Catena: A Memory-Consuming Password-Scrambling Framework

Christian Forler, Stefan Lucks, and Jakob Wenzel

Bauhaus-Universität Weimar, Germany
{Christian.Forler, Stefan.Lucks, Jakob.Wenzel}@uni-weimar.de

Abstract. It is a common wisdom that servers should store the one-way hash of their clients' passwords, rather than storing the password in the clear. In this paper we introduce a set of functional properties a key-derivation function (password scrambler) should have. Unfortunately, none of the existing algorithms satisfies our requirements and therefore, we introduce a novel and provably secure password scrambling framework (PSF) called Catena. Furthermore, we introduce two instantiations of Catena based on a memory-consuming one-way functions. Thus, Catena excellently thwarts massively parallel attacks on cheap memory-constrained hardware, such as recent graphical processing units (GPUs). Additionally, we show that Catena is also a good key-derivation function, since – in the random oracle model – it is indistinguishable from a random function. Furthermore, the memory-access pattern of both instantiations is password-independent and therefore, Catena provides resistance against cache-timing attacks. Moreover, Catena is the first PSF which naturally supports (1) *client-independent updates* (the server can increase the security parameters and update the password hash without user interaction or knowing the password), (2) an optional *server relief* protocol (saving the server's resources at the cost of the client), and (3) a variant Catena-KG for secure *key derivation* (to securely generate many cryptographic keys of arbitrary lengths such that compromising some keys does not help to break others). We denote a password scrambler as a PSF with a certain instantiation.

Keywords: password hashing, memory-hard, cache-timing attack

Table of Contents

1 Introduction

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 "offline" password guessing, see [6] for an early example. Otherwise, the best protection are cryptographic password scramblers performing "key stretching". The basic idea of such schemes is to use an intentionally slow one-way function for hashing a password. Therefore, the password processing takes some time for both kinds of users legitimate ones and adversaries. A good password scrambler PS has to satisfy at least the following basic conditions:

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

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

(2) Computing $PS(pwd)$ should be "as slow as possible", without contradicting Condition (1).
(3) Given $y = PS(pwd)$, there must be no significantly faster way to test q password candidates $x_1, x_2, \ldots, x_q$ for $PS(x_i) = y$ than by actually computing $PS(x_i)$ for each x_i.

Traditionally, most password scramblers realize "as slow as possible" by iterating a cryptographic primitive (a block cipher or a hash function) many times, e.g., `md5crypt` [26] or `sha512crypt` [14]. However, an adversary who happens to have b computing units ("cores") can easily try out b different passwords in parallel. With recent technological trends, such as the availability of graphical processing units (GPUs) with hundreds of cores [39], the question of how to slow down such adversaries becomes a pressing one. Memory is expensive; so, a typical GPU or other cheap and massively-parallel hardware with lots of cores can only have a limited amount of memory for each single core. More importantly, each core will have only a very limited amount of fast ("cache") memory. So the way to prevent b-core adversaries from gaining some close-to-b-times speed-up is by making a password scrambler PS not only intentionally slow on standard sequential computers, but also intentionally memory-consuming. Thus, any adversary using b cores in parallel with less than about b times the memory of a sequential implementation must experience a strong slow-down. The first password scrambler that took this condition into account was `scrypt` [44]. To the best of our knowledge, it is also the only one – up to now.

However, a memory-consuming password scrambler may suffer from new problems: (1) If the memory-access pattern depends on the password, and the adversary can observe that pattern, this may open the way to another kind of shortcut attack. For example, a spy process, running on the same machine as the password scrambler (without access to the internal memory of the password scrambler) may gather information about the password scrambler's memory-access pattern by measuring cache-timings. This information can be used to greatly speed-up massively parallel attacks with low memory for each core. (2) Assume that the memory which is directly derived from the secret input (password) is rarely or never overwritten. Then, a malicious garbage collector can built an efficient password candidate filter by using the knowledge gained from this memory.

To analyze memory-consuming algorithms regarding to its memory-hardness, one can apply the so called "pebble game" approach, which dates back to the early days of Theoretical Computer Science [11,24,27,32,43].

Background. As observed by Wilkes in the late 1960s [58], storing plain authentication passwords is insecure. About 10 years later, the UNIX system integrated some of Wilkes ideas [36] by deploying the DES-based one-way encryption function `crypt`, to "encrypt" a given password. Actually, there is no efficient way to recover the original password from the result of the "encryption", i.e., `crypt` is a one-way hash function, or a *password scrambler*, as we call it.

Since the introduction of `crypt`, storing the hash of a password and avoiding to store the plain password itself has become the minimum standard for secure password-based user authentication. But, even as late as 2012, major players like Yahoo and CSDN (China Software Developer Network) seem to store plain user-passwords [33].

Two important innovations from `crypt` were *key stretching* and *salts*. Key stretching is the answer to the typically low entropy of user-chosen passwords: The password scrambler is intentionally slow, but not too slow for the regular operation, e.g., a password-based log-in. This makes exhaustively searching through all likely passwords more expensive.

A salt refers to an additional random input value for the password scrambler, stored together with the password hash. It enables a password scrambler to derive lots of different password hashes from a single password like an initialization vector enables an encryption scheme to derive lots of different ciphertexts from a single plaintext. Since the salt must be chosen uniformly at random, it is most likely that different users have different salts. Thus, it hinders against attacks where password hashes from many different users are known to the adversary, e.g., against the usage of rainbow tables [40].

Note that there are further ways to thwart adversaries, i.e., ways to perform key stretching. One is to keep p bits of the salt secret, turning them into *pepper* [34]. Both adversaries and legitimate users have to try out all 2^p values the pepper can have (or 2^{p-1} on the average). Note that a careless implementation of this approach could leak a few bits of the pepper via timing information, when trying out all possible values in a specific order. Thus, a better approach would be to start at a random value and wrap around at 2^p. Kelsey et al. [28] analyzed another key stretching approach where a cryptographic operation is iterated n times, where n is secret. Boyen proposed in [9] a user-defined implicit choice for n by iterating until the user presses a “halt” button.

Contribution. In this paper we introduce the password scrambling framework (PSF) Catena (Latin for “chain”, due the sequential structure of its core function). Catena provides innovative features like *client-independent update*, i.e., if the security parameter of the password scrambler is increased, the new hash value can be computed without requiring user interaction. For Catena, this can for example be done by increasing the security parameter *garlic* or by turning some bits of the salt into pepper. The notion of *garlic* reflects the property that incrementing this parameter by '1' doubles the memory usage and at least doubles the computational time. Further, Catena provides built-in support for *server relief*, which allows to shift the main effort for computing a password hash to the client. Note that the idea of server relief is not new (see [13] and [38]), but Catena is the first PSF which supports this technique in the first place.

We present two instantiations (1) Catena-BRG– based on a structure called (G, λ)-bit-reversal graph and (2) Catena-DBG– based on a structure called (G, λ)-double-butterfly graph. Further, we introduce the notion of λ-Memory-Hardness and discuss why Catena-DBG has the properties of a $\lambda-$memory-hard function, whereas Catena-BRG only satisfies the properties of a memory-hard function. For the analysis of both instantiations we refer to the work of Lengauer and Tarjan [31], and restate as well as discuss their results. For both instantiations it holds that Catena thwarts back massively parallelized attacks using GPUs and similar hardware. Additionally, it has been designed to be resistant against (1) cache-timing attacks and (2) garbage-collector attacks (GCA) – a notion which we introduce in this work. Resistance against GCA is naturally provided by Catena-DBG, whereas Catena-BRG only satisfies this kind of resistance for $\lambda \geq 2$. The need to address these issues has been inspired by revealing vulnerability of `scrypt` against cache-timing as well as garbage-collector attacks. To substantiate these vulnerability of `scrypt`, we present a rather *academical* cache-timing attack and a GCA. To the best of our knowledge, such attacks for `scrypt` have not been known before.

Moreover, Catena fits very well in a proof of work scenario, and we show that it is well suited for usage as a password-based key-derivation function – called Catena-KG. Note that we often refer Catena to as a password scrambler. In this situations it is meant that the considered property holds for both presented instantiations.

Outline. Section 2 briefly overviews common password scramblers and their properties. In Section 3 we present desired properties of modern password-scrambling algorithms. In Section 4 we introduce some notations and preliminaries which help to understand the paper. Section 5 provides the specification and novel properties of Catena. In Section 6 we discuss the security properties of the Catena framework. Section 7 introduces two instantiations of Catena. A comprehensive security analysis of these instantiations is given in Section 8. The choice of the parameters and the inner hash function for an implementation of Catena are presented in Section 9. The application of Catena for the scenario of proof of work, a discussion about Catena in different environment, and the application of Catena as a key-derivation function are given in Section 10, Section 11, and Section 12. Section 13 concludes the paper.

2 Frequently used Password Scramblers

Table 1 provides an overview of password scramblers that are or have been in frequent use, compared with Catena. Note that this comparison holds for both presented instantiations. It indicates the amount of memory used and the cost factor to generate a password hash, as well as if the certain algorithm supports *server relief* and *client-independent updates*. Furthermore, it points out issues from which the considered password scrambler may suffer.

Hash Function Based Password Scramblers. Not long ago, `md5crypt` [26] was used in nearly all Free-BSD and Linux-based systems to scramble user passwords. It is based on the well-known MD5 [47] hash function with a fixed number of 1,000 iterations. Due to the fact that CPUs and GPUs become more and more powerful, `md5crypt` can now be computed too fast, e.g., over 5 million times per second on a AMD HD 6990 graphic card [53]. Additionally, its own author does not consider `md5crypt` secure anymore [26]. Common Linux distributions nowadays employ `sha512crypt` [14], e.g., Debian, Ubuntu, or Arch Linux. It provides similar features as `md5crypt`, but uses SHA-512 [41] instead of MD5. Furthermore, the number of iterations can be chosen by the user. `NTLMv1` [21] is a fast password scrambler, which is deployed to generate hash values for several versions of Microsoft Windows passwords. It is very efficient to compute: one can check over nine billion password candidates per second on a single COTS graphic card [53]. For this and other reasons, we recommend that `NTLMv1` should not be used anymore. The "Password-Based Key Derivation Function 2" (`PBKDF2`) has been specified by the National Institute of Standards and Technology (NIST) [57]. It is widely used either as a KDF (e.g., in WPA, WPA2, OpenOffice, or WinZip) or as a password scrambler (e.g., in Mac OS X, LastPass). The security of `PBKDF2` is based on c iterations of HMAC-SHA-1 [30], where c is a user-chosen value, which is given by default with $c = 1000$.

`bcrypt`. The `bcrypt` [45] algorithm is built upon the Blowfish block cipher [51]. Internally, Blowfish uses a slow key scheduler to generate an internal state of 4,168 bytes for the key-dependent S-boxes ($4 \times 1,024$ bytes) and the round keys (72 bytes). Thus, while `bcrypt` was not designed with the intention to thwart parallelized adversaries by exhaustive memory usage, the state is sufficiently large to slow down `bcrypt` significantly on current GPUs, e.g., it can only computed about 4,000 times per second on an AMD HD 7970 graphic card [53]. However, the state size is fixed – so if future GPUs have a larger cache, it may actually run *much faster*. There is no tunable parameter to increase the memory requirement of this password scrambler. For key stretching, `bcrypt` invokes the Blowfish key scheduler 2^c times, e.g., OpenBSD uses $c = 6$ for users and $c = 8$ for the superuser.

Algorithm	Cost Factor (default)	Memory	Server Relief	Client-Indep. Updates	Issues
`crypt` [36]	25	small	-	-	"too fast"
`md5crypt` [26]	1,000	small	-	-	"too fast"
`sha512crypt` [14]	1,000–999,999 (5,000)	small	-	-	(small memory)
`NTLMv1` [21]	1	small	-	-	"too fast"
`PBKDF2` [57]	1–∞ (1,000)	small	-	-	(small memory)
`bcrypt` [45]	2^4–2^{99} (2^6, 2^8)	4,168 bytes	-	-	(constant memory)
`scrypt` [44]	1–∞ (2^{14}, 2^{20})	flexible, big	-	-	cache-timing attacks
`Catena-BRG` (this paper)	2^1–∞ (2^{17}, 2^{20})	flexible, big	✓	✓	performance
`Catena-DBG` (this paper)	2^1–∞ (2^{14}, 2^{16})	flexible, big	✓	✓	performance

Table 1. Comparison of state-of-the-art password scramblers and Catena. Note that all of the mentioned algorithms support salt values. The term $2^1 - \infty$ denotes that the cost factor can be chosen arbitrarily large.

scrypt. Occupying a lot of memory hinders attacks using special-purpose hardware (storage is expensive) and GPUs. We are aware of one single password scrambler that has been designed to occupy a lot of memory: `scrypt` [44]. (There was HEKS [46], but it has been broken by the author of `scrypt`.) As its core, `scrypt` uses the sequentially memory-hard function `ROMix` (see [44] for a formal definition of sequential memory-hardness), which can take G units of memory and performs $2G$ operations. With only G/K units of memory, the number of operations goes up to $2G \cdot K$. Unfortunately, `ROMix` is vulnerable against cache-timing attacks, due to its password-dependent memory-access pattern. In [44], Percival recommends $G = 2^{14}$ and $G = 2^{20}$ for password hashing and key derivation, respectively.

3 Properties of Modern Password Scramblers

In this section we introduce a listing of desired properties a modern password scrambler should have.

Memory-Hardness. To describe memory requirements, we adopt and slightly change the notion from [44]. The intuition is that for any parallelized attack, using b cores, the required memory per core is decreased by a factor of $1/b$, and vice versa.

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

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

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

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

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

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

Thus, we have

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

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

λ-Memory-Hard vs. Sequential Memory-Hard. In [44], Percival introduced the notion of sequential memory-hardness (SMH), which is satisfied by his introduced password scrambler `scrypt`. Bases on this notion, an algorithm is sequential memory-hard, if an adversary has no computational advantage in using multiple CPUs, i.e., using b cores requires b times the effort used for one core. It is easy to see that, in the parallel computation setting, SMH is a stronger notion than that of λ-memory-hardness (λMH). Thus, SMH is a desirable goal when designing a memory-consuming password scrambler. In this section we discuss why our presented password scrambler Catena satisfies at most λMH instead of SMH, without referring to details of Catena, which are presented in Section 5 and 7.

Note that a further goal of our design was to provide resistance against cache-timing attacks, i.e., both instantiations of Catena should satisfy a password-independent memory-access pattern. This goal can be achieved by providing a control flow which is independent of its input. It follows that Catena can be seen as a straight-line program, which on the other hand can be represented by a directed acyclic graph (DAG, see Definition 3).

Definition 3 (Directed Acyclic Graph). *Let $\Pi(\mathcal{V}, \mathcal{E})$ be a graph consisting of a set of vertices $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$ and a set of edges $\mathcal{E} = (e_0, e_1, \ldots, e_{\ell-1})$, where $\mathcal{E} = \emptyset$ is a valid variant. $\Pi(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph, if every edge in $\mathcal{E}$ consists of a starting vertex v_i and an ending vertex v_j, with $i \neq j$. A path through $\Pi(\mathcal{V}, \mathcal{E})$ beginning at vertex v_i must never reach v_i again (else, there would be a cycle). If there exists a path from a vertex v_i to a vertex v_j in the graph with $i \neq j$, we will write $v_i \leq v_j$.*

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

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

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

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

Definition 4 (Password-Recovery Advantage). *Let s denote a randomly chosen salt value and pwd a password randomly chosen from a source $\mathcal{Q}$ with e bits of min-entropy. Let PS denote a password-scrambling algorithm. Then, given a hash value h with $PS(s, pwd) = h$, we define the password-recovery advantage of an adversary A as*

$$\mathbf{Adv}_{PS}^{REC}(A) = \Pr_{s,h}\left[A^{PS,s,h} \Rightarrow x : PS(s,x) \overset{?}{=} h\right].$$

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

In Section 6 we show that for Catena it holds 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 [35], 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 [37] and 73% of all Twitter accounts do not have at least one tweet per month [48]. 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 [38] and more recent in [13]. We realized this idea by splitting the password-scrambling process into two parts: (1) a slow (and possibly memory-demanding) one-way function F and (2) an efficient one-way function H. By default, the server computes the password hash $h = H(F(pwd, s))$ from a password pwd and a salt s. Alternatively, the server sends s to the client who responds $y = F(pwd, s)$. Finally, the server just computes $h = H(y)$. While it is probably easy to write a generic *server relief* protocol using any password scrambler, none of the existing password scramblers has been designed to naturally support this property. Note that this property is optional, e.g., for the proof of work scenario the server relief idea makes no sense, since the whole effort should be already on the side of the client.

Resistance against Cache-Timing Attacks. Consider the implementation of a password scrambler, where data is read from or written to a password-dependent address $a = f(pwd)$. If, for another password pwd', we would get $f(pwd') \neq a$ *and* the adversary could observe whether we access the data at address a or not, then it could use this information to filter out certain passwords. Under certain circumstances, timing information related to a given machine's cache behavior may enable the adversary to observe which addresses have been accessed. Thus, we formally introduce resistance against cache-timing attacks, which we recommend to be fulfilled by upcoming password scramblers.

Definition 5 (Resistance against Cache-Timing Attacks). *Let $\mathcal{F} : \{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^n$ denote a function processing arbitrary large data together with a secret value $K \in \{0,1\}^k$, and outputs a fixed length value of size n bits. 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 6 (Random-Oracle Security). *Let $PS : \{0,1\}^* \rightarrow \{0,1\}^n$ be 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.

4 Preliminaries

In this section we first introduce the notions used throughout this paper (see Table 2). Next, to increase the understanding of the proofs given in [31], we describe a technique for analyzing directed acyclic graphs, called *pebble game*. Then, we introduce the definition of a garbage-collector attack.

Identifier	**Description**
pwd	password
λ	depth of F_λ
s	salt (public random value)
p	pepper (secret bits of the salt)
t	tweak
d	domain (application specifier) of Catena
g_0, g	minimum garlic; current garlic with $G = 2^g$
PS/PSF	Password Scrambler/Password-Scrambling Framework
\$	function returning a fixed-size random value
h, y	password hash (or intermediate hash)
$S(g)$	memory (space) consumption; depends on the garlic
$T(g)$	time consumption; depends on the garlic
$\Pi(\mathcal{V}, \mathcal{E})$	graph based on $\mathcal{V}$ vertices and $\mathcal{E}$ edges
r^i	i-th row
$v^k_{i,j}$	j-th vertex of the i-th row of the k-th DBG
b	number of cores
$A^{O_0,\ldots,O_\ell}$	adversary A with access to the oracles $O_0, \ldots, O_\ell$
q	number of total queries A is allowed to ask
τ	Bit-Reversal Permutation
σ	function determining the index of the diagonal edges (DBG)
AD	associated data
K	secret key
$\lvert X\rvert$	size of X in bits or size of a set X

Table 2. Notions used throughout this paper.

Pebble Game. Hellman presented in [23] a possibility to trade memory/space S against time T in attacking cryptographic algorithms, i.e., he has introduced the idea of a time-memory trade-off (TMT) 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 TMT. In 1970, Hewitt and Paterson introduced a method for analyzing TMTs on directed acyclic graphs (DAG, see Definition 3, Section 3) [42], called *pebble game*. It has been occasionally used in cryptographic context, see, e.g. [20] for a recent example.

The pebble game model is restricted to DAGs with bounded in-degree and can be seen as a single-player game. Let $\Pi(\mathcal{V}, \mathcal{E})$ be a DAG and let $N = |\mathcal{V}|$ be the number of nodes within $\Pi(\mathcal{V}, \mathcal{E})$. For our scheme, we restrict the in-degree to 2. In the setup phase of the game, the player gets S pebbles (tokens) with $S \leq N$. A pebble can be placed on a node (*mark*) or be removed from a node (*unmark*) under certain requirements (where $v \in \mathcal{V}$ denotes a node within the set $\mathcal{V}$ of all nodes of $\Pi(\mathcal{V}, \mathcal{E})$):

1. A pebble may be removed from a vertex v at any time.
2. A pebble can be placed on a node v if all predecessors of the node v are marked.

3. If all immediate predecessors of an unpebbled node v are marked, a pebble may be moved from a predecessor of v to v.

A *move* is the application of either the second or the third action stated above. The goal of the game is to mark (to *pebble*) all nodes of a graph $\Pi(\mathcal{V}, \mathcal{E})$ at least once. The time-memory trade-off (TMT) is then defined by counting the minimum number of moves (T) and the maximum simultaneously placed pebbles on the graph (S), which are necessary to reach the goal. Based on the following two trivial observations (see [31]), we can define a lower and an upper bound for the time-memory trade-off $S \cdot T$. On the one hand, any graph of size N can be pebbled with N pebbles in time N (in topological order). On the other hand, if a graph $\Pi(\mathcal{V}, \mathcal{E})$ of size N can be pebbled with S pebbles at all, it can be pebbled with S pebbles in time

$$T \leq \sum_{0 \leq i \leq S} \binom{N}{i} \leq 2^N.$$

Therefore, the interest of T is bounded to the range $N \leq T(S) \leq 2^N$. As the TMT has always to be satisfied, $T(S)$ has to increase if S decreases, and vice versa. In general, a pebble game is a common model to derive and analyze TMTs as shown in [49,50,52,54,55].

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

Definition 7 (Garbage-Collector Attack). *Let PS be a memory-demanding password scrambler depending on a memory-cost parameter g with $G = 2^g$. Furthermore, let $v_0, \ldots, v_{G-1}$ denote the internal state of PS after its termination. Let $\mathcal{A}$ be a computationally unbounded but always halting adversary conducting a garbage-collector attack. We say that $\mathcal{A}$ is successful if the knowledge about $v_0, \ldots, v_{G-1}$ reduces the runtime of $\mathcal{A}$ for testing a password candidate x from $\mathcal{O}(PS(x))$ to $\mathcal{O}(f(x))$ with $\mathcal{O}(f(x)) < \mathcal{O}(PS(x))$.*

5 Catena – A Memory-Hard Password-Scrambling Framework

In this section we introduce our password-scrambling framework Catena. First, we specify Catena and explain its properties regarding to password hashing, i.e., client-independent update and server relief. Thereupon, we present two instantiations of Catena, called Catena-BRG and Catena-DBG. Both instances are designed to provide a high resilience against cache-timing attacks, and the latter naturally defends against garbage-collector attacks, whereas the former provides this kind of resistance only for $\lambda \geq 2$.

Specification. A formal definition is shown in Algorithm 1, where the function F_λ (see Line 3) is a placeholder for a certain instantiation. The password-dependent input of H is appended to a prefix c, which denotes the iteration counter (garlic factor). For the initial deployment of Catena, we recommend to set the initial garlic value g_0 to g to achieve the best ratio between running time and memory usage.

A secure password scrambler must satisfy preimage security. Catena inherits the preimage security from the underlying hash function H. Next, we discuss the tweak and two further novel features of Catena.

Algorithm 1 Catena

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

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

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

The first byte d specifies the application (domain) of Catena. More detailed, we use $d = 0$ for password hashing and $d = 1$ for key derivation. The other values of d are reserved for future applications. The byte value λ defines together with the value g (see above) the security parameters for Catena. The 32-bit value $|s|$ denotes the total length of the salt in bits. Finally, 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 does not matter as long as they are fixed for a certain application.

The tweak is processed together with the secret password and the salt (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.

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

$$h' = H(g' \mid\mid F_\lambda(g', h)).$$

It is easy to see that the equation $h' = \text{Catena}_\lambda(pwd, t, s, g', F_\lambda)$ 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 4). The current output of Catena_λ is then transmitted to the server. Next, the server computes the password hash by applying the hash function H. 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.

6 Security Analysis of the Catena Framework

We denote a password scrambler to be secure if it provides at least 1-memory-hardness and preimage security. Furthermore, it should be resistant against cache-timing attacks. Catena-DBG (see Section 7) inherits its λ-memory-hardness (see Definition 2) from F_λ, whereas Catena-BRG provides only 1-memory-hardness, i.e., memory-hardness (see Definition 1).

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

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

Theorem 1 (Catena$_\lambda$ is Password-Recovery Resistant). *Let m denote the min-entropy of a password source $\mathcal{Q}$. Then, it holds that*

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

Proof. Note that an adversary $\mathcal{A}$ can always guess a (weak) password by trying out about 2^m password candidates. For a maximum of q queries, it holds that the success probability is given by $q/2^m$. Instead of guessing 2^m password candidates, an adversary can also try to find a preimage for a given hash value h. It is easy to see from Algorithm 1 that an adversary thus has to find a preimage for H in Line 4. More detailed, for a given value h with $h \leftarrow H(g,x)$, $\mathcal{A}$ has to find a valid value for x. The success probability for this can be upper bounded by $\mathbf{Adv}^{\mathrm{pre}}_{H}(q,t)$. Our claim follows by adding up the individual terms. □

Pseudorandomness. In the following we analyze the advantage of an adversary $\mathcal{A}$ in distinguishing the output of Catena$_\lambda$ from a random bitstring of the same length as the output of Catena$_\lambda$. Therefore, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle.

Theorem 2 (PRF Security of Catena$_\lambda$). *Let q denote the number of queries made by an adversary and s a randomly chosen salt value. Furthermore, let H be modelled as a random oracle and $g \geq g_0 \geq 1$. Then, it holds that*

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

Proof. Let $a^i = (pwd^i \;||\; u^i \;||\; s^i \;||\; g)$ represent the i-th query, where pwd^i denotes the password, u^i denotes the tweak, s^i the salt, and g the garlic. For this proof, we impose the reasonable condition that all queries of an adversary are distinct, i.e., $a^i \neq a^j$ for $i \neq j$.

Suppose that y^j denotes the output of $F_\lambda(g, a^j)$ of the j-th query (cf. Algorithm 1, Line 3). Then, $H(g \;||\; y^j)$ is the output of Catena$_\lambda(a^j)$. In the case that $y^1, \ldots, y^q$ are pairwise distinct, an adversary $\mathcal{A}$ cannot distinguish $H(g \;||\; \cdot)$ from a random function $\$(\cdot)$ since in the random-oracle model, both functions return a value chosen uniformly at random from $\{0,1\}^n$.

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

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

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

7 Instantiations

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

7.1 Catena-BRG

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

Definition 8 (Bit-Reversal Permutation τ). *Fix a number $k \in \mathbb{G}$ and represent $i \in \mathbb{Z}_{2^k}$ as a binary k-bit number, $(i_0, i_1, \ldots, i_{k-1})$. The bit-reversal permutation $\tau : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ is defined by*

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

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

Definition 9 ((G,λ)-Bit-Reversal Graph). *Fix a natural number g, let $\mathcal{V}$ denote the set of vertices, and $\mathcal{E}$ the set of edges within this graph. Then, a (G,λ)-bit-reversal graph $\Pi_g^\lambda(\mathcal{V},\mathcal{E})$ 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 $v_{i-1}^j \to v_i^j$ for $i \in \{1, \ldots, 2^g-1\}$ and $j \in \{0, 1, \ldots, \lambda\}$.*
- *$\lambda \cdot 2^g$ edges $v_i^j \to v_{\tau(i)}^{j+1}$ for $i \in \{0, \ldots, 2^g-1\}$ and $j \in \{0, 1, \ldots, \lambda-1\}$.*
- *λ additional edges $v_{2^g-1}^j \to v_0^{j+1}$ where $j \in \{0, \ldots, \lambda-1\}$.*

For example, Figure 1 illustrates an (8,1)-BRG. Note that this graph is almost identical – except for one additional edge $e = (v_7^0, v_0^1)$ – to the bit-reversal graph presented by Lengauer and Tarjan in [31].

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

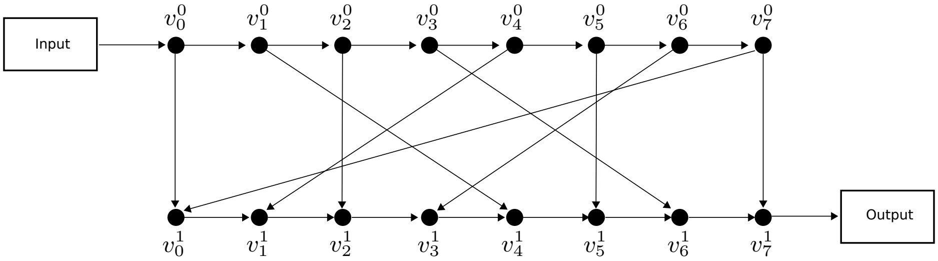

Fig. 1. An $(8, 1)$-BRG.

Algorithm 2 (G, λ)-Bit-Reversal Hashing ((G, λ)-BRH)

```
Input: $g$ {Garlic}, $x$ {Value to Hash}, $\lambda$ {Depth}, $H$ {Hash Function}
Output: $x$ {Password Hash}
 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 $k = 1, \dots, \lambda$ do
 6:     $r_0 \leftarrow H(v_0 \mid\mid v_{2^g-1})$
 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 $r_{2^g-1}$
```

7.2 Catena-DBG

Note that a (G, λ)-Double-Butterfly Graph is based on a stack of λ G-superconcentrators. The following definition of a G-superconcentrator is a slightly adapted version of that introduced in [31].

Definition 10 (G-Superconcentrator). *A directed acyclic graph $\Pi(\mathcal{V}, \mathcal{E})$ with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$, a bounded indegree, G inputs, and G outputs is called a G-superconcentrator if for every k such that $1 \leq k \leq G$ and for every pair of subsets $V_1 \subset \mathcal{V}$ of k inputs and $V_2 \subset \mathcal{V}$ of k outputs, there are k vertex-disjoint paths connecting the vertices in V_1 to the vertices in V_2.*

A double-butterfly graph (DBG) is a special form of a G-superconcentrator which is defined by the graph representation of two back-to-back placed Fast Fourier Transformations [10]. More detailed, it is a representation of twice the Cooley-Tukey FFT algorithm [12] omitting one row in the middle (see Figure 2 for an example where $G = 8$). Therefore, a DBG consists of $2 \cdot g$ rows.
Based on the DBG, we define the sequential and stacked (G, λ)-double-butterfly graph. In the following, we denote $v_{i,j}^k$ as the j-th vertex in the i-th row of the k-th double-butterfly graph.

Definition 11 ((G, λ)-Double-Butterfly Graph). *Fix a natural number $g \geq 1$ and let $G = 2^g$. Then, a (G, λ)-double-butterfly graph $\Pi(\mathcal{V}, \mathcal{E})$ consists of $2^g \cdot (\lambda \cdot (2g - 1) + 1)$ vertices*

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

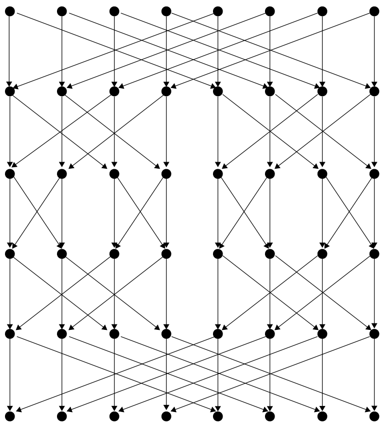

Fig. 2. A Cooley-Tukey FFT graph with eight input and output vertices.

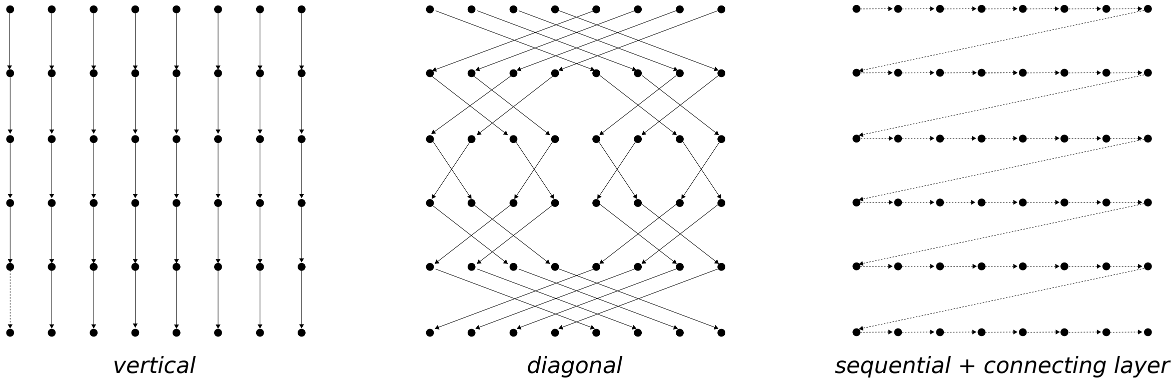

Fig. 3. Types of edges as we use them in our definitions.

- $\{v^{\lambda}_{2g-1,0}, \ldots, v^{\lambda}_{2g-1,2^g-1}\}$,

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

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

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

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

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

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

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

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

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

Figure 3 illustrates the individual types of edges we used in our definition above. Moreover, an example for $G = 8$ and $\lambda = 1$ can be seen in Figure 4.

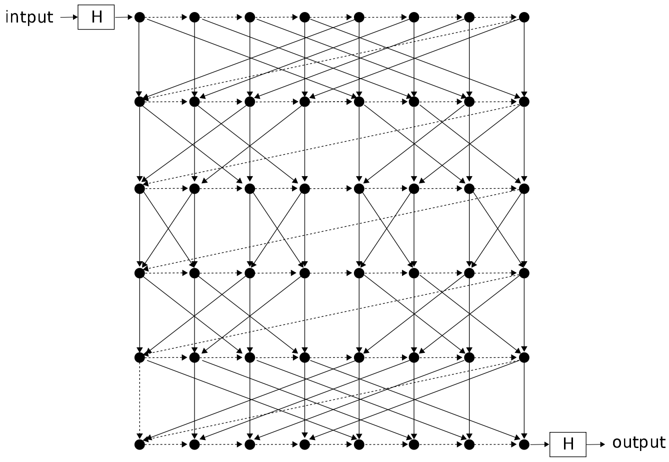

Fig. 4. An $(8,1)$-double-butterfly graph.

Algorithm 3 (G,λ)-Double-Butterfly Hashing ((G,λ)-DBH)

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

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

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

Thus, σ determines the indices of the vertices of the diagonal edges (see Figure 3).

Since the security of Catena in terms of password hashing is based on a time-memory tradeoff, it is desired to implement it in an efficient way, making it possible to increase the required memory. We recommend to use BLAKE2b [4] as the underlying hash function, implying a block size of 1024 bits with 512 bits of output. Thus, it can process two input blocks within one compression function call. This is suitable for Catena-BRG since a bit-reversal graph satisfies a fixed indegree of at most 2. When considering Catena-DBG, we cannot simply concatenate the inputs to H while keeping the same performance per hash function call, i.e., three inputs to H require two compression function calls, which is a strong slow-down in comparison to (G,λ)-BRG. Therefore, we compute $H(X,Y,Z) = H(X \oplus Y \mid\mid Z)$

instead of $H(X, Y, Z) = H(X \mid\mid Y \mid\mid Z)$ obtaining the same performance as Catena-BRG per hash function call. Obviously, this doubles the probability of input collision. Nevertheless, for a 512-bit hash function the advantage for an adversary is still negligible.

Based on the approach above, the number of hash function calls to compute Row r_i from Row r_{i-1} is the same for Catena-BRG and Catena-DBG. Moreover, for both instantiations it holds that the number of hash function calls is equal to the number of compression function calls (when used with BLAKE2b). More detailed, the (G, λ)-BRG requires $2^g - 1 + \lambda \cdot 2^g$ calls to H and the (G, λ)-DBG requires $2^g - 1 + \lambda \cdot (2g - 1) \cdot 2^g$ calls to H. It is easy to see, that the performance of Catena-DBG in comparison to Catena-BRG is decreased by a logarithmic factor.

8 Security Analysis of Catena-BRG and Catena-DBG

In this section we discuss the security of Catena-BRG and Catena-DBG against side-channel attacks. Furthermore, we discuss the memory-hardness and pseudorandomness of both instantiations.

8.1 Resistance Against Side-Channel Attacks

Straightforward implementations of either Catena-BRG or Catena-DBG provide neither a password-dependent memory-access pattern nor password-dependent branches. Therefore, both instantiations are resistant against cache-timing attacks (see Definition 5).

Considering a malicious garbage collector, each of Algorithms 2 and 3 exposes the arrays v and r. Both arrays are overwritten multiple times. Therefore, Catena-DBG is resistant against garbage-collector attacks. *It follows that any variant of* Catena *with some fixed $\lambda \geq 2$ is at least as resistant to garbage-collector attacks as the same variant with $\lambda - 1$ in the absence of a malicious garbage collector.*

8.2 Memory-Hardness

The memory-hardness of an algorithm which can be represented as a DAG with bounded indegree, can be shown by "playing" the pebble game (see Section 4). Here, we restate and discuss the results presented by Lengauer and Tarjan in [31].

Catena-BRG. In [31], Lengauer and Tarjan have proven the lower bound of pebble movements for a $(G, 1)$-bit-reversal graph.

Theorem 3 (Lower Bound for a $(G, 1)$-BRG [31]). *If $S \geq 2$, then, pebbling the bit-reversal graph $\Pi_g(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input nodes with S pebbles takes time*

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

Biryukov and Khovratovich have shown in [8] that stacking more than one bit-reversal graph only adds some linear factor to the quadratic time-memory tradeoff. Hence, a (G, λ)-BRG with $\lambda > 1$ does not achieve the properties of a λ-memory-hard function.

Catena-DBG. Likewise, the authors of [31] analyzed the time-memory tradeoff for a stack of λ G-superconcentrators. Since the double-butterfly is a special form of a G-superconcentrators, their bound also holds for (G, λ)-DBG.

Theorem 4 (Lower Bound for a (G, λ)-Superconcentrator [31]). *Pebbling a (G, λ)-superconcentrator using $S \leq G/20$ black and white pebbles requires T placements such that*

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

Discussion. For scenarios where a quadratic time-memory tradeoff is sufficient, we recommend the efficient Catena-BRG with either $\lambda = 1$ or – if garbage-collector attacks pose a relevant threat – with $\lambda = 2$. Note that the benefit of greater values for λ is very limited since the costs for pebbling the bit-reversal graph remain quadratic. For scenarios that require a higher time-memory tradeoff, we highly recommend the λ-memory-hard Catena-DBG with $\lambda = 2$ or $\lambda = 3$, which is sufficient for most practical applications.

We have to point out that the computational effort for (G, λ)-DBH with reasonable values for G, e.g., $G \in [2^{17}, 2^{21}]$, may stress the patience of many users since the number of vertices and edges grows logarithmic with G. Thus, it remains an open research problem to find a (G, λ)-superconcentrator – or any other λ-memory-hard function – that can be computed more efficiently than a (G, λ)-DBH.

8.3 Pseudorandomness.

For proving the pseudorandomness of Catena-BRG and Catena-DBG, we refer to the definition Random-Oracle Security which was introduced in Section 3 (see Definition 6). Therefore, we model the internally used hash function $H : \{0, 1\}^* \to \{0, 1\}^n$ as a random oracle.

Theorem 5 (Collision Security of (G, λ)-BRH). *Let q denote the number of queries made by an adversary and s a randomly chosen salt value. Furthermore, let H be modelled as a random oracle. Then, we have*

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

Proof. Let $a^i = (pwd^i \mid\mid u^i \mid\mid s^i \mid\mid g^i)$ represent the i-th query, where pwd^i denotes the password, u^i denotes the tweak, s^i the salt, 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 (G, λ)-BRH(g, x, λ) of the j-th query (Line 4 of Algorithm 1, where F^{λ}_H is replaced by the function (G, λ)-BRH shown in Algorithm 2). Then, $H(g \mid\mid y^j)$ is the output of Catena$_{\lambda}(a^j)$. In the case that $y^1, \ldots, y^q$ are pairwise distinct, an adversary $\mathcal{A}$ cannot distinguish $H(g \mid\mid \cdot)$ from $\$(\cdot)$, 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 $\mathcal{A}'s$ 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 `bad event`

Then, we define a new game Catena$'$ which maintains two initially empty list B_{in} and B_{out} and works as follows: When calling Catena$_\lambda$, it stores all inputs and outputs of H which are gained from the *inner* calls to the (G, λ)-BRH operation (see Line 3 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 4 of Algorithm 1) are stored in the set B_{out}. Then, it returns a random value R and finally, an adversary wins iff one of the sets B_{in} or B_{out} contains `bad event`. Remark, the advantage of winning the game Catena$'$ is higher than the advantage for distinguishing Catena$_\lambda(\cdot)$ 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 `bad event`. One call to (G, 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 `bad event`. Since there are at most $(c + 1)$ outer calls to the function H, this can be upper bounded by

$$\frac{(c + 1)^2}{2^n}.$$

Since an adversary is allowed to ask at most q queries, the probability that either B_{in} or B_{out} contains `bad event`, can be upper bounded by

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

Our claim follows from the union bound. □

Finally, we analyze the collision resistance of (G, λ)-DBH. Again, we model the internally used hash function $H : \{0, 1\}^* \to \{0, 1\}^n$ as a random oracle.

Theorem 6 (Collision Security of (G, λ)-DBH). *Let q denote the number of queries. Furthermore, let H be modelled as a random oracle for some fixed integers $g, g_0, \lambda \geq 1$ with $g \geq g_0$ and $G = 2^g$. Then, it holds that*

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

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

For our analysis, we replace the random oracle H by $H'(x) := H(\text{truncate}_n(x))$ that truncates any input to n bits before hashing. Thus, any collision in the first n bits of the input of H in Line 7 and 9 of Algorithm 3 leads to a collision of the output of H, regardless of the remaining inputs.

Output Collision. In this case we upper bound the collision probability for H by deducing the total amount of invocations of H' per query. There are 2^g invocations of H' in Lines 1–4 of Algorithm 3. In addition, there are $\lambda \cdot (2g - 1) \cdot 2^g$ invocations in Lines 5–13 leading to a total of $\lambda \cdot 2g \cdot 2^g$ invocations of H. Since H is modelled as a random oracle, we can upper bound the collision probability for q queries by

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

Input Collision. In this case we have to take into account that an input collision for distinct queries a and b in Line 7 and 9 can occur:

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

or

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

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

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

Our claim follows from the union bound. □

9 Implementation and Benchmarking of Catena

In this section we (1) clarify our choice of the internally used hash function and (2) present an optimized implementation of Catena. Recommendations for the parameters of Catena can be found in Table 3.

Parameter	Description	Encoding	Instantiations	
			(G,λ)-BRH	(G,λ)-DBH
g_p	garlic (password hashing)	1 byte	17	15
g_k	garlic (key derivation)	1 byte	20	18
λ	depth	1 byte	2	2
d	domain	1 byte	-	-
s	salt	byte string	16 bytes	16 bytes
$\|s\|$	salt length	UInt32	-	-

Table 3. 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.

Hash Function Choice. For the practical application of Catena, we were 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 the following hash functions: SHA2-512 [41], SHA3-512, and BLAKE2b [4]. Note that SHA3-512 is not standardized yet and thus, we refer to Keccak-512 [7] with $c = 1024$, where c denotes the capacity.

The advantage of SHA2-512 is that it is well-analyzed [2,22,29], standardized, and widely used, e.g., in `sha512crypt`, the common password scrambler in several Linux distributions [14]. We decided on the use of SHA3-512 as an alternative, since it will soon become a standardized hash function, it is easy to analyze, and it maintains a 1600-bit state, which can provide, depending on the choice of the capacity, a high security margin. Additionally, we point out that SHA3-512 has a high performance in hardware, which suits very well for adversaries using dedicated hardware. Our decision for BLAKE2b was motivated by its 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.

Note that the security of Catena does not rely on the performance of a specific hash function, but on the size of F_λ, i.e., the depth λ and the width g. Thus, even in the case of a secure but very fast cryptographic hash function, which may be counter-intuitive in the password-scrambling scenario, one can adapt the security parameter to reach the same computational effort.

10 Catena for Proof of Work

The concept of proofs of work was introduced by Dwork and Naor [16] in 1992. The principle design goal was to combat junk mail under the usage of CPU-bounded functions, i.e., the goal was to gain control over the access to shared resources. The main idea is "*to require a user to compute a moderately hard, but not intractable, function in order to gain access to the resource* " [16]. Therefore, they introduced so called CPU-bound *pricing functions* based on certain mathematical problems which may be hard to solve (depending on the parameters), e.g., extracting square roots modulo a prime. Tromp recently proposed the "first trivially verifiable, scalable, memory-hard and tmto-hard proof-of-work system" in [56].

As an advancement to CPU-bound function,Abadi et al. [1], and Dwork et al. [15] considered moderately hard, memory-bound functions, since memory access speeds do not vary so much on different machines like CPU accesses. Therefore, they may behave more equitably than CPU-bound functions. These memory-bound function base on a large table which is randomly accesses during the execution, causing a lot of cache misses. Dwork et al. presented in [17] a compact representation for this table by using a time-memory trade-off for its generation. Dziembowski et al. [19] as well as Ateniese et al. [3] put forward the concept of proofs of space, i.e., they do not consider the number of accesses to the memory (as memory-bound function do) but the amount of disk space the prover has to use. In [19], the authors proposed a new scheme using "graphs with high pebbling complexity and Merkle hash-trees".

For Catena, there exist at least two possible attempts to be used for proofs of work. We denote by C the client which has to fulfill the challenge to gain access to a server S. Furthermore, the methods explained below work for both introduced instantiations of Catena and let λ and F_λ be fix.

Guessing Secret Bits (Pepper). At the beginning, S chooses fixed values for pwd, t, s and g, where s denotes a randomly chosen k-bit salt value, where p bits of s are secret, i.e., p-bit pepper with $p \leq k$. Then, S computes $h = \text{Catena}_\lambda(pwd, t, s, g)$ and sends the tuple $(pwd, t, s_{[0,k-p-1]}, g, h, p)$ to C, where $s_{[0,k-p-1]}$ denote the $k - p$ least significant bits of s (the public part). Now, C has to guess the secret bits of the salt by computing $h' = \text{Catena}_\lambda(pwd, t, s', g)$ about 2^p times and comparing if $h = h'$. If so, C gains access to S. The effort of C is given by about 2^p computations of Catena (and about 2^p comparisons for $h = h'$). Hence, the effort of C is scalable by adapting p.

Guessing the Correct Password. In this scenario S chooses an m-bit password pwd, t, s, and g. Then, S computes $h = \text{Catena}_\lambda(pwd, t, s, g)$ and sends the tuple (t, s, g, m, h) to C. The client C then has to guess the password by computing about 2^m times $h' = \text{Catena}_\lambda(pwd', t, s, g)$ for different values of pwd', and comparing if $h' = h$. If so, C gains access to S. The effort of C is given by about 2^m computations of Catena (and about 2^m comparisons for $h = h'$). Hence, in this case the effort of C is scalable by adapting the length m of the password. Furthermore, S can adjust the effort of C by excluding m from the tuple sent to C. Then, since C does not know the length of the original password, the time for finding *pwd'* with $pwd' = pwd$ highly depends on the way C performs password cracking. Note that the latter may not really be suitable for the proof-of-work scenario since a prover with experience in password cracking can access the server significantly faster than a non-expert.

11 Catena in Different Environments

In this section we briefly discuss applications and usages of Catena.

11.1 Using Catena with Multiple Number of Cores

Catena is designed to initially thwart the possibility of being parallel computed. Nevertheless, one can make use of a multiple number of cores when some bits of the salt are kept secret, i.e., using pepper. Assume that $|p| = 2$, where p denotes the pepper. If a client is capable of using only one core, it will compute $h_i = \text{Catena}_\lambda(pwd, t, s_{0,\ldots,|s|-2} \mid\mid i, g)$ for $i = 0, \ldots, 3$ sequentially until one of the values h_i is verified by the server. In average, the client will compute two out of four values. On the other hand, if a client is capable of using four cores, it will compute $h_i = \text{Catena}_\lambda(pwd, t, s_{0,\ldots,|s|-2} \mid\mid i, g)$ for $i = 0, \ldots, 3$ in parallel using one core for each possible value of i. Thus, reducing the effort for the login phase to the time of one call to Catena_λ, in average. To keep the login time constant, the client will always compute h_i in parallel for all possible values of i.

Now, we briefly discuss the impact on the side of an adversary $\mathcal{A}$. Assume that $\mathcal{A}$ is provided with a database of ℓ leaked password hashes h_j for $j = 1, \ldots, \ell$, where each value h_j was generated using a different value s_j (salt), and a randomly chosen value for p_j (pepper). To rule out a given password candidate, $\mathcal{A}$ has to try all possible values p_j for each given pair (s_j, h_j). Thus, if $\mathcal{A}$ needed ℓ cores to test one password candidate for all hash values h_j in parallel, it now needs $2^{|p|} \cdot \ell$ cores to reach the same speedup.

11.2 Application of Server Relief

The application of server relief leads to significantly reduced effort on the side of the server for computing the output of Catena_λ 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.

12 The Key-Derivation Function Catena-KG

In this section we introduce Catena-KG – a mode of operation based on Catena, which can be used to generate different keys of different sizes (even larger than the natural output

Algorithm 4 Catena-KG

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

size of Catena, see Algorithm 4). 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

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

The call to $\textsc{Catena}_\lambda$ is followed by an output transform that takes the output x of $\textsc{Catena}_\lambda$, a one-byte *key identifier* $\mathcal{I}$, and a parameter ℓ for the key length as the input, and generates key material of the desired output size. Catena-KG is even able to handle the generation of extra-long keys (longer than the output size of H), by applying H in Counter Mode [18]. Note that longer keys do not imply improved security, in that context.

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

In contrast to the password hashing scenario where a user want to log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [57], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, for Catena-KG, we recommend to use $g = 20$ (when instantiated with a (G, λ)-BRG) and $g = 18$ (when instantiated with a (G, λ)-DBG).

Security Analysis. It is easy to see that Catena-KG inherits its memory-hardness from Catena (see Section 8, Theorems 3 and 4) since it invokes $\textsc{Catena}_\lambda$ (Line 1 of Algorithm 4). Next, we show that Catena-KG a good pseudorandom function (PRF) in the random oracle model.

Theorem 7. *In the random oracle model we have*

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

Proof. Suppose that H is modeled as random oracle. For the sake of simplification, we omit the truncation step and let the adversary always get access to the untruncated key k. Suppose x^i denotes the output of $\textsc{Catena}_\lambda$ of the i-th query. In the case $x^i \neq x^j$ for all values with $1 \leq i < j \leq q$, the output k is always a random value, since H is always invoked with a fresh input (see Line 4, Algorithm 4). The only chance for an adversary to distinguish Catena-KG$(\cdot)$ from the random function $\$(\cdot)$ is a collision in Catena$_\lambda$. The probability for this event can be upper bounded by similar arguments as in the proof of Theorem 2. □

13 Conclusion and Outlook

In this paper we have presented Catena, a novel password-scrambling framework which is based on the memory-hard F_λ function, where we presented two possible instantiations for F_λ: (1) Catena-BRG– providing memory-hardness and (2) Catena-DBG– providing λ-memory-hardness. Catena is the first algorithm which naturally supports *client-independent updates* and *server relief*. It consists of two security parameters λ (depth) and g (garlic), where λ reflects the memory-hardness (for Catena-DBG) and g the memory consumption. Note that Catena-BRG provides memory-hardness independent of λ.

Further, we have shown that Catena is provably secure in the random oracle model. The motivation for the overall design is to thwart GPU-based attacks and to provide resistance against cache-timing as well as garbage-collector attacks. Thereby, we were inspired by the discovery of cache-timing attacks on `scrypt`. Nevertheless, the application of Catena is not restricted to the generation of password hashes. It also fits very well for the usage as a key-derivation function, since we have shown that it behaves like a pseudorandom permutation in the random oracle model. Additionally, Catena is well-suitable for the usage in proofs of work.

Finally, based on its high flexibility and security properties, Catena (and its instantiations Catena-BRG and Catena-DBG) seems to be the reasonable choice for current and future applications in comparison with existing algorithms (see Section 2). Also, we expect that future password scrambler designs will borrow the two novel features of Catena, i.e., *client-independent update* and *server relief*.

14 Acknowledgement

Thanks to Bill Cox, Eik List, Colin Percival, Stephen Touset, Steve Thomas, Alexander Peslyak, as well as the reviewers of the FSE'14 and ASIACRYPT'14 for their helpful hints and comments, as well as fruitful discussions. Finally, we thank Alexander Biryukov and Dimitry Khovratovich for pointing out a flaw in our proof of the time-memory tradeoff for the (G, λ)-BRH operation by providing a tradeoff cryptanalysis [8].

References

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, 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.

5. Anne Barsuhn. Cache-Timing Attack on scrypt. Bauhaus-Universität Weimar, Bachelor Dissertation, December 2013.
6. S.M. Bellovin and M. Merrit. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. Proceedings of the I.E.E.E. Symposium on Research in Security and Privacy (Oakland), 1992.
7. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.
8. Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of Catena. PHC mailing list: discussions@password-hashing.net.
9. Xavier Boyen. Halting Password Puzzles – Hard-to-break Encryption from Human-memorable Keys. In *16th USENIX Security Symposium—SECURITY 2007*, pages 119–134. Berkeley: The USENIX Association, 2007. Available at `http://www.cs.stanford.edu/~xb/security07/`.
10. William F. Bradley. Superconcentration on a Pair of Butterflies. *CoRR*, abs/1401.7263, 2014.
11. Stephen A. Cook. An Observation on Time-Storage Trade Off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.
12. J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.
13. Solar Designer. Enhanced challenge/response authentication algorithms. `http://openwall.info/wiki/people/solar/algorithms/challenge-response-authentication`. Accessed January 22, 2014.
14. Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. `http://www.akkadia.org/drepper/SHA-crypt.txt`. Accessed May 16, 2013.
15. Cynthia Dwork, Andrew Goldberg, and Moni Naor. On Memory-Bound Functions for Fighting Spam. In *CRYPTO*, pages 426–444, 2003.
16. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *CRYPTO*, pages 139–147, 1992.
17. Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and Proofs of Work. In *CRYPTO*, pages 37–54, 2005.
18. Morris Dworkin. *Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation.* National Institute of Standards, U.S. Department of Commerce, December 2001.
19. Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of Space. *IACR Cryptology ePrint Archive*, 2013:796, 2013.
20. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-Evolution Schemes Resilient to Space-Bounded Leakage. In *CRYPTO*, pages 335–353, 2011.
21. Eric Glass. The NTLM Authentication Protocol and Security Support Provider. `http://davenport.sourceforge.net/ntlm.html`. Accessed May 16, 2013.
22. 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.
23. Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
24. John Hopcroft, Wolfgang Paul, and Leslie Valiant. On Time Versus Space. *J. ACM*, 24(2):332–337, April 1977.
25. B. Kaliski. RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000.
26. Poul-Henning Kamp. The history of md5crypt. `http://phk.freebsd.dk/sagas/md5crypt.html`. Accessed May 16, 2013.
27. Takumi Kasai, Akeo Adachi, and Shigeki Iwata. Classes of Pebble Games and Complete Problems. In *ACM Annual Conference (2)*, pages 914–918, 1978.
28. John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure Applications of Low-Entropy Keys. In Eiji Okamoto, George I. Davida, and Masahiro Mambo, editors, *ISW*, volume 1396 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 1997.
29. Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family. In *FSE*, pages 244–263, 2012.
30. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.
31. 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.
32. Andrzej Lingas. A PSPACE Complete Problem Related to a Pebble Game. In *ICALP*, pages 300–321, 1978.

33. T. Alexander Lystad. Leaked Password Lists and Dictionaries - The Password Project. http://thepasswordproject.com/leaked_password_lists_and_dictionaries. Accessed May 16, 2013.
34. Udi Manber. A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack. *Computers & Security*, 15(2):171–176, 1996.
35. Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.
36. Robert Morris and Ken Thompson. Password Security - A Case History. *Commun. ACM*, 22(11):594–597, 1979.
37. Krishna Neelamraju. Facebook Pages: Usage Patterns | Recommend.ly. http://blog.recommend.ly/facebook-pages-usage-patterns/. Accessed May 16, 2013.
38. 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.
39. Nvidia. Nvidia GeForce GTX 680 - Technology Overview, 2012.
40. Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. *Advances in Cryptology-CRYPTO 2003*, 3:617–630, 2003.
41. NIST National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. April 1995. See http://csrc.nist.gov.
42. 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.
43. Wolfgang J. Paul and Robert Endre Tarjan. Time-Space Trade-Offs in a Pebble Game. In *ICALP*, pages 365–369, 1977.
44. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
45. Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
46. A.G. Reinhold. HEKS: A Family of Key Stretching Algorithms, 1999.
47. Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm.* Internet Activities Board, April 1992.
48. Semiocast SAS. Brazil becomes 2nd country on Twitter, Japan 3rd Netherlands most active country. http://goo.gl/QOeaB. Accessed May 16, 2013.
49. J. Savage and S. Swamy. Space-time trade-offs on the FFT algorithm. *Information Theory, IEEE Transactions on*, 24(5):563 – 568, sep 1978.
50. John E. Savage and Sowmitri Swamy. Space-Time Tradeoffs for Oblivious Interger Multiplications. In *ICALP*, pages 498–504, 1979.
51. Bruce Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *FSE*, pages 191–204, 1993.
52. Ravi Sethi. Complete Register Allocation Problems. *SIAM J. Comput.*, 4(3):226–248, 1975.
53. Jens Steube. oclHashcat-plus - Advanced Password Recovery. http://hashcat.net/oclhashcat-plus/. Accessed May 16, 2013.
54. Sowmitri Swamy and John E. Savage. Space-Time Tradeoffs for Linear Recursion. In *POPL*, pages 135–142, 1979.
55. Martin Tompa. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. In *STOC*, pages 196–204, 1978.
56. John Tromp. Cuckoo Cycle; a memory-hard proof-of-work system. Cryptology ePrint Archive, Report 2014/059, 2014. http://eprint.iacr.org/.
57. 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.
58. M.V. Wilkes. *Time-Sharing Computer Systems.* MacDonald computer monographs. American Elsevier Publishing Company, 1968.

A The scrypt Password Scrambler

Algorithm 5 describes the scrypt password scrambler and its core operation ROMix. For pre- and post-processing, scrypt invokes the one-way function PBKDF2 [25] to support inputs and outputs of arbitrary length. ROMix uses a hash function H with n output bits, where n is the size of a cache line (at current machines usually 64 bytes). To support hash functions with

Algorithm 5 The `scrypt` algorithm and its core operation `ROMix` [44].

```
scrypt
Input:
    $pwd$ {Password}
    $s$ {Salt}
    $G$ {Cost Parameter}
Output: $x$ {Password Hash}
10: $x \leftarrow \text{PBKDF2}(pwd, s, 1, 1)$
11: $x \leftarrow \text{ROMix}(x, G)$
12: $x \leftarrow \text{PBKDF2}(pwd, x, 1, 1)$
13: return $x$

ROMix
Input: $x$ {Initial State} , $G$ {Cost Parameter}
Output: $x$ {Hash value}
20: for $i = 0, \ldots, G-1$ do
21:     $v_i \leftarrow x$
22:     $x \leftarrow H(x)$
23: end for
24: for $i = 0, \ldots, G-1$ do
25:     $j \leftarrow x \bmod G$
26:     $x \leftarrow H(x \oplus v_j)$
27: end for
28: return $x$
```

Algorithm 6 The algorithm `ROMixMC`, performing `ROMix` with K/G storage.

```
Input:
    $x$ {Initial State},
    $G$ {1st Cost Parameter},
    $K$ {2nd Cost Parameter}
Output: $x$ {Hash Value}
 1: for $i = 0, \ldots, G-1$ do
 2:     if $i \bmod K = 0$ then
 3:         $v_i \leftarrow x$
 4:     end if
 5:     $x \leftarrow H(x)$
 6: end for
 7: for $i = 0, \ldots, G-1$ do
 8:     $j \leftarrow x \bmod G$
 9:     $\ell \leftarrow K(j/K)$ { "/" is the integer division }
10:     $y \leftarrow v_\ell$
11:     for $m = \ell+1, \ldots, j$ do
12:         $y \leftarrow H(y)$ { invariant: $y \leftarrow v_m$ }
13:     end for
14:     $x \leftarrow H(x \oplus y)$
15: end for
16: return $x$
```

smaller output sizes, [44] proposes to instantiate H by a function called BlockMix, which we will not elaborate on. For our security analysis of `ROMix`, we model H as a random oracle.

`ROMix` takes two inputs: an initial state x, which depends on both salt and password, and the array size G that defines the required storage. One can interpret $\log_2(G)$ as the garlic factor of `scrypt`. In the first phase (lines 20–23), `ROMix` initializes an array v. More detailed, the array variables $v_0, v_1 \ldots, v_{G-1}$ are set to $x, H(x), \ldots, H(\ldots(H(x)))$, respectively. In the second phase (lines 24–27), `ROMix` updates x depending on v_j. The sequential memory-hardness results from the way how the index j is computed, depending on the current value of x, i.e., $j \leftarrow x \bmod G$. After G updates, the final value of x is returned and undergoes the post-processing.

A minor issue is that `scrypt` uses the password pwd as one of the inputs for post-processing. Thus, it has to stay in storage during the entire password-scrambling process. This becomes a risk if there is any chance that the memory can be compromised during the time `scrypt` is running. Compromising the memory should not happen, anyway, but this issue could easily be fixed without any bad effect on the security of `scrypt`, e.g., one could replace Line 12 of Algorithm 5 by $x \leftarrow \text{PBKDF2}(x, s, 1, 1)$.

A.1 Brief Analysis of `ROMix`

The following part deals with a variant of `ROMix` which can be computed with less than G units of memory. Suppose we only have $S < G$ units of storage for the values in v. For convenience, we assume G is a multiple of S and set $K \leftarrow G/S$. As it will turn out, the memory-constrained algorithm `ROMixMC` (see Algorithm 6) generates the same result as `ROMix` with less than G storage places and is $\Theta(K)$ times slower than `ROMix`. From the array

v, we will only store the values v_0, v_K, v_{2k}, $\ldots$, $v_{(S-1)K}$ – using all the S memory units available.

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

Next, we briefly discuss why `ROMix` is sequentially memory-hard (for the full proof see [44]). The intuition is as follows. The indices j are determined by the output of the random oracle H and thus, essentially, uniformly distributed random values over $\{0, \ldots, G-1\}$. With no way to anticipate the next j, the most suitable strategy is to minimize the size of the "gaps", i.e., the number of consecutively unknown v_j. This is indeed what `ROMixMC` does, by storing one v_i every K-th step.

A.2 Cache-Timing Attacks

Algorithm 5 (`scrypt/ROMix`) revisited. What could possibly go wrong?

The Spy Process. As it turns out, the idea to compute a "random" index j and then ask for the value v_j, which is immensely useful for sequential memory-hardness, is also an issue. Consider a spy process, running on the same machine as `scrypt`. This spy process cannot read the internal memory of `scrypt`, but, as it is running on the same machine, it shares its cache memory with `ROMix`. The spy process interrupts the execution of `ROMix` twice:

1. When `ROMix` enters the second phase (Line 24 of Algorithm 5), the spy process reads from a bunch of addresses, to force out all the v_i that are still in the cache. Thereupon, `ROMix` is allowed to run for another very short time.
2. Now, the spy process interrupts `ROMix` again. By measuring access times when reading from different addresses, the spy process can figure out which of the v_i has been read by `ROMix`, in between.

So, the spy process can tell us the indices j for which v_j has been read, and with this information, we can mount the following cache-timing attack.

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

1. Run the first phase of `ROMix`, without storing the v_i (i.e., skip Line 21 of Algorithm 5).
2. Compute the index $j \leftarrow x \bmod G$.
3. If v_j is one of the values that have been read by `ROMix`, then store pwd in a list.
4. Else, conclude that pwd is a wrong password.

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

Next, we discuss the gain of this attack. Let r denote the number of iterations the loop in lines 24–27 of ROMix has performed, before the second interrupt by the spy process. So, there are at most r indices j with v_j being read. That means, we expect this approach to sort out all but r/G candidates. If our spy process manages to interrupt very soon, after allowing it to run again, we have $r \ll G$. This may enable us to use conventional hardware to run full ROMix to search for the correct password among the candidates on the list.

Final Cache-Timing Attack. In this attack we allow the second interrupt to arrive very late – maybe even as late as the termination time of ROMix. So, the loop in lines 24–27 of ROMix has been run $r = G$ times. As it seems, each v_i has been read once. But actually, this is only true *on average*; some v_i have been read more then once, and we expect about $(1/e)G \approx 0.37\,G$ array elements v_i not to have been read at all. So applying the basic attack allows us to eliminate about 37 % of all password candidates – a rather small gain for such hard work.

In the following we introduce a way to push the attack further, inspired by Algorithm 6, the memory-constrained ROMixMC. Our final cache-timing attack on scrypt only needs the smallest possible amount of memory: $S = 1, K = G/S = G$, and thus, we only have to store the single value v_0. Like the second phase of ROMixMC, we will compute the values v_j on-the-fly when needed. Unlike ROMixMC, we will stop execution whenever one of our values j is such that v_j has not been read by ROMix (according to the info from our spy process).

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

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

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

As it turns out, rejecting a wrong password with constant memory is faster then computing ordinary ROMix with all the required storage, which actually makes $2G$ calls to H, without computing any v_i on-the-fly. We stress that the ability to abort the computation, thanks to the information gathered by the spy process, is crucial. Meanwhile, we are working on an implementation to verify this attack.

A.3 The Garbage-Collector Attack

Memory-demanding password scramblers such as ROMix defend against a massively-parallel password-cracking approach, since the required memory is proportional to the number of passwords scrambled in parallel.

But, memory-demanding password scrambling may also open the gates for new attack opportunities for the adversary. If we allocate a huge block of memory for password scrambling, holding v_0, v_1, …, v_{G-1}, 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 termination of the password scrambler.

For ROMix, the value $v_0 = H(x)$ is a plain hash of the original secret x. Hence, a malicious garbage collector can completely bypass ROMix and search directly for x with $H(x) = v_0$, implying that each password candidate can be checked in time and memory complexity of $\mathcal{O}(1)$. Furthermore, if the adversary fails to learn v_0, but any of the other values $v_i = H(v_{i-1})$, the computational effort grows to $\mathcal{O}(i)$, but the memory complexity is still $\mathcal{O}(1)$.

Thus, ROMix does not provide much defense against garbage-collector attacks. A possible countermeasure would be to overwrite $v_0, \ldots, v_{G-1}$ after running ROMix. But, this step might be removed by a compiler due to optimization, since it is algorithmically ineffective.

A.4 Discussion

Currently, the attacks above are of theoretical nature. The garbage-collector attack requires the adversary to be able to read the memory occupied by ROMix, after its usage. Whereas the cache-timing attack requires to (1) run a spy process on the machine ROMix is running, (2) interrupt ROMix twice at the right points of time, and (3) precisely measure the timings of memory reads. Moreover, other processes running on the same machine can add a huge amount of noise to the cache timings. It is not clear if a "real" server can ever be attacked that way.

However, in an idealized "laboratory" setting, the applicability of cache-timing attacks against ROMix has been demonstrated [5]. The idealized conditions included execution rights on the system.

Remark 2. Even without knowing the password hash at all,

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

just from knowing the memory-access pattern.

Note that severe security issues can be caused by the second point. Consider any offline attack on the password. When passwords are hashed using an old-style password hash function, e.g., md5crypt [26], the adversary first needs to read the file containing the password hash. Without the password hash, mounting an offline attack is not possible. Even plaintext passwords are safe from offline adversaries which are not capable of reading the file containing the plaintext passwords. But, using the seemingly strong password hash function scrypt may enable offline password cracking, even when the adversary fails to ever learn the password hash.