The Catena Password Scrambler

Submission to the Password Hashing Competition (PHC)

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

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

Bauhaus-Universität Weimar

Version 1.1

May 15, 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 1.0 to Version 1.1

- Prepend the version ID byte, currently `0xFF`, to the tweak. (cf. Chapter 3 and Section 7.3)
- Swapped the two input parameters of the hash function H in
  - Line 6 of Algorithm 2 in Chapter 3 and
  - Line 5 of Algorithm 3 in Section 5.1.

Executive Summary

Catena is a novel and provably secure password scrambler 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. Catena can be instantiated by a single standard cryptographic primitive, *i.e.,* a hash function H, *e.g.,* BLAKE2b [5] or SHA-512 [29]. Further, Catena has a sound and elegant design given by a simple and well-understood graph-based structure called "Bit-Reversal Graph", where the depth of the graph is determined by user-chosen parameter λ. The time-memory tradeoff (TMTO) analysis of Catena is based on the pebble-game approach, which was – mostly in the 1970s and 1980s – extensively used to study time-memory tradeoffs. Using this technique, 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 the parameter λ and a cryptographic hash function H – 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 *server relief* protocols 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.

The latter point implies that Catena provides high resilience against massively parallel attacks, *e.g.,* GPU-based attacks. More detailed, if the available memory is halved, the required computational time is increased 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` [22] and `sha512crypt` [11].

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 [8]. 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 [28] 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` [31].

Nevertheless, two aspects of `scrypt` did trouble us. First, `scrypt` is quite complex, since it combines two independent cryptographic primitives (the SHA-256 hash function and the Salsa20/8 core operation) and four generic operations (HMAC, PBKDF2, BlockMix, and ROMix). Second, the data flow of the ROMix operation is data dependent, *i.e.,* ROMix reads data from addresses which are password-dependent. This renders ROMix, and thus `scrypt`, vulnerable to cache timing attacks [19]. Moreover, we show that `scrypt` is vulnerable against the *garbage-collector attack*, *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 5.2).

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 garbage collector adversary will then only learn the information written at the end of the scrambler operation.

Flexibility Desirables. 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 let a computer at the client side allocate the memory and perform the password scrambling, without burdening the server,
- *pepper* and *garlic*: security parameters to tune time and memory requirements,
- *client-independent updates*, to set the security parameters to higher values, 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 [19].

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 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 locks out 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 cryptograhic operation, based on a cryptographic hash function. An alternative would have been some new primitive with the structure of `Catena`. Section 5.3 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 flexibility desirables, 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 scrambler properties we use in the entire paper. Chapter 3 introduces the specification of Catena, our new password scrambler, as well as our parameter recommendations for the PHC. In Chapter 4 we discuss the functional and security properties of Catena. Chapter 5 contains background information and the origins of the Catena design. A comprehensive security analysis of Catena is given in Chapter 6. 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 7. 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 we later use to show that `Catena` satisfies a certain time-memory tradeoff (see Section 6, Security Analysis). Furthermore, we introduce necessary definition and notations used throughout this paper.

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

On the other hand, edges represent the data flow of an operation, *i.e.,* the operation $x \leftarrow y \circ z$ 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 [21] 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 [30] introduced the pebble game as a method for analyzing TMTOs on directed acyclic graphs, which became an important tool for that purpose, see [34–38]. The pebble game has been occasionally used in cryptographic context (see [17] 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.

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 $S = 1$.

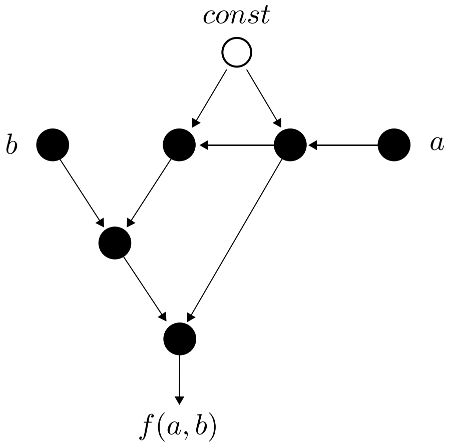

Figure 2.1.: Directed acyclic graph for $f(a, b) = (a \circ const) \circ const) \circ b) \circ (a \circ const)$.

Pebble-Game and Password Scramblers. Note that the upper and lower bounds we prove are abstract theoretical results, and 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 two 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 5.3.

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 generalized the notion of memory-hardness from [31].

Definition 2.2 ($\lambda-$Memory-Hard Function).
Let g denote the memory-cost factor. Let f be a function, which is computed on a Random Access Machine using $S(g)$ space and $T(g)$ operations with $G = 2^g$. We call f λ-memory hard, if it holds that

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

It is easy to see that if S decreases by a factor of about x, T increases by a factor of about x^{λ}, *e.g.,* using b cores, the required memory per core is decreased by a factor of $1/b^{\lambda}$, and vice versa. Note that the notion *memory-hard* from [31] is equivalent to the notion *1-memory-hard.*

λ-Memory-Hardness vs. Sequential Memory-Hardness. In [31], Percival introduced the notion of Sequential Memory-Hardness (SMH), which is satisfied by his introduced password scrambler `scrypt`. Based on this notion, an algorithm is sequential memory-hard if an adversary has no computational advantage from the use of multiple CPUs, *i.e.,* using b cores to compute the output of `scrypt` 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 λ-Memory-Hardness (λMH). Thus, SMH is a desirable goal when designing a memory-demanding password scrambler. In this section we discuss why our presented password scrambler `Catena` is λMH but not SMH, without referring to details of `Catena`, which are presented in Chapter 3.

Note that a further goal of our design was to provide resistance against cache-timing attacks, *i.e.,* `Catena` should satisfy 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 DAG.

Usually, a DAG can be at least partially computed in parallel. Assuming that one has b cores 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`, one can always partition its corresponding DAG into such subgraphs and hence, `Catena` can be at least partially computed in parallel, which is a contradiction to the definition of SMH. 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 published password scrambler which is both λMH and has a password-independent memory-access pattern.

Password Recovery (Preimage Security). For a modern password scrambler it should hold that the advantage of an adversary (modelled 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 2.3 (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*

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

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

In Section 6.2 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 [25], 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 [26] and 73% of all Twitter accounts do not have at least one tweet per month [33]. 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 [27] and more recent in [10]. 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 2.4 (Resistance against Cache-Timing Attacks). *Suppose the function $\mathcal{F} : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n$ processes arbitrary large data together with a secret value K with $|K| = k$, and outputs a fixed length value of size n. We call $\mathcal{F}$ resistant against cache-timing attacks iff its control flow does not depend on the secret input K.*

Key-Derivation Function (KDF). Beyond authentication, passwords are also used to derive symmetric keys. Obviously, one can just use the output of the password scrambler as a symmetric key – perhaps after truncating it to the required key size. This is a disadvantage if one 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 2.5 (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.

2.3. Notational Conventions

Identifier	Description
pwd	password
λ	depth of a λ-Bit-Reversal Graph (λ-BRG)
s	salt (public random value)
p	pepper (secret bits of the salt)
t	tweak
g_0, g	minimum garlic; current garlic with $G = 2^g$
PS	Password Scrambler
m	output length 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
$\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$	λ-BRG with depth λ and 2^g input vertices
r^i	i-th row of a $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$
e_i	i-th edge of a $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$
v_i^j	i-th vertex in the j-th row of a $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$
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
AD	associated data
K	secret key
$\|X\|$	size of X in bits or size of a set X

Table 2.1.: Notations used throughout this document, leaving out notation used once to explain a single issue.

Chapter 3

Specification

In this chapter we introduce our λ-memory-hard password scrambler 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 λ-Bit-Reversal Hashing (λ-BRH) operation is called (see Algorithm 2), where the input is padded with as many 0's as necessary so that $x \mid\mid 0^*$ fits the output size of the underlying hash function. The password-dependent input of a cryptographic hash function H. By default, `Catena` uses BLAKE2b for H.

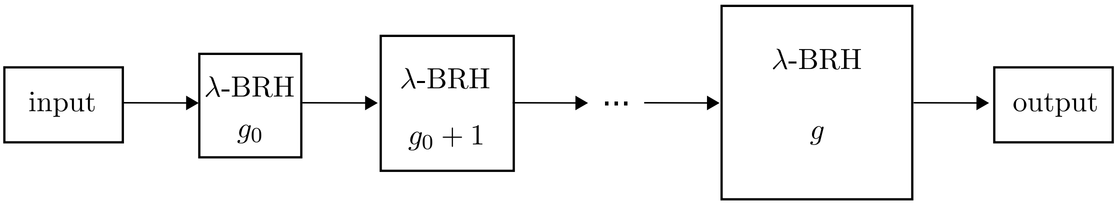

Figure 3.1.: The general idea of applying the λ-Bit-Reversal Hashing operation $(g - g_0)$ times whereas the value for g is increased by 1 in each iteration.

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

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

Where the first byte V denotes the version ID, currently `0xFF`, 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 7.3), and $d = 2$ for the proof of work scenario (see Section 7.1). The remaining

Algorithm 1 Catena

```
Input: $pwd$ {Password}, $t$ {Tweak} $s$ {Salt}, $g_0$ {MinGarlic}, $g$ {Garlic}, $m$ (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 $c = g_0, \dots, g$ do
 4:     $x \leftarrow \lambda\text{-}BRH(c, x \mid\mid 0^*)$
 5:     $x \leftarrow H(c \mid\mid x)$
 6:     $x \leftarrow truncate(x, m)$
 7: end for
 8: return $x$
```

Algorithm 2 λ-Bit-Reversal Hashing ($\lambda-$BRH)

```
Input: $g$ {Garlic}, $x$ {Value to Hash}, $\lambda$ {Depth}
Output: $x$ {Hash Value}
 1: $v_0 \leftarrow H(x)$
 2: for $i = 1, \dots, 2^g - 1$ do
 3:     $v_i \leftarrow H(v_{i-1})$
 4: end for
 5: for $k = 0, \dots, \lambda - 1$ do
 6:     $r_0 \leftarrow H(v_{2^g-1} \mid\mid v_0)$
 7:     for $i = 1, \dots, 2^g - 1$ do
 8:         $r_i \leftarrow H(r_{i-1} \mid\mid v_{\tau(i)})$
 9:     end for
10:     $v \leftarrow r$
11: end for
12: return $r_{2^g-1}$
```

possible values for d are reserved for future applications. The second 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(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 doe not matter.

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 different 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.

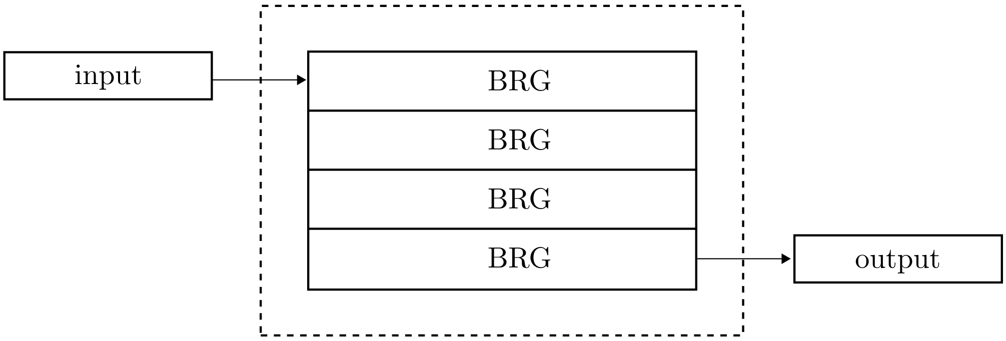

Figure 3.2.: General idea of a 4-Bit-Reversal Graph (see Definition 3.2). A detailed depiction of a 4-BRG can be found in Appendix D.

3.1. The Core of `Catena`

The core of `Catena` is a λ-memory-hard function called λ-Bit-Reversal Hashing (λ-BRH), whose formal definition is given in Algorithm 2 and an abstract depiction when $\lambda = 4$ is given in Figure 3.2 (see Figure D.1 for a detailed depiction). It requires $\mathcal{O}(2^g)$ invocations of H for a fixed value of x. The first input of λ-BRH denotes the garlic parameter g with $g = \log_2(G)$ (see Definition 2.2), the second input denotes the value to process, and the third input denotes the depth (number of stacked Bit-Reversal Permutations). Thus, g determines the units of memory required to execute λ-BRH. Moreover, increasing g by one doubles the computational effort for computing the password hash. It represents a graph-based structure called λ-Bit-Reversal Graph (λ-BRG), which itself is generated by stacking λ Bit-Reversal Permutations. The definition of a Bit-Reversal Permutation is given below (see Definition 3.1).

Definition 3.1 (Bit-Reversal Permutation τ). *Fix a number $k \in \mathbb{N}$ and represent $i \in \mathbb{Z}_{2^k}$ as a binary k-bit number, $(i_0, i_1, \dots, i_{k-1})$, i.e.,*

$$i = \sum_{j=0}^{k-1} 2^j i_j.$$

The bit-reversal permutation $\tau : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ is defined by

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

In Definition 3.2 we introduce a generic definition of the λ-BRG, where the edges within such a graph define the control flow of the λ-BRH operation (see Algorithm 2).

Definition 3.2 (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 λ-Bit-Reversal Graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ consists of $(\lambda + 1) \cdot 2^g$ vertices*

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

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

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

For example, we call a λ-BRG with $\lambda = 1$ a *Sequential Bit-Reversal Graph* (SBRG), which, for $G = 2^3 = 8$ input vertices, is shown in Figure 3.3. Note that an SBRG is almost identical – except for one additional edge $e = (v_{2^g-1}^0, v_0^1)$ – to the Bit-Reversal Graph presented by Lengauer and Tarjan in [24].

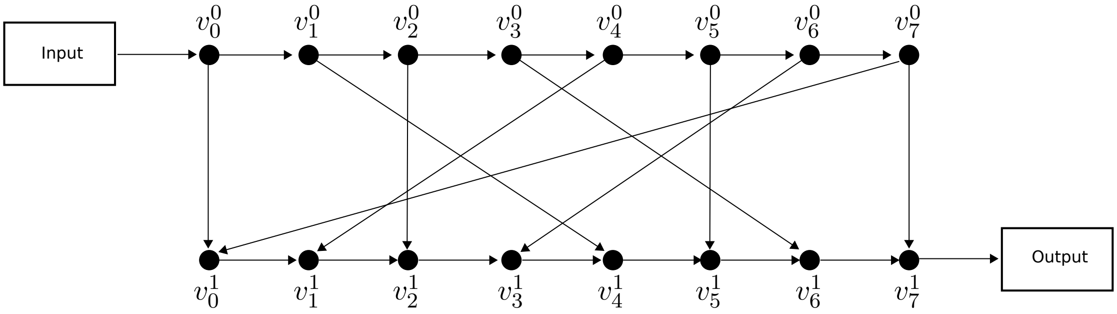

Figure 3.3.: A λ-Bit-Reversal Graph with $\lambda = 1$ (SBRG) and $g = 3$.

3.2. 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, and 2) SHA2-512 [29] since it is well-analyzed [2, 20, 23], standardized, and widely used, *e.g.,* in sha512crypt, the common password scrambler in several Linux distributions [11].

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 $\lambda-$BRG, *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 [18] or SHA3-512 [7].

Cost Parameter. Table 3.1 presents the recommended parameter sets for Catena 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.

Algorithm	Hash Function	Min/Current Garlic	λ	Salt size	Time
Catena	BLAKE2b	18/18	3	128 bits	0.34 sec
Catena	SHA-512	17/17	3	128 bits	0.41 sec
Catena-KG	BLAKE2b	21/21	4	128 bits	3.2 sec
Catena-KG	SHA-512	20/20	4	128 bits	3.9 sec

Table 3.1.: Recommended parameter sets for COTS systems. All timings are measured on a Intel Core i5-3210M CPU (2.50GHz) system.

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

Implementation. A current *optimized* reference implementation can be found on

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

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

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
s	salt	byte string
$\lvert s \rvert$	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

Properties of Catena

4.1. 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. If enough memory is available (either for an adversary or an honest entity), all input vertices (plus one for the output path – see Chapter 6 for details) can be stored in the cache. 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 $h \leftarrow \texttt{Catena}(pwd, t, s, g_0, g, m)$ be the hash of a specific password pwd, where t, s, g_0, g, and m denote tweak, the salt, the minimum garlic, the garlic, 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:

$$h' = truncate(H(g' \mid\mid \lambda\text{-BRH}(g', h)), m).$$

It is easy to see that the equation $h' = \texttt{Catena}(pwd, t, s, g_0, g', m)$ 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` is then transmitted to the server. Next, the server computes the password hash by applying the hash function H. 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.

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` 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 [32]. Let $X := \{pwd, t, s, g_0, g, m\}$. Then, the output of `Catena-KG` is computed as follows:

$$y = \texttt{Catena}_K(userID, X) := \texttt{Catena}(X) \oplus H(K \mid\mid userID \mid\mid g \mid\mid K),$$

where `Catena` is defined as in Algorithm 1 and the $userID$ is a unique and user-specific identification number which is assigned by the server. Now, we show what happens during the client-independent update, *i.e.,* when $g = g + r$ 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. Computes $x = y \oplus z$, where y denotes the current keyed hash value.
3. Update x, *i.e.,* $x = H(c \mid\mid \lambda\text{-BRH}(c, x \mid\mid 0^*))$ 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 [9] for details). Another possibility is to derive the 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.

Observation on Cache-Time Misses. The following observation holds for a $1-$BRG, but can also be easily generalized for λ-BRG 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 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, $\ldots$, 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 11 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.

4.2. Security Properties

$\lambda-$Memory-Hard Function. In Chapter 6 we analyze the core of `Catena`, *i.e.,* the λ-BRG, and show that it satisfies the properties of a $\lambda-$memory-hard function (see Definition 2.2), *i.e.,* given $S(g)$ memory units and $T(g)$ time, `Catena` satisfies

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

where g denotes the garlic parameter and λ the depth of the λ-BRG. We present a proof for this fact in Chapter 6 by using the proof technique described in 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 the input (pwd) in practical time. This requirement becomes mostly crucial in a situation of a leaked password-hash database. In Section 6.2 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, one wants the resulting secret key to be indistinguishable from a random string of the same length. In Section 6.3 we show that for a secret input (pwd), the output of `Catena` looks random. The presented proof is based on the assumption that the underlying hash function behaves like a random oracle.

Cache-Time Resistance. From Definition 2.4 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 a λ-BRG, 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).

Chapter 5

Design Discussion

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

5.1. From ROMix to Bit-Reversal Hashing

The core idea for the `scrypt` password scrambler is given by the definition of **ROMix** (see Algorithm 3 (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 $v_0, \ldots, v_{G-1}$, and then runs through a main loop reading each v_i once, on the average. It has a time-memory tradeoff satisfying $ST = O(G^2)$. Thus, for $S = O(G)$ units of space, the time required to evaluate ROMix is $T = O(G)$. 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 [31], 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 $v_0, \ldots, v_{G-1}$ 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 5.2.

Algorithm 3 ROMix and Bit-Reversal Hashing (BRH)

ROMix(x,G)

```
 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(x,G)

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

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 Line 8). This minor modification does not have a strong advantage over $H(v_{i-1} \oplus v_j)$. However, the size of v_{i-1} and v_j is 512 bits, each, our default choice for H is BLAKE2b, where the evaluation of an 512-bit input is as fast as the evaluation of an 1024-bit input. If one chooses another H, one may tweak `Catena` by using XOR 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.

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

This approach leads to the realization of the **bit-reversal hashing** operation (see Algorithm 3). It is the core of `Catena` with $\lambda = 1$, it trivially solves the first ROMix issue, and its contribution in solving the second ROMix issue is discussed in Section 5.2.

Algorithm 4 Main Loop of λ-Bit-Reversal Hashing

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

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:

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

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 just make λ repetitions of Lines 5–10 of the bit-reversal hashing algorithm for the main loop (see Algorithm 4; and Algorithm 2 for the specification of the λ-bit-reversal hashing operation). Thus, we get a λ-memory-hard hash function where, in practice, any $\lambda \geq 3$ appears to be sufficient to render that approach all but useless.

Remark. For ease of reading, our presentation of bit-reversal hashing maintains not just the memory $v_0, \dots, v_{2^g-1}$, but also a *shadow copy* $r_0, \dots, r_{2^g-1}$ for the newly computed v_i. In Line 10 of Algorithm 3, v is overwritten by r. Our reference implementation realizes the same function in a more memory-efficient way, where it maintains a single copy of the most recently computed v_{i-1} and otherwise overwrites v directly.

5.2. 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 $v_0, v_1, \ldots, v_{G-1}$ (with $G = 2^g$ for bit-reversal hashing), 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.

- For ROMix, the value
$$v_0 = H(x) \tag{5.1}$$
is a plain hash of the original secret x. That means, the garbage collector adversary 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 $O(1)$.

- At a first look, bit-reversal hashing seems to provide some defense. We have
$$r_0 = H(v_0, v_{2^g-1}) = H(H(x), v_{2^g-1}), \tag{5.2}$$
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 $O(1)$ space in time $O(2^g)$, *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 5.1 would apply, and the same attack as for ROMix would become possible.

- Fortunately, 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 4. For λ-bit-reversal hashing, the x in Equation 5.2 is not the secret, but rather the output of $(\lambda - 1)$-bit-reversal hashing.

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

5.3. Justification of the Generic Design

`Catena` can be seen as a mode of operation for cryptographic hash function H, and therefore, it fulfill the properties of an 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, whereas a primitive password scrambler could not inherit security assurance from an underlying primitive. Furthermore, Catena is easy to analyze since it is built solely on an existing cryptographic hash function and a novel mode of operation. 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 cryptanalyzed 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 6

Security Analysis

Since the λ-BRG defines the core structure of `Catena`, and therefore `Catena` inherits the properties of the λ-BRG. Thus, it is sufficient to provide a security analysis of the λ-BRG. We already mentioned that a λ-BRG is described by the λ-BRH operation (see Section 3.1), and furthermore, it is easy to see that the control flow of the λ-BRH operation can be represented as a DAG, *i.e.,* the vertices of the defined graph represent the inputs and outputs of the hash function and an edge denotes a hash function invocation.

Thus, we adapt the model from [30], *i.e.,* the pebble game, to analyze our scheme regarding to the properties of a $\lambda-$memory-hard function. Before we present our security claims and their corresponding proofs, we clarify, why the pebble game fits well for analyzing a λ-BRG regarding to its memory hardness.

- A λ-BRG can be seen as a DAG, since the inputs and outputs of a hash function invocation determined the direction of the edges.
- A λ-BRG is a DAG with bounded indegree, since the number of inputs to one hash function invocation is limited by two.
- It follows the pebble game approach, since the output of the underlying hash function is not computed till all necessary inputs are available.
- The goal is to compute the output of the λ-BRG, *i.e.,* the only vertex without a successor.
- Due to the sequential structure of a λ-BRG, each vertex in the graph must be pebbled at least once to reach the goal.
- Given a certain number of memory units (registers), find a lower/upper bound on the number of hash function invocations.

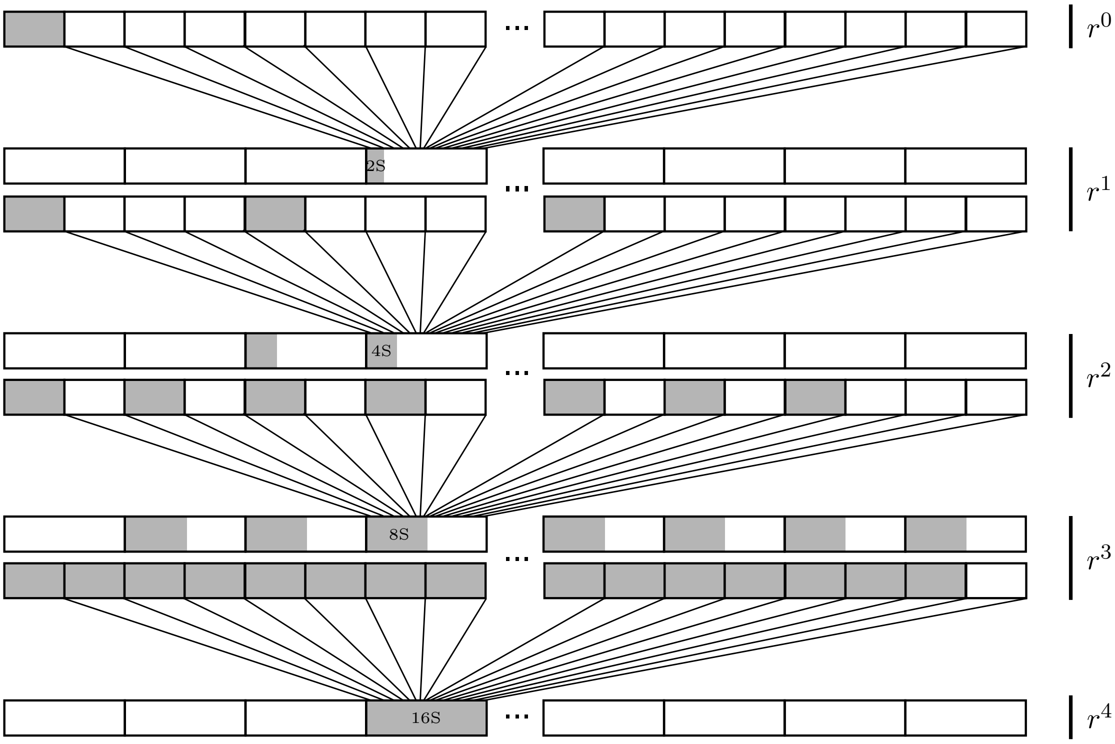

Figure 6.1.: Depiction of our proof idea (lower bound) for a λ-BRG with $\lambda = 4$ and arbitrary number of input vertices. It shows the parts of each row (gray cells) which have to be pebbled to pebble one interval of the bottom row (of size $16S$).

6.1. Memory Hardness

In this section we first present a lower bound for pebbling a λ-BRG, and second, we show one possible way to pebble a λ-BRG given a certain number of pebbles (upper bound).

Theorem 6.1 (Lower Bound). *Let λ be a fixed value and $\lambda + 1 \leq S \leq G/2^\lambda$. Then, pebbling a λ-BRG $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input vertices with S pebbles takes time*

$$T(g) \;\geq\; \frac{G^2}{2^{\lambda+1}S} \cdot \prod_{i=1}^{\lambda-1} \left(\frac{G \cdot \left(2^\lambda - 2^i\right)}{\left(2^{2\lambda} - 2^{\lambda+i+1} - 2^\lambda\right) \cdot S} \right) \;=\; \Omega\left(\frac{G^{\lambda+1}}{S^\lambda} \right).$$

Proof Idea. Let $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ denote a λ-BRG as defined in Definition 3.2. Then, we divide each row, consisting of $2^g = G$ pebbles, into intervals of length $2^\lambda S$. Thus, each row consists of $2^{g-s} = G/2^{\lambda} \cdot S$ intervals. Note that the latter equation only holds when S is a power of 2. To ensure this we give an adversary the minimum number of pebbles to fulfill the equation $2^s = 2^{\lambda}S$. Next, starting at the bottom row r^{λ} looking at one particular interval, we estimate the minimum number of moves which have to be done to pebble this interval. It is easy to see that the number of movements depend on the configuration of the adjacent row $r^{\lambda-1}$. Therefore, we again divide this row into $2^{g-s} = G/2^{\lambda}S$ intervals of length $2^{\lambda}S$ and looking on the minimum pebble movements to pebble a particular interval at row $r^{\lambda-1}$. This approach is executed *recursively* until the top row is reached (see Figure 6.1). Then, the minimum amount of pebble movements, needed to pebble a single bottom row interval, is determined by multiplying up the individual bounds. This enables us to come up with a lower bound of necessary pebble movements to pebble a λ-BRG.

Proof. At first we divide the bottom row r^{λ} into 2^{g-s} intervals of length 2^s. Let I_i^{λ} denote the i-th interval of r^{λ} consisting of the vertices $v_{i2^s}^{\lambda}, \ldots, v_{(i+1)2^s-1}^{\lambda}$. Furthermore, we denote z_i^{λ} as the first time that a pebble is placed on the last vertex of the interval I_i^{λ}, *i.e.,* $v_{(i+1)2^s-1}^{\lambda}$. Suppose that $z_i^{\lambda} = 0$. Then, we have $z_i^{\lambda} < z_{i+1}^{\lambda}$ for $i = 0, \ldots, 2^{g-s} - 1$. In order to find a lower bound on $z_{i+1}^{\lambda} - z_i^{\lambda}$, *i.e.,* the number of moves necessary to pebble the last vertex in I_{i+1}^{λ}, we observe that at time z_i the interval I_{i+1}^{λ} is pebble-free and thus, all 2^s vertices of this interval have to be pebbled between z_i^{λ} and z_{i+1}^{λ}. By definition of the Bit-Reversal Permutation, the immediate predecessors on the input path of the vertices in I_{i+1}^{λ} divide the Row $r^{\lambda-1}$ naturally into 2^s intervals $I_i^{\lambda-1}$ of length 2^{g-s}, where each vertex of I_{i+1}^{λ} defines the high end of an interval $I_j^{\lambda-1}$. The *high end* of an interval denotes the vertex with the largest index within an interval, *i.e.,* the high end of $I_j^{\lambda-1}$ is given by $v_{(j+1)2^s-1}^{\lambda-1}$.

Since we have at most $S-1$ pebbles left to pebble $r^{\lambda-1}$, at least $2^s - (S-1) > (2^{\lambda}-1)S$ intervals are pebble-free at z_{j-1}. In the following we denote a vertex as *marked* if it is part of an interval which already contains a pebble, and *unmarked* otherwise. Thus, at most $1/2^{\lambda}$ of the vertices in $r^{\lambda-1}$ are *marked*, whereas $(2^{\lambda}-1)/2^{\lambda}$ of the vertices are *unmarked*.

Then we divide $r^{\lambda-1}$ again in 2^{g-s} intervals $J_j^{\lambda-1}$ of length 2^s to re-establish the same configuration as in Row r^{λ}. We denote an interval $J_j^{\lambda-1}$ as *bad* if more than $thresh(\lambda - 1) = 1 - (2^{\lambda-1}/2^{\lambda})$ of its vertices are marked. Since $r^{\lambda-1}$ has at most $G/2^{\lambda}$ marked vertices, at most

$$ratio_{bad}(\lambda - 1) = \left(\frac{2^{\lambda} - (2^{\lambda} - 1)}{2^{\lambda}} \cdot \frac{1}{thresh(\lambda - 1)} \right)$$

of the intervals in $r^{\lambda-1}$ are bad, *i.e.,* at least $1 - ratio_{bad}(\lambda - 1)$ of the intervals are *good*. We apply this strategy for each row until we reach the second row r^1. Thus, the threshold for the i-th row is given by $thresh(i) = 1 - (2^i/2^{\lambda})$ with $1 \leq i < \lambda$. Moreover, for the i-th row it holds that

$$ratio_{bad}(i) = \left(\frac{2^\lambda - \left(2^{i+1} - 1\right)}{2^\lambda} \cdot \frac{1}{thresh(i)}\right). \tag{6.1}$$

Now we consider the final step from Row r^1 to r^0. In r^1, there exist $1/(2^\lambda - 2) \cdot G/2^\lambda S$ *good* intervals of size $2^\lambda S$. Each of these good intervals naturally divides the top row (Row r^0) into $2^\lambda S$ intervals of length $2^{g-s} = G/2^\lambda S$. We denote an interval as *bad*, if the vertex which defines the high end of an interval is marked. Thus, there are at least $2^\lambda S \cdot (1 - tresh(1)) = 2S$ good intervals. Since we have at most $S - 1$ pebbles left to pebble r^0, at least S intervals of length 2^{g-s} have to be pebbled. Thus, the effort for pebbling one interval of r^1 is given by

$$S \cdot 2^{g-s} = 2^{s-\lambda} \cdot 2^{g-s} = 2^{g-\lambda} = G/2^\lambda$$

pebble movements. Based on Equation (6.1), the total number of pebble movements for pebbling one bottom row interval is given by

$$T_{bot} \;\geq\; \frac{G}{2^\lambda} \cdot \prod_{i=1}^{\lambda-1} \left(1 - \left(\left(\frac{2^\lambda - (2^{i+1} - 1)}{2^\lambda}\right) \cdot \left(\frac{1}{1 - \frac{2^i}{2^\lambda}}\right)\right)\right) \cdot \frac{G}{2^\lambda \cdot S}.$$

Since there exist $G/2^\lambda S$ bottom-row intervals, the effort for pebbling a λ-BRG is given by

$$T \;\geq\; \frac{G}{2^\lambda S} \cdot T_{bot},$$

which proves our claim. ■

For better understanding of our proof technique, we provide the proof for $\lambda = 3$ in Appendix B.

Upper Bound. Obviously, a λ-BRG consists of a sequential structure, since each vertex within the graph is at least derived from its predecessor. We denote by $G = 2^g$ the number of vertices within one row of a λ-BRG. Let v^i_j denote the j-th vertex of the i-th row of a λ-BRG with $i \in \{0, \ldots, \lambda\}$ and $j \in \{0, \ldots, G - 1\}$. An λ-BRG can be pebbled in time $T = (\lambda + 1) \cdot G$ using at least $S = G$ pebbles. Therefore, the G pebbles are placed sequentially on the input vertex $v^0_0, \ldots, v^0_{G-1}$. Then, the pebble placed on v^0_0 is moved to v^1_0 and the $G - 1$ remaining pebbles of the first row are sequentially placed on the vertices $v^1_1, \ldots, v^1_{G-1}$. Now, the pebble on the vertex v^1_0 is moved to v^2_0 and the $G - 1$ pebbles are sequentially placed on the vertices $v^2_1, \ldots, v^2_{G-1}$. This line of action is repeated for all $i \in \{0, \ldots, \lambda\}$, *i.e.,* until the output of the λ-BRG is produced.

As the memory requirement for $S = G$ pebbles is usually too large, we have to consider scenarios with $S < G$ pebbles. Now, we argue that the smallest number of pebbles a λ-BRG can be pebbled with is $S = \lambda + 1$. Therefore, assume that $S = \lambda$, *i.e.,* there are λ pebbles $p^0, \ldots, p^{\lambda-1}$ available. A λ-BRG consists of the $\lambda+1$ rows $r^0, \ldots, r^\lambda$. We w.l.o.g. assume that one pebble ($p^{\lambda-1}$) is used to move along the last row, *i.e.,* Row r^λ (we call that pebble the *runner pebble*). Furthermore, we w.l.o.g. assume that the remaining $\lambda-1$ pebbles are placed on the rows $r^1, \ldots, r^{\lambda-1}$. Let x denote the current position of $p^{\lambda-1}$. Then, $p^{\lambda-1}$ can be moved *at most one step* from $v_x^{\lambda+1}$ to $v_{x+1}^{\lambda+1}$ without additional effort *iff* $p^{\lambda-2}$ is already on $v_{\tau(x+1)}^{\lambda}$. Else, due to the structure of the λ-BRG, $p^{\lambda-2}$ has to be moved to $v_{\tau(x+1)}^{\lambda}$. Therefore, $p^{\lambda-3}$ has to be moved to the predecessor of $v_{\tau(x+1)}^{\lambda}$. Since this observation holds for the whole λ-BRG and r^0 does not contain a pebble, none of the pebbles $p^0, \ldots, p^{\lambda-1}$ can be moved (or at least one step depending on the current position of the corresponding predecessors). Obviously, this holds for all $S \leq \lambda$ and thus, pebbling a λ-BRG with $S \leq \lambda$ pebbles is impossible. We now present a formal proof for pebbling a λ-BRG using $\lambda+1$ pebbles.

Lemma 6.2 (Upper Bound ($S = \lambda + 1$)). *Let $\lambda \in \mathbb{N}^+$ be a fixed value and let $G = 2^g$. Then, the λ-Bit-Reversal Graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ can be pebbled with $\lambda+1$ pebbles in time*

$$T = O\left(G^{\lambda+1}\right).$$

Note that we also consider the case $\lambda = 0$, since it eases the understanding of the proof, even if $\lambda \in \mathbb{N}^+$ excludes this case.

Proof.

Induction Hypothesis ($n = \lambda$):

For any fixed $\lambda \in \mathbb{N}$, a λ-BRG can be pebbled with $\lambda+1$ pebbles in time $T = O\left(G^{\lambda+1}\right)$.

Basis ($n \in \{0, 1\}$):

$n = 0$**:**

In this case $\lambda = 0$ and thus, we consider a $0-$BRG, *i.e.,* a graph which consists of G vertices $v_0, \ldots, v_{G-1}$, which are all connected in a sequential way. Let p^0 denote the one pebble we are allowed to use (remind that we are allowed to use $\lambda+1$ pebbles and $\lambda = 0$ in this case). Since each vertex v_i has only one predecessor v_{i-1} for $1 \leq i \leq G-1$, a 0-BRG can be pebbled by moving p^0 in a sequential way G times to pebble v_{G-1}. Thus, the effort for pebbling a $0-$BRG is given by G pebble movements.

$n = 1$**:**

In this case we consider a $1-$BRG, *i.e.,* a graph consisting of $2G$ vertices $v_0^0, \ldots, v_{G-1}^0$, $v_0^1, \ldots, v_{G-1}^1$. Let p^0 and p^1 denote the two pebbles we are allowed to use. Then, a $1-$BRG can be pebbled as follows: We start by placing p^0 on v_0^0 and use p^1 to pebble the first row in a sequential order (this can be done in G steps). Then, we move p^1 from v_{G-1}^0 to v_0^1. Let denote x the current position of p^1, *i.e.,* $x = 0$ at the beginning. Now, to move p^1 from v_x^1 to v_{x+1}^1, the pebble p^0 has to be moved to $v^0_{\tau(x+1)}$. For each move of p^1, p^0 has to be moved $G/2$ times in average to reach the certain predecessor. Thus, the effort for pebbling a 1−BRG is given by $G + G \cdot G/2 = O\left(G^2\right)$.

Induction Step ($n \rightsquigarrow n+1$):
In this case we consider a $(\lambda+1)$−BRG, which consists of $G(\lambda+2)$ vertices, *i.e.,* $(\lambda+2)$ rows of G vertices each. Let $p^0, \dots, p^{\lambda+1}$ denote the $\lambda+2$ pebbles we are allowed to use. Furthermore, we denote by $p^{\lambda+1}$ the *runner pebble* which is placed on vertex $v_0^{\lambda+2}$. Based on the inductive hypothesis, pebbling $v_{G-1}^{\lambda+1}$ needs at most $O\left(G^{\lambda+1}\right)$ pebble movements using the pebbles $p^0, \dots, p^\lambda$. To pebble $v_{G-1}^{\lambda+2}$, $p^{\lambda+1}$ has to be moved G times in a sequential order from $v_{G-1}^{\lambda+1}$ to $v_{G-1}^{\lambda+2}$. Since for each movement of $p^{\lambda+1}$ at most the whole λ-BRG has to be pebbled, it holds that the maximum number of movements for pebbling a $(\lambda+1)$−BRG is given by

$$G \cdot O\left(G^{\lambda+1}\right) = O\left(G^{\lambda+2}\right).$$

■

Next, we analyze the case of $(\lambda+1) < S \leq G-1$.

Theorem 6.3 (Upper Bound). *Let $\lambda \in \mathbb{N}$ be a fixed value and let $G = 2^g$. Then, the λ-Bit-Reversal Graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ can be pebbled with $(\lambda+1) < S \leq G-1$ pebbles in time*

$$T \;=\; O\left(G^{\lambda+1}/S^\lambda\right).$$

Proof. Note that any reasonable adversary A with $S+1$ pebbles will use at most one pebble at the bottom row (runner pebble). Furthermore, each adversary using $S+1$ pebbles can be easily replaced by an adversary A' using S pebbles in each row $r^0, \dots, r^{\lambda-1}$ and one pebble in Row r^λ (bottom row), requiring less movements than A. It follows that $\mathbf{Adv}(A) < \mathbf{Adv}(A')$ and thus, it is sufficient to upper bound A'. In the following we estimate the number of pebble movements of A'.

It follows that A' is in possession of $\lambda \cdot S + 1$ pebbles. First, we divide the top row (r^0) into S intervals of size G/S, *i.e.,* we place a pebble on every vertex $v^0_{\lfloor iG/S \rfloor}$ for $0 \leq i < S-1$. Then, we use an extra pebble to pebble v^0_{G-1}. This can be achieved in G steps. Thereafter, we move the pebble from v^0_{G-1} to v^1_0 and use the next S pebbles to place a pebble on each vertex $v^1_{\lfloor iG/S \rfloor}$ for $1 \leq i < S-1$. This takes about $(S-1) \cdot G/S \cdot G/2S$ steps. Now, we take again one additional pebble which is moved from $v^1_{G-G/S}$ to v^1_{G-1} and later to v^2_0. The effort for moving this additional pebble is given by $G/S \cdot G/2S = G^2/2S^2$. Thus, assuming S pebbles on Row r^0, the effort for pebbling r^1 is given by $S \cdot G^2/2S^2 = G^2/2S$. Then, placing $S+1$ pebbles (S pebbles for generating the intervals and again one additional pebble which is later moved to v_0^3) on Row r^2 requires $S \cdot G/S \cdot (G/2S)^2 = G^3/4S^2$ pebble movements.

Since A' is in possession of $\lambda \cdot S + 1$ pebbles, we can repeat the procedure for all but the last row. Thus, the effort for pebbling λ rows $r^0, \ldots, r^{\lambda-1}$ generating S intervals in each row, and placing one additional pebble (runner pebble) on v_0^λ, is given by

$$\underbrace{G}_{\text{1st row}} + \underbrace{\prod_{i=1}^{\lambda-2} (G^2/2S)}_{\text{rows } r^1,\ldots,r^{\lambda-1}} = G + \left(G^\lambda/2^{\lambda-1}S^{\lambda-1}\right).$$

Then, moving the runner pebble from v_0^λ to v_{G-1}^λ in a sequential order requires

$$G \cdot (G/2S)^\lambda = \left(G^{\lambda+1}/2^\lambda S^\lambda\right)$$

pebble movements. Thus, the total effort for pebbling a λ-BRG using $\lambda \cdot S + 1$ pebble is given by

$$G + \left(G^\lambda/2^{\lambda-1}S^{\lambda-1}\right) + \left(G^{\lambda+1}/2^\lambda S^\lambda\right) = O\left(G^{\lambda+1}/S^\lambda\right).$$

■

Corollary 6.4 (Pebbling a λ-BRG). *For pebbling a λ-Bit-Reversal Graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ with $G = 2^g$ input vertices and S pebbles with $(\lambda + 1) \leq S \leq G/2^\lambda$, it holds that*

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

6.2. Preimage Security

In this section we discuss the success probability of an adversary for guessing a valid password, given a salt value s, a hash value h, and an algorithm PS (password scrambler). Then, the password-recovery advantage (see Section 2.2, Definition 2.3) for `Catena` is given by

$$\mathbf{Adv}_{\mathrm{REC}}^{\texttt{Catena}}(q) \leq \frac{q}{2^e} + \mathbf{Adv}_{\mathrm{PRE}}^{H}(q), \tag{6.2}$$

where $\mathbf{Adv}_{\mathrm{PRE}}^{H}(q)$ denotes the preimage security of H.

Sketch. We measure the quality of a password population by the min-entropy, defined as the negative base-2 logarithm of the largest probability of any password from that population. Let e denote the min-entropy of passwords generated by `Catena`. Then, an adversary can guess a password by trying out 2^e password candidates. For a maximum of q queries, it holds that the success probability is given by $q/2^e$. Instead of guessing

2^e password candidates, an adversary can also try to find a preimage for a given hash value h. It is easy to see from Algorithm 1 that an adversary thus has to find a preimage for H in Line 5. More detailed, for a given value h with $h \leftarrow H(c \mid\mid x)$, A has to find a valid value for x. We denote by $\mathbf{Adv}^{H}_{\mathrm{PRE}}(q)$ the advantage of an adversary A to find a preimage for H using at most q queries. Equation 6.2 is then given by summing up the individual terms. ■

6.3. Pseudorandomness

For proving the pseudorandomness of `Catena`, we refer to the definition Random-Oracle Security, which was introduced in Section 2.2 (see Definition 2.5). Therefore, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle.

Theorem 6.5 (Random-Oracle Security of `Catena`). *Let q denote the number of queries made by an adversary and s a randomly chosen salt value. Furthermore, let H be modelled as a random oracle. Then, we have*

$$\mathbf{Adv}^{\mathtt{Catena}_H}_{\$}(q) \ \leq \ \frac{q^2 \cdot (\lambda+1)^2}{2^{n-2g}} + \frac{q^2 \cdot (g+1)^2}{2^m}.$$

Proof. Suppose that $a^i = (p^i||s^i||t^i||g^i)$ represents the i-th query, where p^i denotes the password, s^i denotes the salt, t^i the tweak, and g^i 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 λ-BRH (g, x, λ) of the j-th query (Line 4 of Algorithm 1, where F^λ_H is replaced by the bit-reversal hash operation shown in Algorithm 2). Then, $H(g \mid\mid y^j)$ is the output of `Catena`(a^j). In the case that $y^1, \ldots, y^q$ are pairwise distinct, A can not distinguish $H(g \mid\mid \cdot)$ from $\$(\cdot)$, since both functions are modeled as random oracles, returning a value chosen uniformly at random from the set $\{0,1\}^n$.

Therefore, we have to upper bound the probability of the event $y^i = y^j$ with $i \neq j$. Due to the assumption that $A's$ queries are pairwise distinct, there must be at least one collision for H, *i.e.*, $z \neq z'$ with $H(z) = H(z')$. We call such a collision a `badevent`

Then, we define a new game `Catena'`-λ which maintains two initially empty list B_{in} and B_{out} and works as follows. When calling `Catena`, it stores all inputs and outputs of H which are gained from the *inner* calls to the $\lambda-$BRH operation (see Line 4 of Algorithm 1) in the list B_{in}. All inputs and outputs which are gained from the *outer* calls to H (see Lines 1 and 5 of Algorithm 1) are stored in the set B_{out}. Then, it returns a random value R and finally, let an adversary win iff one of the sets B_{in} or B_{out} contains `badevent`. Remark, the advantage of winning the game `Catena'`-λ is higher than the advantage for distinguishing `Catena`$(\cdot)$ from $\$(\cdot)$, since there are `bad events` that most likely lead to a list of unique outputs.

First, we upper bound the probability that B_{in} contains badevent. One call to λ-BRH consists of $(\lambda+1)\cdot 2^c$ hashing operations, where c denotes the current garlic factor. Thus, the probability for a collision is given by

$$\frac{((\lambda+1)\cdot 2^c)^2}{2^n},$$

where n denotes the output size of H. Second, we upper bound the probability that B_{out} contains badevent. Since there are at most $(c+1)$ outer calls to the function H, this can be upper bounded by

$$\frac{(c+1)^2}{2^m},$$

where m denotes the output size after the truncation (see Line 6 of Algorithm 1). Since an adversary is allowed to ask at most q queries, the probability that either B_{in} or B_{out} contains badevent, can be upper bounded by

$$\frac{(q\cdot(\lambda+1)\cdot 2^c)^2}{2^n}+\frac{(q\cdot(c+1))^2}{2^m}<\frac{q^2\cdot(\lambda+1)^2}{2^{n-2g}}+\frac{q^2\cdot(g+1)^2}{2^m},$$

which proves our claim. ■

Chapter 7

Usage

7.1. Proof of Work/Space

Proof of Work. The concept of proofs of work (POW) was introduced by Dwork and Naor [13] in 1992. 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* " [13]. 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. Thus, POW can be used as a counter measure against spam and denial-of-service (DOS) attacks, since an adversary is forced to use a high computational effort, usually rendering these attack to become unprofitable regarding to its time costs.

Proof of Space. As a generalization of POW, Abadi *et al.* [1], and Dwork *et al.* [12] considered memory-hard 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 [14] a compact representation for this table by using a time-memory trade-off for its generation. Recently, Dziembowski *et al.* [16], and Ateniese *et al.* [3] introduced independently from each other two similar notions for proof of space.

It is easy to see that one can use `Catena` to realize a proof-of-work/proof-of-space scenario, which we introduce next, where we denote by P the prover which has to fulfill the challenge, and by V the verifier.

Guessing Secret Input. Let t, g_0, g, m, and e denote fixed and public inputs. At the beginning, after P requests access to some resource administrated by the verified V, V chooses $x \overset{\$}{\leftarrow} \{0,1\}^e$ and $s \overset{\$}{\leftarrow} \{0,1\}^{128}$. Then, V computes $h = \texttt{Catena}(x, t, s, g_0, g, m)$

and sends the challenge (h, s) to the prover P. Next, P responds with a value r, and the verifier grants access to the resource, if $r = x$; otherwise, it rejects the request of the prover P. Note that the salt is part of the challenge since for small values of e, the prover can precompute the (full) codebook, rendering the protocol purposeless.

Since Catena is preimage secure, P can only achieve a success probability greater than 0.5 by computing about 2^{e-1} times $h' = \texttt{Catena}(x', t, s, g_0, g, m)$ for 2^{e-1} values $x' \in \{0,1\}^e$ and comparing if $h = h'$.

7.2. 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 4.1). 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 $h_i = \texttt{Catena}(pwd, t, s_{0,\dots,|s|-2} \mid\mid i, g_0, g, m)$ for $i = 0, \dots, b-1$. The login is successful, if and only if one of the values h_i is valid. This approach is fully transparent for the user, since due to the parallelism, the login time is not effected. Nevertheless, the total memory usage and the computational effort are increased by a factor b. This also holds for an adversary, since it has to try all possible values for the pepper p to rule out a password candidate.

Low-Memory Environments. The application of the server relief technique leads to significantly reduced effort on the side of the server for computing the output of Catena by splitting it into two functions F 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 Catena, the application of server relief does not make sense.

Algorithm 5 Catena-KG

```
Input: $pwd$ {Password}, $t'$ {Tweak}, $s$ {Salt}, $g_0$ {MinGarlic}, $g$ {Garlic}, $\ell$ {Key Size}, $\mathcal{I}$ {Key Identifier}
Output: $k$ {$\ell$-bit key derived from the password}
x ← Catena(pwd, t', s, g_0, g, m)   $x \leftarrow \textbf{Catena}(pwd, t', s, g_0, g, m)$
$k \leftarrow \emptyset$
for $i = 1, \ldots, \lceil \ell/|n| \rceil$ do
    $k \leftarrow k \mid\mid H(0 \mid\mid i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$
end for
return $\textbf{Truncate}(k, \ell)$ {truncate $k$ to the first $\ell$ bits}
```

7.3. Key Derivation Function

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 5). 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

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

Note that for key derivation is makes no sense to give the user control over the output length m of `Catena` itself. 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. Then, the call of `Catena` is followed by an output transform that takes the output x of `Catena`, a one-byte *key identifier* $\mathcal{I}$, and a parameter ℓ for the key length as the input, and generates key material of the desired output size. `Catena-KG` is even able to handle the generation of extra-long keys (longer than the output size of H), by applying H in Counter Mode [15]. 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. E.g., when Alice and Bob set up a secure connection, they may need four keys: an encryption and a message authentication key for messages from Alice to Bob, and another two keys for the opposite direction. One could argue that $\mathcal{I}$ should also become a part of the associated data. But actually, this would be a bad move, since setting up the connection would require legitimate users to run `Catena` several times. But, the adversary can search for the password for one key, and just derive the other keys, once that password has been found. For a given budget for key derivation, one should rather employ one single call to `Catena` with larger security parameters and then run the output transform for each key.

In contrast to the password hashing scenario, where a user want to log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [39], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, we recommend to use $g = 20$ when `Catena` is used for key derivation.

Security Analysis. It is easy to see that `Catena-KG` inherits its $\lambda-$memory hardness from `Catena` (see Chapter 6, Corollary 6.4), since it invokes `Catena` (Line 1 of Algorithm 5). Next, we show that `Catena-KG` a good pseudorandom function (PRF) in the random oracle model.

Theorem 7.1 (Random-Oracle Security of `Catena-KG`). *In the random oracle model we have*

$$\mathbf{Adv}_{\$}^{\mathit{Catena\text{-}KG}_H} = \left|\Pr[A^{\mathit{Catena\text{-}KG}_H} \Rightarrow 1] - \Pr[A^{\$} \Rightarrow 1]\right| \leq \frac{q^2}{2^{n-g-1}}.$$

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.

Note that all inputs to H from Algorithm 5 (see Line 4) contain zero as their first byte. Hence, they never occur as inputs in any invocations in $\mathtt{Catena}_H(pwd, t', s, g)$, because the first byte of the input therein is always a value in the interval $[1, g]$. Suppose x^i denotes the output of `Catena` of the i-th query. In the case $x^i \neq x^j$ for all values with $1 \leq i < j \leq q$, the output k is always a random value, since H is always invoked with a fresh input (see Line 4, Algorithm 5). The only chance for an adversary to distinguish $\mathtt{Catena\text{-}KG}_H(\cdot)$ from the random function $\$(\cdot)$ is a collision in $\mathtt{Catena}_H$. The probability for this event can be upper bounded by similar arguments as in the proof of Theorem 6.5. ■

Chapter 8

Acknowledgement

Thanks to Bill Cox, Eik List, Colin Percival, Stephen Touset, Steve Thomas, Alexander Peslyak, as well as the reviewers of the FSE'14 for their helpful hints and comments, as well as fruitful discussions. Finally, we want to thank Ben Harris who discovered two minor miss-matches between the specification and reference implementation of CATENA.

Chapter 9

Legal Disclaimer

To the best of our knowledge, neither Catena, the BLAKE2b Function Family, nor the structure of the bit-reversal 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] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

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

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

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

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

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

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

[15] Morris Dworkin. *Special Publication 800-38A: Recommendation for block cipher modes of operation.* National Institute of Standards, U.S. Department of Commerce, December 2001.

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

[17] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In *CRYPTO*, pages 335–353, 2011.

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

[30] 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.

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

[32] 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.

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

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

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

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

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

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

[39] 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 a λ-BRG, 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 λ-BRG to be sorted in their topological order, each vertex v_i depends at least on the vertex v_{i-1} for $1 \leq i \leq (\lambda + 1) \cdot 2^g - 1$.

Appendix B

Lower Bound for Pebbling a 3-BRG

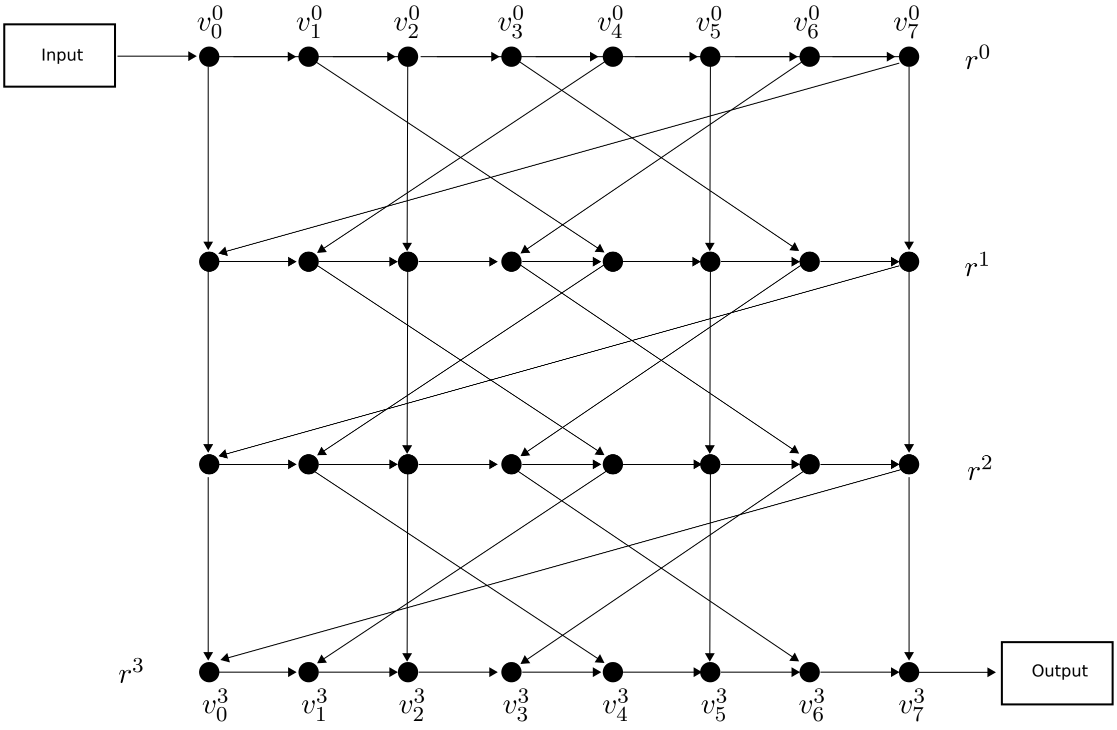

Figure B.1.: A λ-Bit-Reversal Graph with $\lambda = 3$ (3-BRG) and $g = 3$.

Lemma B.1 (Lower Bound for $\lambda = 3$). *If $4 \leq S \leq G/8$, then, pebbling the $3-BRG$ $\Pi_g^3(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input vertices with $S \leq 2^\sigma + 1$ pebbles takes time*

$$T \geq \frac{G^4}{2^{18} S^3}.$$

Proof. Note that if an adversary is not in possession of $2^\sigma + 1$ pebbles, we give it as many pebbles as necessary for free until it has $2^\sigma + 1$ pebbles. Let the Row r^3 be divided into 2^{g-s} intervals of length 2^s, where $2^s \geq 2^{\sigma+3}$ and where the j-th interval I_j^3 consists of the vertices $v_{j2^s}^3, \ldots, v_{(j+1)2^s-1}^3$ with $0 \leq j < 2^{g-s}$. Let z_j be the first time (*i.e.,* the number of the first move) that a pebble is placed on the last vertex of the interval I_j^3 (*i.e.,* $v_{(j+1)2^s-1}^3$). Let $z_{j-1} = 0$. Then, $z_j > z_{j-1}$ for $0 \leq j < 2^{g-s}$. In order to find a lower bound on $z_j - z_{j-1}$, we observe that at time z_{j-1} the interval I_j^3 is pebble-free and thus, all 2^s vertices in I_j^3 have to be pebbled between z_{j-1} and z_j. By definition of the bit-reversal permutation, the immediate predecessors on the input path of the vertices in I_j^3 divide the Row r^2 naturally into 2^s intervals I_i^2 of length 2^{g-s}, where each vertex of I_j^3 defines the high end of an interval I_i^2 with $0 \leq i < 2^s$.

Since we have at most $S - 1$ pebbles left to pebble r^2, at least $2^s - (S - 1) > 7S$ intervals are pebble-free at z_{j-1}. In the following we denote a vertex as *marked* if it is part of an interval which already contains a pebble, and *unmarked* otherwise. Thus, at most $1/8$ of the vertices in r^2 are *marked*, whereas $7/8$ of the vertices are *unmarked.*

Then we divide r^2 again in 2^{g-s} intervals J_j^2 of length 2^s to re-establish the same configuration as in Row r^3. We denote an interval J_j^2 as *bad* if more than half of its vertices ($> 2^{s-1}$) are marked. Since r^2 has at most $G/8$ marked vertices, at most $1/4$ of the intervals in r^2 are bad, *i.e.,* at least $3/4$ of the intervals are *good.* Now, each good interval in r^2 divides the Row r^1 naturally into 2^s intervals I_i^1 of length 2^{g-s} where each vertex of J_j^2 defines the high end of an interval I_i^1. We denote an interval I_i^1 in r^1 as bad, if the vertex of J_j^2 that defines its high end is *marked.* Thus, we have at least $2^s - 4S$ good intervals in r^1. Furthermore, at most $S - 1$ pebbles can be placed within these intervals and thus, we have at least $2^s - 5S \geq 3S$ good intervals. It follows that at most $5/8$ of the vertices of r^1 are marked.

Then again, we divide r^1 in 2^{g-s} intervals J_j^1 of length 2^s to re-establish the same configuration as in Row r^2. We denote an interval J_j^1 as bad if more than $3/4$ of its vertices are marked. Since r^1 has at most $5/8$ marked vertices, at most $5/6$ of the intervals in r^1 are bad. It follows, that at least $1/6$ of the intervals in r^1 can be considered as good. For each of those intervals, r^0 is divided into 2^s intervals J_j^0 of length 2^{g-s}. Since at most $3/4$ of these intervals are generated by a marked vertex in an interval J_j^1, we have $2^s - 6S \geq 2S$ good intervals in the top row. Furthermore, since at most $S - 1$ of these intervals can contain a pebble, there remain S good intervals in the Row r^0. Thus, the cost for pebbling a good interval of r^1 is given by $S \cdot 2^{g-s} \geq G/8$. Further, the effort for pebbling one interval of r^2 is given by $1/6 \cdot 2^{g-s} \cdot G/8 \geq G^2/384S$. Next, the effort for pebbling on interval of r^3 is given by $1/8 \cdot 2^{g-s} \cdot G^2/768S \geq G^3/24576S^2$. Since this has to be done for each interval of r^3, the total costs for pebbling a 3−BRG are at least

$$2^{g-s} \cdot G^3/24576S^2 \geq \frac{G^4}{2^{18}S^3}.$$

■

Appendix C

Test Vectors

C.1. Test Vectors for Catena-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`
	`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`
		(64 octets)

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`
	`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`
		(64 octets)

C.2. Test Vectors for Catena-**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	

Appendix D

Illustration of 4-BRG

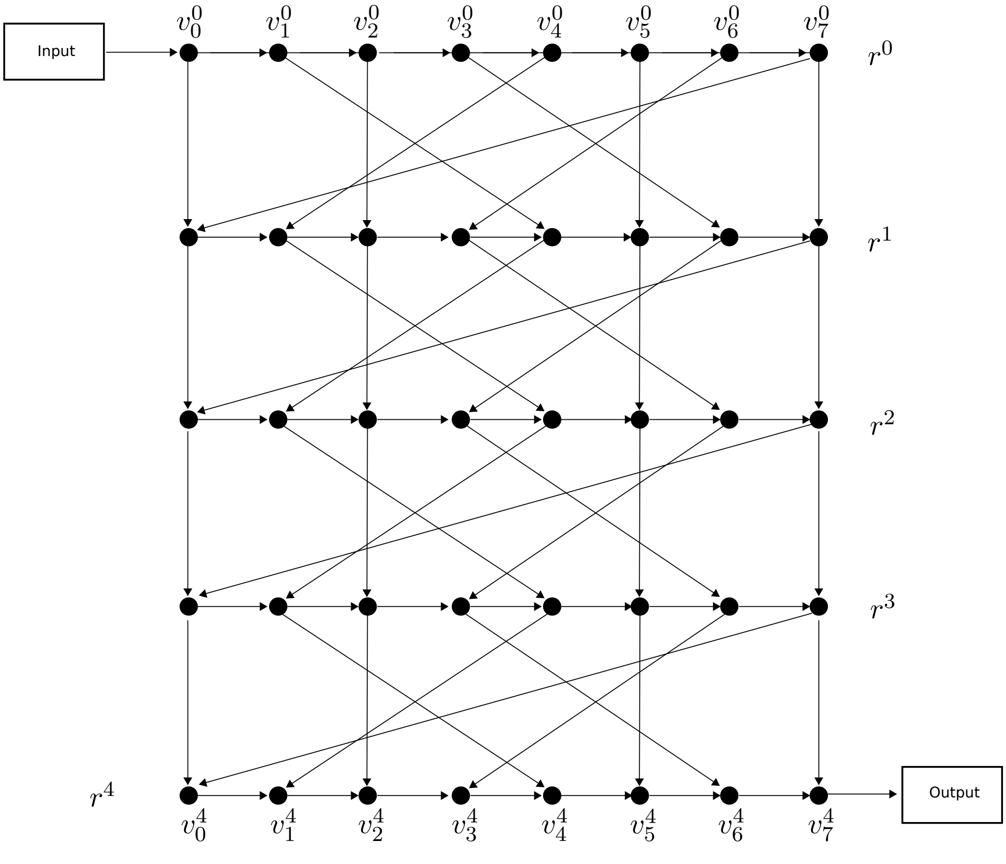

Figure D.1.: A λ-Bit-Reversal Graph with $\lambda = 4$ (4-BRG) and $g = 3$.