The Catena Password-Scrambling Framework

Christian Forler* Stefan Lucks† Jakob Wenzel

cforler@posteo.de
stefan.lucks@uni-weimar.de
jakob.wenzel@uni-weimar.de

Bauhaus-Universität Weimar

Version 3.2

September 17, 2015

*The research leading to these results received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement no. 307952.

†A part of this research was done while Stefan Lucks was visiting the National Institute of Standards and Technologies (NIST), during his sabbatical.

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

– Albert Einstein

Changelog

From Version 3.1 to Version 3.2

- adjusted the tweak description corresponding to the reference implementation (m and $|s|$ are now given by byte-sized values and describe the output length and the salt length in bytes, respectively)
- updated performance values for parameter recommendations (cost parameter remain the same to preserve consistency)
- updated test vectors
- added pseudocode and description of BLAKE2b-1
- reference implementation now supports client-independent for keyed password hashing and keyed server relief
- changed the link to the reference implementation
- renamed g_0 and g to g_{low} and g_{high}
- other minor changes (introduction of variables, typos, ...)
- introduced Catena-Axungia: an automated benchmark-tool to find optimal parameters for given constraints (see Section 3.4)
- introduced Catena-Variants: an implementation of Catena that allows extension and modifications (see Section 3.4)

From Version 3.0 to Version 3.1

- added test vectors for keyed password hashing (Appendix C)

From Version 2.1 to Version 3.0

- introduce Catena-Dragonfly as a specific instance of Catena-BRG and Catena-Butterfly as a specific instance of Catena-DBG.
- introduce BLAKE2b-1, a round-reduced variant of BLAKE2b
- change the way the memory is initialized: (1) using two predecessors instead of only one and (2) introducing a new function Γ, which updates the memory using a memory-access pattern based on an arbitrary public input.
- introduce SaltMix as a particular instance of Γ.
- adding one invocation of the underlying memory-hard function $flap$ with $\lceil g_0/2 \rceil$
- added a new chapter called "Lessons Learned: The Tweak" (Chapter 9).
- some additional minor changes

From Version 2.0 to Version 2.1

Line 2 of Algorithm 4 in Chapter 5. Swapping the input parameters.

From Version 1.1 to Version 2.0

- removed the flawed proof for λ-memory-hardness of the (g, λ)-bit-reversal hashing operation (based on the cryptanalysis by Biryukov and Khovratovich [9])
- Catena is now designated as a password-scrambling framework (PSF) instead of a pure password scrambler
- introducing the name Catena-BRG: Catena instantiated with the memory-hard (g, λ)-bit-reversal hashing operation (BRH^g_λ)
- introducing a new instance Catena-DBG: Catena instantiated with the λ-memory-hard (g, λ)-double butterfly hashing operation (DBH^g_λ)
- new recommendations for the usage of either Catena-BRG or Catena-DBG depending on the required memory-hardness
- set version ID to `0xFF` for Catena-BRG
- set version ID to `0xFE` for Catena-DBG

From Version 1.0 to Version 1.1

- prepend the version ID byte, currently `0xFF`, to the tweak (cf. Chapter 3 and Section 8.3)
- swapped the two input parameters of the hash function H in Line 6 of Algorithm 4 in Chapter 3

Executive Summary

Catena is a novel and provably secure password-scrambling framework with cutting-edge properties. Catena supports flexible usage in multiple environments. Catena can also be used as a *key-derivation function* and for *proofs of work/space*.

Catena is flexible. An instantiation of Catena is defined by: (1) a cryptographic primitive H, e.g., BLAKE2b [5], (2) a "reduced" primitive H' (typically a reduced-round version of H, though $H' = H$ is also possible), (3) an optional randomization layer Γ, to harden the memory initialization, and (4) a "memory-hard" function F, using both H and H', and with some graph-driven data flow.

Catena has tunable parameters. The *garlic* defines time and memory requirements for Catena, increasing the *pepper* allows to increase the time without affecting the memory, and the *salt* size determines the defense against precomputation attacks. Catena supports a *server relief* protocol to shift the effort (both time and memory) for computing the password hash from the server to the client. Catena provides the *client-independent update* feature allowing the defender to increase the main security parameters (*garlic* and *pepper*) at any time, even for inactive accounts.

Catena provides two default instances. Catena is a framework. The user can plug in any suitable H, H', and F. This may be too much too choose from, for the less experienced users, and this lacks a fixed target for the cryptanalysts. Thus, we recommend two instances, with different choices for F. **Catena-Butterfly** runs in limited memory and maximizes the work for lower-memory adversaries, **Catena-Dragonfly** maximizes the amount of memory used.

Catena has a sound theoretical foundation. Both default instances have a sound and elegant design given by a simple and well-understood graph-based structure. The data flow in Catena-Butterfly is based on a stack of double-butterfly graphs, the data flow in Catena-Dragonfly is based on bit-reversal graphs. The time-memory tradeoff analysis follows the research from [30] and is based on the pebble game, which was – mostly in the 1970s and 1980s – extensively used to study time-memory tradeoffs.

Catena is secure. We claim the following security properties of Catena: **preimage security** (this important for most applications of password hashes), **indistinguishability from random** (important for key derivation), **lower bounds on the time-memory tradeoffs** (for high resilience against massively parallel attacks with constrained memory, such as, e.g., when using GPUs), and **resistance to side-channel attacks**, such as cache-timing and 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 the secret key.

Contents

Chapter 1

Introduction

This document introduces Catena, our submission to the Password Hashing Competition (PHC). We elaborate on the requirements for password hashing in general, and on some of the specific design choices for Catena.

Passwords[1] are user-memorizable secrets that are commonly used for user authentication and cryptographic key derivation. Typical (user-chosen) passwords often 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. In some scenarios, when a password is used to initiate an interactive session, the security of password-based authentication and key derivation can be enhanced by dedicated cryptographic protocols defeating "off-line" password guessing, (see, e.g., [6] for an early example). Otherwise, the best protection is given by performing "key stretching".

Key Stretching. Let X be a password with μ 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 $2^{\mu-1}$ password candidates. To slow down the adversary by a factor of 2^σ, one iterates the hash function 2^σ times by computing $Y_i = H(Y_{i-1})$ for $i \in \{2, \ldots, 2^\sigma\}$ and then uses Y_{2^σ} as the final password hash. There are variations of this approach, but iterating H is the core idea behind the majority of current password scramblers, such as `md5crypt` [28] and `sha512crypt` [16]. This forces the adversary to call the hash function $2^{\sigma+\mu}$, rather than 2^μ times. But the defender is also slowed down by 2^σ. 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 σ. Thus, there is no protection against password-cracking adversaries for users with weak passwords[2]. Furthermore, in the rare case that a user has a high-entropy password (say, $\mu > 100$), key stretching is unnecessary. But, for users with mid-entropy passwords, key

[1] In our context, "passphrases" and "personal identification numbers" (PINs) are also "passwords".

[2] A study from 2012 reports a min-entropy $\mu < 7$ bit for typical user groups [11]. For any such group, an adversary trying the group's most frequent password succeeds for $\approx 1\,\%$ of the users.

stretching can *hold the balance* in terms of security. Thus, we define the basic conditions for any password scrambling function PS are as follows:

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

(2) Computing PS(pwd) should be "as slow as possible", without contradicting (1).

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

Memory-Demanding Key Stretching. The established approach of performing key stretching by iterating a conventional primitive many times has become less useful over the years. 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, graphical processing units (GPU) with hundreds of cores [35] 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 are running at full speed. However, the memory – and, especially, the fast ("cache") memory – on a typical GPU are about as large (at least by the order of magnitude) as the memory and cache on a typical CPU as used by typical defenders. Thus, the idea behind a memory-demanding password scrambler 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` [38].

Nevertheless, two aspects of `scrypt` did trouble us: First, `scrypt` is quite complex since it combines two independent cryptographic primitives (the SHA-256 hash function and the Salsa20/8 core operation) and four generic operations (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 thus `scrypt`, vulnerable to cache-timing attacks [24]. Moreover, we have shown in [24] that `scrypt` is vulnerable against *garbage-collector attacks*, i.e., a malicious garbage collector

can obtain internal states of the algorithm from memory fragments, which allows to test password candidates in a highly efficient manner (see Section 2.2).

There is a growing amount of resarch on cache-timing attacks, mostly with a focus on recovering secret keys by measuring which entries in an S-box table have been read. See [27] for a recent example. We believe that many of the techniques in this context are applicable to the internal memory of password-hashing functions. To the best of our knowledge, practical exploits for garbage-collector attacks have not been studied much, so far.

In any case, both cache-timing and garbage-collector attacks are frightening risks, which we want both to avoid. Therefore, our challenge was to design a *new* memory-demanding password scrambler PS which complies to the six previously stated properties and also the following additional properties:

(7) Easy to analyze.

(8) Resilient to cache-timing attacks.

(9) Resilient to garbage-collector attacks.

To accomplish Property (7), we focused on a single generic operation, using a single cryptographic primitive. The analysis should prove its expected security properties under well-established assumptions and models for the underlying 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. A malicious garbage collector will then learn only the information written at the end of the scrambler operation.

Desired Flexibility Properties. The current generation of password scramblers is rather inflexible and we would like future password scramblers to support the following features:

- *server relief*: the option to shift the main memory and time effort from a authentication server to the client, without burdening the server,
- *pepper* and *garlic*: security parameters to adjust time and memory requirements,
- *client-independent update*: adjust (increase) the security parameters, even without knowing the password.

To the best of our knowledge, the idea for *client-independent updates* has been introduced in [24].

Design Choices for Catena. Informally, Properties (1)-(6) can be translated into the rule of thumb "fast enough on the defender's machine" and "as slow as possible on the adversaries' machines". This is what any password scrambler is trying to achieve – and

the design of a password scrambler depends on the designers' understanding of these machines. 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 certain amount of fast memory, i.e., cache. On the other hand, 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 (i.e., FPGAs) and non-reprogrammable hardware (ASICs). Catena has been designed as defense with all these adversaries in mind, with the highest priority on commodity hardware, followed by reprogrammable hardware. However, the second-round tweak has, in part, been motivated by the desire to improve resistance against adversaries using expensive non-reprogrammable hardware.

While our concrete proposal suggests to use BLAKE2b for H and BLAKE2b-1 (one round of BLAKE2b including finalization) for H', Catena enables the defender to actually choose any strong hash function H and a reduced hash function H' that run well on its machine. The freedom to change H and H' has the additional side effect of frustrating well-funded adversaries who use expensive non-reprogrammable hardware: For every defender with different H and H', they would have to buy new hardware.

Note that Catena is a composed cryptographic operation, based on a cryptographic hash function. An alternative would have been some new primitive with the structure of Catena. Section 7.2 elaborates on the reasons why we avoided that alternative.

Specific Choices for Catena Related to PHC. Beyond meeting Properties (1)-(9) and support for our desired flexibility properties, the design of Catena also meets the requirements of the PHC [4]:

- Support passwords of any length between 0 and 128 bytes.
- Support salts of 16 bytes.
- Provide at least one cost parameters to tune time and/or space usage.
- Produce (but not limited to) 32-byte outputs.
- Support of optional inputs such as a personalization string, a secret key, or any application-specific parameter.

Actually, Catena allows to choose paswwords and salts of arbitrary length. Furthermore, the maximum length of the password hash value depends on the underlying hash function. The the time and/or memory usage can be adjusted:

- Keeping bits of the salt secret (pepper).
- Increasing the memory-cost parameter (garlic).
- Increasing the number of stacks of the inner structure (λ).

Furthermore, Catena is designed to fulfill the following security properties:

- Standard cryptographic security: preimage resistance, collision resistance, immunity to length extension, infeasibility to distinguish outputs from random.
- High computational costs for massively parallel cracking devices with limited amount of fast memory, e.g., GPUs, low-cost ASICs, and FPGAs.
- Resilience against side-channel attacks, such as cache timing.

We present a comprehensive security analysis to show that Catena provides the desired cryptographic security.

Outline. the remainder of the paper is structured as follows. Chapter 2 lists the necessary preliminaries, definitions, and fundamental password-scrambling properties we use throughout this paper. Chapter 3 provides the specification of Catena, our new password-scrambling framework, as well as our parameter recommendations for the PHC. Furthermore, we discuss functional and security properties of Catena. In Chapter 4, we analyze the Catena framework in terms of its preimage security and pseudorandomness. In Chapter 5 we introduce two instances of Catena: Catena-Dragonfly and Catena-Butterfly . In Chapter 6, we discuss their security properties in terms of memory-hardness, pseudorandomness, and resistance against side-channel attacks. Chapter 7 contains our design rationing. The usage of Catena for the proof of work scenario, a discussion about Catena in different environments, and its application as key-derivation function are given in Chapter 8. Chapter 9 summariezes the proposed modifications for the second-round tweak. We conclude with an acknowldegement, a legal disclaimer, the bibliography and some appendices.

Chapter 2

Preliminaries

In this section we describe a technique called *Pebble Game*, which will help to understand the proofs of our underlying graph-based structures presented in [30]. Furthermore, we introduce necessary definitions and notations used throughout this paper. Note that we often refer CATENA to one specific instance than to the generic framework. In this situations it is meant that the considered property holds for all presented instances.

2.1. The Pebble Game

The pebble game is a well-known method from theoretical computer science to analyze time-memory tradeoffs for a restricted set of programs. The restrictions are as follows:

1. 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 "loop unrolling".

2. 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 *directed acyclic graph* (DAG, see Definition 2.1) of vertices and directed edges, where vertices without ingoing edges represent inputs, and all remaining vertices represent the result of an operation.

Definition 2.1 (Directed Acyclic Graph (DAG)). *Let $\Pi(\mathcal{V}, \mathcal{E})$ be a directed graph consisting of a set of vertices $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$ and a set of edges $\mathcal{E} = (e_0, e_1, \ldots, e_{\ell-1})$. $\Pi(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph iff it does not contain any directed cycle, i.e., a path from a node $v \in \mathcal{V}$ to itself.*

On the other hand, edges represent the data flow of an operation, i.e., the operation $r \leftarrow (x \diamond y) \circ z$ would be represented by three edges $x \rightarrow r$, $y \rightarrow r$, and $z \rightarrow r$. While the pebble game is defined for vertices with fan-in $\leq d$, for some constant d on operations with at most three inputs, i.e., "$(x \diamond y) \circ z$", implying that all vertices within the DAG have a fan-in of at most 3. Then, "$\diamond$" and "$\circ$" can be any operation which take two inputs x and y or $(x \circ y)$ and z, respectively, and generate one output r, such as $(x \diamond y) \circ z = H((x \oplus y) \mid\mid z)$. Moreover, for any two vertices $s \neq s'$, with $s \leftarrow x \diamond y$ and $s' \leftarrow x' \diamond z'$, the symbol "$\diamond$" can represent different operations, depending on the target s resp. s'. The same holds for the symbol "$\circ$".

Playing the Pebble Game. The background for the pebble game is to determine a time-memory tradeoff 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 performss a number of certain actions until a predefined output vertex has been pebbled. The following two actions are possible:

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

1. Put a pebble from the heap onto v (all w_i remain pebbled).
2. Move a pebble from one of the w_i to v (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.

Time-Memory Tradeoffs. Hellman presented in [26] the approach to trade memory/space S against time T in attacking cryptographic algorithms, i.e., he has introduced the idea of a time-memory tradeoff (TMTO) in terms of generic attacks. Hence, we can assume that an adversary with access to this algorithm and restricted resources is always looking for a sweet spot to minimize $S \cdot T$. To analyze the effort for a given adversary, one needs to choose a certain model for studying the TMTO. In 1970, Hewitt and Paterson [37] introduced the pebble game as a method for analyzing TMTOs on directed acyclic graphs, which became an important tool for that purpose (see [41–43, 45, 46]). The pebble game has been occasionally used in cryptographic context (see [22] for a recent example).

Figure 2.1 presents a simple example. In spite of its simplicity, it reveals an interesting tradeoff between space S and time T, where S denotes the number of pebbles, and T

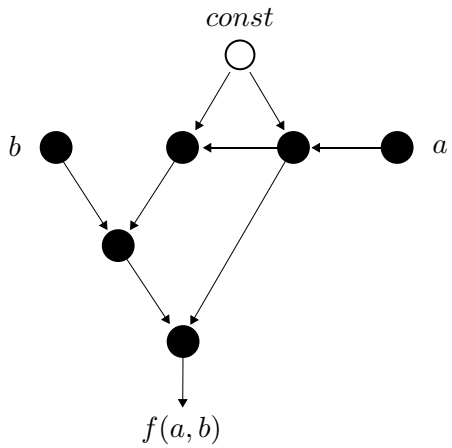

Figure 2.1.: Directed acyclic graph for $f(a, b) = (a \circ const) \circ const) \circ b) \circ (a \circ const)$.

the number of moves: Note that the value "const" denotes a fixed value which is always in the memory, i.e., one gets this vertex for free.

With $S \geq 3$, time $T = 6$ is sufficient for pebbling the output vertex. With less than three pebbles, this needs more time. The reason is that common subexpressions cannot be stored, any more, but must be recomputed. The graph can still be pebbled with $S = 2$ and $T = 8$. The graph cannot be pebbled with $S = 1$.

The Pebble Game and Password Scramblers. Note that the upper and lower bounds proved in [30] highly depend on the operation "$\circ$". Recall that the computation of $x = y \circ z$ can represent different operations, depending on the target x, and "time" is just the total number of such operations. If all "$\circ$"-operations take approximately the same time, upper bounds ("given space S, one can pebble the graph in time T", for some specific T and S) allow to meaningfully estimate *the computational effort of the defender*. However, the *security of the defender* depends on the non-existence of efficient algorithms for the adversary. Such non-existence results are lower bounds ("given space S, it is impossible to pebble the graph in less than time T"). But the lower bounds are only applicable if the computations of an adversary follow the DAG. Depending on the operation "$\circ$", this may be the case, or not.

With algebraic operations, the lower bound usually collapses. For example, let "$\circ$" denote the integer addition "+", or the XOR-operation "$\oplus$". The function $f(a, b)$ from Figure 2.1 degenerates to $f(a, b) = 2a + 3const + b$ in the case of the addition, and $b \oplus const$ for the XOR. With the addition, the function f can be computed with $S = 2$ pebbles in less then eight operations, with the XOR, $f(a, b)$ can even be computed with $S = 1$. On the other hand, if one models the operation $x \circ y$ as a call to a random oracle $H(x \,||\, y)$, there is no alternative way to compute f. Since it is well-established practice in cryptography to instantiate random oracles by hash functions, Catena instantiates its internal operation by a strong cryptographic hash function, where we suggest BLAKE2b

as default primitive.

Of course, there is a middle-ground between using a simple algebraic operation on one side, and an entire cryptographic primitive on the other side. BLAKE2b consists of several rounds, where each round is a cunning composition of xor operations, additions, and bit-wise rotations. If we use such an operation for "$\circ$" (or in general, if we use the internal round or step operations of a cryptographic primitive), the relevance of the lower bounds is completely unclear, and finding "shortcut attacks" with improved time-memory tradeoffs becomes a new challenge for cryptanalysts. We elaborate on this approach, which would turn a password scrambler into a cryptographic primitive of its own right, in Section 7.2.

2.2. Properties and Definitions

Below, we describe and define the desired properties of a modern password scrambler. Note that we refer to the garlic in general by g (or when used to describe a particular graph instance) and only differ between g_{low} and g_{high} when necessary.

Memory-Hardness. To describe memory requirements, we adopt and slightly change the notion from [38]. 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.

Definition 2.2 (Memory-Hard Function). *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 $S(g) \in \Omega(T(g)^{1-\alpha})$.*

Thus, for $S \cdot T = G^2$ with $G = 2^g$, using b cores, we have

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

A formal generalization of this notion is given in the following definition.

Definition 2.3 ($\lambda-$Memory-Hard Function). *Let g denote the memory cost factor. For a $\lambda-$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*

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

Thus, we have

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

Remark. Note that for a λ-memory-hard function f, the relation $S(g) \cdot T(g)$ is always in $\Omega(G^{\lambda+1})$, i.e., it holds that if S decreases, T has to increase, and vice versa.

λ-Memory-Hard vs. Sequential Memory-Hard. In [38], Percival introduced the notion of sequential memory-hardness (SMH), which is satisfied by his password scrambler `scrypt`. Based on this notion, an algorithm called sequential memory-hard, if no adversary has a computational advantage from the use of multiple CPUs, i.e., using b cores requires b times the effort for a single core. It is easy to see that, in the parallel computation setting, SMH is a stronger notion than that of λ-memory-hardness (λMH). Thus, SMH is a desirable goal when designing a memory-consuming password scrambler. In this section, we discuss why our presented password scrambler Catena satisfies at most λMH instead of SMH, without referring to details of Catena, which are presented in Chapters 3 and 5.

Note that a further goal of our design was to provide resistance against cache-timing attacks, i.e., all instances of Catena should satisfy a password-independent memory-access pattern. This goal can be achieved by providing a control flow which is independent of its inputs or at least independent of its secret inputs. If none of the inputs determine the control flow, Catena can be seen as a straight-line program, which on the other hand can be represented by a directed acyclic graph.

Usually, a DAG can be computed (at least partially) in parallel. Assuming that one has b processors to compute a graph $\Pi(\mathcal{V}, \mathcal{E})$, one can partition $\Pi(\mathcal{V}, \mathcal{E})$ into b disjunct subgraphs $\pi_0, \ldots, \pi_{b-1}$. Let $\mathcal{R}_{i,j}$ denote the set of crossing edges between two subgraphs π_i and π_j. If the available shared memory units are at least equal to the order of $\mathcal{R}_{i,j}$, one can compute π_i and π_j in parallel. More detailed, in the first step one computes each vertex corresponding to a crossing edge and stores them in the global shared memory. Next, both subgraphs can be processed in parallel by accessing this memory. It follows that if the available memory is

$$\sum_{i=0}^{b-1} \sum_{j=0}^{b-1} |\mathcal{R}_{i,j}|,$$

then, one can compute all subgraphs $\pi_0, \ldots, \pi_{b-1}$ in parallel. Due to the structure of Catena (or more specifically, the main structure of our proposed instances) one can always partition its corresponding DAGs into such subgraphs and therefore, Catena can be at least partially computed in parallel, which is a contradiction to the definition of sequential memory-hardness. Thus, we introduced the notion of λMH as described above, which is a weaker notion in the parallel computing setting but a stronger notion in the single-core setting. To the best of our knowledge, Catena is the first password scrambler which satisfies both to be memory-consuming (by satisfying at least λMH) and providing resistance against cache-timing attacks.

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 PS, we define the Password-Recovery Advantage of an adversary A as follows:

Definition 2.4 (Password-Recovery Advantage). *Let s denote a randomly chosen salt value, and let $\mathcal{Q}$ be a entropy source with e bits of min-entropy. Then, we define the password-recovery advantage of an adversary A against a password scrambler PS as*

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

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

In Section 4.1 we provide an analysis of Catena which shows that for guessing a valid password, an adversary either has to try all possible candidates or it has to find a preimage for the underlying hash function.

Client-Independent Update. According to Moore's Law [32], 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 [33] and 73% of all Twitter accounts do not have at least one tweet per month [40]. 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* of the password hash. When key stretching is done by iterating an operation, client-independent updates may or may not be possible, depending on the details of the operation, e.g., when the original password is one of the inputs for every operation, client-independent updates 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, i.e., to shift the effort from the side of the server to the side of the client, can be found in [34] and more recently in [15]. We realized this idea by splitting the password-scrambling process 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 $h = H(F(pwd, s))$ from a password pwd and a salt s. Alternatively, the server sends s to the client who responds $y = F(pwd, s)$. Finally, the server just computes $h = H(y)$. While it is probably easy to write a generic *server-relief* protocol, none of the existing password scramblers has been designed to naturally support this property. Note that this property is optional, e.g., the server-relief idea makes no sense for the proof-of-work scenario since the whole effort should be already on the side of the client.

Resistance Against Cache-Timing Attacks. Consider the implementation of a password scrambler, 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 out passwords candidates. Under certain circumstances, timing information related to a given machine's cache behavior may enable the adversary to observe which addresses have been accessed. Thus, we formally introduce resistance against cache-timing attacks.

Definition 2.5 (Resistance against Cache-Timing Attacks). *Suppose the function $\mathcal{F} : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n$ processes arbitrary large data together with a secret value K with $|K| = k$, and outputs a fixed-length value of size n. We call $\mathcal{F}$ resistant against cache-timing attacks iff its control flow does not depend on the secret input K.*

Key-Derivation Function (KDF). Beyond authentication, passwords are also used to derive symmetric keys. Obviously, one can just use the output of the password scrambler 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 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 of the keys does not endanger the other ones. Note that it is required for a KDF that the input and output behavior cannot be distinguished from a set of random functions. Thus, we define the random-oracle security of a password scrambler as follows:

Definition 2.6 (Random-Oracle Security). *Let $PS : \{0,1\}^* \to \{0,1\}^n$ be a password scrambler which gets an input of arbitrary length and produces a fixed-length output. Let A be a fixed adversary which is allowed to ask at most q queries to an oracle. Further, let $\$: \{0,1\}^* \to \{0,1\}^n$ be a random function which, given an input of arbitrary length, always returns uniformly at random values from $\{0,1\}^n$. Then, the Random-Oracle Security of a password scrambler PS is defined by*

$$\mathbf{Adv}^{\$}_{PS}(A) = \left| \Pr\left[A^{PS} \Rightarrow 1\right] - \Pr\left[A^{\$} \Rightarrow 1\right] \right|.$$

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

Note that the input (of arbitrary length) of PS contains the password, the salt, and optional parameters, e.g., parameters to adjust the memory consumption or the computational time.

Resistance Against (Weak) Garbage-Collector ((W)GC) Attacks. The basic idea of this attack type is to exploit that PS leave the internal state or (efficiently computable) password -dependend 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 Garbage-Collector Attack.

Definition 2.7 (Garbage-Collector Attack). *Let $PS_G(\cdot)$ be a memory-consuming password scrambler that depends on a memory-cost parameter G and let Q be a positive constant. Furthermore, let v denote the internal state of $PS_G(\cdot)$ after its termination. Let $\mathcal{A}$ be a computationally unbounded but always-halting adversary conducting a garbage-collector attack. We say that $\mathcal{A}$ is successful if some knowledge about v reduces the runtime of $\mathcal{A}$ for testing a password candidate x from $\mathcal{O}(PS_G(x))$ to $\mathcal{O}(f(x))$ with $\mathcal{O}(f(x)) \lll \mathcal{O}(PS_G(x))/Q, \forall x \in \{0,1\}^*$.*

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

Definition 2.8 (Weak Garbage-Collector Attack). *Let $PS_G(\cdot)$ be a password scrambler that depends on a memory-cost parameter G, and let $R(\cdot)$ be an underlying function of $PS_G(\cdot)$ that can be computed efficiently. We say that an adversary is successful in terms of a weak garbage-collector attack if a value $y = R(pwd)$ remains in memory during (almost) the entire runtime of $PS_G(pwd)$, 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 WGC attacks trivial.

Endianness. All values used within the reference implementation of Catena are assumed to be represented in little-endian byte order.

2.3. Notational Conventions

Identifier	Description
pwd	password
λ	security parameter of F (depth)
s	salt (public random value)
p	pepper (secret bits of the salt)
t	tweak
d	domain (application specifier) of Catena
V	unique version identifier
γ	public input (e.g., salt)
g_{low},g_{high}	minimum garlic; current garlic with $G = 2^{g_{\text{high}}}$
H	underlying cryptographic hash function
H'	reduced version of H
PS/PSF	Password Scrambler/Password-Scrambling Framework
m	output length of Catena
F	memory-hard function replaced in a particular instance of Catena
Γ	function depending on the public input γ
\$	function returning a fixed-size random value
h, y	password hash (or intermediate hash)
$S(g)$	memory (space) consumption; depends on the garlic
$T(g)$	time consumption; depends on the garlic
$\Pi(\mathcal{V}, \mathcal{E})$	graph based on $\mathcal{V}$ vertices and $\mathcal{E}$ edges
r^i	i-th row of a $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$
v_i^j	i-th vertex of the j-th BRG
$v_{i,j}^k$	i-th vertex of the j-th row of the k-th DBG
b	number of cores
$A^{O_1,\ldots,O_\ell}$	adversary A with access to the oracles $O_1, \ldots, O_\ell$
q	number of total queries A is allowed to ask
τ	Bit-Reversal Permutation
σ	function determining the index of the diagonal edges (DBG)
AD	associated data
K	secret key
$\lvert X \rvert$	size of X in bits or size of a set X

Table 2.1.: Notations used throughout this document.

Chapter 3

Catena – A Memory-Hard Password-Scrambling Framework

In this chapter we introduce our password-scrambling framework (PSF) called Catena. Besides providing novel and sustainable properties, it provides high resilience against cache-timing attacks on the secret input.

3.1. Specification

A formal definition is shown in Algorithm 1, whereas the general idea is given in Figure 3.1. The function *truncate(x,m)* (see Line 6 of Algorithm 1) outputs the m least significant bytes of x, where m is the user-chosen output length of Catena. After processing the password, the tweak, and the salt, the function $flap$ is called once with $\lceil g_{\text{low}}/2 \rceil$ to provide resistance against weak garbage-collector attacks. Then, as the first step of each invocation of the main loop, the function $flap$ is called, where the password-dependent input x is padded with as many 0's as necessary so that $x \mid\mid 0^*$ fits the output size of the underlying hash function. By default, Catena uses BLAKE2b for H and BLAKE2b-1 for H' (Catena-Dragonfly and Catena-Butterfly), where BLAKE2b-1 denotes one single round of BLAKE2b including finalization (see Appendix A for a detailed description of BLAKE2b-1). Note that, depending on the requirements of the application, it is also possible to set $H' = H = BLAKE2b$ (e.g., we followed this approach for Catena-Dragonfly-Full and Catena-Butterfly-Full).

The function $flap$ (see Algorithm 2) consists of three phases: (1) an initialization phase (see Lines 1 to 6), where the memory of size $2^g \cdot n$ bits is written in a sequential order (where n denotes the output size of the underlying hash function in bits), (2) the function Γ (see Line 7) depending on the public input γ, which, can for example, be instantiated with a graph-based structured satisfying a random memory-access pattern depending on the salt, and (3) a call to a memory-hard function F (see Line 8).

Algorithm 1 Catena

Input: pwd {Password}, t {Tweak}, s {Salt}, g_{low} {Min. Garlic}, g_{high} {Garlic}, m {Output Length}, γ {Public Input}
Output: x {Hash of the Password}

$x \leftarrow H(t \mid\mid pwd \mid\mid s)$
$x \leftarrow flap(\lceil g_{\text{low}}/2 \rceil, x, \gamma)$
for $g = g_{\text{low}} \ldots, g_{\text{high}}$ **do**
  $x \leftarrow flap(g, x \mid\mid 0^*, \gamma)$
  $x \leftarrow H(g \mid\mid x)$
  $x \leftarrow truncate(x, m)$
end for
return x

Algorithm 2 Function $flap$ of Catena

Input: g {Garlic}, x {Value to Hash}, γ {Public Input}
Output: x {Intermediate Hash Value}

$v_{-2} \leftarrow x \oplus 1$
$v_{-1} \leftarrow x$
$v_0 \leftarrow H(v_{-1} \mid\mid v_{-2})$
for $i = 1, \ldots, 2^g - 1$ **do**
  $v_i \leftarrow H'(v_{i-1} \mid\mid v_{i-2})$ {initialize the memory}
end for
$v \leftarrow \Gamma(g, v, \gamma)$ {one layer with γ-based memory accesses}
$x \leftarrow F(v)$ {memory-hard function}
return x

We denote Catena as Catena-BRG when F is instantiated with BRH^g_λ (see Section 5.2) and as Catena-DBG when F is instantiated with DBH^g_λ (see Section 5.3). Thus, Catena-Dragonfly is a particular instance of Catena-BRG and Catena-Butterfly is a particular instance of Catena-DBG. For these instances, we fix the functions F, H, H', and Γ (see Table 3.1), where the functions $H = BLAKE2b$ and Γ (see Algorithm 4) are equal for all our instances. Furthermore, for Catena-Dragonfly-Full and Catena-Butterfly-Full, we set $H = H' = BLAKE2b$. Depending on the application, one can adapt the parameters $g_{\text{low}}, g_{\text{high}}, \lambda$, and γ, where we present our recommendations in Section 3.4.

Tweak. The parameter t is an additional multi-byte value which is given by:

$$t \leftarrow H(V) \mid\mid d \mid\mid \lambda \mid\mid m \mid\mid |s| \mid\mid H(\text{AD}),$$

where the first, n-bit value $H(V)$ denotes the hash of the unique version identifier V. An overview of the values V for our proposed instances is given in Table 3.1. The second,

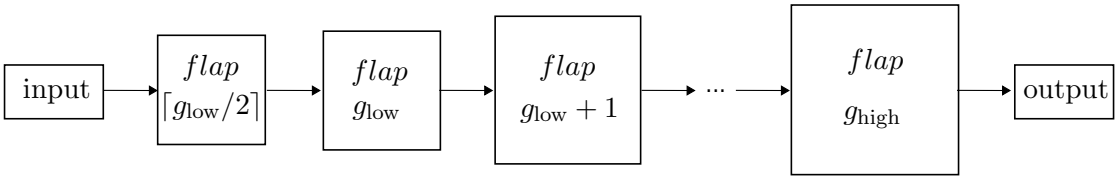

Figure 3.1.: The general idea of Catena applying the function *flap*.

Name	F	$\mathbf{H}'$	**V**
Catena-Dragonfly	BRH^g_λ	BLAKE2b-1	"Dragonfly"
Catena-Dragonfly-Full	BRH^g_λ	BLAKE2b	"Dragonfly-Full"
Catena-Butterfly	DBH^g_λ	BLAKE2b-1	"Butterfly"
Catena-Butterfly-Full	DBH^g_λ	BLAKE2b	"Butterfly-Full"

Table 3.1.: Overview of the default instances of Catena.

byte-sized value d denotes the domain (i.e., the mode) for which Catena is used. We fix $d = 0$ for the usage of Catena as a password scrambler, $d = 1$ when used as a key-derivation function (see Section 8.3), and $d = 2$ for the proof-of-work scenario (see Section 8.1). The remaining possible values for d are reserved for future applications. The third, byte-sized value λ defines, together with the garlic g_{high} (see above), the security parameters for Catena. The byte-sized value m denotes the output length of Catena in bytes, and the byte-sized value $|s|$ denotes the total length of the salt in bytes. The n-bit value $H(\text{AD})$ is the hash of the associated data AD, which can contain additional information like hostname, user-ID, name of the company, or the IP of the host, with the goal to customize the password hashes. Note that the order of the values does not matter as long as they are fixed for a certain application.

The tweak is processed together with the salt and the secret password (see Line 1 of Algorithm 1). Thus, t can be seen as a weaker version of a salt increasing the additional computational effort for an adversary when using different values. Furthermore, it allows to differentiate between diverse applications of Catena, and can depend on all possible input data. Note that one can easily provide unique tweak values (per user), when including the user-ID in the associated data.

3.2. Functional Properties

Garlic. Catena employs a graph-based structure, where the memory requirement highly depends on the number of input vertices of the permutation graph. Since the goal is to hinder an adversary to make a reasonable number of parallel password checks using the same memory, we have to consider a minimal number of input vertices. In

general, we use $G = 2^g$ input vertices, where g denotes the *garlic* parameter. Note that as in our default instances, we set $g_{\text{low}} = g_{\text{high}} = g$.

Client-Independent Update (CI-update). 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:

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

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. In the final iteration of the **for**-loop in Algorithm 1, the client has to omit the last invocation of the hash function H (see Line 5). 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 further thwart off-line attacks, 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. Note that K is the same for all users, and thus, has to be stored on server-side. To preserve the server-relief property (see above), we encrypt the output of Catena using the XOR operation with $H(K \mid\mid userID \mid\mid g_{\text{high}} \mid\mid K)$, which, under the reasonable assumption that the value $(userID \mid\mid g_{\text{high}})$ is a nonce, was proven to be CPA-secure in [39]. Let $X := \{pwd, t, s, g_{\text{low}}, g_{\text{high}}, m, \gamma\}$. Then, the output of Catena^K is computed as follows:

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

where Catena is defined as in Algorithm 1 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 $z = H(K \mid\mid userID \mid\mid g_{\text{high}} \mid\mid K)$.
2. Compute $x = y \oplus z$, where y denotes the current keyed hash value.
3. Update x, i.e., $x = H(g \mid\mid flap(g_{\text{high}}, x \mid\mid 0^*, \gamma))$ for $g \in \{g_{\text{high}} + 1, \ldots, g_{\text{high}} + r\}$.

4. Compute the new hash value $y = y \oplus H(K \mid\mid userID \mid\mid g_{\text{high}} + r \mid\mid K)$.

Remark. Obviously, it is a bad idea to store the secret key K together with the password hashes since it can be leaked 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 by making use of hardware security modules (HSM), 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 [13] for details). Another possibility is to derive K from a password during the bootstrapping phase. Afterwards, K will be kept in the 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.

3.3. Security Properties

Memory-Hardness. In Chapter 6, we present and discuss the results of Lengauer and Tarjan [30]. They analyzed the underlying structures which we use in our instances regarding to their memory-hardness. In short, Catena-BRG (see Section 5.2) provides a time-memory tradeoff of the form $S \cdot T = G^2$ (see Definition 2.2), where S denotes the memory, T the time, and $G = 2^g$ the garlic. On the other hand, Catena-DBG (see Section 5.3) is λ-memory-hard function, i.e., $S^\lambda \cdot T = G^{\lambda+1}$ (see Definition 2.3), where λ denotes the depth of the DBG_λ^g. The security analysis is based on the *pebble game* proof technique (see Section 2.1). This property enables Catena to thwart massively parallel adversaries.

Preimage Security. One major requirement for password scramblers is described by the preimage security, i.e., given a fresh password hash $h = \text{PS}(pwd)$, 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 4.1, we show that the preimage security of Catena depends on 1) the assumption that the underlying hash function H is a one-way function and 2) the entropy of the password (pwd).

Random-Oracle 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 4.2 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.

Cache-Time Resistance. From Definition 2.5, it follows that an algorithm is cache-time-resistant if its control flow does not depend on the input. One can easily see that Catena provides this property since it is based on the function F, whose control flow only depends on the security parameters g (garlic) and λ (depth). Given these two

parameters, it provides a predetermined memory-access pattern, which is independent from the secret input (pwd).

3.4. Parameter Recommendation

Hash Function. For the practical application of Catena, we looked 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. In any case, we assume that both the output size of H (and H') and the cache-line size are powers of two. So if, they are not equal, the bigger number is a multiple of the smaller one. Moreover, the output of H (and thus, H') should be byte-aligned. For Catena, we decided to use BLAKE2b [5] for H since its high performance in software allows to use a large value for the `garlic` g, resulting in a higher memory effort than for, e.g., SHA3-512 [7], and BLAKE2b-1 for H', which is actually one single round of BLAKE2b including finalization. Further versions of Catena may be instantiated with, e.g., SHA2-512 [36] since it is well-analyzed [2, 25, 29], standardized, and widely used, e.g., in `sha512crypt`, the common password scrambler in several Linux distributions [16].

Note that the security of Catena does not only rely on the performance of a specific hash function, but also on the size of the underlying graph (BRG^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 in the password-scrambling scenario, one can adapt the security parameter to reach the desired computational effort.

Remark: Our primary recommendation for the Password Hashing Competition (PHC) is BLAKE2b for H and BLAKE2b-1 for H'. Nevertheless, we highly encourage users to plug in their favorite cryptographic hash function such as SKEIN-512 [23] or SHA3-512 for H, and, if desired, a reduced version of these hash functions for H'.

Cost Parameter. Table 3.2 presents the recommended parameter sets for Catena (depending on the particular instance) when considering COTS systems. The parameter set for keyed password hashing is similar to the parameter set for key-less password hashing, plus an additional 128-bit key. For non-COTS systems, the parameter sets must be individually adjusted corresponding to the underlying hardware, e.g., for embedded systems one would choose smaller values g_{low} (minimum garlic) and g_{high} (garlic). Note that for semantic reasons we set the minimum time cost value $\lambda = 1$ and the minimum memory cost values $g_{\mathrm{high}} = g_{\mathrm{low}} = 1$. The reference implementation currently bonds these values to 255 and 63 for λ and g_{high}, respectively. Nevertheless, these upper bounds can change depending on the implementation.

Other instances. Users are free to choose other instances of Catena according to their specific needs. If, e.g., the defender's machine is constrained so, that even Catena-Butterfly does not fit into the memory, we recommend to use Catena-Butterfly

Password Hashing			
Algorithm	$g_{\text{low}}/g_{\text{high}}$	λ	**Time**
Catena-Dragonfly	21/21	2	0.36 sec
Catena-Dragonfly-Full	18/18	2	0.19 sec
Catena-Butterfly	16/16	4	0.28 sec
Catena-Butterfly-Full	14/14	4	0.38 sec
Key Derivation			
Catena-Dragonfly-Full	22/22	2	3.15 sec
Catena-Butterfly-Full	17/17	4	3.76 sec

Table 3.2.: Recommended parameter sets for COTS systems. All timings are measured on an Intel(R) Core(TM) i7-3930M CPU @ 3.20GHz system. For all instances we set $\gamma = s$.

with smaller g_{high}, and to increase λ accordingly. E.g., Catena-Butterfly with $g_{\text{high}} = 14$ would fit into about 1 MB of memory. With Catena-Axungia we provide a search tool for optimal parameters under given constraints. Catena-Axungia can be found on:

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

Moreover, note that both Catena-Dragonfly and Catena-Butterfly allow to leak the public γ by cache-timing attacks. 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.

Encoding. The parameter encoding table can be found in Table 3.3.

Implementation. A current reference implementation can be found on

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

This implementation was used to create the test vectors given in Appendix C. An implementation that allows extension and modification can be found on:

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

Parameter	Description	Encoding
g_p	garlic (password hashing)	1 byte
g_k	garlic (key derivation)	1 byte
λ	depth	1 byte
d	domain	1 byte
m	output length in bytes	1 byte
s	salt	byte string
$\|s\|$	salt length in bytes	1 byte

Table 3.3.: Parameter choices for the practical usage of Catena.

Chapter 4

Security Analysis of the Catena Framework

We denote a password scrambler to be secure if it provides at least 1-memory-hardness and preimage security. Furthermore, it should be resistant against cache-timing attacks. Catena-Butterfly (see Section 5.3) inherits its λ-memory-hardness (see Definition 2.3) from F, whereas Catena-Dragonfly (see Section 5.3) provides only 1-memory-hardness, i.e., memory-hardness (see Definition 2.2).

Since the memory-access pattern of Catena is independent from the password, it provides resistance against cache-timing attacks against the secret input. Finally, we show that Catena is a secure password scrambler that behaves like a good random function, which is useful for using Catena as a secure KDF.

4.1. Password-Recovery Resistance.

In this section we show that Catena is a good password scrambler, i.e., given the hash value h it is infeasible for an adversary to do better than trying out password candidates in likelihood order to obtain the correct password.

Theorem 4.1 (Catena is Password-Recovery Resistant). *Let m denote the min-entropy of a password source $\mathcal{Q}$. Then, it holds that*

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

Proof. Note that an adversary $\mathcal{A}$ can always guess a (weak) password by trying out about 2^m password candidates. For a maximum of q queries, it holds that the success probability is given by $q/2^m$. Instead of guessing 2^m password candidates, an adversary can also try to find a preimage for a given hash value h. It is easy to see from Algorithm 1

that an adversary thus has to find a preimage for H in Line 5. More detailed, for a given value h with $h \leftarrow H(g_{\text{high}} \mid\mid x)$, $\mathcal{A}$ has to find a valid value for x. The success probability for this can be upper bounded by $\mathbf{Adv}^{\text{pre}}_{H}(q,t)$. Our claim follows by adding up the individual terms. ■

4.2. Pseudorandomness.

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. Therefore, we model the internally used hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ as a random oracle. Note that the output length m, the depth λ, and the value g_{low} (minimum garlic) are constant values which are set once when initializing a system the first time.

Theorem 4.2 (PRF Security of Catena). *Let q denote the number of queries made by an adversary and s a randomly chosen salt value. Furthermore, let H be modelled as a random oracle and $g_{high} \geq g_{low} \geq 1$. Then, it holds that*

$$\mathbf{Adv}^{PRF}_{\text{Catena}}(q,t) \;\leq\; \frac{(q \cdot g_{high} + q)^2}{2^n} + \mathbf{Adv}^{coll}_{F}(g_{high} \cdot q).$$

Proof. Let $a^i = (pwd^i \mid\mid u^i \mid\mid s^i \mid\mid g_{\text{high}} \mid\mid \gamma^i)$ represent the i-th query, where pwd denotes the password, u denotes the tweak, s the salt, g_{high} 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 $flap(g_{\text{high}}, x \mid\mid 0^*, \gamma^j)$ of the j-th query (cf. Algorithm 1, Line 4). Then, $H(g_{\text{high}} \mid\mid y^j)$ is the output of Catena(a^j). In the case that $y^1, \ldots, y^q$ are pairwise distinct, an adversary $\mathcal{A}$ cannot distinguish $H(g_{\text{high}} \mid\mid \cdot)$ from a random function $\$(\cdot)$ since in the random-oracle model, both functions return a value chosen uniformly at random from $\{0,1\}^n$.

Therefore, we have to upper bound the probability of the event $y^i = y^j$ with $i \neq j$. Due to the assumption that $\mathcal{A}'s$ queries are pairwise distinct, there must be at least one collision for H or $flap$. For q queries, we have at most $q(g_{\text{high}} + 1)$ invocations of H. Thus, we can upper bound the collision probability by

$$\frac{(q \cdot g_{\text{high}} + q)^2}{2^n}.$$

Furthermore, we have $q \cdot g_{\text{high}}$ invocations of the memory-consuming function $flap$. We can upper bound the probability of a collision by $\mathbf{Adv}^{\text{coll}}_{F_\lambda}(g_{\text{high}} \cdot q)$. Our claim follows from the union bound. ■

Chapter 5

Instances

In this section, we introduce two concrete instances: Catena-Dragonfly and Catena-Butterfly. Prior, we describe the function SaltMix which is shared by both instances.

5.1. SaltMix

The purpose of the function Γ from Algorithm 2 (see Chapter 3) is to update the state array v by accessing its elements in this salt-dependent manner. Usual time-memory tradeoffs could exploit predictable memory accesses. Therefore, we introduced a random-access layer that bases on a public input, e.g., the salt. This would render the implementation of time-memory tradeoffs much more difficult. Furthermore, we claim that such a construction would also increase the costs of attacks with expensive non-reprogrammable hardware: either, adversaries would have to design a new computational circuit for each salt, or to support multiple salts – which would turn them into almost "reprogrammable" hardware.

The function SaltMix, as defined in Algorithm 3, is our proposed instantiation for Γ. In each iteration of the for-loop (see Lines 12-16), two random g-bit values j_1 and j_2 are computed using a random-number generator (the `xorshift1024star` function on the left side in Algorithm 2). There, j_1 determines the index for both the updated word and the first input, and j_2 that of the second input (see Line 15). The use of H may appear as a natural choice for random-number generator. Though, due to its better performance, we decided for the non-cryptographic but statistically sound `xorshift1024star` RNG [49], and used H only to seed it. The seed is given by the 1024-bit value $(H(s) \mid\mid H(H(s))$ as shown in Line 11 of Algorithm 4. To further increase the performance, SaltMix updates only $2^{\lceil 3g/4 \rceil}$ (out of 2^g) values of the state v. Note that the state array v consists of 512-bit values, whereas the 1024-bit state p used in the `xorshift1024star` functions consists of sixteen 64-bit words $s_0, \ldots, s_{15}$.

Algorithm 3 The functions SaltMix (left) and `xorshift1024star` (right).

SaltMix

Input: g {Garlic}, v {State}, s {Salt}
Output: v {Updated State}

10: $r \leftarrow H(s) \mid\mid H(H(s))$
11: $p \leftarrow 0$
12: **for** $i \leftarrow 0, \ldots, 2^{\lceil 3g/4 \rceil} - 1$ **do**
13: $\quad (j_1, r, p) \leftarrow$ `xorshift1024star`(r, p)
14: $\quad (j_2, r, p) \leftarrow$ `xorshift1024star`(r, p)
15: $\quad v_{j_1} \leftarrow H'(v_{j_1} \mid\mid v_{j_2})$
16: **end for**
17: **return** v

`xorshift1024star`

Input: r, p
Output: idx {Current Index}

20: $s_0 \leftarrow r_p$
21: $p \leftarrow (p + 1) \bmod 16$
22: $s_1 \leftarrow r_p$
23: $s_1 \leftarrow s_1 \oplus (s_1 \ll 31)$
24: $s_1 \leftarrow s_1 \oplus (s_1 \gg 11)$
25: $s_0 \leftarrow s_0 \oplus (s_0 \gg 30)$
26: $r_p \leftarrow s_0 \oplus s_1$
27: $idx \leftarrow r_p \cdot 1181783497276652981$
28: **return** $(idx \gg (64 - g), r, p)$

5.2. Catena-Dragonfly

For Catena-Dragonfly, the function $flap$ (see Line 4 of Algorithm 1) is instantiated by SaltMix for the function Γ, and by (g, λ)-Bit-Reversal Hashing (BRH^g_λ), as defined in Algorithm 4, for the function F. In the remainder of this section, we discuss the origin of the function BRH^g_λ, where v^j_i denotes the i-th vertex of the j-th bit-reversal graph.

Algorithm 4 (g, λ)-Bit-Reversal Hashing (BRH^g_λ).

Input:
g {Garlic},
v {State Array},
λ {Depth}
Output: x {Password Hash}

1: **for** $k \leftarrow 1, \ldots, \lambda$ **do**
2: $\quad r_0 \leftarrow H(v_{2^g-1} \mid\mid v_0)$
3: $\quad$ **for** $i = 1, \ldots, 2^g - 1$ **do**
4: $\quad\quad r_i \leftarrow H'(r_{i-1} \mid\mid v_{\tau(i)})$
5: $\quad$ **end for**
6: $\quad v \leftarrow r$
7: **end for**
8: **return** r_{2^g-1}

Definition 5.1 (Bit-Reversal Permutation τ). *Fix a number $k \in \mathbb{N}$ and represent $i \in \mathbb{Z}_{2^k}$ as a binary k-bit number, $(i_0, i_1, \ldots, i_{k-1})$. The bit-reversal permutation $\tau : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ is defined by*

$$\tau(i_0, i_1, \ldots, i_{k-1}) = (i_{k-1}, \ldots, i_1, i_0).$$

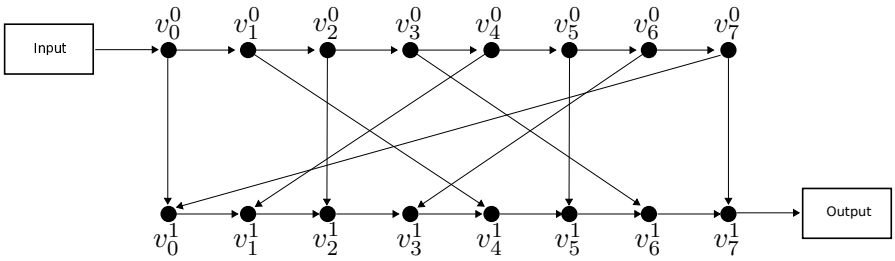

Figure 5.1.: A $(3, 1)$-bit-reversal graph (BRG_1^3).

The bit-reversal permutation τ defines the (g, λ)-Bit-Reversal Graph (BRG_λ^g).

Definition 5.2 ((g, λ)-Bit-Reversal Graph). *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 (g, λ)-bit-reversal graph $BRG_\lambda^g(\mathcal{V}, \mathcal{E})$ consists of $(\lambda + 1) \cdot 2^g$ vertices*

$$\{v_0^0, \ldots, v_{2^g-1}^0\} \cup \{v_0^1, \ldots, v_{2^g-1}^1\} \cup \cdots \cup \{v_0^{\lambda-1}, \ldots, v_{2^g-1}^{\lambda-1}\} \cup \{v_0^\lambda, \ldots, v_{2^g-1}^\lambda\},$$

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

- *$(\lambda + 1) \cdot (2^g - 1)$ edges $v_{i-1}^j \to v_i^j$ for $i \in \{1, \ldots, 2^g - 1\}$ and $j \in \{0, 1, \ldots, \lambda\}$.*
- *$\lambda \cdot 2^g$ edges $v_i^j \to v_{\tau(i)}^{j+1}$ for $i \in \{0, \ldots, 2^g - 1\}$ and $j \in \{0, 1, \ldots, \lambda - 1\}$.*
- *λ additional edges $v_{2^g-1}^j \to v_0^{j+1}$ where $j \in \{0, \ldots, \lambda - 1\}$.*

For example, Figure 5.1 illustrates an BRG_1^3. A BRG_4^3 can be seen in Appendix D. Note that this graph is almost identical – except for one additional edge $e = (v_7^0, v_0^1)$ – to the bit-reversal graph presented by Lengauer and Tarjan in [30].

Bit-Reversal Hashing. The (g, λ)-Bit-Reversal Hashing function is defined in Algorithm 4. It requires $\mathcal{O}(2^g)$ invocations of a given hash function H' for a fixed value of x. The three inputs g, x, and λ of BRH_λ^g represent the garlic $g = \log_2(G)$, the value to process, and the depth, respectively. Thus, g specifies the required units of memory. Moreover, incrementing g by one doubles the time and memory effort for computing the password hash.

5.3. Catena-Butterfly

For Catena-Butterfly, the function $flap$ (see Line 4 of Algorithm 1) is instantiated by SaltMix for the function Γ, and by (g, λ)-Double-Butterfly Hashing (DBH_λ^g), as

defined in Algorithm 5, for the function F. In the remainder of this section, we discuss the origin of the function DBH^g_λ, which is based on a stack of λ G-superconcentrators.

Algorithm 5 (g, λ)-Double-Butterfly Hashing (DBH^g_λ).

Input:
g {Garlic},
v {State Array},
λ {Depth}
Output: x {Password Hash}

for $k = 1, \ldots, \lambda$ **do**
  for $j = 1, \ldots, 2g - 1$ **do**
    $r_0 \leftarrow H(v_{2^g-1} \oplus v_0 \mid\mid v_{\sigma(g,j-1,0)})$
    for $i = 1, \ldots, 2^g - 1$ **do**
      $r_i \leftarrow H'(r_{i-1} \oplus v_i \mid\mid v_{\sigma(g,j,i-1)})$
    end for
    $v \leftarrow r$
  end for
end for
return v_{2^g-1}

The following definition of a G-superconcentrator is a slightly adapted version of that introduced in [30].

Definition 5.3 (G-Superconcentrator). *A directed acyclic graph $\Pi(\mathcal{V}, \mathcal{E})$ 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 k such that $1 \leq k \leq G$ and for every pair of subsets $V_1 \subset \mathcal{V}$ of k inputs and $V_2 \subset \mathcal{V}$ of k outputs, there are k vertex-disjoint paths connecting 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 back-to-back placed Fast Fourier Transformations [12]. More detailed, it is a representation of twice the Cooley-Tukey FFT algorithm [14] omitting one row in the middle (see Figure 5.2 for an example where $g = 3$). Therefore, a DBG consists of $2 \cdot g$ rows.
Based on the DBG, we define the sequential and stacked (g, λ)-double-butterfly graph. In the following, we denote $v^k_{i,j}$ as the i-th vertex of the j-th row of the k-th double-butterfly graph.

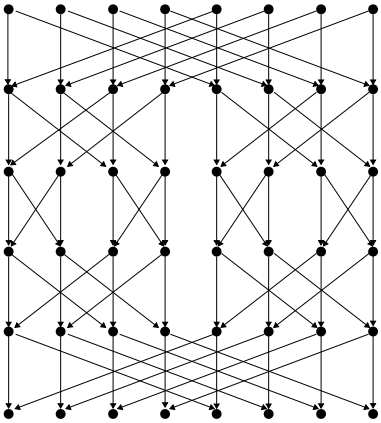

Figure 5.2.: A Cooley-Tukey FFT graph with eight input and output vertices.

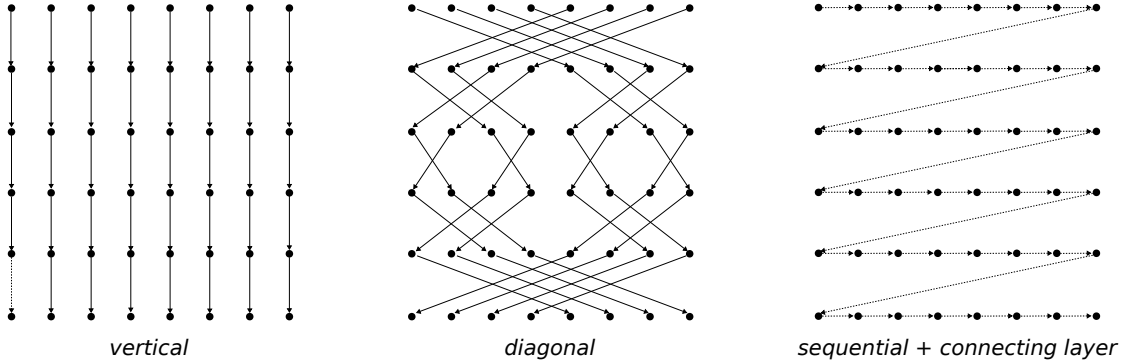

Figure 5.3.: Types of edges as we use them in our definitions.

Definition 5.4 ((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*

- $\{v^k_{0,0}, \ldots, v^k_{2^g-1,0}\} \cup \ldots \cup \{v^k_{0,2g-2}, \ldots, v^k_{2^g-1,2g-2}\}$ *for* $1 \leq k \leq \lambda$ *and*
- $\{v^\lambda_{0,2g-1}, \ldots, v^\lambda_{2^g-1,2g-1}\}$,

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

- *vertical: $2^g \cdot (\lambda \cdot (2g-1))$ edges*

$$(v^k_{i,j}, v^k_{i,j+1}) \text{ for } 0 \leq j \leq 2g-2, 0 \leq i \leq 2^g-1, \text{ and } 1 \leq k \leq \lambda,$$

- *diagonal: $2^g \cdot \lambda \cdot g \; + 2^g \cdot \lambda \cdot (g-1)$ edges*

$$(v^k_{i,j}, v^k_{i \oplus 2^{g-1-i}, j+1}) \text{ for } 0 \leq j \leq g-1, 0 \leq i \leq 2^g-1, \text{ and } 1 \leq k \leq \lambda.$$
$$(v^k_{i,j}, v^k_{i \oplus 2^{i-(g-1)}, j+1}) \text{ for } g \leq j \leq 2g-2, 0 \leq i \leq 2^g-1, \text{ and } 1 \leq k \leq \lambda.$$

- *sequential: $(2^g-1) \cdot (\lambda \cdot (2g-1)+1)$ edges*

$$(v^k_{i,j}, v^k_{i+1,j}) \quad \text{for} \quad 1 \leq j \leq 2g-1, 0 \leq i \leq 2g-2, 1 \leq k \leq \lambda, \text{ and}$$
$$(v^\lambda_{i,2g-1}, v^\lambda_{i+1,2g-1}) \quad \text{for} \quad 0 \leq i \leq 2g-2$$

- *connecting layer: $\lambda \cdot (2g-1)$ edges*

$$(v^k_{2^g-1,j}, v^k_{0,i+1}) \quad \text{for} \quad 1 \leq k \leq \lambda, \quad 0 \leq j \leq 2g-2.$$

In Appendix E you can see a DBG^3_2. Figure 5.3 illustrates the individual types of edges we used in our definition above. Moreover, an example for $g = 3$ and $\lambda = 1$ can be seen in Figure 5.4.

Double-Butterfly Hashing. The (g, λ)-double-butterfly hashing operation is defined in Algorithm 5. The structure is based on a (g, λ)-double-butterfly graph. Note that the function σ (see Lines 3 and 5) is given by

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

Thus, σ determines the indices of the vertices of the diagonal edges (see Figure 5.3).

Since the security of Catena in terms of password hashing is based on a time-memory tradeoff, it is desired to implement it in an efficient way, making it possible to increase the required memory. We recommend to use BLAKE2b [5] as the underlying hash function,

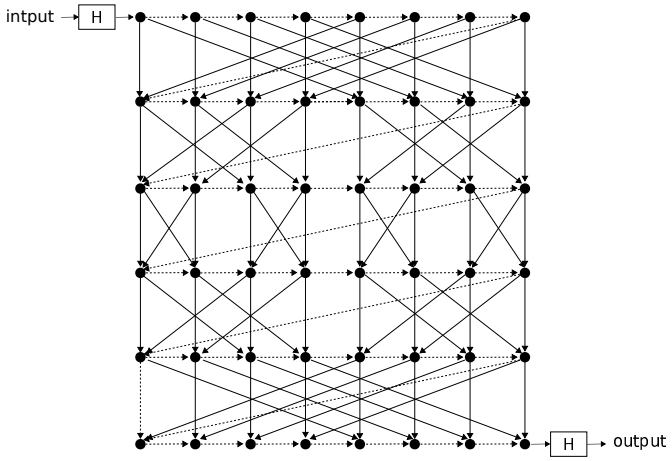

Figure 5.4.: A $(3,1)$-double-butterfly graph (DBG_1^3).

implying a block size of 1024 bits with 512 bits of output. Thus, it can process two input blocks within one compression function call. This is suitable for the BRH_λ^g operation since a bit-reversal graph satisfies a fixed indegree of at most 2. When considering the DBH_λ^g 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 require two compression function calls, which is a strong slow-down in comparison to BRH_λ^g. Therefore, we compute $H(X,Y,Z) = H(X \oplus Y \mid\mid Z)$ instead of $H(X,Y,Z) = H(X \mid\mid Y \mid\mid Z)$ obtaining the same performance as for the BRH_λ^g operation per hash function call. Obviously, this doubles the probability of an input collision. Nevertheless, for a 512-bit hash function, the success probability for a collision of an adversary is still negligible. Based on the approach above, the number of hash function calls to compute Row r_i from Row r_{i-1} is the same for BRH_λ^g and DBH_λ^g. Moreover, for both operations it holds that the number of hash function calls is equal to the number of compression function calls (when used with BLAKE2b). More detailed, the BRH_λ^g operation requires $2^g - 1 + \lambda \cdot 2^g$ calls to H (or H') and the DBH_λ^g operation requires $2^g - 1 + \lambda \cdot (2g-1) \cdot 2^g$ calls to H (or H'). It is easy to see, that the performance of Catena-DBG (and thus Catena-Butterfly) in comparison to Catena-BRG (and thus Catena-Dragonfly) is decreased by a logarithmic factor.

Remark. Note that the performance optimization discussed above has no influence on the λ-memory hardness of the DBH_λ^g operation since the first input $X \oplus Y$ is given by XORing vertices from the *sequential* or *connecting layer* and the *vertical* layer. In Chapter 6 we discuss the results of [30] who have shown that even without the sequential input, the DBH_λ^g operation provides λ-memory-hardness. Thus, adding additional inputs operation does not invalidate their results. The objective of the sequential layer is to thwart the possibility of computing DBH_λ^g in parallel.

Chapter 6

Security Analysis of Catena-BRG and Catena-DBG

In this section we discuss the security of Catena-BRG and Catena-DBG against side-channel attacks. Furthermore, we discuss the memory-hardness and pseudorandomness of both instances. Note that to prove memory-hardness of an instantiation of Catena it is sufficient to elaborate on the particular instantiation of the function F. Moreover, each of our particular instances (Catena-Dragonfly and Catena-Butterfly) inherits the security from Catena-BRG or Catena-DBG (depending on the instantiation of F).

6.1. Resistance Against Side-Channel Attacks

Straightforward implementations of either Catena-BRG or Catena-DBG provide neither a password-dependent memory-access pattern nor password-dependent branches. Therefore, both instances are resistant against cache-timing attacks (see Definition 2.5).

Considering a malicious garbage collector (see Definition 2.7), each of Algorithms 4 and 5 exposes the arrays v and r. Both arrays are overwritten multiple times (depending on the choice of λ). Lets consider both instances with $g = 3$ and $\lambda = 1$. Then, for BRH_2^3, the array v is overwritten twice and the array r once, whereas for DBH_2^3, v is overwritten 10 times and r nine times. Even for $\lambda = 1$, Catena-DBG is resistant against garbage-collector attacks and furthermore, it follows that *any variant of* Catena *with some fixed* $\lambda \geq 2$ *is at least as resistant to garbage-collector attacks as the same variant with* $\lambda - 1$ *in the absence of a malicious garbage collector.*

Remark. Note that cache-timing and garbage-collector attacks have even more severe consequences. They do not only speed-up regular password-guessing attacks where the password hash is already in possession of the adversary. They also enable an adversary $\mathcal{A}$ to recover a password without knowing the password hash at all by just verifying the

memory-access pattern.

6.2. Memory-Hardness

The memory-hardness of an algorithm which can be represented as a DAG with bounded indegree, can be shown by "playing" the pebble game (see Section 2.1). Here, we restate and discuss the results presented by Lengauer and Tarjan in [30].

Catena-BRG. In [30], Lengauer and Tarjan have proven the lower bound of pebble movements for a $(G, 1)$-bit-reversal graph.

Theorem 6.1 (Lower Bound for a BRG_1^g [30]). *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*

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

Biryukov and Khovratovich have shown in [9] that stacking more than one bit-reversal graph only adds some linear factor to the quadratic time-memory tradeoff. Hence, a BRG_λ^g with $\lambda > 1$ does not achieve the properties of a λ-memory-hard function.

Catena-DBG. Likewise, the authors of [30] analyzed the time-memory tradeoff for a stack of λ G-superconcentrators. Since the double-butterfly is a special form of a G-superconcentrators, their bound also holds for DBG_λ^g.

Theorem 6.2 (Lower Bound for a (G, λ)-Superconcentrator [30]). *Pebbling a (G, λ)-superconcentrator using $S \leq G/20$ black and white pebbles requires T placements such that*

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

Discussion. For scenarios where a quadratic time-memory tradeoff is sufficient, we recommend the efficient Catena-BRG (i.e., its particular instance Catena-Dragonfly) with either $\lambda = 1$ or – if garbage-collector attacks pose a relevant threat – with $\lambda = 2$. Note that the benefit of greater values for λ is very limited since the costs for pebbling the bit-reversal graph remain quadratic. For scenarios that require a higher time-memory tradeoff, we highly recommend the λ-memory-hard Catena-DBG (i.e., its particular

instance Catena-Butterfly) with $\lambda = 4$, which is sufficient for most practical applications. A detailed parameter recommendation can be found in Section 3.4.

We have to point out that the computational effort for DBH^g_λ 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 logarithmic with G. Thus, it remains an open research problem to find a (G, λ)-superconcentrator – or any other λ-memory-hard function – that can be computed more efficiently than a DBH^g_λ.

6.3. Pseudorandomness

For proving the pseudorandomness of Catena-BRG and Catena-DBG, we refer to the definition Random-Oracle Security which was introduced in Section 2.2 (see Definition 2.6). Therefore, we set $H = H'$ and model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle. Furthermore, for simplicity, we do not include the function Γ (see Line 7 of Algorithm 2) since it is a user-chosen function and can even be neglected, i.e., it can be the identity function.

Theorem 6.3 (Collision Security of BRH^g_λ). *Let q denote the number of queries made by an adversary and s a randomly chosen salt value. Furthermore, let H be modelled as a random oracle. Then, we have*

$$\mathbf{Adv}^{coll}_{BRH^g_\lambda}(q,t) \;\leq\; \frac{(q\cdot(\lambda+1))^2}{2^{n-2g}}.$$

Proof. It is easy to see from Algorithm 4 that a collision $\mathrm{BRH}^g_\lambda(x) = \mathrm{BRH}^g_\lambda(x')$ 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 2^g invocations of H in Lines 4-5 (initialization) of Algorithm 2. In addition, there are $\lambda \cdot 2^g$ invocations in Lines 2-4 of Algorithm 4 leading to a total of $(\lambda+1)\cdot 2^g$ invocations of H. Since H is modelled as a random oracle, we can upper bound the collision probability for q queries by

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

Thus, our claim follows. ∎

Finally, 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 6.4 (Collision Security of DBH^g_λ). *Let q denote the number of queries. Furthermore, let H be modelled as a random oracle for some fixed integers $g_{high}, g_{low}, \lambda \geq 1$ with $g_{high} \geq g_{low}$ and $G = 2^{g_{high}}$. Then, it holds that*

$$\mathbf{Adv}^{coll}_{DBH^g_\lambda}(q,t) \;\leq\; \frac{(q\cdot\lambda\cdot g_{high})^2}{2^{n-2g_{high}-3}}.$$

Proof. It is easy to see from Algorithm 5 that a collision $\mathrm{DBH}^g_\lambda(x) = \mathrm{DBH}^g_\lambda(x')$ for $x \neq x'$ implies either an input or 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 5 leads to a collision of the output of H, 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 4-5 (initialization) of Algorithm 2. In addition, there are $\lambda \cdot (2g-1) \cdot 2^g$ invocations in Lines 3-5 of Algorithm 5 leading to a total of $\lambda \cdot 2g \cdot 2^g$ invocations of H^t. Since H (and thus H^t) is modelled 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 5 can occur:

$$v^a_{2^g-1} \oplus v^a_0 = v^b_{2^g-1} \oplus v^b_0 \qquad \text{(Algorithm 5, Line 3)}$$

or

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

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. ■

Remark. The proof presented here does clearly hold for $H = H'$, which is our recommendation for the usage of Catena as a key-derivation function. Nevertheless, for password hashing, we iterate $H' =$ BLAKE2b-1 thousands of times and have implemented it in a compatible way to H, i.e., 12 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 12 times the application of H' provides similar security as one invocation of H. But, since H' is a user-chosen parameter, this assumption can obviously change for different instances of Catena.

Chapter 7

Design Discussion

In this section, we give an informal overview over the main observations and ideas that lead to the development of Catena.

7.1. Default instances

Catena is a *framework* that allows users to choose their own instantiation that suits their specific needs best. Making the proper choice may be a 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 two default instances of Catena:

Catena-Butterfly: This is based on several layers of Catena-DBG, i.e., a λ-memory-hard function for $\lambda > 1$, where each layer is somewhat slow (time proportional to $g \cdot 2^g$ for garlic g). Thus, the amount S of memory which can be claimed in a given amount of time is limited (typically less than 10 MB). Catena-Butterfly should be used

- by defenders who cannot afford to use a huge amount of memory for password hashing, anyway,
- mainly as a defense against typical "memory-constrained" adversaries who cannot afford S units of memory for many parallel cores.

Catena-Dragonfly: This is based on Catena-BRG, and therefore "only" memory-hard. Since a layer of Catena-BRG can be executed much faster than a layer of Catena-DBG (time proportional to 2^g rather than $g \cdot 2^g$ for garlic g), this allows to allocate much more memory (e.g., 100 MB or more). Catena-Dragonfly shall be used

- by defenders who can afford to allocate so much memory, and

- as a technique to maximize the cost of password cracking even for high-budget adversaries.

Note that the version that a user should choose depends on her machine, on the usage scenario, and on the assumed adversary to defend against.

Defender's machine. Catena has been designed to run on any machine, from cheap smartphones to high-end servers. The large-memory variant Catena-Dragonfly may not always be acceptable, either because that amount of memory unavailable, or because allocating too much memory would hinder other running processes on the same machine. However, if at least one MB of storage is available for password scrambling, Catena-Butterfly should run well. If this is too fast, we recommend increasing the pepper value, or, maybe preferably, the value of λ.

Usage scenarios. Three typical usage scenarios for a password scrambler PS are the following:

1. *Password-based authentication.* Given a triple $(username, salt, PS(salt, pwd))$, a user with a given username is authenticated by typing a password pwd', such that

$$PS(salt, pwd') = PS(salt, pwd).$$

   This can be used to log in at a server, providing a service for a multitude of different users, or at a machine servicing a single user.

2. *Password-based key derivation.* A value

$$key := PS(salt, pwd)$$

   is used as a cryptographic key.

3. *Proof of work / proof of space.* The prover has to find a specific X, such that $PS(X)$ satisfies a statistically rare property p. This means, the prover searches for

$$\text{an input } X \text{ such that } p(PS(X)) = 1 \text{ holds.}$$

   This is supposed to be a challenging task for the prover, while the verifier, given pwd, just needs to compute $PS(pwd)$ once and then check the property. As a simple example, $p(y) = 1$ could mean "the n least significant bits of y are 0". For a well-designed password scrambler, the prover should need to call PS about 2^n 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) [47]. They also have been used for cryptocurrencies [31].

Regarding the first scenario, security against cache-timing and garbage-collector attacks is highly important. It has recently been shown that cache-timing attacks can be used to perform attacks on virtual machines, even if the attack program is running on a different core than the defender's program [27].

Also regarding the first scenario, maximizing the amount of used memory may not always be feasible. On some low-end systems, so much memory may just be unavailable. On high-end servers, allocating so much memory may hinder other processes running on the same server. In fact, a log-in process requiring too much memory may ease denial-of-service attacks.

For the remaining two scenarios, cache-timing and garbage-collector attacks are less of an issue, and large amounts of memory should usually be available for the defender. Thus, the first scenario may require Catena-Butterfly, while the second and third may benefit from Catena-Dragonfly.

Adversaries' capabilities. We distinguish adversaries by the hardware they are using for their attacks:

1. *Typical password crackers*, pragmatically using cheap off-the-shelf hardware for their purpose, e.g., GPUs.
2. *Low-cost hardware-based adversaries*, using low-cost reprogrammable hardware, e.g., FPGAs.
3. *Clock-cycle thieves*, performing password-cracking on a bot-net with hundreds or thousands of machines.
4. *High-end adversaries* with a multi-million budget, who can afford to build dedicated hardware, e.g., ASICs.

Regarding the first two types of adversaries, the limited memory allocated by Catena-Butterfly is likely to cause trouble for a massively parallel adversary. The third type of adversary would be kind of indifferent to the choice between Catena-Butterfly and Catena-Dragonfly since the range of machines being part of a bot-net is similar to the range of defenders' machines Catena has been designed for. To maximize the fourth adversary's cost, we would recommend the usage of Catena-Dragonfly with maximal memory, if the defender can afford it.

7.2. Justification of the Generic Design

Catena can be seen as a mode of operation for a cryptographic hash function H, an eventually reduced version H' of a cryptographic hash function, and a (λ)-memory-hard function $flap$, and therefore, it fulfills the properties of a generic design. Alternatively, one can design a primitive password scrambler of its own right, with the structure of Catena, but internally using something similar to the round- or step-function of a

cryptographic primitive. This approach would lead to a faster but less flexible password scrambler which would allow to allocate more memory and therefore – eventually – to hinder the adversary more.

Advantages of our Generic Design. Catena inherits the security assurance and the cryptanalytic attempts from the underlying hash function and the function $flap$, whereas a primitive password scrambler could not inherit security assurance from an underlying primitive. Furthermore, Catena is easy to analyze since the underlying structure is defined by a cryptographic primitive and a well-analyzed graph-based structure. Therefore, cryptanalysts can benefit from decades of experience. Finally, it is quite easy to replace the cryptographic hash function, e.g., for performance or security issues, which leads to incompatible variants of Catena. This *diversity* can frustrate well-funded adversaries using fast but expensive non-programmable hardware for password-cracking: For each variant of Catena, they must build new hardware – or have to adapt existing hardware.

Disadvantages of a Primitive Password Scrambler. Note that a primitive password scrambler would actually be *a new type of primitive*. Thus, cryptographers would have to develop new methods for cryptanalysis, and understand new attack surfaces, such as 1) the garbage-collector attack and 2) disproving lower bounds from the pebble game. This would be a scientifically interesting development, and we hope some people will actually design primitive password scramblers for PHC. But this would add more years to the time to wait before deploying the new password scrambler since many cryptographic primitives have been broken within a few years after their publications. Primitives that have been deeply analyzed without researchers finding an attack gain confidence in their security, over the years. Note that it is not sufficient to just *wait* a couple of years before the adoption of a new primitive. One needs to *catch the cryptanalysts' attention* and make them try to find attacks against the primitive.

Chapter 8

Usage

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

8.1. Catena for Proof of Work

The concept of proofs of work was introduced by Dwork and Naor [18] in 1992. The principle design goal was to combat junk mail under the usage of CPU-bounded functions, i.e., the goal was 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* " [18]. 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 [47].

As an advancement to CPU-bound function,Abadi et al. [1], and Dwork et al. [17] considered moderately hard, memory-bound functions, since memory access speeds do not vary so much on different machines like CPU accesses. Therefore, they may behave more equitably than CPU-bound functions. These memory-bound function base on a large table which is randomly accesses during the execution, causing a lot of cache misses. Dwork et al. presented in [19] a compact representation for this table by using a time-memory trade-off for its generation. Dziembowski et al. [21] as well as Ateniese et al. [3] 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 [21], the authors proposed a new scheme using "graphs with high pebbling complexity and Merkle hash-trees".

For Catena, there exist at least two possible attempts 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 introduced instances.

Guessing Secret Bits (Pepper). At the beginning, S chooses fixed values for pwd, t, s and g_{high}, where s denotes a randomly chosen k-bit salt value, where p bits of s are secret, i.e., p-bit pepper with $p \leq k$. Then, S computes $h = \textsc{Catena}(pwd, t, s, g)$ and sends the tuple $(pwd, t, s_{[0,k-p-1]}, g_{\text{high}}, h, p)$ to C, where $s_{[0,k-p-1]}$ denote the $k-p$ least significant bits of s (the public part). Now, C has to guess the secret bits of the salt by computing $h' = \textsc{Catena}(pwd, t, s', g_{\text{high}})$ about 2^p times and comparing if $h = h'$. If so, C 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 an e-bit password pwd, a tweak t, a salt s, and the garlic g_{high}. Then, S computes $h = \textsc{Catena}(pwd, t, s, g_{\text{high}})$ and sends the tuple $(t, s, g_{\text{high}}, e, h)$ to C. The client C then has to guess the password by computing about 2^e times $h' = \textsc{Catena}(pwd', t, s, g_{\text{high}})$ for different values of pwd', and comparing if $h' = h$. If so, C gains access to S. The effort of C is given by about 2^e computations of Catena (and about 2^e comparisons for $h = h'$). Hence, in this case the effort of C is scalable by adapting the length e of the password. Furthermore, S can adjust the effort of C by excluding e from the tuple sent to C. Then, since C does not know the length of the original password, the time for finding pwd' with $pwd' = pwd$ highly depends on the way C performs password cracking. Note that the latter may not really be suitable for the proof-of-work scenario since a prover with experience in password cracking can access the server significantly faster than a non-expert.

8.2. Catena in Different Environments

Backup of User-Database. When maintaining a database of user data, e.g., password hashes, a storage provider (server) 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 unwanted publication. Therefore, we highly recommend to use Catena in the keyed password hashing mode (see Section 3.2). 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, i.e., it must not be stored together with the backup.

Using Catena with Multiple Number of Cores. Catena is initially designed to run on a modern single-core machine. To make use of 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 choose $p = \log_2(b)$. During the login process, the i-th core will then compute the value $h_i = \textsc{Catena}(pwd, t, s_{0,\ldots,|s|-2} \mid\mid i, g_{\text{high}})$ 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

Algorithm 6 Catena-KG

Input: pwd {Password}, t' {Tweak}, s {Salt}, g_{low} {Min Garlic}, g_{high} {Garlic}, m {Output Length}, γ {Public Input}, ℓ {Key Size}, $\mathcal{I}$ {Key Identifier}
Output: k {ℓ-Bit Key Derived from the Password}

1: $x \leftarrow \text{Catena}(pwd, t', s, g_{\text{low}}, g_{\text{high}}, m, \gamma)$ {with $m = |H(\cdot)|$}
2: $k \leftarrow \emptyset$
3: **for** $i = 1, \ldots, \lceil \ell/n \rceil$ **do**
4: $\quad k \leftarrow k \mid\mid H(i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$
5: **end for**
6: **return** $truncate(k, \ell)$ {truncate k to the first ℓ bits}

adversary, since it has to try all possible values for the pepper p to rule out a password candidate.

Low-Memory Environments. The application of the server relief technique leads to significantly reduced effort on the side of the server for computing the output of Catena by splitting it 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, e.g., social networks. Then, each client has to compute an intermediate hash $y = P(\cdot)$ and the server only has to compute $h = H(y)$ for each y, i.e., for each user.

On the other hand, e.g., if Catena is used in the proof-of-work scenario, i.e., a client has to proof that it took a certain amount of time and memory to compute the output of Catena, the application of server relief does not make sense.

8.3. 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 6). To provide uniqueness of the inputs, the domain value d of the tweak is set to 1, i.e., the tweak 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 is makes no sense to give the user control over the output length m of Catena. It has only control over the output of Catena-KG by adapting ℓ. Thus, within Catena-KG, the value for m is set to default, i.e., 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 one-byte *key identifier* $\mathcal{I}$, and a parameter 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 [20]. 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. 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 [48], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, for Catena-KG, we recommend to use $g_{\text{high}} = 22$ (when instantiated with a BRG^g_λ) and $g_{\text{high}} = 17$ (when instantiated with a DBG^g_λ).

Security Analysis. It is easy to see that Catena-KG inherits its memory-hardness from Catena (see Chapter 6, Theorems 6.1 and 6.2) since it invokes Catena (Line 1 of Algorithm 6). Next, we show that Catena-KG is a good pseudorandom function (PRF) in the random oracle model. Again, as in the proof presented in Section 6.3, we assume that $H = H'$.

Theorem 8.1 (PRF Security of Catena-KG). *In the random oracle model we have*

$$\begin{aligned}\mathbf{Adv}^{PRF}_{\text{Catena-KG}}(q,t) &= \left|\Pr[A^{\text{Catena-KG}} \Rightarrow 1] - \Pr[A^{\$} \Rightarrow 1]\right| \\ &\leq \frac{(q \cdot g_{high} + q)^2}{2^n} + \mathbf{Adv}^{coll}_F(q \cdot g_{high}).\end{aligned}$$

Proof. Suppose that H is modeled as random oracle. For the sake of simplification, we omit the truncation step and let the adversary always get access to the untruncated key k. 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 always invoked with a fresh input (see Line 4, Algorithm 6). The only chance for an adversary to distinguish Catena-KG$(\cdot)$ from the random function $\$(\cdot)$ is a collision in Catena. The probability for this event can be upper bounded by similar arguments as in the proof of Theorem 4.2. ■

Chapter 9

Lessons Learned: The Tweak

After publishing Catena at the IACR ePrint server, and even more after submitting Catena to the first round of the Password Hashing Competition (PHC), we received a lot of suggestions, potential improvements, critique, ect. We are grateful for this input, which turned out to be a valuable resource for ideas to improve Catena.

First-Round. The core operation of first-round Catena worked as follows:

1. **Parameters.** Determine the storage size 2^g, and a cryptographic primitive H.
2. **Initialization.** Derive v_0 from $H(x)$ and v_i from v_{i-1}, where x denotes the hash of the tweak, the password, and the salt.
3. **Memory-Hard Transformation.** Transform $v_0, \ldots, v_{2^g-1}$ into new values, based on the memory-hard bit-reversal structure.
4. **Output.** $H(v_{2^g-1})$.

Proposed Tweak. The main differences between first-round-Catena and the proposed new version are the following:

1. **Parameters.** Determine the storage size 2^g, a strong cryptographic primitive H, and a "light" cryptographic primitive H'.

   We anticipate H' to be a "reduced-round" version of H, though $H' = H$ is also possible.
2. **Initialization.** Derive v_0 from $H(x)$ (where x is derived as above) and v_i from v_{i-1} and v_{i-2}. Then apply an (optional) random step Γ: Derive v_i from v_i and v_j. The indices i and j are pseudorandom, deterministically generated from a public value (typically the salt).

3. **Memory-Hard Transformation.** Transform $v_0, \ldots, v_{2^g-1}$ into new values, based on a memory-hard structure. The proposed version still supports the bit-reversal structure, but, as an alternative, also the double-butterfly structure.

4. **Output.** $H(v_{2^g-1})$ (no modification).

Note that the proposed tweak generalizes the Catena framework. In other words, the first-round version is essentially a restricted set of instances of the proposed tweak: Set $H' = H$, skip the optional random step in the initialization, and use the bit-reversal structure. The only remaining difference in the core function[1] is the derivation of v_i from two values, v_{i-1} and v_{i-2}, where the first-round version used v_{i-1}, only.

Background for the Double-Butterfly Structure. Initially, we proposed a stack of λ bit-reversal graphs for Catena, and we claimed this to be λ-memory-hard. Unfortunately, it turned out that such a stack only provides 1-memory-hardness. Alexander Biryukov and Dmitry Khovratovich pointed out a flaw in our proof of the time-memory tradeoff by providing a tradeoff cryptanalysis [9].

Thus, we decided to give the user the choice: Either stick with 1-memory-hardness by using the fast bit-reversal structure, and maximize the memory, or use a decent amount of memory and maximize the pain for memory-constrained adversaries by λ-memory hardness, now based on a stack of double-butterfly graphs.

Background for the "Light" Primitive H'. Since the first publication of Catena, many people asked us to replace H by a weaker and faster function. Note that H is supposed to be cryptographically very strong and thus cannot be very fast. Such a change would allow us to speed-up the internal operations of Catena. Consequently, we would increase the garlic, while maintaining the same overall speed as before the change. Bill Cox was the first to suggest this. He even implemented this approach in his "waywardgeek" branch of Catena. This is, of course, entirely in the spirit of Catena, which we understand as a framework that can be instantiated by its users, according to their needs.

But by default, our choice for H is BLAKE2b, a cryptographically strong hash function that has essentially been designed to behave like a "random oracle" for a user who has full control over the inputs. In the context of a Catena, the user may choose the password, but then H is iterated thousands of times, without the user choosing any inputs to H. Using a cryptographically strong H appears to be an overkill.

On the other hand, our goal for Catena was *not* to design a new primitive, but to design a framework, using an underlying primitive H. And we wanted to claim the security of Catena, based on the (plausible) assumption that H is secure.

Moreover, our focus was on using "sufficiently large" memory to defend against typical password crackers with massively parallel off-the-shelf hardware, such as GPUs, and we where able to do so without an ultra-fast H. While we still consider using a "sufficiently

[1] There are also some minor differences in the overall structure, such as computing $flap(\lceil g_{\text{low}}/2 \rceil, x, \gamma)$ in Line 2 of Algorithm 1, which serves the purpose of hindering weak garbage-collector attacks.

large" memory as the essential security feature for any memory-demanding password hashing function, we now understand that further increasing the memory usage beyond the "sufficiently large" point can make sense, e.g., as a defense against password crackers using dedicated hardware. Thus, we eventually decided to speed-up Catena internally, by including the "light" primitive H'.

Nevertheless, we still desired to claim the security of Catena, based on a (plausible) assumption about the security of H, not on an (implausible) assumption about the security of H'. Fortunately, a typical H iterates some "round function" several times, though the "round function" itself is not very strong. This justifies the idea to define Catena by iterating the "round function" of H thousands of times, wherever we originally have been iterating H itself thousands of times. We (very informally) argue that any severe weakness of such an instance of Catena would indicate that iterating the "round function" of H is not secure, which on the other hand would raise doubts about the security of H itself.

The hash function BLAKE2b, our default choice for H, iterates a round 12 times. Our default choice fore H' is a version of BLAKE2b using a single round.

Background for the Modified Initialization. The first change is determining v_i not only from v_{i-1}, but also from v_{i-2}. The main reason for this change is the introduction of a cryptographically "weak" H', and the avoidance of fixed points for H', i.e., values v_i with $H'(v_i) = v_i$. For a full-blown cryptographic hash function H it would be very unlikely to choose a password such that some fixed-point v_i is generated, but for some reduced-round primitive H', this probability may be higher.

The second change is the introduction of the (optional) random layer in the initialization. Note that the bit-reversal permutation has a very regular structure, e.g., it is its own inverse. The time-memory-tradeoff cryptanalysis presented in [9] makes use of such regularities. We conjecture that introducing another layer with a mostly irregular structure makes such tradeoff attacks harder. We leave it as an open problem to prove or disprove this conjecture. We have seen the idea to use a layer with pseudorandom indices at a presentation given by Dmitry Khovratovich [8]. Though, he did not mention how to initialize the random generation for the indices. We argue that the salt is an excellent choice. First, the salt is usually assumed to be public and thus, does not need to be hidden from the adversary. Under this assumption, salt-dependent memory accesses do not leak secret information to cache-timing adversaries. Second, this increases the cost for time-memory tradeoff attacks on dedicated hardware, especially on ASICs. A fixed salt implies a fixed memory-access pattern, which one could easily implement on an ASIC. Changing the salt implies an entirely different memory-access pattern. But, creating new ASICs for every new salt would hardly be economic. On the other hand, ASICs with support for different salts must support different memory-access patterns – which would turn them into almost "reprogrammable" hardware.

Slow-Memory and Cache-Timing Attacks. One major feature of the Catena structure are the password-independent memory accesses. This feature is both a sacrifice and a blessing. It is a sacrifice, because it allows the adversary to reduce the cost for full-memory password guessing by using cheap memory and mounting *slow-memory attacks* [44]. It is a blessing, because it prevents *cache-timing attacks*.

It seems that many other PHC candidates neglect the danger from cache-timing attacks, for example: *"cache-timing attacks are extremely hard to mount on a remote machine, so we presume their application on Argon being purely theoretical"* [10]. Recent results on cache-timing attacks seem to contradict this reasoning, e.g., see [27] for practical cache-timing attacks, where target and victim are residing on different cores and separated by virtual machine boundaries. Thus, vulnerabilities to cache-timing attacks could introduce dramatic weaknesses, e.g., in cloud-computing environments.

Some candidates, such as Lyra2, follow a hybrid approach: A first phase satisfies password-independent memory accesses and a second phase reads or writes the memory at password-dependent locations [44]. The second phase hinders slow-memory attacks, while the first phase is a good defense against cheapening password-guessing in a cache-timing scenario.

Nevertheless, cache-timing attacks are not always about *reducing the cost* for password-guessing. Sometimes they are about *enabling* password-guessing. This is a somewhat critical situation: If one uses an old-style password scrambler, e.g., `md5crypt`, or even stores passwords in the clear, password-guessing is impossible if the password file has not been compromised. Of course, it makes sense to defend against a potential password compromise. However, switching to a modern memory-intense password scrambler, e.g., Lyra2, can backfire. Now, the adversary can mount a password-guessing attack even *without the password file being compromised*, just by observing the memory-access pattern. A password-independent memory-access pattern would avoid this.

Comment for the PHC Committee. The proposed modifications are based on our conclusions from comments and criticism we received, and on discussions at the PHC mailing list. **We believe that these modifications make Catena a better and more general password scrambler, while maintaining its original structure and design ideas.** But, we will forgo any modifications, if these are depreciated by the PHC committee, or considered too heavy for a "tweak".

Future Work. Our work on Catena is part of an ongoing research project. We would like to improve Catena even further, and we are open to any such request from the PHC committee, or the public in general. Some ideas, which we consider, even though they did not make it into our second-round submission, are the following:

- Further techniques to raise the cost for ASIC-based attacks, without significantly increasing the cost of a defender. One approach could be replacing the "light" hash function H' by 64-bit multiplications.

Chapter 10

Acknowledgement.

Thanks to Bill Cox, Ben Harris, Eik List, Colin Percival, Stephen Touset, Steve Thomas, and Alexander Peslyak for their helpful hints and comments, as well as fruitful discussions. Also, we thank Sascha Schmidt for the current reference implementation, and Alexander Biryukov and Dmitry Khovratovich for their time-memory tradeoff cryptanalysis of Catena-BRG. Finally, we thank the members of the PHC committee for their work, and for promoting Catena into the second round of PHC.

Chapter 11

Legal Disclaimer

To the best of our knowledge, neither Catena, the BLAKE2b Function Family, nor the structure of the bit-reversal graph or the double-butterfly graph are encumbered by any patents. We have not, and will not apply for patents on any part of our design or anything in this document. Furthermore, we assure that there are no deliberately hidden weaknesses within the structure or the source code of Catena.

Bibliography

[1] Martin Abadi, Michael Burrows, and Ted Wobber. Moderately Hard, Memory-Bound Functions. In *NDSS*, 2003.

[2] 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.

[3] 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.

[4] Jean-Philippe Aumasson. Password Hashing Competition. `https://password-hashing.net/call.html`. Accessed September 3, 2015.

[5] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[6] 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.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.

[8] Alex Biryukov, Daniel Dinu, Johann Groschaedl, and Dmitry Khovratovich. Design and analysis of memory-hard functions. Presentation at Early Symmetric Crypto (ESC) in Clervaux, Luxembourg, 2015. `https://www.cryptolux.org/mediawiki-esc2015/images/1/1b/Esc2015.pdf`.

[9] Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of Catena. PHC mailing list: discussions@password-hashing.net.

[10] Alex Biryukov and Dmitry Khovratovich. Argon v1: Password Hashing Scheme. Password Hashing Competition, 1st round submission, 2014. `https://password-hashing.net/submissions/specs/Argon-v1.pdf`.

[11] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[12] William F. Bradley. Superconcentration on a Pair of Butterflies. *CoRR*, abs/1401.7263, 2014.

[13] Tom Caddy. FIPS 140-2. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005.

[14] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[15] Solar Designer. Enhanced challenge/response authentication algorithms. `http://openwall.info/wiki/people/solar/algorithms/challenge-response-authentication`. Accessed January 22, 2014.

[16] Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. `http://www.akkadia.org/drepper/SHA-crypt.txt`. Accessed May 16, 2013.

[17] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On Memory-Bound Functions for Fighting Spam. In *CRYPTO*, pages 426–444, 2003.

[18] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *CRYPTO*, pages 139–147, 1992.

[19] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and Proofs of Work. In *CRYPTO*, pages 37–54, 2005.

[20] Morris Dworkin. *Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation*. National Institute of Standards, U.S. Department of Commerce, December 2001.

[21] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of Space. *IACR Cryptology ePrint Archive*, 2013:796, 2013.

[22] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-Evolution Schemes Resilient to Space-Bounded Leakage. In *CRYPTO*, pages 335–353, 2011.

[23] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Submission to NIST, 2010.

[24] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A Memory-Consuming Password Scrambler. Cryptology ePrint Archive, Report 2013/525, 2013. `http://eprint.iacr.org/`.

[25] 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 LNCS, 2010.

[26] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[27] 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/`.

[28] Poul-Henning Kamp. The history of md5crypt. `http://phk.freebsd.dk/sagas/md5crypt.html`. Accessed May 16, 2013.

[29] 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.

[30] 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.

[31] 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.

[32] Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[33] Krishna Neelamraju. Facebook Pages: Usage Patterns | Recommend.ly. `http://blog.recommend.ly/facebook-pages-usage-patterns/`. Accessed May 16, 2013.

[34] 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.

[35] Nvidia. Nvidia GeForce GTX 680 - Technology Overview, 2012.

[36] NIST National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. April 1995. See http://csrc.nist.gov.

[37] 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.

[38] Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.

[39] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[40] Semiocast SAS. Brazil becomes 2nd country on Twitter, Japan 3rd — Netherlands most active country. `http://goo.gl/QOeaB`. Accessed May 16, 2013.

[41] J. Savage and S. Swamy. Space-time trade-offs on the FFT algorithm. *Information Theory, IEEE Transactions on*, 24(5):563 – 568, sep 1978.

[42] John E. Savage and Sowmitri Swamy. Space-Time Tradeoffs for Oblivious Interger Multiplications. In *ICALP*, pages 498–504, 1979.

[43] Ravi Sethi. Complete Register Allocation Problems. *SIAM J. Comput.*, 4(3):226–248, 1975.

[44] 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`.

[45] Sowmitri Swamy and John E. Savage. Space-Time Tradeoffs for Linear Recursion. In *POPL*, pages 135–142, 1979.

[46] Martin Tompa. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. In *STOC*, pages 196–204, 1978.

[47] John Tromp. Cuckoo Cycle; a memory-hard proof-of-work system. Cryptology ePrint Archive, Report 2014/059, 2014. `http://eprint.iacr.org/`.

[48] 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.

[49] Sebastiano Vigna. An experimental exploration of marsaglia's xorshift generators, scrambled. *CoRR*, abs/1402.6246, 2014.

Appendix A

BLAKE2b-1

BLAKE2b-1 describes a modification of BLAKE2b reduced to one single round instead of twelve rounds. The difference between the function BLAKE2b-1 and the original BLAKE2b can be seen in Algorithm 7, where the lengths are given in bytes.

Algorithm 7 BLAKE2b-1 (left) and BLAKE2b (right)

Input: I1 {Input1}, I2 {Input2},
v {Vertex Index}
Output: h {Hash}

1: $S.buf \leftarrow$ I1 || I2
2: $S.buflen \leftarrow 128$
3: $increment\_counter(S, S.buflen)$
4: $set\_last\_block(S)$
5: $r \leftarrow v \bmod 12$
6: $compress(S, r)$
7: $h \leftarrow S.h$
8: **return** h

Input: I {Input Array}, l {Input Length},
m {Output Length}
Output: h {Hash}

1: $blake2b\_init(S, m)$ {Init State}
2: **for** i **from** 0 **to** $(\lfloor l/128 \rfloor - 1)$ **do**
3: $S.buf \leftarrow$ I$[i]$
4: $S.buflen \leftarrow 128$
5: $increment\_counter(S, S.buflen)$
6: $compress(S)$
7: **end for**
8: $S.buf \leftarrow$ I$[\lfloor l/128 \rfloor]$ || 0^*
9: $S.buflen \leftarrow l \bmod 128$
10: $increment\_counter(S, S.buflen)$
11: $set\_last\_block(S)$
12: $compress(S)$
13: $h \leftarrow S.h$
14: **return** h

Following from Line 1 in Algorithm 7 (right), BLAKE2b initializes the internal state S in every invocation, whereas BLAKE2b-1 does not. Thus, for Catena, the internal state 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 4 and 5. This assures that twelve invocations of BLAKE2b-1 are as close as possible to the original BLAKE2b and also saves computation time as shown in Table A.1. To further ensure this similarity, we compute the round index as shown in Line 5 of Algorithm 7 (left).

The functions *blake2b_init*, *increment_counter*, and *set_last_block* are used as speci-

Algorithm	**Medien Clocks per Byte**
BLAKE2b-1	2.44
BLAKE2b-1 without initialization	0.86
BLAKE2b	9.81

Table A.1.: Benchmark comparison of BLAKE2b-1, BLAKE2b-1 without initialization, and BLAKE2b. Timings are measure on an Intel(R) Core(TM) i7-3930M CPU @ 3.20GHz system.

fied in the reference implementation of BLAKE2b. For BLAKE2b-1, we fixed the input length to 128 bytes and neglect the padding since the size of the inputs never changes when using BLAKE2b-1 within Catena. The compression functions of BLAKE2b-1 and BLAKE2b are shown in Algorithm 8, where σ denotes the message schedule (not to be confused with the indexing function σ of Catena-Dragonfly).

Algorithm 8 Functions *compress* of BLAKE2b-1 (left) and BLAKE2b (right)

Input: S {BLAKE2b State}, r {Round Index}
Output: S {Updated BLAKE2b State}
1: $v[0\ldots7] \leftarrow S.h$
2: $v[8\ldots15] \leftarrow \text{IV}$
3: $v[12,13] \leftarrow v[12,13] \oplus S.t$
4: $v[14,15] \leftarrow v[14,15] \oplus S.f$
5: $s[0\ldots15] \leftarrow \sigma[r \mod 10][0\ldots15]$
6: $v \leftarrow G(v,0,4,8,12,S.buf[s[0]],S.buf[s[1]])$
7: $v \leftarrow G(v,1,5,9,13,S.buf[s[2]],S.buf[s[3]])$
8: $v \leftarrow G(v,2,6,10,14,S.buf[s[4]],S.buf[s[5]])$
9: $v \leftarrow G(v,3,7,11,15,S.buf[s[6]],S.buf[s[7]])$
10: $v \leftarrow G(v,0,5,10,15,S.buf[s[8]],S.buf[s[9]])$
11: $v \leftarrow G(v,1,6,11,12,S.buf[s[10]],S.buf[s[11]])$
12: $v \leftarrow G(v,2,7,8,13,S.buf[s[12]],S.buf[s[13]])$
13: $v \leftarrow G(v,3,4,9,14,S.buf[s[14]],S.buf[s[15]])$
14: $S.h \leftarrow S.h \oplus v[0\ldots7] \oplus v[8\ldots15]$

Input: S {BLAKE2b State}
Output: S {Updated BLAKE2b State}
1: $v[0\ldots7] \leftarrow S.h$
2: $v[8\ldots15] \leftarrow \text{IV}$
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 **from** 0 **to** 11 **do**
6: $s[0\ldots15] \leftarrow \sigma[r \mod 10][0\ldots15]$
7: $v \leftarrow G(v,0,4,8,12,S.buf[s[0]],S.buf[s[1]])$
8: $v \leftarrow G(v,1,5,9,13,S.buf[s[2]],S.buf[s[3]])$
9: $v \leftarrow G(v,2,6,10,14,S.buf[s[4]],S.buf[s[5]])$
10: $v \leftarrow G(v,3,7,11,15,S.buf[s[6]],S.buf[s[7]])$
11: $v \leftarrow G(v,0,5,10,15,S.buf[s[8]],S.buf[s[9]])$
12: $v \leftarrow G(v,1,6,11,12,S.buf[s[10]],S.buf[s[11]])$
13: $v \leftarrow G(v,2,7,8,13,S.buf[s[12]],S.buf[s[13]])$
14: $v \leftarrow G(v,3,4,9,14,S.buf[s[14]],S.buf[s[15]])$
15: **end for**
16: $S.h \leftarrow S.h \oplus v[0\ldots7] \oplus v[8\ldots15]$

Appendix B

The Name

The name Catena comes from the Latin word for "chain". It was chosen based on the fact that the underlying structure of Catena is given by the BRG_λ^g or the DBG_λ^g, where each vertex within these graph depends at least on its predecessor, thus, providing a sequential structure. More detailed, if one thinks of representing all vertices within a BRG_λ^g or a DBG_λ^g to be sorted in their topological order, each vertex v_i depends at least on the vertex v_{i-1} for $1 \leq i \leq (\lambda + 1) \cdot 2^g - 1$ (Catena-BRG) or $1 \leq i \leq (\lambda \cdot (2g - 1) + 1) \cdot (2^g - 1)$ (Catena-DBG).

Appendix C

Test Vectors

C.1. Test Vectors for Catena-Dragonfly

Lambda:	2	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	26 4b 83 7b 14 c4 7b 07 ac 4d f4 6d 61 39 b0 78 c1 f5 5a f6 cd 03 65 ef 9f 0c 28 e8 dd ee a2 0d 73 e5 7f 0f d9 7c 3f 08 5c 34 af 8f 1d 11 44 29 5b dc 07 d5 77 68 0a b2 6c 22 8b 33 15 46 8c 1a	(64 octets)

Lambda:	2	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	97 8e 99 44 3b d2 07 dd 7c 26 92 4e c2 00 c9 db aa c2 d4 a4 d8 60 d8 22 e0 12 b8 42 56 e0 e2 5a 3e 7a 87 63 ad e4 5e 2c 12 ec 10 13 6d ea af ca 2d 85 18 ad d1 22 3f a4 e4 26 02 94 4d c4 5d f4	(64 octets)

C.2. Test Vectors for Catena-Butterfly

Lambda:	4	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	2b 57 5a 52 12 36 36 cf 20 cb 2d 49 97	(64 octets)
	59 18 da d1 bd 16 9c a2 59 44 83 38 f4	
	12 4b 37 7a 27 a6 7d 69 68 ee 13 33 db	
	82 05 b5 8d a3 97 da 25 d4 34 07 38 d8	
	ba 82 21 72 5c 54 16 53 18 e2 df 90	

Lambda:	4	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	2c fb c1 12 5d b3 db 95 19 d4 34 12 58	(64 octets)
	61 9c 05 fe 56 3b a1 5d 2e 77 10 01 f1	
	0c b4 f5 89 7d d1 ee e7 69 12 e5 a0 75	
	ed 4e b9 c0 40 37 46 01 0c 6e ab a4 01	
	86 5f a8 9b 1d 21 9b 2d 1e 74 9a b0	

C.3. Test Vectors for Catena-Dragonfly-Full

Lambda:	2	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	3e 10 70 76 cd da 86 96 07 7d 8b 0e 61	(64 octets)
	0b 43 4e 86 59 e1 5c 09 12 33 47 0f 9f	
	b4 1f 7f ad e1 e1 47 5b 2d 09 32 d3 c9	
	d1 4c f8 a1 4f 4b af b7 9f f6 58 9c 33	
	aa 86 ef 9a 7d 9a 44 70 26 ce 40 ce	

Lambda:	2	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	f2 f3 8d 02 e3 27 a9 d5 4b 4b 30 6b cc	(64 octets)
	e0 db 4a 96 62 b2 c8 e6 4c 45 fb 53 6c	
	95 8d f8 cd 1d 65 31 f6 99 12 6d d5 31	
	18 ca 15 ac 8a 17 72 c1 38 16 36 54 b2	
	23 42 75 16 fd cf aa 22 fd 1d e6 01	

C.4. Test Vectors for Catena-Butterfly-Full

Lambda:	4	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	41 8c 2a b1 a8 84 04 24 a1 59 2a bb 9e	(64 octets)
	aa 0f 3f 3c be 76 4b 13 30 f3 c5 53 83	
	5f 37 d6 e5 c1 7a b8 a9 f5 6f eb e0 41	
	75 8c 15 ff 07 e1 48 ee 20 50 cc fa 64	
	75 75 d3 db 0b bf 4d 32 7c 1e 50 fe	

Lambda:	4	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
Output:	1b 5e ce d6 bf 1c 4c f3 a1 46 57 f7 2c	(64 octets)
	48 93 35 ee 49 85 06 e7 b3 0f b5 7f e4	
	1a f3 46 24 f3 11 1d 16 1a dc fa 41 a9	
	38 72 89 6d 59 e1 f9 a3 c6 00 0b b2 28	
	4f 7f b1 0f 1a 95 2a 42 99 f1 d7 a3	

C.5. Test Vectors for Keyed Catena-Dragonfly

Lambda:	2	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	11 26 20 53 c8 4e 71 ed 1e 1f dd b5 16 6c da b2 94 29 99 b1 41 12 01 70 b8 cf 84 d4 8c da 41 98 3f c9 bc 32 18 8b 68 b3 0a fc 22 36 00 7a bb ee 8d 51 98 37 0b aa 54 33 bf 84 e1 97 1f bc 30 70	(64 octets)

Lambda:	2	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	32 cf 47 96 33 61 35 60 ef 49 8d d4 5d 46 9a 52 51 08 ad 0a 2f 3b 7a 4f a3 7f 5c 1a 48 e6 5a bc bd 5a 24 84 cf a5 25 7b 6d c3 37 3f 03 22 23 c5 a7 03 45 7e 0e 8c 6a ee cb 1a 95 15 f2 78 a0 69	(64 octets)

C.6. Test Vectors for Keyed Catena-Butterfly

Lambda:	4	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	1c 3a f9 7a ce bc 3c 25 92 99 04 91 e0 0c 72 10 84 61 d5 db 2e 48 20 1c 1f 37 be 77 66 4e c4 33 31 45 ab d3 d2 c4 8c 39 53 7d 00 1a 8a b1 da 13 e2 8a a7 3a c6 40 7f f3 8f f2 7c f7 12 18 63 fa	(64 octets)

Lambda:	4	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	89 ba 1f c0 55 00 e9 28 8a bb 2b 88 c7 27 cf 8c 05 9c 42 0f aa 75 d5 7d 42 9c e8 ec eb 8f c5 37 6d c7 ca f5 87 e1 0e ba 31 96 e7 6c 59 8e 8d 03 e4 2d f9 d2 59 f1 fd d1 32 1d 0c ac a1 c8 67 2d	(64 octets)

C.7. Test Vectors for Keyed Catena-Dragonfly-Full

Lambda:	2	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	09 7d d3 5e 11 50 8c 7c b5 2f a2 d6 16 5e 29 84 d3 85 22 1b 85 03 57 d8 28 5c 18 23 2e 99 02 74 0b 77 ee 34 f3 24 9e 6a 1a 30 2c f6 56 c4 48 58 20 d5 03 d1 d6 44 b1 1b ae 3c 2e d4 2c 34 fc a4	(64 octets)

Lambda:	2	
Garlic:	10	
Associated data:		(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	57 b2 53 d0 eb 94 9b 68 d8 24 2f f1 53 a6 88 c3 6d a8 cb 66 11 17 e7 96 10 01 71 d5 e6 cb a5 83 b2 d6 3a f5 0f 94 4a 4f b5 3a 8b a6 79 ba 4d 37 9c b0 09 61 fc ec 20 5c d2 f3 3d a3 42 a1 1b 9c	(64 octets)

C.8. Test Vectors for Keyed Catena-Butterfly-Full

Lambda:	4	
Garlic:	1	
Associated data:		(0 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	76 e1 89 99 74 0e 0e ce 13 0b 03 63 e9 ff 65 f5 69 62 b5 0c 9f 21 97 5a 74 40 f3 0b 87 d1 22 ef f4 85 36 52 2a 17 16 ce da dd 72 be fc 23 11 e7 86 41 65 86 09 b7 8d 5a d8 19 27 96 76 e4 ec 94	(64 octets)

Lambda:	4	
Garlic:	10	
Associated data:	64 61 74 61	(4 octets)
Password:	70 61 73 73 77 6f 72 64	(8 octets)
Salt:	73 61 6c 74	(4 octets)
UUID:	18446744073709551615	(octets)
Key:	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF	(16 octets)
Output:	be 1f 10 04 b7 af 7e 4e 32 29 48 6d b3 0e c0 bc 15 83 fc a8 10 e8 ad d8 3c 89 fe ab 58 22 4b f7 9e 36 b9 3b 98 00 d2 6f 0d a6 4a 75 8f 31 2f c9 8a 8d ef fb 90 d1 e4 45 35 a9 bd c3 26 4d 2a 3e	(64 octets)

Appendix D

Illustration of a BRG_4^3

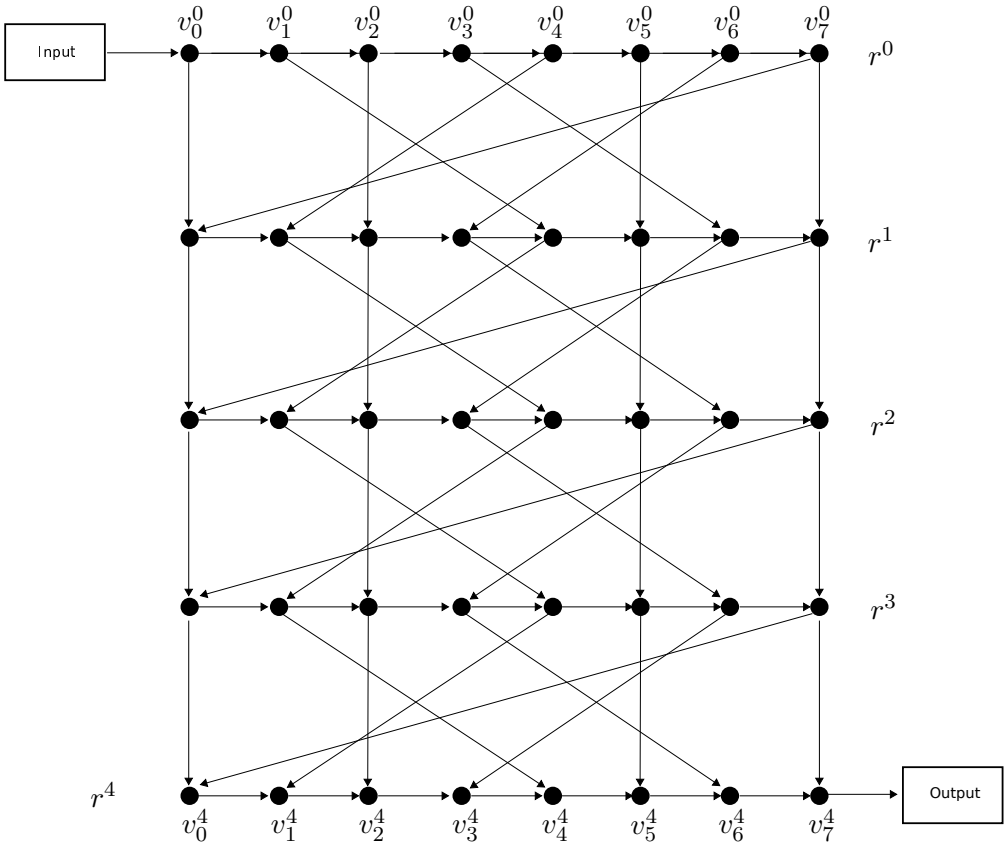

Figure D.1.: A $(3, 4)$-bit-reversal graph.

Appendix E

Illustration of a DBG_2^3

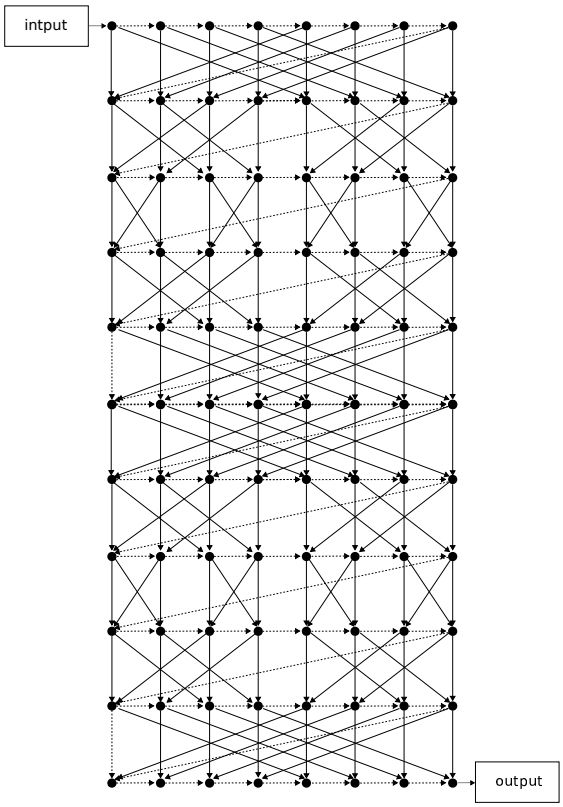

Figure E.1.: A $(3, 2)$-double-butterfly graph.