The Catena Password-Scrambling Framework

Submission to the Password Hashing Competition (PHC)

Christian Forler[*] Stefan Lucks[†] Jakob Wenzel

`<first name>.<last name>@uni-weimar.de`

Bauhaus-Universität Weimar

Version 2.0

October 10, 2014

[*]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.

“Make everything as simple as possible, but not simpler.”

– Albert Einstein

Changelog

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 [8])
- 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_λ^G)
- introducing a new instance Catena-DBG: Catena instantiated with the λ-memory-hard (G, λ)-double butterfly hashing operation (DBH_λ^G)
- 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 2 in Chapter 3 and
  - Line 6 of Algorithm 4 (right) in Section 7.1.

Executive Summary

Catena is a novel and provably secure password-scrambling framework (PSF) that provides cutting-edge properties to allow a flexible usage in multiple environments. Furthermore, Catena can be used as a *key derivation function* and for the scenario of *proof of work/space.*

Catena simple and easy to analyze. An instantiation of Catena is defined by two inputs: (1) a single standard cryptographic primitive, i.e., a hash function H, e.g., BLAKE2b [5] or SHA-512 [32] and (2) a (λ-)memory-hard hashing operation F_λ, e.g., (G, λ)-Double-Butterfly Hashing [27] or (G, λ)-Bit-Reversal Hashing [27]. Further, for both instantiations, it holds that Catena has a sound and elegant design given by a simple and well-understood graph-based structure, where the user-chosen parameter λ determines the depth of the graph. The time-memory tradeoff (TMTO) analysis of the underlying graph-based structures of both instances (Catena-BRG and Catena-DBG) is given in [27], and is based on the pebble game, which was – mostly in the 1970s and 1980s – extensively used to study time-memory tradeoffs. In addition, we show that Catena is provably secure under standard security assumption on the underlying cryptographic hash function H.

Catena is flexible. Any instance of Catena – determined by a cryptographic hash function H and a (λ-)memory-hard function F_λ – can be used with different parameters for *garlic* (tweaking time and memory used for scrambling a password), *pepper* (tweaking the time, only), and *salt* (defending against precomputation attacks). Furthermore, Catena supports a *server relief* protocol to allow shifting (most of) the effort for computing the password hash from the server to the client. This becomes handy whenever a single server is supposed to handle many user requests in parallel.

Moreover, 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 is secure. We claim the following security properties of Catena:

- Preimage security (this important for most applications of password hashes).
- Indistinguishability from random (this is important for key derivation).
- Lower bounds on the time-memory tradeoff for Catena-BRG and Catena-DBG.

The latter point implies that Catena provides high resilience against massively parallel attacks, e.g., GPU-based attacks. More detailed, Catena-BRG satisfies memory-hardness, i.e., halving the available memory double the required computational time. On the other hand, Catena-DBG satisfies λ-memory-hardness, i.e., if the available memory is halved, the required computational time is increase by a factor of about 2^λ.

Furthermore, CATENA provides resistance against cache-timing attacks since both its control flow and its data flow are independent from any secret input, e.g., a password. Finally, 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, 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 password candidates in likelihood-order, until the right one has been found. In some scenarios, when a password is used to open 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, knowing the password hash $Y_1 = H(X)$, can expect to find X by trying out about 2^μ password candidates, calling H about 2^μ times. 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` [25] and `sha512crypt` [14].

This forces the adversary to call the hash function $2^{\sigma+\mu}$ times, rather than 2^μ. 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 rather rare case that a user has a high-entropy password (say, $\mu > 100$), key stretching is unnecessary. But, for users with mid-entropy

[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 [9]. For any such group, an adversary trying the group's single most frequent password succeeds for $\approx 1\,\%$ of the users.

passwords, key 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 $y = \mathrm{PS}(pwd)$, there must be no significantly faster way to test q password candidates $x_1, \ldots, x_q$ for $\mathrm{PS}(x_i) = y$ than by actually computing $\mathrm{PS}(x_i)$ for each single 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 the devices available for potential adversaries. Even without special-purpose hardware, graphical processing units (GPU) with hundreds of cores [31] have nowadays 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 above six conditions) is `scrypt` [34].

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, Block-Mix, and ROMix). Second, the data flow of the ROMix operation is data-dependent, i.e., ROMix reads data from addresses which are password-dependent. This renders ROMix, and thus `scrypt`, vulnerable to cache-timing attacks [22]. Moreover, we have shown in [22] that `scrypt` is vulnerable against the *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 7.2).

Even though we currently are not aware of any practical exploits for either cache-timing or garbage-collector attacks, both are frightening properties that we prefer to avoid. Thus, our challenge was to design a *new* memory-demanding password scrambler PS with the following additional properties:

(7) Simple and easy to analyze.

(8) Resilient to cache-timing attacks.

(9) Resilient to garbage-collector attacks.

Based on Property (7), we focused on a single generic operation, using a single cryptographic primitive. The analysis should prove the 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 any secret input, 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 only learn the information written at the end of the scrambler operation.

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

- *server relief*: the option to shift the main memory and time effort to the client, without burdening the server,
- *pepper* and *garlic*: security parameters to tune 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 first been developed by ourselves, as part of the current research, see [22].

Design Choices for Catena. Informally, Properties (1)-(6) can be translated into "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 an adversary may seem like a futile exercise, since it will actually use all computational power which is within its budget. Though, since todays COTS GPUs and CPUs are rather affordable and can be easily rent from cloud-computing or a botnet provider, we think it is more important to focus on such hardware instead of potentially expensive reprogrammable hardware, e.g., FPGAs. On the other hand, it is more important to slow down password cracking on reprogrammable hardware than on even more expensive non-reprogrammable hardware, e.g., ASICs. For Catena, we anticipate typical defenders to tune the parameters such that Catena runs in the rather slow Random-Access Memory (RAM) in reasonable time. Since then Catena will produce a large amount of cache misses, this is a good defense against adversaries using a GPU, or similar hardware, with plenty of cores but small cache memory for every core. It also thwarts adversaries using cheap (memory-constrained) reprogrammable hardware.

While our concrete proposal suggests to use BLAKE2b, Catena enables the defender to actually choose any strong hash function H that runs well on its machine. The freedom to change H has the additional side effect of frustrating well-funded adversaries using expensive non-reprogrammable hardware: For every defender using a different 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 7.3 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-bytes outputs.
- Possibility of optional inputs such as personalization string, a secret key, or any application-specific parameter.

Actually, using Catena allows to choose arbitrary values for the lengths of the password and the salt. Furthermore, the maximum length of the password hash value depends on the underlying hash function. The adjustment of the time and/or memory usage can be realized by using one or more of the following ways:

- Keep bits of the salt secret (pepper).
- Increase the memory cost parameter (garlic).
- Increase 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 output from random.
- High computational costs for massively parallel cracking devices, 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. Chapter 2 introduces the necessary preliminaries, definitions, and fundamental password-scrambling properties we use in the entire paper. Chapter 3 introduces 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 preimage security and pseudorandomness. In Chapter 5 we introduce two instantiations of Catena– Catena-BRG and Catena-DBG and in Chapter 6 we discuss whose security properties in terms of memory-hardness, pseudorandomness, and resistance against side-channel attacks. Chapter 7 contains background information and the origins of the Catena design. The usage of Catena for the scenario of proof of work, a discussion about Catena in different environments, and the application of Catena as a key derivation function are given in Chapter 8. The paper concludes with acknowledgements, a legal disclaimer, the bibliography and some appendices.

Chapter 2

Preliminaries

In this section we discuss a technique called *Pebble Game*, which will help to understand the proofs of our underlying graph-based structures presented in [27]. Furthermore, we introduce necessary definitions and notations used throughout this paper. Note that we often refer CATENA to as a password scrambler. In this situations it is meant that the considered property holds for both presented instantiations.

2.1. The Pebble Game

The pebble game is an old 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 $v_0, \dots, v_{n-1}$ in memory is only allowed if the index i is statically determined – and thus, independent from the input.

Programs following these two restrictions can be represented as a *directed acyclic graph* (DAG, see Definition 2.4) of vertices and directed edges, where vertices without ingoing edges represent an input, 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 $\mathcal{V} = (v_0, v_1, \dots, v_{n-1})$ and a set of edges $\mathcal{E} = (e_0, e_1, \dots, e_{\ell-1})$. $\Pi(\mathcal{V}, \mathcal{E})$ 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 $x \leftarrow y \circ z$ would be represented by two edges $y \rightarrow x$ and $z \rightarrow x$. Even though the pebble game is defined for vertices with fan-in $\leq d$, for some constant d, we focus, based on the structure of CATENA, only on binary operations "$y \circ z$", implying that all vertices within the DAG have a fan-in of at most 2. Then, "$\circ$" can be any computation which takes two inputs y and z and generates one output x, such as $y \circ z = H(y \mid\mid z)$. Also, for any two vertices $x \neq x'$, with $x \leftarrow y \circ z$ and $x' \leftarrow y' \circ z'$, the symbol "$\circ$" can represent different operations, depending on the target x resp. x'.

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 performs 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 [24] 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 optimize $S \cdot T$. To analyze the effort for a given adversary, one needs to choose a certain model for studying the TMTO. In 1970, Hewitt and Paterson [33] introduced the pebble game as a method for analyzing TMTOs on directed acyclic graphs, which became an important tool for that purpose, see [37–41]. The pebble game has been occasionally used in cryptographic context (see [20] 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 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.

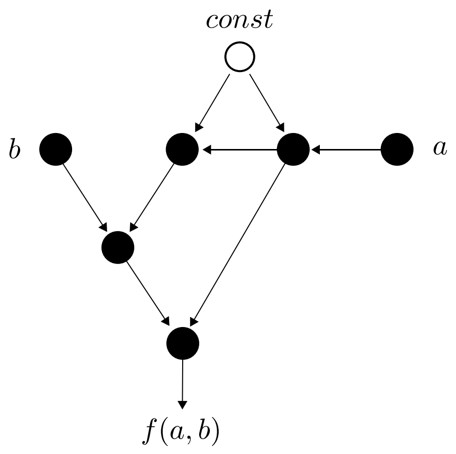

Figure 2.1.: Directed acyclic graph for $f(a,b) = (a \circ const) \circ const) \circ b) \circ (a \circ const)$.

Now, 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 $S = 1$.

Pebble-Game and Password Scramblers. Note that the upper and lower bounds proved in [27] 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 collapse. 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 $y \circ z$ as a call to a random oracle $H(x \mid\mid y)$, there is no alternative way to compute f. Since it is well-established practice in cryptography to instantiate random oracles by hash functions, Catena follows this approach and 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 single round is a cunning composition of xor-operations, additions, and bit-wise rotations over an input word. 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 7.3.

2.2. Properties and Definitions

Below, we describe and define the desired properties of a modern password scrambler.

Memory-Hardness. To describe memory requirements, we adopt and slightly change the notion from [34]. 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 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 [34], Percival introduced the notion of sequential memory-hardness (SMH), which is satisfied by his introduced password scrambler `scrypt`. Bases on this notion, an algorithm is sequential memory-hard, if an adversary has no computational advantage in using multiple CPUs, i.e., using b cores requires b times the effort used for one 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 Chapter 3 and 5.

Note that a further goal of our design was to provide resistance against cache-timing attacks, i.e., both instantiations 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 input. It follows that Catena can be seen as a straight-line program, which on the other hand can be represented by a directed acyclic graph (DAG, see Definition 2.4).

Definition 2.4 (Directed Acyclic Graph). *Let $\Pi(\mathcal{V},\mathcal{E})$ be a graph consisting of a set of vertices $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$ and a set of edges $\mathcal{E} = (e_0, e_1, \ldots, e_{\ell-1})$, where $\mathcal{E} = \emptyset$ is a valid variant. $\Pi(\mathcal{V},\mathcal{E})$ is a directed acyclic graph, if every edge in $\mathcal{E}$ consists of a starting vertex v_i and an ending vertex v_j, with $i \neq j$. A path through $\Pi(\mathcal{V},\mathcal{E})$ beginning at vertex v_i must never reach v_i again (else, there would be a cycle). If there exists a path from a vertex v_i to a vertex v_j in the graph with $i \neq j$, we will write $v_i \leq v_j$.*

Usually, a DAG can be at least partially computed 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 structure of the two proposed instantiations, one can always partition its corresponding DAGs into such subgraphs and hence, 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 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 reasonable 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 2.5 (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 an password scrambler PS as*

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

Furthermore, by $\mathbf{Adv}_{PS}^{REC}(q)$ we denote 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 [28], 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 [29] and 73% of all Twitter accounts do not have at least one tweet per month [36]. It is desirable to be able to compute a new password hash (with some higher security parameter) from the old one (with the old and weaker security parameter), without having to involve 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 [30] and more recent in [13]. 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 using any password scrambler, none of the existing password scramblers has been designed to naturally support this property. Note that this property is optional, e.g., for the proof of work scenario the server relief idea makes no sense, 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 certain passwords. 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 2.6 (Resistance against Cache-Timing Attacks). *Suppose the function $\mathcal{F} : \{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^n$ 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 either needs a key longer than the password hash or has to derive more than one key. Thus, 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 behaviour cannot be distinguished from a set of random functions. Thus, we define the Random-Oracle Security of a password scrambler as follows:

Definition 2.7 (Random-Oracle Security). *Let PS* : $\{0,1\}^* \rightarrow \{0,1\}^n$ *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* \$: $\{0,1\}^* \rightarrow \{0,1\}^n$ *be a random function which, given an input of arbitrary length, always returns randomly chosen 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 some other (optional) parameters, e.g., parameters to adjust the memory consumption or the computational time.

Resistance against Garbage-Collector (GC) Attacks. The basic idea of this attack is to exploit the management of the memory and the internal state a password-hashing algorithm. More detailed, the goal of an adversary is to find out a valid preimage (password) for a given hash value without taking the whole effort of computing the corresponding password-hashing algorithm for each candidate (shortcut attack). Next, we formally define the term Garbage-Collector Attack.

Definition 2.8 (Garbage-Collector Attack). *Let* PS *be a memory-demanding password scrambler depending on a memory-cost parameter g with* $G = 2^g$. *Furthermore, let* $v_0, \ldots, v_{G-1}$ *denote the internal state of* PS *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 the knowledge about* $v_0, \ldots, v_{G-1}$ *reduces the runtime of* $\mathcal{A}$ *for testing a password candidate x from* $\mathcal{O}(PS(x))$ *to* $\mathcal{O}(f(x))$ *with* $\mathcal{O}(f(x)) < \mathcal{O}(PS(x))$.

2.3. Notational Conventions

Identifier	**Description**
pwd	password
λ	security parameter of F_λ (depth)
s	salt (public random value)
p	pepper (secret bits of the salt)
t	tweak
d	domain (application specifier) of CATENA
V	version identifier
g_0, g	minimum garlic; current garlic with $G = 2^g$
PS/PSF	Password Scrambler/Password-Scrambling Framework
m	output length of CATENA
F_λ	function replaced in a particular instance of CATENA
\$	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 $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$
$v_{i,j}^k$	j-th vertex of the i-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. A formal definition is shown in Algorithm 1, whereas the general idea is given in Figure 3.1. The function *truncate(x,m)* (see Lines 2 and 6 of Algorithm 1) outputs the m least significant bits of x, where m is the user-chosen output length of CATENA. After the first truncation step, the function F_λ 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. Note that the function F_λ is replaced by either BRH_λ^G (CATENA-BRG, see Section 5.1) or DBH_λ^G (CATENA-DBG, see Section 5.2).

3.1. Specification

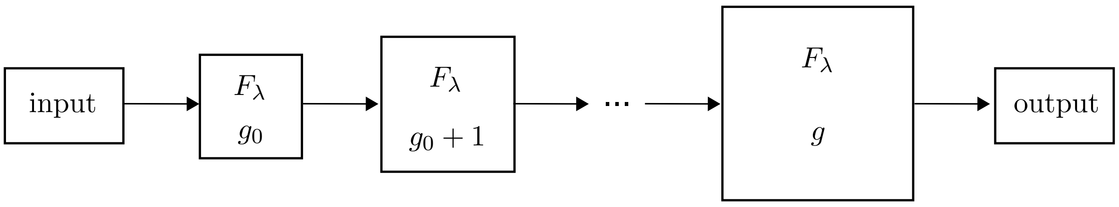

Figure 3.1.: The general idea of applying the function F_λ $(g - g_0)$ times whereas the value for g is increased by 1 in each iteration.

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

Algorithm 1 CATENA

```
Input: $\lambda$ {Depth}, $pwd$ {Password}, $t$ {Tweak} $s$ {Salt}, $g_0$ {Min. Garlic}, $g$ {Garlic},
    $F_\lambda$ {Instance}, $m$ {Output Length}
Output: $x$ {Hash of the Password}
1: $x \leftarrow H(t \mid\mid pwd \mid\mid s)$
2: $x \leftarrow truncate(x)$
3: for $c = g_0, \ldots, g$ do
4:     $x \leftarrow F_\lambda(c, x \mid\mid 0^*)$
5:     $x \leftarrow H(c \mid\mid x)$
6:     $x \leftarrow truncate(x, m)$
7: end for
8: return $x$
```

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

where the first byte V denotes the version ID (`0xFF` for CATENA-BRG and `0xFE` for CATENA-DBG), and the second byte d denotes the domain (i.e., the mode) for which CATENA is used. We set $d = 0$ 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 λ defines together with the value g (see above) the security parameters for CATENA. The 16-bit value m denotes the output length of CATENA in bits, and the 32-bit value $|s|$ denotes the total length of the salt in bits. The n-bit value $H(\mathrm{AD})$ 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. As 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.

Client-Independent Update (CI-update). Its sequential structure does enable Catena to provide client-independent updates. Let $h \leftarrow \text{Catena}_\lambda(pwd, t, s, g_0, g, F_\lambda, m)$ be the hash of a specific password pwd, where t, s, g_0, g, F_λ, and m denote tweak, the salt, the minimum garlic, the garlic, the instance, and the output length, respectively. After increasing the security parameter from g to $g' = g + 1$, we can update the hash value h without user interaction by computing:

$$h' = truncate(H(g' \mid\mid F_\lambda(g', h \mid\mid 0^*)), m).$$

It is easy to see that the equation $h' = \text{Catena}_\lambda(pwd, t, s, g_0, g', F_\lambda, m)$ holds.

Server Relief. In the last 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_λ 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, it has to be stored on server-side. To preserve the server-relief property (see above), we encrypt the output of Catena_λ using the XOR operation with $H(K \mid\mid userID \mid\mid g \mid\mid K)$, which, under the reasonable assumption that the value $(userID \mid\mid g)$ is a nonce, was proven to be CPA-secure in [35]. Let $X := \{pwd, t, s, g_0, g, F_\lambda, m\}$. Then, the output of Catena^K_λ is computed as follows:

$$y = \text{Catena}^K_\lambda(userID, X) := \text{Catena}_\lambda(X) \oplus H(K \mid\mid userID \mid\mid g \mid\mid K),$$

where Catena_λ 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 $g = g + r$ for arbitrary $r \in \mathbb{N}$. The process takes the following four steps:

1. Given K and $userID$, compute $z = H(K \mid\mid userID \mid\mid g \mid\mid K)$.
2. Compute $x = y \oplus z$, where y denotes the current keyed hash value.
3. Update x, i.e., $x = H(c \mid\mid F_\lambda(c, x \mid\mid 0^*))$ for $c \in \{g + 1, \ldots, g + r\}$.
4. Compute the new hash value $y = y \oplus H(K \mid\mid userID \mid\mid g + r \mid\mid K)$.

Remark. Obviously, it is a bad idea to store the secret key K on the same place as 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 [11] 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 [27]. They analyzed the underlying structures which we use in our instantiations regarding to its memory-hardness. In short, CATENA-BRG (see Section 5.1) 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 5.2) provides the property of a 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 a proof technique called *pebble game* (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 $h = \mathrm{PS}(pwd)$, one cannot gain any information about pwd in practical time. This requirement becomes mostly crucial in a 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_λ 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 2.6, it follows that an algorithm is cache-time resistance 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_λ, whose control flow only depend on the security parameters g (garlic) and λ (depth), i.e., 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 where looking for a hash function with a 512-bit (64 byte) output, since it 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 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 should be byte-aligned. For Catena, we decided to use 1) BLAKE2b [5] since it high performance in software, which allows to use a large value for the `garlic` parameter, resulting in a higher memory effort than for, e.g., SHA3-512 [7], and 2) SHA2-512 [32] since it is well-analyzed [2, 23, 26], standardized, and widely used, e.g., in `sha512crypt`, the common password scrambler in several Linux distributions [14].

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_λ^G or DBG_λ^G), 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 same computational effort.

Remark: Our primary recommendation for the Password Hashing Competition (PHC) is BLAKE2b, and our secondary recommendation is SHA-512. Nevertheless, we highly encourage users to plug in their favourite cryptographic hash function such as SKEIN-512 [21] or SHA3-512.

Cost Parameter. Table 3.1 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 system, the parameter sets must be individually adjusted corresponding to the underlying hardware, e.g., for embedded systems one would chose smaller garlic values.

Encoding. The parameter encoding table can be found in Table 3.2.

Implementation. A current reference implementation can be found on

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

This implementation was used to create the test vectors given in Appendix B.

Algorithm	H	F_λ	g_0/g	λ	$\|s\|$	**Time**
CATENA	BRG_λ^G	BLAKE2b	14/14	3	128 bits	0.36 sec
		SHA-512	13/13	2	128 bits	0.43 sec
CATENA	DBG_λ^G	BLAKE2b	18/18	2	128 bits	0.23 sec
		SHA-512	17/17	2	128 bits	0.34 sec
CATENA-KG	BRG_λ^G	BLAKE2b	21/21	4	128 bits	3.16 sec
		SHA-512	20/20	3	128 bits	3.77 sec
CATENA-KG	DBG_λ^G	BLAKE2b	17/17	4	128 bits	4.65 sec
		SHA-512	15/15	4	128 bits	3.75 sec

Table 3.1.: Recommended parameter sets for COTS systems. All timings are measured on a Intel Core i5-2520M CPU (2.50GHz) system.

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	4 bytes
s	salt	byte string
$\|s\|$	salt length	UInt32

Table 3.2.: Parameter choices for the practical usage of CATENA. By UInt32 we denote a 32-bit unsigned integer which is always encoded in little-endian way.

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-DBG (see Section 5.2) inherits its λ-memory-hardness (see Definition 2.3) from F_λ, whereas Catena-BRG (see Section 5.2) provides only 1-memory-hardness, i.e., memory-hardness (see Definition 2.2).

Since the memory-access pattern of Catena is static and therefore, independent from the password, it provides resistance against cache-timing attacks. 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$_\lambda$ is Password-Recovery Resistant). *Let m denote the min-entropy of a password source $\mathcal{Q}$. Then, it holds that*

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

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 4. More detailed, for a given value h with $h \leftarrow H(g, x)$, $\mathcal{A}$ has to find a valid value for x. The success probability for this can be upper bounded by $\mathbf{Adv}_{H}^{\text{pre}}(q, t)$. 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_λ from a random bitstring of the same length as the output of CATENA_λ. Therefore, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle. Note that the output length m, the depth λ, and the value g_0 (minimum garlic) are constant values which are set once when initializing a system the first time.

Theorem 4.2 (PRF Security of Catena$_\lambda$). *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 $g \geq g_0 \geq 1$. Then, it holds that*

$$\mathbf{Adv}_{\text{CATENA}_\lambda}^{PRF}(q, t) \;\leq\; \frac{(q \cdot g + q)^2}{2^n} + \mathbf{Adv}_{F_\lambda}^{coll}(g \cdot q).$$

Proof. Let $a^i = (pwd^i \mid\mid u^i \mid\mid s^i \mid\mid g)$ represent the i-th query, where pwd^i denotes the password, u^i denotes the tweak, s^i the salt, and g the garlic. 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 $F_\lambda(g, a^j)$ of the j-th query (cf. Algorithm 1, Line 3). Then, $H(g \mid\mid y^j)$ is the output of $\text{CATENA}_\lambda(a^j)$. In the case that $y^1, \ldots, y^q$ are pairwise distinct, an adversary $\mathcal{A}$ cannot distinguish $H(g \mid\mid \cdot)$ 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 F_λ. For q queries, we have at most $q(g+1)$ invocations of H. Thus, we can upper bound the collision probability by

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

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

Chapter 5

Instantiations

In this section we introduce two concrete instantiations of Catena: Catena-BRG and Catena-DBG.

5.1. Catena-BRG

For Catena-BRG, F_λ is implemented by the (G, λ)-Bit-Reversal Hashing (BRH^G_λ) algorithm, which is based on the bit-reversal permutation.

Definition 5.1 (Bit-Reversal Permutation τ). *Fix a number $k \in \mathbb{G}$ 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*

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

The bit-reversal permutation τ defines the (G, λ)-Bit-Reversal Graph (BRG^G_λ).

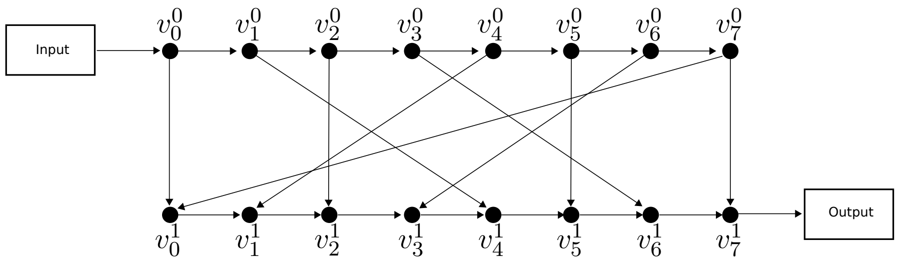
Input
Output
v_0^0 v_1^0 v_2^0 v_3^0 v_4^0 v_5^0 v_6^0 v_7^0
v_0^1 v_1^1 v_2^1 v_3^1 v_4^1 v_5^1 v_6^1 v_7^1

Figure 5.1.: An $(8,1)$-bit-reversal graph (BRG_1^8).

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*

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

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

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

For example, Figure 5.1 illustrates an BRG_1^8. A BRG_4^8 can be seen in Appendix C. 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 [27].

Bit-Reversal Hashing. The (G,λ)-Bit-Reversal Hashing function is defined in Algorithm 2. 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.

Observation on Cache-Time Misses. The following observation holds for a $(G,1)$-bit-reversal graph, but can also be easily generalized for (G,λ)-bit-reversal graph with arbitrary values of $\lambda \in \mathbb{N}$. When the output size of H is equal to the size of a cache line (or a multiple), each time a value is read from or written to a location v_i, the time to access v_i is the same, to first order. Now, assume the output size of H (i.e., the number of bits for each of the v_i) is k times the cache line size. In this case the adversary may

Algorithm 2 (G, λ)-Bit-Reversal Hashing (BRH_λ^G)

Input: g {Garlic}, x {Value to Hash}, λ {Depth}, H {Hash Function}
Output: x {Password Hash}

```
1:  $v_0 \leftarrow H(x)$
2:  for $i = 1, \ldots, 2^g - 1$ do
3:      $v_i \leftarrow H(v_{i-1})$
4:  end for
5:  for $k = 1, \ldots, \lambda$ do
6:      $r_0 \leftarrow H(v_0 \,||\, v_{2^g-1})$
7:      for $i = 1, \ldots, 2^g - 1$ do
8:          $r_i \leftarrow H(r_{i-1} \,||\, v_{\tau(i)})$
9:      end for
10:     $v \leftarrow r$
11: end for
12: return $r_{2^g-1}$
```

try to optimize the memory layout (the order in which the v_i are stored in memory) to minimize the number of cache misses. However, a nice property of the bit-reversal permutation τ is that one cannot gain much from such an optimization. If the values are stored in their natural order: v_0, v_1, ..., v_{2^g-1}, then, the number of cache misses in the first phase (Lines 2 and 4 of Algorithm 2) are drastically reduced to $2^g/k$. But, in the second phase (Lines 5 and 12 of Algorithm 2), the number of cache misses is 2^g. If an adversary stores the v_i in their bit-reversal order, the number of cache misses in the second phase is $2^g/k$, but, in the first it is now 2^g. A more complex mixture between natural and bit-reversal order would allow $2^g/\sqrt{k}$ cache misses in each of the first and the second phase. If k is not really huge, the benefit from such an optimization would remain small.

5.2. Catena-DBG

Note that a (G, λ)-Double-Butterfly Graph (DBG_λ^G) is based on a stack of λ G-superconcentrators. The following definition of a G-superconcentrator is a slightly adapted version of that introduced in [27].

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 Trans-

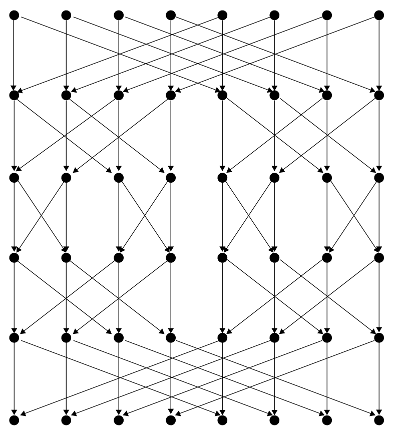

Figure 5.2.: A Cooley-Tukey FFT graph with eight input and output vertices.

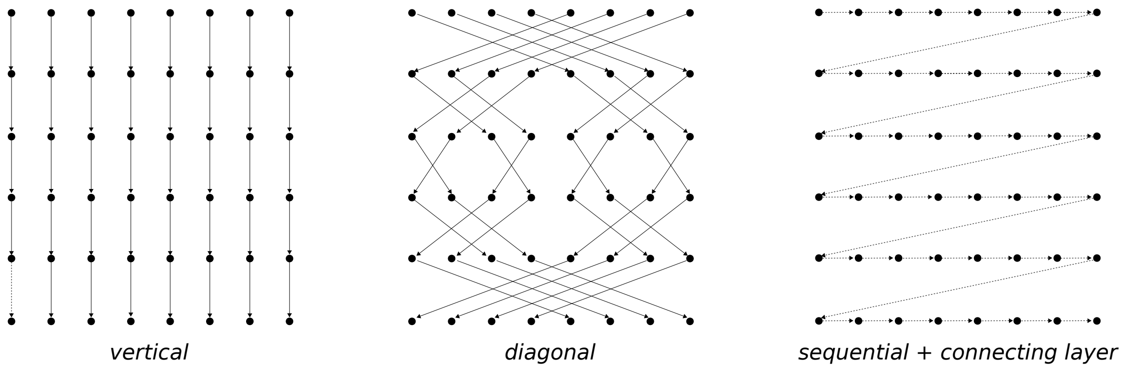

Figure 5.3.: Types of edges as we use them in our definitions.

formations [10]. More detailed, it is a representation of twice the Cooley-Tukey FFT algorithm [12] omitting one row in the middle (see Figure 5.2 for an example where $G = 8$). 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 $v_{i,j}^{k}$ as the j-th vertex in the i-th row of the k-th double-butterfly graph.

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 $DBG_\lambda^G(\mathcal{V},\mathcal{E})$ consists of $2^g \cdot (\lambda \cdot (2g-1)+1)$ vertices*

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

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

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

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

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

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

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

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

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

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

In Appendix D you can see a DBG_2^8. Figure 5.3 illustrates the individual types of edges we used in our definition above. Moreover, an example for $G = 8$ and $\lambda = 1$ can be seen in Figure 5.4.

Double-Butterfly Hashing. The (G,λ)-double-butterfly hashing operation is defined in Algorithm 3. The structure is based on a (G,λ)-double-butterfly graph. Note that the function σ (see Lines 7 and 9) is given by

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

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

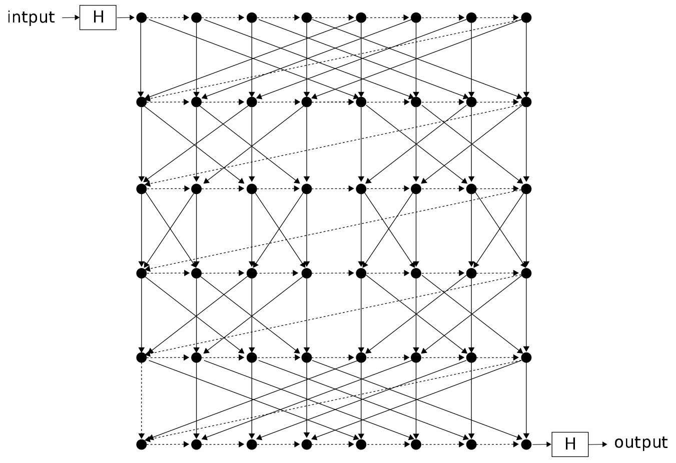

Figure 5.4.: An $(8,1)$-double-butterfly graph (DBG_1^8).

function, 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 Catena-BRG since a bit-reversal graph satisfies a fixed indegree of at most 2. When considering Catena-DBG, we cannot simply concatenate the inputs to 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 BRG_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 Catena-BRG 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 Catena-BRG and Catena-DBG. Moreover, for both instantiations 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 BRG_λ^G requires $2^g - 1 + \lambda \cdot 2^g$ calls to H and the (G,λ)-DBG requires $2^g - 1 + \lambda \cdot (2g-1) \cdot 2^g$ calls to H. It is easy to see, that the performance of Catena-DBG in comparison to Catena-BRG 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 [27] 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.

Algorithm 3 (G, λ)-Double-Butterfly Hashing (DBH_λ^G)

Input: g {Garlic}, x {Value to Hash}, λ {Depth}, H {Hash Function}

Output: x {Password Hash}

```
1:  $v_0 \leftarrow H(x)$
2:  for $i = 1, \ldots, 2^g - 1$ do
3:      $v_i \leftarrow H(v_{i-1})$
4:  end for
5:  for $k = 1, \ldots, \lambda$ do
6:      for $i = 1, \ldots, 2g - 1$ do
7:          $r_0 \leftarrow H(v_{2^g-1} \oplus v_0 \mid\mid v_{\sigma(g,i-1,0)})$
8:          for $j = 1, \ldots, 2^g - 1$ do
9:              $r_i \leftarrow H(r_{i-1} \oplus v_i \mid\mid v_{\sigma(g,i-1,j)})$
10:         end for
11:         $v \leftarrow r$
12:     end for
13: end for
14: return $v_{2^g-1}$
```

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 instantiations.

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 instantiations are resistant against cache-timing attacks (see Definition 2.6).

Considering a malicious garbage collector (see Definition 2.8), each of Algorithms 2 and 3 exposes the arrays v and r. Both arrays are overwritten multiple times (depending on the choice of λ). Lets consider both instantiations with $G = 8$ and $\lambda = 1$. Then, for BRH_2^8, the array v is overwritten twice and the array r once, whereas for DBH_2^8, v is overwritten 10 times and r 9 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 [27].

Catena-BRG. In [27], 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 [27]). *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 [8] 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 [27] 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 [27]). *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 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 with $\lambda = 2$ or $\lambda = 3$, 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_λ^G with reasonable values for G, e.g., $G \in [2^{17}, 2^{21}]$, 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 $\mathrm{DBH}_{\lambda}^{G}$.

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 2.7). Therefore, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle.

Theorem 6.3 (Collision Security of $\mathbf{BRH}_{\lambda}^{G}$). *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*

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

Proof. From Algorithm 2 it is easy to see that a collision $\mathrm{BRH}_{\lambda}^{G}(x) = \mathrm{BRH}_{\lambda}^{G}(x')$ 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 1–4 of Algorithm 2. In addition, there are $\lambda \cdot 2^g$ invocations in Lines 5–11 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

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

Thus, our claim follows. ■

Finally, we analyze the collision resistance of $\mathrm{DBH}_{\lambda}^{G}$. 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 $\mathbf{DBH}_{\lambda}^{G}$). *Let q denote the number of queries. Furthermore, let H be modelled as a random oracle for some fixed integers $g, g_0, \lambda \ge 1$ with $g \ge g_0$ and $G = 2^g$. Then, it holds that*

$$\mathbf{Adv}_{DBH_{\lambda}^{G}}^{coll}(q,t) \;\le\; \frac{(q \cdot \lambda \cdot g)^2}{2^{n-2g-3}}.$$

Proof. From Algorithm 3 it is easy to see that a collision $\mathrm{DBH}_{\lambda}^{G}(x) = \mathrm{DBH}_{\lambda}^{G}(x')$ for $x \neq x'$ implies either an input or output collision for H.

For our analysis, we replace the random oracle H by $H'(x) := H(\mathrm{truncate}_n(x))$ that truncates any input to n bits before hashing. Thus, any collision in the first n bits of the input of H in Lines 7 and 9 of Algorithm 3 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' per query. There are 2^g invocations of H' in Lines 1–4 of Algorithm 3. In addition, there are $\lambda \cdot (2g-1) \cdot 2^g$ invocations in Lines 5–13 leading to a total of $\lambda \cdot 2g \cdot 2^g$ invocations of H'. Since H (and thus H') 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 Line 7 and 9 can occur:

$$v^a_{2^g-1} \oplus v^a_0 = v^b_{2^g-1} \oplus v^b_0 \qquad \text{(Algorithm 3, Line 7)}$$

or

$$r^a_{i-1} \oplus v^a_i = r^b_{i-1} \oplus v^b_i \qquad \text{(Algorithm 3, Line 9).}$$

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', 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. ■

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. From ROMix to BRH_λ^G and DBH_λ^G

The core idea for the `scrypt` password scrambler is given by the definition of ROMix (see Algorithm 4 (left)). ROMix is a memory-demanding function based on a cryptographic hash function H, namely the "BlockMix" operation. At first, ROMix generates G password-dependent hash values $v_0, \dots, v_{G-1}$, and then runs through a main loop reading each v_i once, on the average. It has a time-memory tradeoff satisfying $S \cdot T = \mathcal{O}(G^2)$. Thus, for $S = \mathcal{O}(G)$ units of space, the time required to evaluate ROMix is $T = \mathcal{O}(G)$. Each iteration of the main loop, an index j is computed (see Line 7), which depends on H and the current x, and is used to update x, and thus, there is no gain by parallel computations. In [34], Percival introduced this notion as *sequentially memory-hard.*

ROMix Issues. Unfortunately, the ROMix operation contains two security issues:

1. In its main loop, ROMix reads the memory at some index j (see Line 7); and j depends on the current x. As it turns out, each secret password x defines it own access pattern to the memory. This makes ROMix by design vulnerable to *cache-timing attacks.*

2. If, after evaluating ROMix, the memory $v_0, \dots, v_{G-1}$ is not carefully wiped out, password search for an adversary with access to that memory may be much simpler than for an adversary who only has access to the final password hash. If, e.g., the value $v_0 = H(x)$ has been compromised, the adversary only needs a single call to the function H for each password candidate, and negligible memory. In other words, ROMix is thus vulnerable to *garbage-collector attacks* (see Section 7.2).

Algorithm 4 ROMix and (G, λ)-Bit-Reversal Hashing

```
ROMix($g, x$)
 1: $v_0 \leftarrow H(x)$
 2: for $i = 1, \ldots, G-1$ do
 3:     $x \leftarrow H(x)$
 4:     $v_i \leftarrow x$
 5: end for
 6: for $i = 0, \ldots, G-1$ do
 7:     $j \leftarrow (x \bmod G)$
 8:     $x \leftarrow H(x \oplus v_j)$
 9: end for
10: return $x$
```

```
BRH$_\lambda^G(g, x)$
 1: $v_0 \leftarrow H(x)$
 2: for $i = 1, \ldots, 2^g - 1$ do
 3:     $v_i \leftarrow H(v_{i-1})$
 4: end for
 5: for $k = 1, \ldots, \lambda$ do
 6:     $r_0 \leftarrow H(v_0 \mid\mid v_{2^g-1})$
 7:     for $i = 1, \ldots, 2^g - 1$ do
 8:         $j \leftarrow \tau(i)$ {bit-reversal perm.}
 9:         $r_i \leftarrow H(r_{i-1} \mid\mid v_j)$
10:     end for
11:     $v \leftarrow r$
12: end for
13: return $r_{2^g-1}$
```

Since both of our instantiations (Catena-BRG and Catena-DBG) follow the same basic idea, we discuss only the differences between ROMix and the BRH$_\lambda^G$ operation. Our approach applied the following two major modifications to fix the mentioned ROMix security issues.

1. The index j depends on i, not on the current x.
2. In the main loop, the v_i are overwritten.

Further, we do concatenate the two inputs of H instead of XOR-ing them (see Lines 6 and 9 of Algorithm 4 (right)). This minor modification does not have a strong advantage over $H(v_{i-1} \oplus v_j)$. However, the sizes of v_{i-1} and v_j are 512 bits and our default choice for H is BLAKE2b, where the evaluation of an 512-bit input is as fast as the evaluation of a 1024-bit input. Note that for DBH$_\lambda^G$, we concatenated the first two inputs to H since otherwise the slow-down of Catena-DBG would be even stronger. If one would choose another instance for H (with a smaller block size of the underlying compression function), one may further tweak Catena by using XOR of all inputs instead of concatenation.

First Modification. A proper way to apply the first modification is hard to find. The security of ROMix greatly depends on the adversary not being able to predict the values of j without actually running ROMix. But if the indices j do not depend on the initial secret, the adversary can predict them. A naive choice of j would clearly help the adversary to make some beneficial time-memory-tradeoffs. We need a way to choose j, depending on the loop index i, which prevents such time-memory tradeoffs.

Algorithm 5 Main Loop of (G, λ)-Bit-Reversal Hashing

```
1: for $k = 0, \dots, \lambda - 1$ do
2:     $r_0 \leftarrow H(v_0 || v_{2^g-1})$
3:     for $i = 1, \dots, 2^g - 1$ do
4:         $j \leftarrow \tau(i)$ {bit-reversal perm.}
5:         $r_i \leftarrow H(v_{i-1} \mid\mid v_j)$
6:     end for
7:     $v \leftarrow r$
8: end for
```

Second Modification. Graphs derived from the *bit-reversal permutation* τ (see Definition 5.1) have interesting properties, provable by the pebble game technique (see Section 2.1). Given an n-bit number $i = (i_0, i_1, \dots, i_n)$ with the numeric value $i = \sum_k i_k \cdot 2^k$, $\tau(i)$ is the n-bit number $\tau(i) = (i_{n-1}, i_{n-2}, \dots, i_0$ with the numeric value $i = \sum_k i_k \cdot 2^{n-k}$.

This approach leads to the realization of the BRH_λ^G operation (see Algorithm 4 (right)). It is the core of Catena-BRG and it trivially solves the first ROMix issue. Further, its contribution in solving the second ROMix issue is discussed in Section 7.2. While the time-memory tradeoff on a sequential machine is $ST = O(2^{2g})$, as for ROMix, a memory-constrained adversary can benefit a lot from parallel processing:

$$c \text{ parallel cores,} \quad \text{time } T = \mathcal{O}(2^g), \quad \text{and space } S = 2^g/c.$$

Formally, this means that bit-reversal hashing is *memory-hard, but not sequentially memory-hard*, and this makes trading cores for memory a potentially interesting deal for the adversary.

Ideally, we would have liked to propose an approach unifying sequential memory-hardness with a memory-access pattern that is independent from a secret. We pose this as an open problem. Instead, we propose a structure based on λ stacks of a bit-reversal graph (see Algorithm 5). For Catena-BRG, this increases the time-memory tradeoff only by a linear factor (as shown in [8]), whereas for Catena-DBG (see Algorithm 3), this technique leads to a λ-memory-hard function (see Definition 2.3).

7.2. The Garbage-Collector Attack

As we argued above (and throughout this entire paper), memory-demanding password scrambling is an excellent defense against common attack patterns. Typical attackers try plenty of password candidates in parallel, and this gets a lot more costly if they need a huge amount of memory for each candidate. The defender, on the other hand, will not try more than one password candidate in parallel, and the parameters (especially the "garlic") should be chosen such that that amount of memory is easily available to the defender.

But, memory-demanding password scrambling may also provide completely new attack opportunities for the adversary. If we allocate a huge block of memory for password scrambling, holding v_0, v_1, …, v_{G-1} (with $G = 2^g$ for BRH_λ^G and DBH_λ^G), this memory becomes "garbage" after the password scrambler has terminated, and will be collected for reuse, eventually. One usually assumes that the adversary learns the hash of the secret. The *garbage-collector attack* assumes that the adversary additionally learns the memory content, i.e., the values v_i, after the termination of the password scrambler. Next, we discuss how an adversary can benefit from such attacks. Based on a similar argument as in Section 7.1, we only consider the BRH_λ^G operation.

- For ROMix, the value
$$v_0 = H(x) \tag{7.1}$$
is a plain hash of the original secret x. That means, the malicious garbage collector can bypass ROMix completely and directly search for x with $H(x) = v_0$. Each password candidate can thus be checked in time and memory $\mathcal{O}(1)$.
- At a first look, $(G, 1)$-bit reversal hashing seems to provide some defense. We have
$$r_0 = H(v_0, v_{2^g-1}) = H(H(x), v_{2^g-1}), \tag{7.2}$$
and later v_0 is overwritten by the assignment $v \leftarrow r$. Thus, the adversary needs to compute the value v_{2^g-1}, which can be done with $\mathcal{O}(1)$ space in time $\mathcal{O}(2^g)$, i.e., the adversary cannot bypass bit-reversal hashing, but can bypass the storage-demanding part of it.
- Even that may be too optimistic. Overwriting v is algorithmically ineffective and might be removed by an optimizing compiler. In that case, v_0 would not be overwritten at all and thus, Equation 7.1 would apply, and the same attack as for ROMix would become possible.
- Fortunately, (G, λ)-bit-reversal hashing provides a decent defense against garbage-collector attacks. Overwriting v is algorithmically effective in all but the last round of Algorithm 5. For (G, λ)-bit-reversal hashing, the x in Equation 7.2 is not the secret, but rather the output of $(G, \lambda - 1)$-bit-reversal hashing.

Thus, neither ROMix nor $(G, 1)$-bit-reversal hashing provide much defense against garbage-collector attacks. On the other hand, (G, λ)-bit-reversal hashing does provide a decent defense against such attacks: it is at least as secure as $(G, \lambda - 1)$-bit-reversal hashing against conventional attacks. Thus, whoever worries about these kind of attacks, just needs to increment λ.

7.3. Justification of the Generic Design

Catena can be seen as a mode of operation for cryptographic hash function H and a (λ)-memory-hard function F_λ, 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 enables us to choose a larger garlic factor, i.e., to use more memory for Catena, and thus, 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 F_λ, 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 parameter λ, g_0, F_λ, and m are fixed values.

8.1. Catena for Proof of Work

The concept of proofs of work was introduced by Dwork and Naor [16] 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* " [16]. 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 [42].

As an advancement to CPU-bound function,Abadi et al. [1], and Dwork et al. [15] 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 [17] a compact representation for this table by using a time-memory trade-off for its generation. Dziembowski et al. [19] 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 [19], 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 both introduced instantiations of Catena and let λ and F_λ be fix.

Guessing Secret Bits (Pepper). At the beginning, S chooses fixed values for pwd, t, s and g, 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 $h = \text{Catena}_\lambda(pwd, t, s, g)$ and sends the tuple $(pwd, t, s_{[0,k-p-1]}, g, h, p)$ 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 $h' = \text{Catena}_\lambda(pwd, t, s', g)$ 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_λ (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 m-bit password $pwd, t, s,$ and g. Then, S computes $h = \text{Catena}_\lambda(pwd, t, s, g)$ and sends the tuple (t, s, g, m, h) to C. The client C then has to guess the password by computing about 2^m times $h' = \text{Catena}_\lambda(pwd', t, s, g)$ 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^m computations of Catena (and about 2^m comparisons for $h = h'$). Hence, in this case the effort of C is scalable by adapting the length m of the password. Furthermore, S can adjust the effort of C by excluding m 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 $h_i = \text{Catena}_\lambda(pwd, t, s_{0,\ldots,|s|-2} \mid\mid i, g)$ 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

Algorithm 6 Catena-KG

```
Input: $\lambda$ {Depth}, $pwd$ {Password}, $t'$ {Tweak}, $s$ {Salt}, $g_0$ {Min Garlic}, $g$ {Garlic},
    $F_\lambda$ {Instance}, $m$ {Output Length}, $\ell$ {Key Size}, $\mathcal{I}$ {Key Identifier}
Output: $k$ {$\ell$-Bit Key Derived from the Password}
1: $x \leftarrow \textsc{Catena}_\lambda(pwd, t', s, g_0, g, F_\lambda, m)$    {with $m = |H(\cdot)|$}
2: $k \leftarrow \emptyset$
3: for $i = 1, \dots, \lceil \ell/|n| \rceil$ do
4:     $k \leftarrow k \mid\mid H(i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$
5: end for
6: return Truncate$(k, \ell)$ {truncate $k$ to the first $\ell$ bits}
```

time is not effected. Nevertheless, the total memory usage and the computational effort are increased by a factor b. This also holds for an adversary, since it has to 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 $\textsc{Catena}_\lambda$ by splitting it into two functions F (typically F_λ) and H, where F 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 $y = F(\cdot)$ 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 $\textsc{Catena}_\lambda$, 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

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

Note that for key derivation is makes no sense to give the user control over the output length m of $\textsc{Catena}_\lambda$. 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 $\textsc{Catena}_\lambda$ is followed by an output transform that takes the output x of $\textsc{Catena}_\lambda$, 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 [18]. 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 $\textsc{Catena}_\lambda$ 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 $\textsc{Catena}_\lambda$ with larger security parameters and then run the output transform for each key.

In contrast to the password hashing scenario where a user want to log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [43], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, for Catena-KG, we recommend to use $g = 21$ (when instantiated with a BRG^G_λ) and $g = 17$ (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 $\textsc{Catena}_\lambda$ (Line 1 of Algorithm 6). Next, we show that Catena-KG a good pseudorandom function (PRF) in the random oracle model.

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

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

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 $\textsc{Catena}_\lambda$ 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 $\textsc{Catena}_\lambda$. The probability for this event can be upper bounded by similar arguments as in the proof of Theorem 4.2. ∎

Chapter 9

Acknowledgement

Thanks to Bill Cox, Eik List, Colin Percival, Stephen Touset, Steve Thomas, and Alexander Peslyak for their helpful hints and comments, as well as fruitful discussions. Next, we would like to thank Ben Harris who discovered two minor miss-matches between the specification and the reference implementation of CATENA. We would also like to thank Sascha Schmidt for writing the current reference implementation of CATENA-DBG. Finally, we thank Alexander Biryukov and Dimitry Khovratovich for pointing out a flaw in our proof of the time-memory tradeoff for the BRH_λ^G operation by providing a tradeoff cryptanalysis [8].

Chapter 10
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 February 20, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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.

[22] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. Cryptology ePrint Archive, Report 2013/525, 2013. `http://eprint.iacr.org/`.

[23] 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.

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

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

[26] 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.

[27] 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.

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

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

[30] 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.

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

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

[33] 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.

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

[35] 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.

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

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

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

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

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

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

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

[43] 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.

Appendix A

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 BRH_λ^G or the DBH_λ^G, where each vertex within this graph depends at least on its predecessor, thus, providing a sequential structure. More detailed, if one thinks of representing all vertices within a BRH_λ^G or a DBH_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 B

Test Vectors

B.1. Test Vectors for Catena-BRG-BLAKE2b

Lambda:	3	
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:	`bb a6 51 dc 8b fb 89 73 d2 6c ef 99 16`	(64 octets)
	`e0 fe 7d 8b 66 7b f6 fb 78 c4 fa 16 9c`	
	`f1 de 97 0b 99 d8 28 70 57 be c1 c8 14`	
	`30 c3 0c 5c 27 b6 ba a0 6a b3 0c da f3`	
	`10 45 cb e6 5e f5 59 88 b0 1d 71 fc`	

Lambda:	3	
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:	`a1 fa 2e d2 cc 66 8f 5e d3 0b 96 b3 ad`	(64 octets)
	`a3 db a0 14 85 93 ff 2b b8 04 fd d1 d1`	
	`7b 3f 3a ba 36 3b b6 be 68 fb 33 0f b4`	
	`51 48 7c 18 b8 e1 8f 1a c5 44 4b 5a e2`	
	`f1 19 8c 14 57 23 1a 1b 2c ea ec e6`	

B.2. Test Vectors for Catena-BRG-SHA-512

Lambda:	3	
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:	`a8 71 61 a2 ba 93 05 d6 50 c4 a6 e5 ee`	(64 octets)
	`9e f9 fb 05 54 f7 7c 5c b2 5e e1 08 c5`	
	`7b 09 76 f9 b3 17 97 40 03 ab 7e 3d 59`	
	`ba 82 4d 6a f6 c2 0e c4 29 f2 a6 1f 92`	
	`85 16 69 f5 79 1e cb 98 16 ec b4 14`	

Lambda:	3	
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:	`70 b8 92 ee 68 98 f1 7a 16 cb 2c c4 35`	(64 octets)
	`37 6c ca 1b e0 b8 d3 98 cd 07 b0 68 24`	
	`ad 3e 3f 91 f4 1f 59 ab b5 ef 18 42 3d`	
	`52 73 ee 3d 0b f0 ac 6d 90 23 09 59 2e`	
	`f8 5c 88 11 cb 01 44 1c 0e 9d 29 85`	

B.3. Test Vectors for Catena-DBG-BLAKE2b

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:	`f2 0e 68 8e e8 09 38 f4 3b 23 7a fe 3c`	(64 octets)
	`d0 7c 56 19 64 c4 68 ef 20 a1 c2 cf 67`	
	`8d 99 04 53 87 80 7e dc 4e 30 65 54 bd`	
	`ec 9c cc 66 aa a3 e0 a1 b3 85 3b 04 01`	
	`e7 ba cc b8 16 cf dc aa 02 42 4e 6e`	

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)
Output:	`94 f9 01 e9 59 ea 36 aa 33 55 ad 3d 0a`	(64 octets)
	`e4 0a 26 9e 43 71 2d ab ea 54 dc 6c 6b`	
	`ca 75 d3 c4 8e bd 62 bd bf af de bb 2a`	
	`dc 91 36 db 93 45 63 d3 ab 1f 80 fb 48`	
	`67 2b 85 55 e4 7c 74 ca 37 00 72 a5`	

B.4. Test Vectors for Catena-DBG-SHA-512

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:	`a7 32 51 3d 96 bf 80 cf 26 d2 fd dd 7c`	(64 octets)
	`fd 8f 4a 80 d0 d3 35 17 a3 61 b2 4a c4`	
	`48 76 1f e8 ac 25 38 63 9b 4a 3b 39 18`	
	`40 f8 85 d7 b2 2d 57 4b 4a 18 5e 73 65`	
	`82 bc a1 27 dc 2f b4 5d 66 e4 55 46`	

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:	`0c 1e 7a 67 c0 d9 96 54 a9 c0 88 a1 21`	(64 octets)
	`8c ba ce 7f 5c 2d 33 3a 8e 30 93 5a 96`	
	`74 d5 cf 80 59 9f f1 95 69 d9 ee 9f be`	
	`fd 6a d7 4a f7 29 bd 3d 7e 21 97 76 3a`	
	`20 4a a0 e4 64 59 37 e6 a2 7b da 7a`	

Appendix C

Illustration of a BRG_4^8

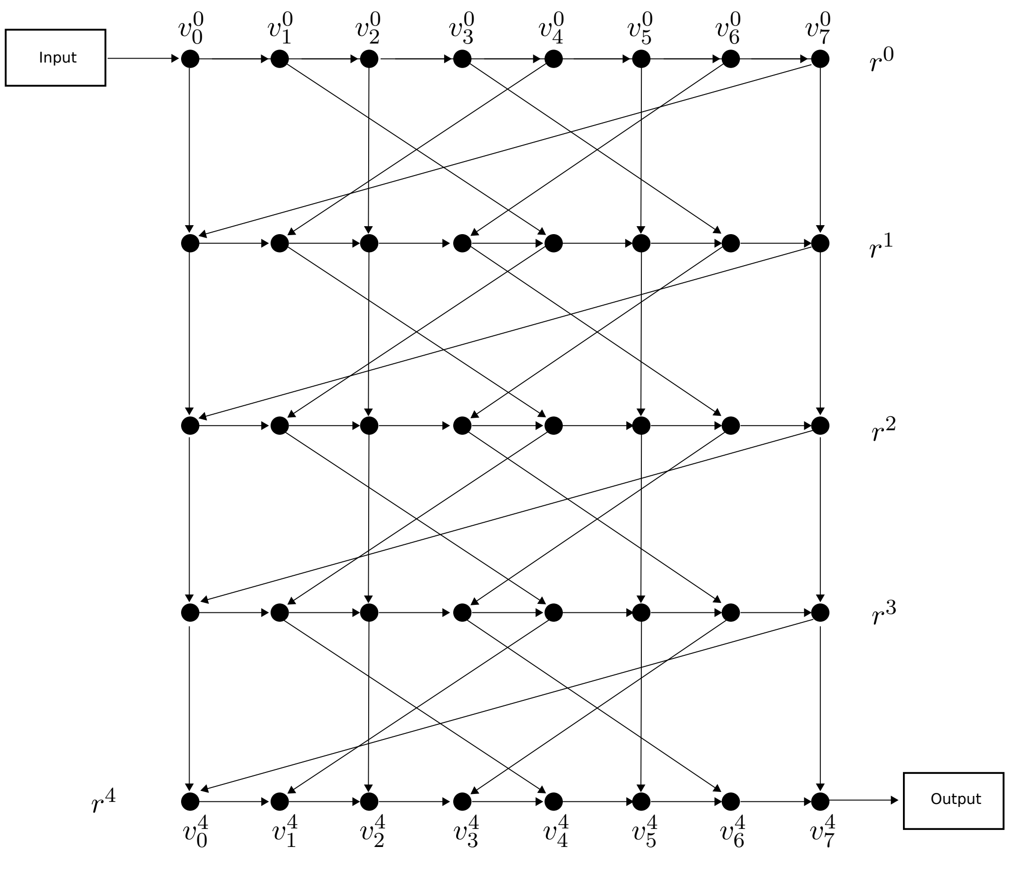

Figure C.1.: An $(8,4)$-bit-reversal graph.

Appendix D

Illustration of a DBG_2^8

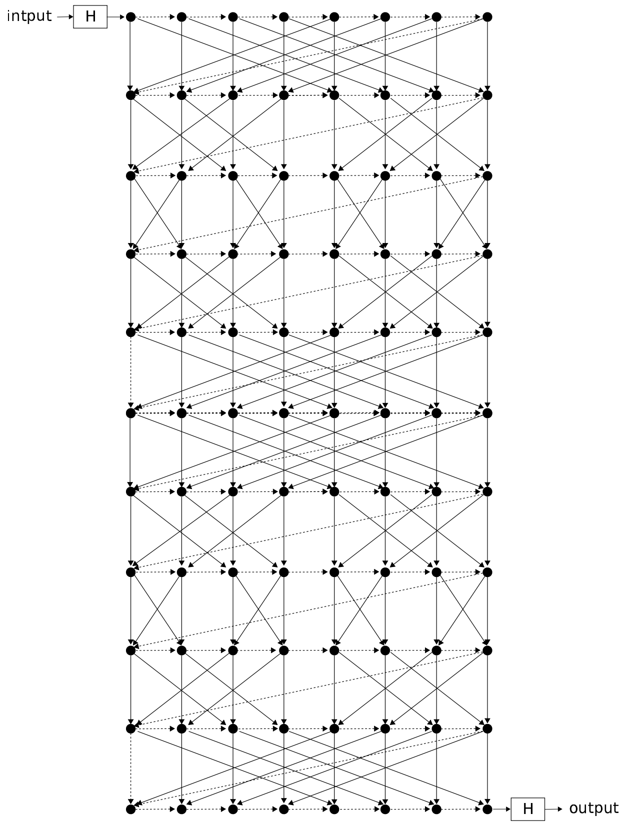

Figure D.1.: An $(8, 2)$-double-butterfly graph.