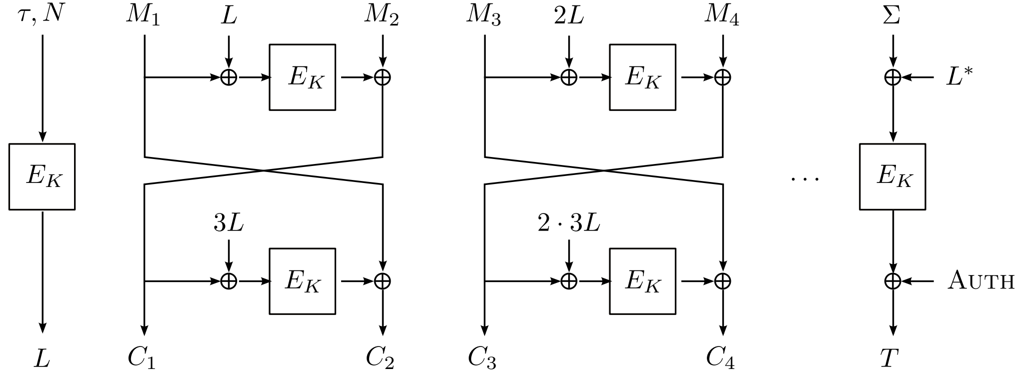

Figure 12.2: Simplified schematic illustration of the encryption process in AES-OTRv3.1 (parallel). The final two message blocks M_{m-1} and M_m are treated differently; $\Sigma = \bigoplus_{i=1}^{m/2} M_{2i}$, $L^* = 2^{m-1} \cdot L$, and AUTH denotes the result of processing the associated data.

among different messages, which also resulted in selective forgeries. The attacks by Ferguson still hold in similar form also for OCBv3, as pointed out by Sun, Wang, and Zhang [251]. In the following, we recall the details briefly.

For versions v1 and v3 of OCB, the designers used Gray codes for masking the block-cipher inputs. The mask for the i-th message/ciphertext block is given by $Z_i := \gamma_i \cdot L \oplus R$, where $L \leftarrow E_K(0^n)$, $R \leftarrow E_K(N \oplus L)$, and γ_i represents the integer i in the canonical Gray code. The multiplications are in $\mathbb{GF}(2^n)$ with primitive polynomial $\mathtt{x}^{128} + \mathtt{x}^7 + \mathtt{x}^2 + \mathtt{x} + \mathtt{1}$.

1. Choose A and N arbitrarily, and choose a long query $M = (M_1, \ldots, M_m)$, such that for all $k \in \{1, \lfloor m/4 \rfloor\}$, it holds that $M_{4k} \oplus M_{4k+1} \oplus M_{4k+2} \oplus M_{4k+3} = 0^n$. Ask for its encryption $C = (C_1, \ldots, C_m)$ and T.

2. If it holds, for any pair $i, j \in \{1, \ldots, m\}$, $i \neq j$, that $M_i \oplus C_i = M_j \oplus C_j$, then it holds with probability 0.5 that this collision is the result of colliding cipher inputs. Then, we can recover L by $L \leftarrow (M_i \oplus M_j) \cdot (\gamma_i \oplus \gamma_j)^{-1}$.

3. For any index d, change $C'_k \leftarrow C_d \oplus (\gamma_i \oplus \gamma_k) \cdot L$ for $k = 4, \ldots, 7$. Leave other ciphertext blocks, T, A, and N unchanged. The so-modified ciphertext C' is still valid and will yield $M'_4 \oplus M'_5 \oplus M'_6 \oplus M'_7 = 0^n$, which also held for the original message. Hence, the tag T remains valid also for the modified ciphertext.

12.3.2 AES-OTR

Similar collision attacks as for OCB can be applied to AES-OTR. As a reaction to Bost and Sanders' [234] polynomial attacks on the v2 version of AES-OTR, Minematsu updated the tweak usage in v3 of AES-OTR to use the masking key L from encrypting N. We describe two attacks on AES-OTR v3.1 with birthday-bound complexity of $2^{n/2}$ that recover L: an attack with a single, long message, and an attack with multiple messages.

Single-Message Attack. The first attack works as follows:

1. Choose A and N arbitrarily, and choose a long query $M = (M_1, \ldots, M_m)$ for even m such that all even blocks are equal, i.e. $M_2 = M_4 = \ldots = M_{2i}$ for all $i \in \{1, \ldots, m/2 - 1\}$, and

random pair-wise distinct odd blocks M_{2i-1}. For simplicity, choose arbitrary full blocks M_{m-1} and M_m. Ask for its encryption $C = (C_1, \ldots, C_m)$ and T.

2. If for any pair $i, j \in \{1, \ldots, m/2-1\}$, it holds that $C_{2i-1} = C_{2j-1}$, it follows from $M_{2i} = M_{2j}$ and from the fact that $E_K(\cdot)$ is a permutation that

$$M_{2i-1} \oplus 2^{i-1}L = M_{2j-1} \oplus 2^{j-1}L, \text{ and hence,}$$
$$L = (M_{2i-1} \oplus M_{2j-1}) \cdot (2^{i-1} \oplus 2^{j-1})^{-1}.$$

3. For any pair of indices $x, y \in \{1, \ldots, m/2-1\}$, derive $\Delta \leftarrow 3 \cdot (2^{x-1} \oplus 2^{y-1}) \cdot L$ and $\nabla \leftarrow (2^{x-1} \oplus 2^{y-1}) \cdot L$, and compute

$$\begin{aligned} C'_{2x-1} &\leftarrow C_{2y-1} \oplus \Delta, & C'_{2x} &\leftarrow C_{2y} \oplus \nabla, \\ C'_{2y-1} &\leftarrow C_{2x-1} \oplus \Delta, \quad \text{and} & C'_{2y} &\leftarrow C_{2x} \oplus \nabla. \end{aligned}$$

Leave other ciphertext blocks, T, A, and N unchanged. The so-modified ciphertext C' is still valid and will yield $M'_{2x-1} = M_{2y-1} \oplus \nabla$, $M'_{2x} = M_{2y} \oplus \Delta$, $M'_{2y-1} = M_{2x-1} \oplus \nabla$, and $M'_{2y} = M_{2x} \oplus \Delta$.

Since the tag generation uses the sum of the even-indexed message blocks,

$$\Sigma = \bigoplus_{i=1}^{m/2} M_{2i},$$

it holds that $\Sigma' = \Sigma$ since $M'_{2x} \oplus M'_{2y} = M_{2y} \oplus M_{2x}$.

Multi-Message Nonce-Respecting Attack. A variant of Ferguson's collision attack with multiple messages on OCB may also be possible for AES-OTR; however, it would allow to recover the relation of $\Delta = 2^{i-1}L \oplus 2^{j-1}L$. For OCB, Ferguson could inject differences of $(4 \oplus 5 \oplus 6 \oplus 7)\Delta$ which cancel out in $\mathbb{GF}(2^n)$. Since AES-OTR employs doublings instead, this would mean, one would obtain $(2^i \oplus 2^j \oplus 2^k \ldots)\Delta$ when swapping double-blocks from between the colliding messages. Thus, Ferguson's collision attack seems not directly applicable to AES-OTR; at least, we could not find a straight-forward way to cancel values. However, we found two attacks with collision among different messages for the serial-ADP version of AES-OTR.

For the serial version, we can derive a multi-message nonce-respecting collision attack. For this version, the masking key is computed from $L \leftarrow (E_K(\tau, N) \oplus \textsc{Auth}) \cdot 2$. Leaving the (assumed constant) parameter τ aside, it is easy to see that two values of L can collide at birthday bound.

1. Choose an integer $m \geq 4$, and fix arbitrary values $M, M_{m-1} \in \{0,1\}^n$.

2. For $i = 1..q$, choose pair-wise distinct random pairs of associated data and nonce (A^i, N^i) such that the nonces N^i are all distinct. Choose M_{2j-1}, for $j = 1, \ldots, m/2-1$ randomly. Ask for the authenticated encryption of (A^i, N^i, M^i), with

$$M^i = (M_1^i, M, M_3^i, M, \ldots, M_{m-1}, M)$$

and store (A^i, N^i, C^i, T^i) as well as the odd-indexed blocks of M^i in a table $\mathcal{L}$.

3. If, for any indices $i \neq j$, it holds that $C_m^i = C_m^j$, then it must follow from $M_{m-1}^i = M_{m-1}^j$ and $M_m^i = M_m^j$ that the masking keys for both messages L^i and L^j must be identical. It follows furthermore from the tag generation of AES-OTR that the tags of both messages M^i and M^j are identical.

4. Denote $t \leftarrow (m/2)-1$. Leaving the final double-block (M^i_{m-1}, M^i_m) aside that served for detecting the collision of the masking keys, we have t double blocks (C^i_{2k-i}, C^i_{2k}) and (C^j_{2k-i}, C^j_{2k}), for $k = 1, \ldots, m/2-1$, in our two involved ciphertexts that can be swapped across messages. Since both yield $M^i_{2k} = M^j_{2k} = M$ after decryption, the swaps do not change the tag. Assuming $M^i_{2k-1} \neq M^j_{2k-1}$ for all $k = 1, \ldots, m/2-1$, which holds with high probability, we can create in total $2 \cdot 2^t$ different authenticated ciphertexts by exhaustive combination of double blocks with either (A^i, N^i) or (A^j, N^j). Note that we already used two of those combinations for finding the collision.

Multi-Message Nonce-Ignoring Attack. Lu [172] published almost-universal forgery attacks on AES-COPA, and Marble. They can be also translated into nonce-ignoring attacks on AES-OTRv3.1. We consider the version with serial ADP since the specification claims that its "*security [...] holds as far as a pair of AD and nonce* (A, N) *is unique for all encryption queries, for privacy and authenticity notions*". Lu proposed attacks with constant associated data, requiring at best about 2^{124} queries and time, at about $2^{120.6}$ bytes and a success probability of approximately 0.32. Moreover, he proposed an attack with 2^{65} queries and time. Both recover the masking key L. We transform the latter with birthday-bound complexity to an attack on AES-OTR that also recovers L. The attack works as follows:

1. Fix any message M and nonce N.
2. For $i = 1, \ldots, 2^{n/2}$, choose a single-block associated data A^i of length $< n$ bits, s. t. all A^i are pair-wise distinct. Ask for the encryption of (A^i, N, M) to C^i, T and store them into a table.
3. For $j = 1, \ldots 2^{n/2}$, choose a single-block associated data data A'^j of length n bits. If there exist i, j with $A^i = A'^j$, then, we can recover the associated-data masking key $Q \leftarrow E_K(0^n)$ from

$$A^i \oplus 2Q \leftarrow A'^j \oplus 4Q \quad \text{and thus} \quad Q \leftarrow (A^i \oplus A'^j) \cdot (2 \oplus 4)^{-1}.$$

   Though, it suffices to compute $A^i \oplus A'^j \leftarrow 2Q \oplus 4Q$.
4. In the following, for each of the $2^{n/2}$ stored tuples (A^i, N, C^i, T^i) with partial A^i, derive the padded n-bit value $A'^i \leftarrow (A^i \parallel 10^*) \oplus (2Q \oplus 4Q)$. All ciphertexts (A'^i, N, C^i, T^i) are valid forgeries.

12.3.3 COLM

The multi-message nonce-ignoring attack by Lu can also be applied in similar form to COLMv1. Here, we can recover first the masking key L, which is also used for encryption and tag generation. Using the notation from the attack description on AES-OTR, it holds for COLMv1 that we obtain

$$A^i \oplus 3 \cdot 2 \cdot 7 \cdot L = A'^i \oplus 3 \cdot 2 \cdot L \quad \text{and thus} \quad L = (A^i \oplus A'^j) \cdot 7^{-1}.$$

For each of the $2^{n/2}$ stored tuples (A^i, N, C^i, T^i) with partial A^i, derive the padded n-bit value $A'^i \leftarrow (A^i \parallel 10^*) \oplus (3 \cdot 2 \cdot 7 \cdot L \oplus 3 \cdot 2 \cdot L)$. Again, all ciphertexts (A'^i, N, C^i, T^i) are valid forgeries. However, the knowledge of L allows almost universal forgeries.

There are various ways to obtain a vast amount of more forgeries. For instance, choose some N', M' and a long associated data $A' = (A_1, \ldots, A_a)$, with $A_1 = A_2 = \ldots = A_a$, such that (N', A', M') has not been queried before. Ask for its corresponding encryption (C', T'). From $A_1 = A_2 = \ldots = A_a$

follows that the masked inputs to the block cipher, $AA_i \leftarrow A_i \oplus 2^i \cdot 3 \cdot L$, are pair-wise distinct. Since we know L, we can modify the blocks A'_i to obtain a permutation of the a values AA_i. Since there exist $a!$ such permutations, we obtain $a! - 1$ forgeries from a single encryption query.

12.4 Countermeasures to j-IV-Collision Attacks

This section briefly describes two possible approaches for providing resistance to j-IV-CAs in the nonce-respecting (NR) as well as in the nonce-ignoring (NI) setting.

Independence of F_{IV} and F_T. For realizing that approach, the pair (AD_i, N_i) has to be processed twice. Let $F_{IV}(AD_i, N_i, *)$ be the IV-generation step of an NAE scheme processing the tuple $(AD_i, N_i, *)$, where '$*$' denotes that F_{IV} can optionally process the message M. Usually, it is proven that F_{IV} behaves like a PRF. Further, let $F_T(*, *, *, AD_i, N_i)$ be the tag-generation step of an AE scheme processing the tuple $(*, *, *, AD_i, N_i)$, where the first three inputs can be the chaining value CV, the message M, and/or the ciphertext C[3], and there exists a proof showing that F_T also behaves like a PRF. Hence, the corresponding scheme would have the class $(11{*}{*}{*}{*}11)$ which belongs to $\mathcal{C}_1$. If one can guarantee independence between F_{IV} and F_T, we can say that the outputs of $F_{IV}(AD_i, N_i, *)$ and $F_T(*, *, *, AD_i, N_i)$ are independent random values. Based on this assumption, a simple collision of the form $F_{IV}(AD_i, N_i, *) = F_{IV}(AD_k, N_k, *)$ (as required by the j-IV-CA) does not suffice to produce a forgery since it is highly likely that $F_T(AD_i, N_i, *) \neq F_T(*, *, *, AD_k, N_k)$ and vice versa. Therefore, this *two-pass processing* realizes a *domain separation* between the IV-generation and the tag-generation step, providing resistance to j-IV-CAs. One way to achieve that goal can be to invoke the same PRF twice (for F_{IV} and F_T) but always guarantee distinct inputs, e.g., $F_{IV}(AD_i, N_i, *, 1)$ and $F_T(*, *, *, AD_i, N_i, 2)$. Another approach would be to use two independent functions.

Wide-State IV. A second approach requires a PRF-processing of the associated data F_{IV} which produces a wide-state output $\tau \leftarrow F_{IV}(AD_i, N_i)$ with $|\tau| > n$ bit. For example, for $|\tau| = 2n$, a pair (AD_i, N_i) would be processed to two independent n-bit values τ_1 and τ_2. Then, one could use τ_1 as initialization vector to the encryption step and τ_2 as initialization vector to the tag-generation step. Therefore, one can always guarantee domain separation between encryption and tag generation, while remaining a one-pass AE scheme. One possible instantiation for such a MAC (which can be utilized for the processing of the associated data) is PMAC2x [171].

12.5 Summary

In this chapter, we pursued on the idea of multi-forgery attacks first described by Ferguson in 2002 and went on with introducing the notion j-INT-CTXT. We introduced a classification of NAE schemes depending of the usage of their inputs to the initialization, encryption, and authentication process, and categorize them regarding to that classification. To allow a systematic analysis of the reforgeability of NAE schemes, we introduced the j-IV-Collision Attack, providing us with expected upper bounds on the hardness of further forgeries. During our analysis, we made the following observations:

[3]Note that at least one of the three inputs must be given since otherwise, the tag would be independent from the message, which would make the scheme trivially insecure.

- None of the considered NAE schemes provides full resistance to j-IV-CAs in the nonce-ignoring as well as in the nonce-respecting setting.
- ACORN, AES-OTR (serial), Ascon, COLM, JAMBU, Ketje, and NORX belong to the class $\mathcal{C}_0$, rendering them implicitly vulnerable or at most semi-resistant to j-IV-CAs.
- AEGIS, Ascon, Ketje, Keyak, MORUS, NORX, NR-NORX, and Tiaoxin are *semi-resistant* to j-IV-CAs in the nonce-respecting setting since all of them employ a wide state.
- NR-NORX is also *semi-resistant* to j-IV-CAs in the nonce-ignoring setting.
- Full resistance in the nonce-respecting setting is only satisfied by CCM, CLOC, Deoxys-I, Deoxys-II, EAX, NR-NORX, and SILC.

Note that the wide-state property has no impact on the applicability of a j-IV-CA itself, but it significantly increases the necessary computational effort for the internal collision. [4] Finally, we briefly proposed two alternative approaches which would render an NAE scheme resistant to j-IV-CAs in the nonce-respecting as well as the nonce-ignoring setting.

[4] For example, if the internal state is of size $2n$ (wide state) instead of n, a generic collision can be found in 2^n instead of $2^{n/2}$.

Part V

Epilog

CHAPTER 13

Conclusion

The scientist is not a person who gives the right answers, he's one who asks the right questions.

CLAUDE LÉVI-STRAUSS

In this chapter, we first summarize the achievements obtained during this thesis and second, discuss possible future works.

13.1 Summary

Block-Cipher-Based Compression Functions. Block-cipher-based compression functions were brought forward first by Rabin [220] back in the 1970s and experienced a strong development of single- and double-block-length compression functions, e.g., [217, 232, 247, 185, 96, 133]. In Part II, we introduced COUNTER-bDM, the first provably secure family of MBL compression functions, which, in contrast to the above, allows arbitrary large outputs. COUNTER-bDM consists of a simple and neat design and is accompanied by a thorough security analysis. Furthermore, to avoid multiple calls to the key schedule of the underlying block cipher, we realized the demand for independent and distinct block-cipher calls within one invocation of the compression function by always guaranteeing distinct plaintext inputs.

Password Hashing. Our work in the field of password hashing in Part III was triggered by showing that the first modern Password Scrambler (PS) `scrypt` [211] is vulnerable to three types of Side-Channel Attacks (SCAs), i.e., Cache-Timing Attacks (CTAs), Garbage-Collector Attacks (GCAs), and Weak Garbage-Collector Attacks (WGCAs), where the two latter were introduced by us. Thereupon, we came up with a list of (functional, security, and general) properties that should be satisfied by modern password scramblers, e.g., memory hardness, simplicity, SR, CIUs, and resistance to SCAs The first design fulfilling all the required properties was CATENA, which we introduced in 2013 and which had a high impact on the requirements stated by the Password Hashing Competition (PHC) [31]. CATENA describes a highly flexible Password-Scrambling Framework for which we provide several instances suitable for a variety of applications. We provided a thorough

security analysis of Catena – including, e.g., preimage security, PRF security, and resistance to SCAs (for some instances) – ensuring that Catena can be used without any concerns.

Finally, we compared all submissions to the PHC that were not withdrawn (and `scrypt`) by the means of the properties mentioned above, and accompanied that comparison by a discussion about the resistance to GCAs and WGCAs.

Authenticated Encryption. In Part IV, we focused on the integrity of Nonce-Based Authenticated Encryptions (NAEs). More detailed, we considered the third-round submissions to the CAESAR competition as well as classical AE schemes (CWC [158], CCM [88], EAX [42], and GCM [180]) in terms of j-INT-CTXT security – a notion which we introduced during our work. It refers to an adversary attempting to obtain j forgeries based on a single given forgery. For a systematic analysis, we first provided a generic AE scheme and introduced a classification of the considered NAE schemes accordingly. Next, we introduced the j-IV-Collision Attack (j-IV-CA) providing us with expected upper bounds regarding to the notion j-INT-CTXT. We have done our analysis in the nonce-ignoring as well as in the nonce-respecting setting and we conducted that in the former, none of the considered NAE schemes provide full resistance, whereas some of them provide semi-resistance based on a wide state. Finally, we outlined two approaches for designing an AE scheme that would provide resistance to j-IV-CAs.

13.2 Future Work

Even though we proposed a block-cipher-based compression function allowing for arbitrary large outputs, there is still room left for improvement:

- *Is it possible to provide a provably secure MBL hash function with $b \cdot n$ bits output size that is not tied to a block cipher from $\texttt{Block}(bn, n)$?*

The research of password hashing has significantly evolved during the last years. Apart from key stretching, and memory hardness, Side-Channel Attacks (SCAs) have come to the fore and motivated many well-designed modern password scramblers. Nevertheless, there still exist many tasks and questions that arose during our work on Catena:

- *A verification of our theoretical Cache-Timing Attacks (CTAs), Garbage-Collector Attacks (GCAs), and Weak Garbage-Collector Attacks (WGCAs) on `scrypt` and other memory-hard password scramblers would require a practical implementation.*

- *Besides GCAs, WGCAs, TMTOs, and SCAs, are there other possible attack vectors on modern password scramblers?*

- *In [170], it is shown that a DBG (from which we use a slight adapted version in our instances* Catena-Butterfly *and* Catena-Mydasfly*) is a superconcentrator. Moreover, it is the basis for the DBG^g_λ as used in our instances* Catena-Butterfly *and* Catena-Mydasfly*, providing λ-Memory Hardness (LMH). Since the depth of a DBG^g_λ depends on the parameter g (memory), we are interested whether there exist superconcentrators with a fixed depth, thus, providing LMH while maintaining an efficient and scalable graph-based PS.*

- *Is it possible to design a sequential memory-hard password scrambler that also satisfies resistance to Cache-Timing Attacks (CTAs)?*

- *Regarding the precomputation attack [55] on different instances of F_λ: could an adversary obtain a significant lower penalty when considering an optimal distribution of all $\lambda \cdot 2^{2g/3}$ sampling points over the whole graph.*
- *Do TMTO attacks similar to that of Biryukov and Khovratovich [55] exist that are optimized for other instances of F_λ, namely $GRG\ell_\lambda^g$?*

With our work on reforgeability of NAE schemes, we introduced a new notion of security when considering the integrity of such schemes. We also presented two possible approaches that provide resistance to j-IV-CAs as introduced in Part IV.

- *Is it possible to design a provable secure NAE scheme that provides resistance to j-IV-CAs in both the nonce-ignoring as well as the nonce-respecting setting?.*

We appeal, as human beings, to human beings: Remember your humanity, and forget the rest. If you can do so, the way lies open to a new Paradise; if you cannot, there lies before you the risk of universal death.

RUSSEL-EINSTEIN MANIFESTO

Bibliography

[1] Martin Abadi, Michael Burrows, and Ted Wobber.
Moderately Hard, Memory-Bound Functions.
In *NDSS*, 2003.
(One citation on page 62.)

[2] Mohamed Ahmed Abdelraheem, Peter Beelen, Andrey Bogdanov, and Elmar Tischhauser.
Twisted Polynomials and Forgery Attacks on GCM.
In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT I*, volume 9056 of *Lecture Notes in Computer Science*, pages 762–786, 2015.
(Citations on pages 129 and 130.)

[3] Farzaneh Abed, Scott Fluhrer, John Foley, Christian Forler, Eik List, Stefan Lucks, David McGrew, and Jakob Wenzel.
The POET Family of On-Line Authenticated Encryption Schemes.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2014.
(One citation on page 124.)

[4] Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and Jakob Wenzel.
Pipelineable On-Line Encryption.
IACR Cryptology ePrint Archive, 2014:297, 2014.
(Citations on pages 4 and 5.)

[5] Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and Jakob Wenzel.
Pipelineable On-line Encryption.
In *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, pages 205–223, 2014.
(One citation on page 5.)

[6] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Biclique Cryptanalysis of the PRESENT and LED Lightweight Ciphers.
IACR Cryptology ePrint Archive, 2012:591, 2012.
(One citation on page 6.)

[7] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
A Framework for Automated Independent-Biclique Cryptanalysis.
In Shiho Moriai, editor, *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 561–581. Springer, 2013.
(One citation on page 6.)

[8] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Counter-bDM: A Provably Secure Family of Multi-Block-Length Compression Functions.

In David Pointcheval and Damien Vergnaud, editors, *AFRICACRYPT*, volume 8469 of *Lecture Notes in Computer Science*, pages 440–458. Springer, 2014.
(Citations on pages 4 and 6.)

[9] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
RIV for Robust Authenticated Encryption.
In Thomas Peyrin, editor, *FSE*, volume 9783 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2016.
(One citation on page 5.)

[10] Farzaneh Abed, Christian Forler, David McGrew, Eik List, Scott Fluhrer, Stefan Lucks, and Jakob Wenzel.
Pipelineable On-line Encryption.
In Carlos Cid and Christian Rechberger, editors, *FSE*, volume 8540 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2014.
(One citation on page 5.)

[11] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel.
Cryptanalysis of the Speck Family of Block Ciphers.
IACR Cryptology ePrint Archive, 2013:568, 2013.
(One citation on page 6.)

[12] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel.
Differential Cryptanalysis of Reduced-Round Simon.
IACR Cryptology ePrint Archive, 2013:526, 2013.
(One citation on page 6.)

[13] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel.
Differential Cryptanalysis of Round-Reduced Simon and Speck.
In Carlos Cid and Christian Rechberger, editors, *FSE*, volume 8540 of *Lecture Notes in Computer Science*, pages 525–545. Springer, 2014.
(Citations on pages 5 and 6.)

[14] Onur Aciiçmez.
Yet another MicroArchitectural Attack: : exploiting I-Cache.
In *Proceedings of the 2007 ACM workshop on Computer Security Architecture, CSAW 2007, Fairfax, VA, USA, November 2, 2007*, pages 11–18, 2007.
(One citation on page 49.)

[15] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher.
New Results on Instruction Cache Attacks.
In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 110–124, 2010.
(One citation on page 49.)

[16] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert.
On the Power of Simple Branch Prediction Analysis.
IACR Cryptology ePrint Archive, 2006:351, 2006.
(One citation on page 49.)

[17] Onur Aciiçmez and Jean-Pierre Seifert.
Cheap Hardware Parallelism Implies Cheap Security.
In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 80–91, 2007.

(One citation on page 49.)

[18] Rafael Alvarez.
CENTRIFUGE – A password hashing algorithm.
`https://password-hashing.net/submissions/specs/Centrifuge-v0.pdf`, 2014.
(One citation on page 107.)

[19] Joël Alwen and Jeremiah Blocki.
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*, pages 241–271, 2016.
(Citations on pages 115 and 116.)

[20] Joël Alwen and Jeremiah Blocki.
Towards practical attacks on argon2i and balloon hashing.
IACR Cryptology ePrint Archive, 2016:759, 2016.
(Citations on pages 115 and 116.)

[21] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak.
Depth-Robust Graphs and Their Cumulative Memory Complexity.
In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, pages 3–32, 2017.
(One citation on page 115.)

[22] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro.
On the Complexity of Scrypt and Proofs of Space in the Parallel Random Oracle Model.
In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 358–387, 2016.
(One citation on page 116.)

[23] Joël Alwen and Vladimir Serbinenko.
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*, pages 595–603, 2015.
(Citations on pages 65, 81, 103, 115, and 117.)

[24] Joël Alwen, Peter GaV zi, Chethan Kamath, Karen Klein, Georg Osang, Krzysztof Pietrzak, Leonid Reyzin, Michal Rolínek, and Michal Rybár.
On the Memory-Hardness of Data-Independent Password-Hashing Functions.
Cryptology ePrint Archive, Report 2016/783, 2016.
`http://eprint.iacr.org/2016/783`.
(One citation on page 115.)

[25] Elena Andreeva, Andrey Bogdanov, Nilanjan Datta, Atul Luykx, Bart Mennink, Mridul Nandi, Elmar Tischhauser, and Kan Yasuda.
COLM v1.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(Citations on pages 122 and 123.)

[26] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda.
AES-COPA.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2014.
(One citation on page 122.)

[27] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang.
Preimages for Step-Reduced SHA-2.
In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, pages 578–597, 2009.
(One citation on page 89.)

[28] Frederik Armknecht, Ewan Fleischmann, Matthias Krause, Jooyoung Lee, Martijn Stam, and John P. Steinberger.
The Preimage Security of Double-Block-Length Compression Functions.
In *ASIACRYPT*, pages 233–251, 2011.
(Citations on pages 29, 30, 34, and 40.)

[29] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi.
Proofs of Space: When Space is of the Essence.
IACR Cryptology ePrint Archive, 2013:805, 2013.
(One citation on page 62.)

[30] Jeff Atwood.
Speed Hashing.
`https://blog.codinghorror.com/speed-hashing/`.
Accessed April 27, 2017.
(One citation on page 44.)

[31] Jean-Philippe Aumasson.
Password Hashing Competition.
`https://password-hashing.net/call.html`.
Accessed September 3, 2015.
(Citations on pages 3, 47, and 139.)

[32] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves.
NORX.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(Citations on pages 123 and 131.)

[33] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein.
BLAKE2: Simpler, Smaller, Fast as MD5.
In *ACNS*, pages 119–135, 2013.
(Citations on pages 65, 90, and 91.)

[34] Anne Barsuhn.
Cache-Timing Attack on scrypt.
Bauhaus-Universität Weimar, Bachelor Dissertation, December 2013.
(One citation on page 51.)

[35] Mihir Bellare, Alexandra Boldyreva, and Adriana Palacio.
An Uninstantiable Random-Oracle-Model Scheme for a Hybrid-Encryption Problem.

In *EUROCRYPT*, pages 171–188, 2004.
(One citation on page 16.)

[36] Mihir Bellare, Ran Canetti, and Hugo Krawczyk.
Keying Hash Functions for Message Authentication.
In *CRYPTO*, pages 1–15, 1996.
(One citation on page 121.)

[37] Mihir Bellare, Oded Goldreich, and Anton Mityagin.
The Power of Verification Queries in Message Authentication and Authenticated Encryption.
IACR Cryptology ePrint Archive, 2004:309, 2004.
(One citation on page 25.)

[38] Mihir Bellare, Joe Kilian, and Phillip Rogaway.
The Security of the Cipher Block Chaining Message Authentication Code.
J. Comput. Syst. Sci., 61(3):362–399, 2000.
(One citation on page 121.)

[39] Mihir Bellare and Chanathip Namprempre.
Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm.
In *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.
(Citations on pages 3 and 24.)

[40] Mihir Bellare and Phillip Rogaway.
Random Oracles are Practical: A Paradigm for Designing Efficient Protocols.
In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.
(One citation on page 15.)

[41] Mihir Bellare and Phillip Rogaway.
Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography.
In Tatsuaki Okamoto, editor, *ASIACRYPT*, volume 1976 of *LNCS*, pages 317–330. Springer, 2000.
(One citation on page 24.)

[42] Mihir Bellare, Phillip Rogaway, and David Wagner.
The EAX Mode of Operation.
In *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004.
(Citations on pages 3, 122, 123, and 140.)

[43] S.M. Bellovin and M. Merrit.
Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks.
Proceedings of the I.E.E.E. Symposium on Research in Security and Privacy (Oakland), 1992.
(One citation on page 3.)

[44] Dan J. Bernstein.
CAESAR Call for Submissions, Final, January 27 2014.
`http://competitions.cr.yp.to/caesar-call.html`.
(Citations on pages 3 and 122.)

[45] Daniel J. Bernstein.
Cache-timing attacks on AES, 2005.

(Citations on pages 3 and 20.)

[46] Daniel J. Bernstein.
The Salsa20 Family of Stream Ciphers.
In *The eSTREAM Finalists*, pages 84–97. Springer, 2008.
(One citation on page 46.)

[47] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche.
Sponge Functions.
Ecrypt Hash Workshop 2007, May 2007.
(One citation on page 2.)

[48] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche.
The Keccak SHA-3 submission.
Submission to NIST (Round 3), 2011.
(One citation on page 90.)

[49] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche.
Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications.
In *Selected Areas in Cryptography*, pages 320–337, 2011.
(One citation on page 109.)

[50] Guido Bertoni, Joan Daemen, Michaël Peeters, and Ronny Van Keer Gilles Van Assche.
CAESAR submission; Ketje v2.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[51] Eli Biham and Orr Dunkelman.
The SHAvite-3 Hash Function.
Submission to NIST (Round 2), 2009.
(One citation on page 2.)

[52] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich.
Argon2.
Password Hashing Competition, Winner, 2015.
`https://www.cryptolux.org/index.php/Argon2`.
(Citations on pages 3, 19, 79, 89, 92, 93, 99, and 100.)

[53] Alex Biryukov and Dmitry Khovratovich.
Tradeoff cryptanalysis of Catena.
PHC mailing list: discussions@password-hashing.net.
(One citation on page 79.)

[54] Alex Biryukov and Dmitry Khovratovich.
ARGON and Argon2: Password Hashing Scheme.
`https://password-hashing.net/submissions/specs/Argon-v2.pdf`, 2015.
(One citation on page 107.)

[55] Alex Biryukov and Dmitry Khovratovich.
Tradeoff Cryptanalysis of Memory-Hard Functions.
IACR Cryptology ePrint Archive, 2015:227, 2015.
(Citations on pages v, 3, 73, 74, 78, 79, 80, 81, 82, 83, 99, and 141.)

[56] John Black.
The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function.

In *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, pages 328–340, 2006.
(One citation on page 10.)

[57] John Black and Martin Cochran.
MAC Reforgeability.
In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 345–362. Springer, 2009.
(One citation on page 121.)

[58] John Black, Martin Cochran, and Thomas Shrimpton.
On the Impossibility of Highly-Efficient Blockcipher-Based Hash Functions.
In *EUROCRYPT*, pages 526–541, 2005.
(One citation on page 30.)

[59] John Black, Phillip Rogaway, and Thomas Shrimpton.
Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV.
Cryptology ePrint Archive, Report 2002/066, 2002.
`http://eprint.iacr.org/`.
(One citation on page 10.)

[60] John Black, Phillip Rogaway, and Thomas Shrimpton.
Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV.
In *CRYPTO*, pages 320–335, 2002.
(One citation on page 30.)

[61] Matt Blaze.
Efficient Symmetric-Key Ciphers Based on an NP-Complete Subproblem, 1996.
(One citation on page 112.)

[62] Joseph Bonneau.
The science of guessing: analyzing an anonymized corpus of 70 million passwords.
In *2012 IEEE Symposium on Security and Privacy*, May 2012.
(Citations on pages 45 and 46.)

[63] Xavier Boyen.
Halting Password Puzzles – Hard-to-break Encryption from Human-memorable Keys.
In *16th USENIX Security Symposium—SECURITY 2007*, pages 119–134. Berkeley: The USENIX Association, 2007.
Available at `http://www.cs.stanford.edu/~xb/security07/`.
(One citation on page 44.)

[64] William F. Bradley.
Superconcentration on a Pair of Butterflies.
CoRR, abs/1401.7263, 2014.
(One citation on page 75.)

[65] Tom Caddy.
FIPS 140-2.
In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005.
(One citation on page 61.)

[66] Justin Cappos.

PolyPassHash: Protecting Passwords In The Event Of A Password File Disclosure.
https://password-hashing.net/submissions/specs/PolyPassHash-v0.pdf, 2014.
(One citation on page 111.)

[67] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya.
Rig: A simple, secure and flexible design for Password Hashing.
https://password-hashing.net/submissions/specs/RIG-v2.pdf, 2014.
(One citation on page 112.)

[68] Donghoon Chang, Mridul Nandi, Jesang Lee, Jaechul Sung, Seokhie Hong, Jongin Lim, Haeryong Park, and Kilsoo Chun.
Compression Function Design Principles Supporting Variable Output Lengths from a Single Small Function.
IEICE Transactions, 91-A(9):2607–2614, 2008.
(One citation on page 31.)

[69] Codenomicon.
The Heartbleed Bug.
http://heartbleed.com/, 2014.
(One citation on page 49.)

[70] Stephen A. Cook and Ravi Sethi.
Storage Requirements for Deterministic Polynomial Time Recognizable Languages.
In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 33–39, 1974.
(One citation on page 80.)

[71] J. W. Cooley and J. W. Tukey.
An Algorithm for the Machine Calculation of Complex Fourier Series.
Math. Comput., 19:297–301, 1965.
(One citation on page 76.)

[72] D. Coppersmith, S. Pilpel, C.H. Meyer, S.M. Matyas, M.M. Hyden, J. Oseas, B. Brachtl, and M. Schilling.
Data Authentication Using Modification Dectection Codes Based on a Public One-Way Encryption Function.
U.S. Patent No. 4,908,861, March, 13. 1990.
(One citation on page 30.)

[73] (C) Intel Corporation.
7th Generation Intel (R) Processor Families for S Platforms.
http://www.intel.com/content/www/us/en/processors/core/7th-gen-core-family-desktop-s-processor-lines-datasheet-vol-1.html.
Accessed April 25, 2017.
(Citations on pages 47 and 89.)

[74] Bill Cox.
TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme.
https://password-hashing.net/submissions/specs/TwoCats-v0.pdf, 2014.
(One citation on page 113.)

[75] Sophie Curtis.

Sony saved thousands of passwords in a folder named 'Password'.
`http://www.telegraph.co.uk/technology/sony/11274727/Sony-saved-thousands-of-passwords-in-a-folder-named-Password.html`.
Accessed: 2017-03-16.
(One citation on page 2.)

[76] Joan Daemen and Vincent Rijmen.
The Design of Rijndael: AES - The Advanced Encryption Standard.
Springer, 2002.
(One citation on page 29.)

[77] Nilanjan Datta and Mridul Nandi.
ELmD.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2014.
(One citation on page 122.)

[78] Des.
Data Encryption Standard.
In *In FIPS PUB 46, Federal Information Processing Standards Publication*, 1977.
(One citation on page 43.)

[79] Solar Designer.
Enhanced challenge/response authentication algorithms.
`http://openwall.info/wiki/people/solar/algorithms/challenge-response-authentication`.
Accessed January 22, 2014.
(One citation on page 19.)

[80] Solar Designer.
New developments in password hashing: ROM-port-hard functions.
`http://distro.ibiblio.org/openwall/presentations/New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf`, 2012.
(Citations on pages 108 and 109.)

[81] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer.
Ascon v1.2.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[82] Ulrich Drepper.
Unix crypt using SHA-256 and SHA-512.
`http://www.akkadia.org/drepper/SHA-crypt.txt`.
Accessed April 27, 2017.
(Citations on pages 44, 45, and 89.)

[83] Markus Dürmuth and Ralf Zimmermann.
AntCrypt.
`https://password-hashing.net/submissions/AntCrypt-v0.pdf`, 2014.
(One citation on page 107.)

[84] Cynthia Dwork, Andrew Goldberg, and Moni Naor.
On Memory-Bound Functions for Fighting Spam.
In *CRYPTO*, pages 426–444, 2003.
(One citation on page 62.)

[85] Cynthia Dwork and Moni Naor.
Pricing via Processing or Combatting Junk Mail.
In *CRYPTO*, pages 139–147, 1992.
(One citation on page 62.)

[86] Cynthia Dwork, Moni Naor, and Hoeteck Wee.
Pebbling and Proofs of Work.
In *CRYPTO*, pages 37–54, 2005.
(One citation on page 62.)

[87] Morris Dworkin.
Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation.
National Institute of Standards, U.S. Department of Commerce, December 2001.
(One citation on page 64.)

[88] Morris J. Dworkin.
SP 800-38C. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality.
Technical report, Gaithersburg, MD, United States, 2004.
(Citations on pages 122, 123, and 140.)

[89] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak.
Proofs of Space.
IACR Cryptology ePrint Archive, 2013:796, 2013.
(One citation on page 62.)

[90] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs.
Key-Evolution Schemes Resilient to Space-Bounded Leakage.
In *CRYPTO*, pages 335–353, 2011.
(One citation on page 79.)

[91] Brandon Enright.
Omega Crypt (ocrypt).
`https://password-hashing.net/submissions/specs/OmegaCrypt-v0.pdf`, 2014.
(One citation on page 111.)

[92] Niels Ferguson.
Collision Attacks on OCB.
unpublished manuscript. Available online, 2002.
`http://web.cs.ucdavis.edu/~rogaway/ocb/fe02.pdf`.
(Citations on pages 121, 123, and 131.)

[93] Niels Ferguson.
Authentication weaknesses in GCM, 2005.
(One citation on page 122.)

[94] Ewan Fleischmann.
Analysis and Design of Blockcipher Based Cryptographic Algorithms.
PhD thesis, Bauhaus-Universität Weimar, 2013.
(One citation on page 30.)

[95] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks.
Collision-Resistant Double-Length Hashing.
In *ProvSec*, pages 102–118, 2010.

(One citation on page 31.)

[96] Ewan Fleischmann, Christian Forler, and Stefan Lucks.
The Collision Security of MDC-4.
In *AFRICACRYPT*, pages 252–269, 2012.
(Citations on pages 2, 31, and 139.)

[97] Ewan Fleischmann, Christian Forler, Stefan Lucks, and Jakob Wenzel.
McOE: A Foolproof On-Line Authenticated Encryption Scheme.
Cryptology ePrint Archive, Report 2011/644, 2011.
`http://eprint.iacr.org/`.
(One citation on page 5.)

[98] Ewan Fleischmann, Christian Forler, Stefan Lucks, and Jakob Wenzel.
Weimar-DM: A Highly Secure Double-Length Compression Function.
In *ACISP*, pages 152–165, 2012.
(Citations on pages 4, 12, 30, 31, and 32.)

[99] Ewan Fleischmann, Michael Gorski, and Stefan Lucks.
On the Security of Tandem-DM.
In *FSE*, pages 84–103, 2009.
(Citations on pages 30 and 31.)

[100] Ewan Fleischmann, Michael Gorski, and Stefan Lucks.
Security of Cyclic Double Block Length Hash Functions.
In *IMA Int. Conf.*, pages 153–175, 2009.
(Citations on pages 30, 31, and 32.)

[101] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Overview of the Candidates for the Password Hashing Competition - And Their Resistance Against Garbage-Collector Attacks.
In Stig Fr. Mjølsnes, editor, *PASSWORDS*, volume 9393 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2014.
(Citations on pages 3, 4, and 43.)

[102] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Efficient Beyond-Birthday-Bound-Secure Deterministic Authenticated Encryption with Minimal Stretch.
In Joseph K. Liu and Ron Steinfeld, editors, *ACISP (II)*, volume 9723 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2016.
(One citation on page 5.)

[103] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Efficient Beyond-Birthday-Bound-Secure Deterministic Authenticated Encryption with Minimal Stretch.
IACR Cryptology ePrint Archive, 2016:395, 2016.
(One citation on page 5.)

[104] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
POEx: A Beyond-Birthday-Bound-Secure On-Line Cipher.
In *ArcticCrypt*, volume 1, 2016.
16 pages.
(One citation on page 5.)

[105] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel.
Reforgeability of Authenticated Encryption Schemes.
In *ACISP*, page to appear, 2017.
(Citations on pages 5 and 6.)

[106] Christian Forler, Stefan Lucks, and Jakob Wenzel.
Designing the API for a Cryptographic Library - A Misuse-Resistant Application Programming Interface.
In *Reliable Software Technologies - Ada-Europe 2012 - 17th Ada-Europe International Conference on Reliable Software Technologies, Stockholm, Sweden, June 11-15, 2012. Proceedings*, pages 75–88, 2012.
(One citation on page 5.)

[107] Christian Forler, Stefan Lucks, and Jakob Wenzel.
Catena: A Memory-Consuming Password Scrambler.
Cryptology ePrint Archive, Report 2013/525, 2013.
`http://eprint.iacr.org/`.
(One citation on page 4.)

[108] Christian Forler, Stefan Lucks, and Jakob Wenzel.
Memory-Demanding Password Scrambling.
In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 289–305, 2014.
(Citations on pages 3, 4, and 43.)

[109] Christian Forler, Stefan Lucks, and Jakob Wenzel.
The Catena Password-Scrambling Framework.
Password Hashing Competition, final submission, 2015.
`https://password-hashing.net/submissions/specs/Catena-v3.pdf`.
(Citations on pages 4 and 107.)

[110] Christian Forler, David McGrew, Stefan Lucks, and Jakob Wenzel.
COFFE: Ciphertext Output Feedback Faithful Encryption.
IACR Cryptology ePrint Archive, 2014:1003, 2014.
(One citation on page 5.)

[111] Pierre-Alain Fouque, Gwenaëlle Martinet, Frédéric Valette, and Sébastien Zimmer.
On the Security of the CCM Encryption Mode and of a Slight Variant.
In *ACNS*, pages 411–428, 2008.
(One citation on page 122.)

[112] Daniel Franke.
The EARWORM Password Hashing Algorithm.
`https://password-hashing.net/submissions/specs/EARWORM-v0.pdf`, 2014.
(One citation on page 108.)

[113] funkysash.
catena-variants.
`https://github.com/medsec/catena-variants`, 2015.
(One citation on page 94.)

[114] John R. Gilbert and Robert E. Tarjan.

VARIATIONS OF A PEBBLE GAME ON GRAPHS.
http://i.stanford.edu/pub/cstr/reports/cs/tr/78/661/CS-TR-78-661.pdf, 1978.
Accessed May 30, 2017.
(One citation on page 80.)

[115] Eric Glass.
The NTLM Authentication Protocol and Security Support Provider.
http://davenport.sourceforge.net/ntlm.html.
Accessed April 27, 2017.
(One citation on page 44.)

[116] Virgil D. Gligor and Pompiliu Donescu.
Fast Encryption and Authentication: XCBC Encryption and XECB Authentication Modes.
In *FSE*, pages 92–108, 2001.
(One citation on page 3.)

[117] Oded Goldreich.
The Foundations of Cryptography - Volume 1, Basic Techniques.
Cambridge University Press, 2001.
(One citation on page 43.)

[118] Shafi Goldwasser and Yael Tauman Kalai.
On the (In)security of the Fiat-Shamir Paradigm.
In *FOCS*, pages 102–113, 2003.
(One citation on page 16.)

[119] Michael Gorski, Thomas Knapke, Eik List, Stefan Lucks, and Jakob Wenzel.
Mars Attacks! Revisited: Differential Attack on 12 Rounds of the MARS Core and Defeating the Complex MARS Key-Schedule.
In Daniel J. Bernstein and Sanjit Chatterjee, editors, *INDOCRYPT*, volume 7107 of *Lecture Notes in Computer Science*, pages 94–113. Springer, 2011.
(One citation on page 6.)

[120] Jeremi M. Gosney.
The Pufferfish Password Hashing Scheme.
https://password-hashing.net/submissions/specs/Pufferfish-v1.pdf, 2015.
(One citation on page 112.)

[121] Frank Gray.
Pulse code communication, March 1953.
US Patent 2,632,058.
(One citation on page 74.)

[122] Michaël Peeters Guido Bertoni, Joan Daemen, Gilles Van Assche, and Ronny Van Keer.
CAESAR submission; Keyak v2.
http://competitions.cr.yp.to/caesar-submissions.html, 2016.
(One citation on page 123.)

[123] David Gullasch, Endre Bangerter, and Stephan Krenn.
Cache Games - Bringing Access-Based Cache Attacks on AES to Practice.
In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505, 2011.
(One citation on page 49.)

[124] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang.
Advanced Meet-in-the-Middle Preimage Attacks: First Results on full Tiger, and improved Results on MD4 and SHA-2.
ASIACRYPT'10, volume 6477 of Lecture Notes in Computer Science, 2010.
(One citation on page 89.)

[125] David Hamilton.
Yahoo's password leak: What you need to know (FAQ).
`https://www.cnet.com/news/yahoos-password-leak-what-you-need-to-know-faq/`.
Accessed: 2017-03-16.
(One citation on page 2.)

[126] H. Handschuh and D. Naccache.
"Shacal".
In B. Preneel, Ed., First Open NESSIE Workshop, Leuven, Belgium, November 13-14, 2000.
(One citation on page 2.)

[127] Helena Handschuh and Bart Preneel.
Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms.
In *CRYPTO*, pages 144–161, 2008.
(One citation on page 121.)

[128] Ben Harris.
Replacement index function for data-independent schemes (Catena).
`http://article.gmane.org/gmane.comp.security.phc/2457/match=grey`, 2015.
(One citation on page 74.)

[129] Mitsuhiro Hattori, Shoichi Hirose, and Susumu Yoshida.
Analysis of Double Block Length Hash Functions.
In *IMA Int. Conf.*, pages 290–302, 2003.
(One citation on page 31.)

[130] George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas.
Password Hashing Competition - Survey and Benchmark.
IACR Cryptology ePrint Archive, 2015:265, 2015.
(One citation on page 104.)

[131] Martin E. Hellman.
A Cryptanalytic Time-Memory Trade-Off.
IEEE Transactions on Information Theory, 26(4):401–406, 1980.
(One citation on page 18.)

[132] Shoichi Hirose.
Provably Secure Double-Block-Length Hash Functions in a Black-Box Model.
In *ICISC*, pages 330–342, 2004.
(Citations on pages 31 and 32.)

[133] Shoichi Hirose.
Some Plausible Constructions of Double-Block-Length Hash Functions.
In *FSE*, pages 210–225, 2006.
(Citations on pages 2, 30, 31, and 139.)

[134] Shoichi Hirose.
Some Plausible Constructions of Double-Block-Length Hash Functions.

In *FSE*, pages 210–225, 2006.
(Citations on pages 11, 29, and 32.)

[135] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway.
AEZ v4.2: Authenticated Encryption by Enciphering.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[136] Walter Hohl, Xuejia Lai, Thomas Meier, and Christian Waldvogel.
Security of Iterated Hash Functions Based on Block Ciphers.
In *CRYPTO*, pages 379–390, 1993.
(One citation on page 31.)

[137] Mei Hong, Hui Guo, and Xiaobo Sharon Hu.
A cost-effective tag design for memory data authentication in embedded systems.
In *Proceedings of the 15th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 17–26, 2012.
(One citation on page 122.)

[138] HPSchilling.
catena-variants.
`https://github.com/HPSchilling/catena-variants`, 2015.
(One citation on page 94.)

[139] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar.
Jackpot Stealing Information From Large Caches via Huge Pages.
Cryptology ePrint Archive, Report 2014/970, 2014.
`http://eprint.iacr.org/`.
(One citation on page 54.)

[140] ISO/IEC.
ISO DIS 10118-2: Information technology - Security techniques - Hash-functions, Part 2: Hash-functions using an n-bit block cipher algorithm. First released in 1992, 2000.
(One citation on page 30.)

[141] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, and Sumio Morioka.
CLOC and SILC v3.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(Citations on pages 123 and 124.)

[142] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu.
Breaking and Repairing GCM Security Proofs.
In *CRYPTO*, volume 7417 of *LNCS*, pages 31–49. Springer, 2012.
(One citation on page 129.)

[143] Jérémy Jean, Ivica Nikolić, Thomas Peyrin, and Yannix Seurin.
Deoxys v1.41.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[144] Magnus Westerlund John Mattsson.
Authentication Key Recovery on Galois Counter Mode (GCM).
Cryptology ePrint Archive, Report 2015/477, 2015.

http://eprint.iacr.org/2015/477.
(One citation on page 122.)

[145] Antoine Joux.
Authentication Failures in NIST version of GCM.
NIST Comment, 2006.
(One citation on page 122.)

[146] Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto.
The Lyra2 reference guide.
https://password-hashing.net/submissions/specs/Lyra2-v3.pdf, 2015.
(One citation on page 110.)

[147] Charanjit S. Jutla.
Encryption Modes with Almost Free Message Integrity.
In *EUROCRYPT*, pages 529–544, 2001.
(One citation on page 3.)

[148] B. Kaliski.
RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0.
Technical report, IETF, 2000.
(Citations on pages 44 and 47.)

[149] Poul-Henning Kamp.
The history of md5crypt.
http://phk.freebsd.dk/sagas/md5crypt.html.
Accessed April 27, 2017.
(Citations on pages 44, 45, and 51.)

[150] Evgeny Kapun.
Yarn password hashing function.
https://password-hashing.net/submissions/specs/Yarn-v2.pdf, 2014.
(One citation on page 113.)

[151] Jonathan Katz and Moti Yung.
Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation.
In *FSE*, pages 284–299, 2000.
(Citations on pages 3 and 24.)

[152] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner.
Secure Applications of Low-Entropy Keys.
In *ISW*, pages 121–134, 1997.
(One citation on page 44.)

[153] Swati Khandelwal.
VK.com HACKED! 100 Million Clear Text Passwords Leaked Online.
http://thehackernews.com/2016/06/vk-com-data-breach.html.
Accessed: 2017-03-16.
(One citation on page 2.)

[154] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva.
Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family.
In *FSE*, pages 244–263, 2012.

(One citation on page 89.)

[155] Lars R. Knudsen, Xuejia Lai, and Bart Preneel.
Attacks on Fast Double Block Length Hash Functions.
Journal of Cryptology, 11(1):59–72, 1998.
(One citation on page 31.)

[156] Lars R. Knudsen and Frédéric Muller.
Some Attacks Against a Double Length Hash Proposal.
In *ASIACRYPT*, pages 462–473, 2005.
(One citation on page 31.)

[157] Donald E. Knuth.
The Art of Computer Programming, Volume III: Sorting and Searching.
Addison-Wesley, 1973.
(One citation on page 15.)

[158] Tadayoshi Kohno, John Viega, and Doug Whiting.
CWC: A High-Performance Conventional Authenticated Encryption Mode.
In *FSE*, pages 408–426, 2004.
(Citations on pages 122, 123, and 140.)

[159] Matthias Krause, Frederik Armknecht, and Ewan Fleischmann.
Preimage Resistance Beyond the Birthday Bound: Double-Length Hashing Revisited.
IACR Cryptology ePrint Archive, 2010:519, 2010.
(One citation on page 31.)

[160] H. Krawczyk, M. Bellare, and R. Canetti.
HMAC: Keyed-Hashing for Message Authentication.
RFC 2104 (Informational), February 1997.
(One citation on page 44.)

[161] Ted Krovetz and Phillip Rogaway.
The Software Performance of Authenticated-Encryption Modes.
In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.
(One citation on page 3.)

[162] Ted Krovetz and Phillip Rogaway.
OCB.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(Citations on pages 122 and 123.)

[163] Xuejia Lai and James L. Massey.
Hash Function Based on Block Ciphers.
In *EUROCRYPT*, pages 55–70, 1992.
(Citations on pages 30 and 31.)

[164] Jooyoung Lee.
Provable Security of the Knudsen-Preneel Compression Functions.
In *ASIACRYPT*, pages 504–525, 2012.
(One citation on page 29.)

[165] Jooyoung Lee and Daesung Kwon.
The Security of Abreast-DM in the Ideal Cipher Model.

Cryptology ePrint Archive, Report 2009/225, 2009.
`http://eprint.iacr.org/`.
(Citations on pages 30 and 31.)

[166] Jooyoung Lee and Daesung Kwon.
The Security of Abreast-DM in the Ideal Cipher Model.
IEICE Transactions, 94-A(1):104–109, 2011.
(Citations on pages 10 and 30.)

[167] Jooyoung Lee and Martijn Stam.
MJH: A Faster Alternative to MDC-2.
In *CT-RSA*, pages 213–236, 2011.
(One citation on page 31.)

[168] Jooyoung Lee, Martijn Stam, and John P. Steinberger.
The Collision Security of Tandem-DM in the Ideal Cipher Model.
In *CRYPTO*, pages 561–577, 2011.
(Citations on pages 10, 30, and 31.)

[169] Jan V. Leeuwen.
Handbook of Theoretical Computer Science: Algorithms and Complexity.
MIT Press, Cambridge, MA, USA, 1990.
(One citation on page 1.)

[170] Thomas Lengauer and Robert Endre Tarjan.
Asymptotically Tight Bounds on Time-Space Trade-offs in a Pebble Game.
J. ACM, 29(4):1087–1130, 1982.
(Citations on pages 73, 75, 78, 79, 80, 81, 115, and 140.)

[171] Eik List and Mridul Nandi.
Revisiting Full-PRF-Secure PMAC and Using It for Beyond-Birthday Authenticated Encryption.
In Helena Handschuh, editor, *CT-RSA*, volume 10147 of *LNCS*, pages 31–49. Springer, 2017.
(One citation on page 135.)

[172] Jiqiang Lu.
On the Security of the COPA and Marble Authenticated Encryption Algorithms against (Almost) Universal Forgery Attack.
IACR Cryptology ePrint Archive, 2015:79, 2015.
(Citations on pages 122 and 134.)

[173] Stefan Lucks.
A Collision-Resistant Rate-1 Double-Block-Length Hash Function.
In *Symmetric Cryptography*, 2007.
(One citation on page 31.)

[174] Stefan Lucks and Jakob Wenzel.
Catena Variants - Different Instantiations for an Extremely Flexible Password-Hashing Framework.
In *Technology and Practice of Passwords - 9th International Conference, PASSWORDS 2015, Cambridge, UK, December 7-9, 2015, Proceedings*, pages 95–119, 2015.
(Citations on pages 4, 82, and 101.)

[175] Yiyuan Luo and Xuejia Lai.

Attacks On a Double Length Blockcipher-based Hash Proposal.
IACR Cryptology ePrint Archive, 2011:238, 2011.
(One citation on page 31.)

[176] T. Alexander Lystad.
Leaked Password Lists and Dictionaries - The Password Project.
`http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.
Accessed April 27, 2017.
(Citations on pages 44 and 46.)

[177] Udi Manber.
A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack.
Computers & Security, 15(2):171–176, 1996.
(One citation on page 17.)

[178] Sérgio Martins and Yang Yang.
Introduction to bitcoins: a pseudo-anonymous electronic currency system.
In *Center for Advanced Studies on Collaborative Research, CASCON '11, Toronto, ON, Canada, November 7-10, 2011*, pages 349–350, 2001.
(One citation on page 54.)

[179] Mikhail Maslennikov.
PASSWORD HASHING SCHEME MCS_PHS.
`https://password-hashing.net/submissions/specs/MCS_PHS-v2.pdf`, 2015.
(One citation on page 111.)

[180] David McGrew and John Viega.
The Galois/Counter Mode of Operation (GCM).
Submission to NIST. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf, 2004.
(Citations on pages 3, 121, 122, 123, and 140.)

[181] David A. McGrew and Scott R. Fluhrer.
Multiple forgery attacks against Message Authentication Codes.
IACR Cryptology ePrint Archive, 2005:161, 2005.
(One citation on page 121.)

[182] David A. McGrew and John Viega.
The Security and Performance of the Galois/Counter Mode (GCM) of Operation.
In *INDOCRYPT*, pages 343–355. Springer, 2004.
(One citation on page 122.)

[183] Mark Baugher David McGrew and Melinda Shore.
Securing Internet Telephony Media with SRTP and SDP.
`http://www.cisco.com/c/en/us/about/security-center/securing-voip.html`.
Accessed June 5, 2017.
(One citation on page 122.)

[184] Bart Mennink.
Optimal Collision Security in Double Block Length Hashing with Single Length Key.
In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 526–543. Springer Berlin Heidelberg, 2012.
(One citation on page 29.)

[185] Ralph C. Merkle.
One Way Hash Functions and DES.
In *CRYPTO*, pages 428–446, 1989.
(Citations on pages 2, 30, and 139.)

[186] Friedhelm Meyer auf der Heide.
A Comparison Between Two Variations of a Pebble Game on Graphs.
In *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*, pages 411–421, 1979.
(One citation on page 80.)

[187] Microsoft.
Security Considerations for Implementers.
`https://msdn.microsoft.com/en-us/library/cc236715.aspx`.
Accessed May 11, 2017.
(One citation on page 44.)

[188] Kazuhiko Minematsu.
AES-OTR v3.1.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(Citations on pages 122 and 123.)

[189] Gordon E. Moore.
Cramming more Components onto Integrated Circuits.
Electronics, 38(8), April 1965.
(One citation on page 19.)

[190] Gordon E. Moore.
Readings in Computer Architecture.
chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
(One citation on page 2.)

[191] Robert Morris and Ken Thompson.
Password Security - A Case History.
Commun. ACM, 22(11):594–597, 1979.
(Citations on pages 17 and 43.)

[192] Haneef Mubarak.
Lanarea DF.
`https://password-hashing.net/submissions/specs/Lanarea-v0.pdf`, 2014.
(One citation on page 110.)

[193] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton.
Reconsidering Generic Composition.
In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 257–274. Springer, 2014.
(Citations on pages iii, 123, 124, and 125.)

[194] Mridul Nandi.
Revisiting Security Claims of XLS and COPA.
Cryptology ePrint Archive, Report 2015/444, 2015.
`http://eprint.iacr.org/2015/444`.
(One citation on page 122.)

[195] Mridul Nandi, Wonil Lee, Kouichi Sakurai, and Sangjin Lee.
Security Analysis of a 2/3-Rate Double Length Compression Function in the Black-Box Model.
In *FSE*, pages 243–254, 2005.
(One citation on page 31.)

[196] National Institute of Standards and Technology.
FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2.
Technical report, US Department of Commerce, August 2002.
(One citation on page 17.)

[197] NIST National Institute of Standards and Technology.
Federal Information Processing Standards Publication (FIPS PUB) 197: Specification for the Advanced Encryption Standard (AES). November 2001. See `http://csrc.nist.gov`.
(One citation on page 10.)

[198] Krishna Neelamraju.
Facebook Pages: Usage Patterns | Recommend.ly.

`http://blog.recommend.ly/facebook-pages-usage-patterns/`.
Accessed May 16, 2013.
(One citation on page 19.)

[199] C. Newman, A. Menon-Sen, A. Melnikov, and N. Williams.
Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms.
RFC 5802 (Proposed Standard), July 2010.
(One citation on page 19.)

[200] Niels Ferguson and Stefan Lucks and Bruce Schneier and Doug Whiting and Mihir Bellare and Tadayoshi Kohno and Jon Callas and Jesse Walker.
Skein Source Code and Test Vectors.
`http://www.skein-hash.info/downloads`.
(One citation on page 2.)

[201] Ivica Nikolić.
Tiaoxin-346.
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[202] Nvidia.
Nvidia GeForce GTX 680 - Technology Overview, 2012.
(Citations on pages 3 and 46.)

[203] Philippe Oechslin.
Making a Faster Cryptanalytic Time-Memory Trade-Off.
Advances in CryptologyCRYPTO 2003, 3:617–630, 2003.
(Citations on pages 17, 43, and 65.)

[204] National Institute of Standards and Technology (NIST).
SHA-3 Competition.
`http://csrc.nist.gov/groups/ST/hash/sha-3/`, 2005.
(One citation on page 2.)

[205] NIST National Institute of Standards and Technology.
Federal Information Processing Standards Publication (FIPS PUB) 46-3: Data Encryption Standard (DES). October 1999. See `http://csrc.nist.gov`.
(One citation on page 10.)

[206] NIST National Institute of Standards and Technology.
FIPS 180-1: Secure Hash Standard. April 1995. See http://csrc.nist.gov.
(Citations on pages 2 and 15.)

[207] NIST National Institute of Standards and Technology.
FIPS 180-2: Secure Hash Standard. April 1995. See http://csrc.nist.gov.
(Citations on pages 2, 15, and 44.)

[208] Onur Özen and Martijn Stam.
Another Glance at Double-Length Hashing.
In *IMA Int. Conf.*, pages 176–201, 2009.
(One citation on page 31.)

[209] Michael S. Paterson and Carl E. Hewitt.
Comparative Schematology.
In Jack B. Dennis, editor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, chapter Computation schemata, pages 119–127. ACM, New York, NY, USA, 1970.
(One citation on page 79.)

[210] Colin Percival.
Cache missing for fun and profit.
In *Proc. of BSDCan 2005*, 2005.
(Citations on pages 20 and 49.)

[211] Colin Percival.
Stronger Key Derivation via Sequential Memory-Hard Functions.
presented at BSDCan'09, May 2009, 2009.
(Citations on pages 17, 18, 43, 45, 46, 47, 48, 49, and 139.)

[212] Alexander Peslyak.
yescrypt - a Password Hashing Competition submission.
`https://password-hashing.net/submissions/specs/yescrypt-v1.pdf`, 2015.
(One citation on page 114.)

[213] Thomas Peyrin, Henri Gilbert, Frédéric Muller, and Matthew J. B. Robshaw.
Combining Compression Functions and Block Cipher-Based Hash Functions.
In *ASIACRYPT*, pages 315–331, 2006.
(One citation on page 31.)

[214] Krisztián Pintér.
Gambit – A sponge based, memory hard key derivation function.
`https://password-hashing.net/submissions/specs/Gambit-v1.pdf`, 2014.
(One citation on page 109.)

[215] Thomas Pornin.
The MAKWA Password Hashing Function.
`https://password-hashing.net/submissions/specs/Makwa-v1.pdf`, 2015.
(One citation on page 110.)

[216] Bart Preneel.
MDC-2 and MDC-4.
In *Encyclopedia of Cryptography and Security*. Springer, 2005.
(One citation on page 11.)

[217] Bart Preneel, René Govaerts, and Joos Vandewalle.
Hash Functions Based on Block Ciphers: A Synthetic Approach.
In *CRYPTO*, pages 368–378, 1993.
(Citations on pages 2, 30, and 139.)

[218] Gordon Procter and Carlos Cid.
On Weak Keys and Forgery Attacks against Polynomial-based MAC Schemes.
In Shiho Moriai, editor, *FSE*, volume 8424 of *Lecture Notes in Computer Science - Lecture Notes in Computer Science*, pages 287–304. Springer, 2013.
(One citation on page 129.)

[219] Niels Provos and David Mazières.
A Future-Adaptable Password Scheme.
In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
(Citations on pages 44 and 45.)

[220] M. Rabin.
Digitalized Signatures.
In R. DeMillo, D. Dobkin, A. Jones and R.Lipton, editors, Foundations of Secure Computation, Academic Press, pages 155–168, 1978.
(Citations on pages 2, 11, and 139.)

[221] A.G. Reinhold.
HEKS: A Family of Key Stretching Algorithms, 1999.
(One citation on page 45.)

[222] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage.
Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds.
In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.
(One citation on page 49.)

[223] Ronald L. Rivest.
RFC 1321: The MD5 Message-Digest Algorithm.
Internet Activities Board, April 1992.
(One citation on page 44.)

[224] Ronald L. Rivest.
The MD6 hash function – A proposal to NIST for SHA-3.
Submission to NIST, 2008.
(One citation on page 2.)

[225] P. Rogaway and D. Wagner.
A Critique of CCM.
Cryptology ePrint Archive, Report 2003/070, 2003.
http://eprint.iacr.org/2003/070.
(One citation on page 122.)

[226] Phillip Rogaway.

Authenticated-Encryption with Associated-Data.
In *ACM Conference on Computer and Communications Security*, pages 98–107, 2002.
(One citation on page 23.)

[227] Phillip Rogaway.
Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC.
In *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.
(One citation on page 61.)

[228] Phillip Rogaway.
Nonce-Based Symmetric Encryption.
In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *LNCS*, pages 348–359. Springer, 2004.
(Citations on pages 3 and 121.)

[229] Phillip Rogaway.
Formalizing Human Ignorance.
In *VIETCRYPT*, pages 211–228, 2006.
(One citation on page 15.)

[230] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz.
OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption.
In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001.
(Citations on pages 3, 121, and 123.)

[231] Phillip Rogaway and Thomas Shrimpton.
Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance.
In *FSE*, pages 371–388, 2004.
(Citations on pages 12 and 30.)

[232] Phillip Rogaway and John P. Steinberger.
Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers.
In *CRYPTO*, pages 433–450, 2008.
(Citations on pages 2, 30, 32, and 139.)

[233] Markku-Juhani Olavi Saarinen.
Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes.
In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2012.
(Citations on pages 122 and 129.)

[234] Raphael Bost; Olivier Sanders.
Trick or Tweak: On the (In)security of OTR's Tweaks.
In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT*, volume to appear of *LNCS*. Springer, 2016.
(One citation on page 132.)

[235] Semiocast SAS.
Brazil becomes 2nd country on Twitter, Japan 3rd — Netherlands most active country.
`http://goo.gl/QOeaB`.

Accessed May 16, 2013.
(One citation on page 19.)

[236] Satoh, Haga, and Kurosawa.
Towards Secure and Fast Hash Functions.
TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems, 1999.
(One citation on page 31.)

[237] J. Savage and S. Swamy.
Space-time trade-offs on the FFT algorithm.
Information Theory, IEEE Transactions on, 24(5):563 – 568, sep 1978.
(One citation on page 79.)

[238] John E. Savage and Sowmitri Swamy.
Space-Time Tradeoffs for Oblivious Interger Multiplications.
In *ICALP*, pages 498–504, 1979.
(One citation on page 79.)

[239] Teath Sch.
Tortuga – Password hashing based on the Turtle algorithm.
`https://password-hashing.net/submissions/specs/Tortuga-v0.pdf`, 2014.
(One citation on page 112.)

[240] Bruce Schneier.
Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish).
In *FSE*, pages 191–204, 1993.
(One citation on page 44.)

[241] Ravi Sethi.
Complete Register Allocation Problems.
SIAM J. Comput., 4(3):226–248, 1975.
(One citation on page 79.)

[242] Adi Shamir.
How to Share a Secret.
Commun. ACM, 22(11):612–613, 1979.
(One citation on page 111.)

[243] Mark Shand, Patrice Bertin, and Jean Vuillemin.
Hardware Speedups in Long Integer Multiplication.
In *SPAA*, pages 138–145, 1990.
(One citation on page 94.)

[244] Marcos Simplicio, Leonardo Almeida, Paulo dos Santos, and Paulo Barreto.
The Lyra2 reference guide.
Password Hashing Competition, 2nd round submission, 2015.
`https://password-hashing.net/submissions/specs/Lyra2-v2.pdf`.
(Citations on pages 90, 92, and 94.)

[245] sneves (Samues Neves).
BLAKE2 Reference Implementation.
`https://github.com/BLAKE2/BLAKE2/tree/master/ref`.
Accessed April 25, 2017.

(One citation on page 90.)

[246] Peter Soderquist and Miriam Leeser.
An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations.
In *12th Symposium on Computer Arithmetic (ARITH-12 '95), July 19-21, 1995, Bath, England, UK*, pages 132–139, 1995.
(One citation on page 94.)

[247] Martijn Stam.
Blockcipher-Based Hashing Revisited.
In Orr Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2009.
(Citations on pages 2, 30, 31, and 139.)

[248] John P Steinberger.
The Collision Intractability of MDC-2 in the Ideal Cipher Model.
Cryptology ePrint Archive, Report 2006/294, 2006.
`http://eprint.iacr.org/`.
(One citation on page 31.)

[249] John P. Steinberger.
The Collision Intractability of MDC-2 in the Ideal-Cipher Model.
In *EUROCRYPT*, pages 34–51, 2007.
(One citation on page 10.)

[250] Jens Steube.
oclHashcat-plus - Advanced Password Recovery.
`http://hashcat.net/oclhashcat-plus/`.
Accessed April 27, 2017.
(Citations on pages 3 and 44.)

[251] Zhelei Sun, Peng Wang, and Liting Zhang.
Collision Attacks on Variant of OCB Mode and Its Series.
In Miroslaw Kutylowski and Moti Yung, editors, *Inscrypt*, volume 7763 of *LNCS*, pages 216–224. Springer, 2012.
(One citation on page 132.)

[252] Sowmitri Swamy and John E. Savage.
Space-Time Tradeoffs for Linear Recursion.
In *POPL*, pages 135–142, 1979.
(One citation on page 79.)

[253] Steve Thomas.
battcrypt (Blowfish All The Things).
`https://password-hashing.net/submissions/specs/battcrypt-v0.pdf`, 2014.
(One citation on page 107.)

[254] Steve Thomas.
Parallel.
`https://password-hashing.net/submissions/specs/Parallel-v0.pdf`, 2014.
(One citation on page 111.)

[255] Martin Tompa.

Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits.
In *STOC*, pages 196–204, 1978.
(One citation on page 79.)

[256] Eran Tromer, Dag Arne Osvik, and Adi Shamir.
Efficient Cache Attacks on AES, and Countermeasures.
Journal of Cryptology, 23(1):37–71, 2010.
(One citation on page 20.)

[257] John Tromp.
Cuckoo Cycle; a memory-hard proof-of-work system.
Cryptology ePrint Archive, Report 2014/059, 2014.
`http://eprint.iacr.org/`.
(Citations on pages 54 and 62.)

[258] Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen.
SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications.
Technical report, NIST, Gaithersburg, MD, United States, 2010.
(Citations on pages 44 and 64.)

[259] Sebastiano Vigna.
An experimental exploration of Marsaglia's xorshift generators, scrambled.
CoRR, abs/1402.6246, 2014.
(One citation on page 79.)

[260] Rade Vuckovac.
schvrch.
`https://password-hashing.net/submissions/specs/Schvrch-v0.pdf`, 2014.
(One citation on page 112.)

[261] M.V. Wilkes.
Time-Sharing Computer Systems.
MacDonald computer monographs. American Elsevier Publishing Company, 1968.
(Citations on pages 17 and 43.)

[262] Robert S. Winternitz.
A Secure One-Way Hash Function Built from DES.
In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.
(One citation on page 11.)

[263] Hongjun Wu.
POMELO: A Password Hashing Algorithm.
`https://password-hashing.net/submissions/specs/POMELO-v3.pdf`, 2015.
(One citation on page 112.)

[264] Hongjun Wu.
ACORN: A Lightweight Authenticated Cipher (v3).
`http://competitions.cr.yp.to/caesar-submissions.html`, 2016.
(One citation on page 123.)

[265] Hongjun Wu and Tao Huang.
The Authenticated Cipher MORUS.

http://competitions.cr.yp.to/caesar-submissions.html, 2016.
(One citation on page 123.)

[266] Hongjun Wu and Tao Huang.
The JAMBU Lightweight Authentication Encryption Mode (v2.1).
http://competitions.cr.yp.to/caesar-submissions.html, 2016.
(One citation on page 123.)

[267] Hongjun Wu and Bart Preneel.
AEGIS: A Fast Authenticated Encryption Algorithm (v1,1).
http://competitions.cr.yp.to/caesar-submissions.html, 2016.
(One citation on page 123.)

[268] Eric A. Young and Tim J. Hudson.
OpenSSL: The Open Source toolkit for SSL/TLS.
http://www.openssl.org/, September 2011.
(One citation on page 49.)

[269] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart.
Cross-VM side channels and their use to extract private keys.
In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 305–316, 2012.
(One citation on page 49.)