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. Therefore, we introduce a novel and provably secure password-scrambling framework called Catena and derive an instantiation Catena-λ, which is based on a memory-consuming one-way function called $\lambda-$bit-reversal graph ($\lambda-$BRG). It is characterized by its $\lambda-$memory hardness. Thus, Catena-λ excellently thwarts massively parallel attacks on cheap memory-constrained hardware, such as recent graphical processing units (GPUs). Additionally, we show that Catena-λ 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 the $\lambda-$BRG is password-independent and therefore resistance against cache-timing attacks.

Moreover, Catena 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) a *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).

Keywords: password, memory-hard, cache-timing attack, pebble game

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 "off-line" password guessing, see, e.g., [5] for an early example. Otherwise, the best protection are cryptographic password scramblers, performing "key stretching". The basic idea of such schemes is using an intentionally slow one-way function for hashing the password. Therefore, the password processing takes some time for both kinds of users legitimate ones and adversaries. A good password scrambler P has to satisfy at least the following basic conditions:

(1) Given a password pwd, computing $P(pwd)$ should be "fast enough" for the user.
(2) Computing $P(pwd)$ should be "as slow as possible", without contradicting condition (1).
(3) Given $y = P(pwd)$, there must be no significantly faster way to test q password candidates $x_1, x_2, \ldots, x_q$ for $P(x_i) = y$ than by actually computing $P(x_i)$ 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 (see [10] and [23] for

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

example). However, an adversary who happens to have c computing units ("cores") can easily try out c different passwords in parallel. With recent technological trends, such as the availability of graphical processing units (GPUs) with hundreds of cores [37], the question of how to slow down such adversaries becomes a pressing one. Memory is expensive; so, a typical GPU or other cheap 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 c-core adversaries from gaining some close-to-c-times speed-up is by making P not only intentionally slow on standard sequential computers, but also intentionally memory-consuming. In the spirit of the basic condition (3), any adversary using c cores in parallel with less than about c 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` [43]. To the best of our knowledge, it is also the only one – up to now.

However, a memory-consuming password scrambler may suffer from a new problem. 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 password scrambler's internal memory) 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. In this paper we will show that this is actually an issue for `scrypt` by presenting a cache-timing attack. Further, by introducing Catena-λ, we show that one can avoid password-defined memory-access pattern and still provide memory hardness for an algorithm. We formally analyze the security of Catena-λ and its memory consumption using the "pebble game" approach, which dates back to the early days of Theoretical Computer Science [9,21,24,30,41].

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 [34] 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 [31].

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 [38].

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* [32]. 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. [25] analyzed another key stretching approach where a cryptographic operation is iterated n times, where n is secret. Boyen proposed in [8] a user-defined implicit choice for n by iterating until the user presses a "halt" button.

Contribution. Our primary contribution is a novel password-scrambling framework called `Catena` (Latin for "chain", due to its sequential structure when instantiated with a λ-memory-hard function F^λ). `Catena` provides innovative features like *client-independent update*, when increasing the security parameter *garlic* or by turning some bits of the salt into pepper. The notion of *garlic* reflects the property that incrementing of parameter by '1' doubles the memory usage and at least doubles the computational time. Further, it provides built-in support for *server relief*, which allows to shift the main effort for computing a password hash to the client. 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` (see Section 7). Further, we provide `Catena`-λ – an instantiation of `Catena` using the $\lambda-$bit-reversal hash operation ($\lambda-$BRH) for F^λ. We prove this construction to be $\lambda-$memory-hard in the random oracle model. `Catena`-λ thus thwarts back massively parallelized attacks using GPUs and similar hardware. Additionally, `Catena`-λ has been designed to be resistant against cache-timing attacks. The need to address this issue has been inspired by revealing the cache-timing vulnerability of scrypt (see Appendix A). To the best of our knowledge, such attacks for `scrypt` have not been known before.

Outline. Section 2 briefly overviews modern password scramblers and their properties. In Section 3 we present desired properties of modern password-scrambling algorithms. Section 4 introduces `Catena`, our new password-scrambling framework satisfying the properties described in Section 3. In Section 5 we introduce an instantiation of `Catena` with a $\lambda-$BRG resulting in a memory-demanding password scrambler called `Catena`-λ. Furthermore, this section contains a brief description of how to implement `Catena`-λ. Section 6 contains a generic security analysis of `Catena`-λ. In Section 7 we show that `Catena` fits very well in the proofs of work scenario by presenting two examples. Finally, Section 8 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`. 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` [23] was used in nearly all Free-BSD and Linux-based systems to scramble user passwords. It is based on the well-known MD5 [46] 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 [52]. Additionally, its own author does not consider `md5crypt` secure anymore [23]. Common Linux distributions nowadays employ `sha512crypt` [10], e.g., Debian, Ubuntu, or Arch Linux. It provides similar features as `md5crypt`, but uses SHA-512 [39] instead of MD5. Furthermore, the number of iterations can be chosen by the user. `NTLMv1` [18] 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 [52]. 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 [28], where c is a user-chosen value, which is given by default with $c = 1000$.

bcrypt. The `bcrypt` [44] algorithm is built upon the Blowfish block cipher [50]. 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 [52]. 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.

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` [43]. (There was HEKS [45], but it has been broken by the author of `scrypt`.) As its core, `scrypt` uses the sequentially memory-hard function `ROMix` (see [43] 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 [43], 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 [43]. The intuition is that for any parallelized attack, using m cores, the required memory per core is decreased by a factor of $1/m$, and vice versa.

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

Algorithm	Cost Factor	Memory	Server	Client-Indep.	Issues
	(default)		Relief	Updates	
crypt [34]	25	small	-	-	"too fast"
md5crypt [23]	1,000	small	-	-	"too fast"
sha512crypt [10]	1,000–999,999 (5,000)	small	-	-	(small memory)
NTLMv1 [18]	1	small	-	-	"too fast"
PBKDF2 [57]	1–∞ (1,000)	small	-	-	(small memory)
bcrypt [44]	2^4–2^{99} (2^6, 2^8)	4,168 bytes	-	-	(constant memory)
scrypt [43]	1–∞ (2^{14}, 2^{20})	flexible, big	-	-	cache-timing attacks
Catena (this paper)	2^1–∞ (2^{16}, 2^{20})	flexible, big	✓	✓	-

Table 1. Comparison of state-of-the-art password scramblers and Catena. Note that all of the mentioned algorithms support salt values.

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

$$\frac{1}{m} \cdot S + m \cdot T = N^2.$$

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

Definition 2 ($\lambda-$Memory-Hard Function).
For a $\lambda-$memory-hard function f, which is computed on a Random Access Machine using $G(g)$ 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

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

λ-Memory-Hard vs. Sequential Memory-Hard. In [43], Percival introduced the notion of sequential memory hardness (we will refer to it as 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 p cores requires p 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 scrambling framework Catena satisfies only λMH instead of SMH, without referring to details of Catena, which are presented in Section 4.

Note that a further goal of our design was to provide resistance against cache-time attacks, i.e., Catena should satisfy password-independent memory access pattern. This goal can be achieved by providing a control flow which is independent of its input. It follows that Catena can be seen as a straight-line program, which on the other hand can be represented by a 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 c processors to compute a graph $\Pi(\mathcal{V},\mathcal{E})$, one can partition $\Pi(\mathcal{V},\mathcal{E})$ into c disjunct subgraphs $\pi_0,\ldots,\pi_{c-1}$. 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

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

then, one can compute all subgraphs $\pi_0,\ldots,\pi_{c-1}$ in parallel. Due to the structure of `Catena`, one can always partition its corresponding DAG into such subgraphs and hence, `Catena` can be at least partially computed in parallel, which is a contradiction to the definition of 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-time attacks.

Nevertheless, we leave the task of designing a memory-consuming password scrambler satisfying sequential memory hardness and resistance against cache-time attacks as an open research problem.

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

Definition 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*

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

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

In Section 6.3 we show that for `Catena`-λ 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 [33], 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 [36] and 73% of all Twitter accounts do not have at least one tweet per month [47]. 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. So we came up with the idea to split 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 $H(F(pwd, s))$ from the password pwd and a salt s. Alternatively, the server sends s to the client who responds $x = F(pwd, s)$. Finally, the server just computes $H(x)$. 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.

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 recommend to realize password-independent memory-access patters for upcoming password scramblers.

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 5 (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*

Algorithm 1 Catena

```
Input: $pwd$ {Password}, $t$ {Tweak} $s$ {Salt}, $g$ {Garlic}
Output: $x$ {hash of the password}
$x \leftarrow H(\texttt{0xFF} \,||\, t \,||\, pwd \,||\, s)$
for $c = 1, \ldots, g$ do
    $x \leftarrow F_H^{\lambda}(c, x)$
    $x \leftarrow H(c \,||\, (\lambda + 1) \cdot 2^c \,||\, x)$
end for
return $x$
```

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 The Catena Password-Scrambling Framework

In this section we introduce our λ-memory-hard password-scrambling framework called Catena. It consist of a family of novel and sustainable password-scrambling algorithms with high resilience against cache-timing attacks. The core of Catena is the λ-memory-hard function $F_H^{\lambda}(\cdot, \cdot)$ requiring $\mathcal{O}(2^g)$ invocations of the hash function H for a fixed value of x. The first input of F_H^{λ} denotes the garlic parameter g with $g = \log_2(G)$ (see Definition 2) and the second input denotes the value to process. Thus, g determines the units of memory required to execute $F_H^{\lambda}(\cdot, \cdot)$. Moreover, increasing g by one doubles the computational effort for computing the password hash. The formal definition of Catena is given in Algorithm 1.

The password-dependent input of H is appended to a prefix tuple (c, i), where c denotes the iteration counter (garlic factor) and i denotes the hash function invocation counter. This position encoding guarantees the uniqueness of inputs for H within a run of Catena. This becomes handy in the security analysis of Catena where H is modeled as a random oracle.

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

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

where the first byte d denotes the domain (i.e., the mode) for which Catena is used. We set $d = 0$ for the usage of Catena as a password scrambler, $d = 1$ when used as a key-derivation function (see Section 7.1), and $d = 2$ for proofs of work (see Section 7.2). The remaining possible values for d are reserved for future applications. The second byte λ defines together with the value g (see above) the security parameters for Catena. The 16-bit value n denotes the output length of the underlying hash function H in bits, and the 32-bit value $|s|$ denotes the total length of the salt in bits. The n-bit value $H(AD)$ is the hash of the associated data AD, which can contain additional information like hostname, user-ID, name of the company, or the IP of the host, with the goal to customize the password hashes. The tweak is processed together with the one-byte value `0xFF`, the salt, and the secret password (see Line 1 of Algorithm 1). Thus, t can be seen as a weaker version of a salt increasing the additional computational effort for an adversary when using different values. Furthermore, it allows to differentiate between password hashing and key derivation. Note that the first byte value `0xFF` can also theoretically be reached by the value c which would destroy the uniqueness. But, based on practical reasons, $c = \texttt{0xFF}$ will never occur, since then `Catena` would require 2^{255} operations, which is infeasible.

Garlic. `Catena` employs a graph-based structure, where the memory requirement highly depends on the number of input nodes of the permutation graph. If enough memory is available (either for an adversary or an honest entity), all input nodes (plus one for the output path – see Section 6 for details) can be stored in the cache. As the goal is to hinder an adversary to make a reasonable number of parallel password checks using the same memory, we have to consider a minimal number of input nodes. In general, we use $G = 2^g$ input nodes. A recommendation for g can be found in Table 2. But, as the available memory for GPU's and CPU's is continually growing, this recommendation will change in the future. Thus, for `Catena`, the parameter *garlic* specifies the number of input nodes G, and can be adapted in the future.

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

$$h' = H(g' \mid\mid (\lambda + 1) \cdot 2^{g'} || F(g', h)).$$

It is easy to see that the equation $h' = \texttt{Catena}(pwd, t, s, g')$ 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` is then transmitted to the server. Next, the server computes the password hash by applying the hash function H. Thus, the vast majority of the effort (memory usage and computational time) for computing the password hash is handed over to the client, freeing the server. This enables someone to deploy `Catena` even under restricted environments or when using constrained devices – or when a single server has to handle a huge amount of authentication requests.

Keyed Password Hashing. To further thwart off-line attacks, we introduce a technique to use `Catena` for keyed password hashing, where the password hash depends on both the password and a secret key K. Note that K is the same for all users, and thus, it has to be stored on server-side. To preserve the server-relief property (see above), we encrypt the output of `Catena` using a CPA-secure encryption scheme like the Counter-AES-Mode (CTR-AES) [14]:

$$y = \texttt{Catena}_K(userID, pwd, t, s, g) := \text{CTR-AES}_K(userID, \texttt{Catena}(pwd, t, s, g)),$$

where `Catena` is defined as in Algorithm 1 and the *userID* is a unique and user-specific identification number which is assigned by the server. For CTR-AES, the starting value of the counter is given by a 64-bit user-ID. Since AES processes 128-bit blocks, the remaining

64 least significant bits are set to zero. Obviously, it is a bad idea to store the secret key for the AES on hard drive, since it can be leaked in the same way as the password-hash database. Instead, we recommend to derive the key K from a password during the bootstrapping phase. Afterwards, K will be kept in the RAM and will never be on the hard drive. Note that one has to care about side-channel attacks in this setting, and hence, the usage of a side-channel resistant implementation of AES is necessary such as presented in [2,54,56].
Now we discuss what happens during the client-independent update, i.e., when $g = g + r$ for arbitrary $r \in \mathbb{N}$. First, y is decrypted using CTR-AES under K, i.e., $x = \text{CTR-AES}_K^{-1}(y)$. Second, x is updated using the new garlic parameter, i.e., $x = H(F_H^\lambda(g, x))$. Last, x is encrypted using the CTR-AES under K leading to the updated password hash.

5 Catena-λ

In this chapter we present an instantiation of the Catena password-scrambling framework using a structure called λ-bit-reversal graph ($\lambda-$BRG), which on the other hand uses a one-way function H. We call this instantiation Catena-λ. In the second part of this section we discuss how to optimize the implementation of Catena-λ.

5.1 The Core of Catena-λ

The core of Catena-λ is the $\lambda-$Bit-Reversal Hashing (λ-BRH) operation. The origin of this operation on the other hand is the $\lambda-$BRG, which is given by stacking λ bit-reversal permutations. The definition of the bit-reversal permutation is given below (see Definition 6).

Definition 6 (Bit-Reversal Permutation τ). *Fix a number $k \in \mathbb{N}$ and represent $i \in \mathbb{Z}_{2^k}$ as a binary k-bit number, $(i_0, i_1, \ldots, i_{k-1})$, i.e.,*

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

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

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

In Definition 7 we introduce a generic definition of the $\lambda-$BRG, where the edges within such a graph define the control flow of the $\lambda-$BRH operation (see Algorithm 2).

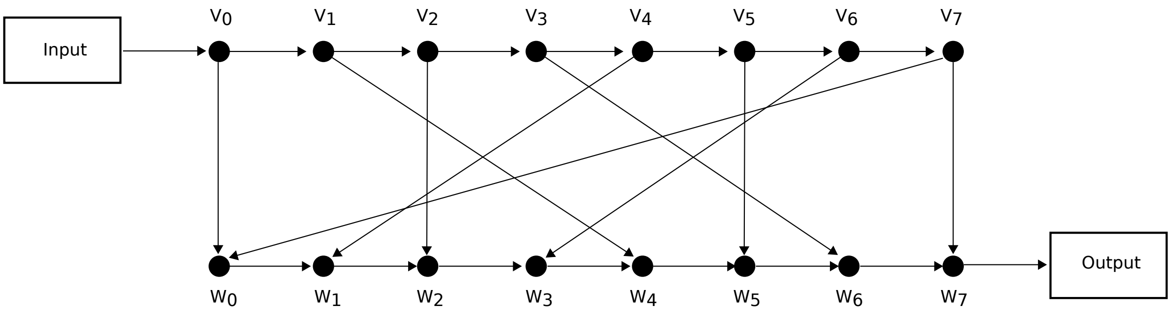

Fig. 1. A $\lambda-$bit-reversal graph with $\lambda = 1$ (SBRG) and $g = 3$ (eight input and eight output nodes).

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

```
Input: $c$ {Garlic}, $x$ {Value to Hash}, $\lambda$ {Depth}
Output: $x$ {Hash Value}
$v_0 \leftarrow H(c \,||\, 0 \,||\, x)$
for $i = 1, \dots, 2^c - 1$ do
    $v_i \leftarrow H(c \,||\, i \,||\, v_{i-1})$
end for
for $k = 0, \dots, \lambda - 1$ do
    $r_0 \leftarrow H(c \,||\, (k+1) \cdot 2^c \,||\, v_0 \,||\, v_{2^c-1})$
    for $i = 1, \dots, 2^c - 1$ do
        $r_i \leftarrow H(c \,||\, (k+1) \cdot 2^c + i \,||\, r_{i-1} \,||\, v_{\tau(i)})$
    end for
    $v \leftarrow r$
end for
return $r_{2^c-1}$
```

Definition 7 ($\lambda-$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 $\lambda-$Bit-Reversal Graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ consists of $(\lambda + 1) \cdot 2^g$ vertices*

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

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

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

For example, we call a $\lambda-$BRG with $\lambda = 1$ a *Sequential Bit-Reversal Graph* (SBRG), which is shown in Figure 1. Note that this structure is almost identical – except for one additional edge $e = (v_{2^g-1}^0, v_0^1)$ – to the Bit-Reversal Graph presented by Lengauer and Tarjan in [29].

5.2 Implementation Details

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

Parameter	Description	Encoding	Recommendation
g_p	garlic (password hashing)	1 byte	17
g_k	garlic (key derivation)	1 byte	20
λ	depth	1 byte	2
d	domain	1 byte	-
s	salt	byte string	16 bytes
$\lvert s \rvert$	salt length	UInt32	-

Table 2. Parameter choices for the practical usage of Catena-λ. 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`-λ, we where looking for a hash function with a 512-bit (64 byte) output, since it often complies with the size of a cache line on common CPUs. In any case, we assume that both the output size of H and the cache-line size are powers of two, so if they are not equal, the bigger number is a multiple of the smaller one. Moreover, the output of H should be byte-aligned. For `Catena`-λ, we decided to use the following hash functions: SHA2-512 [39], 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 [3,19,26], standardized, and widely used, e.g., in `sha512crypt`, the common password scrambler in several Linux distributions [10]. 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 the $\lambda-$BRG, i.e., the depth λ and the width g. Thus, even in the case of a secure but very fast cryptographic hash function, which may be counter-intuitive in the password-scrambling scenario, one can adapt the security parameter to reach the same computational effort.

Observation. The following observation holds for a $1-$BRG. Nevertheless, it can be applied to a $\lambda-$BRG for arbitrary values of $\lambda \in \mathbb{N}$. When the output size of H is equal to the size of a cache line (or a multiple), each time a value is read from or written to a location v_i, the time to access v_i is the same, to first order. Now, assume the output size of H (i.e., the number of bits for each of the v_i) is k times the cache line size. In this case the adversary may try to optimize the memory layout (the order in which the v_i are stored in memory) to minimize the number of cache misses. However, a nice property of the bit-reversal permutation τ is that one cannot gain much from such an optimization. If the values are stored in their natural order: v_0, v_1, $\ldots$, v_{2^g-1}, then, the number of cache misses in the first phase (lines 1 and 3 of Algorithm 2) are drastically reduced to $2^g/k$. But, in the second phase (lines 6 and 8 of Algorithm 2), the number of cache misses is 2^g. If an adversary stores the v_i in their bit-reversal order, the number of cache misses in the second phase is $2^g/k$, but, in the first it is now 2^g. A more complex mixture between natural and bit-reversal order would allow $2^g/\sqrt{k}$ cache misses in each of the first and the second phase. If k is not really huge, the benefit from such an optimization would remain small.

Optimization. In this section we describe an optimized implementation of `Catena` instantiated with a $\lambda-$BRG, where we transform the explicit unique prefix tuple (c, i) (see Line 4 of Algorithm 1 and lines 3,6,8 of Algorithm 2) into an implicit mode of operation. The prefix tuple (c, i) is really handy when analyzing the security of `Catena`-λ (see Section 6). But, in real world implementations, the tuple (c, i) should match the block size, which would imply one additional compression function invocation. In the following we discuss how to omit this extra invocation of the compression function without violating the unique prefix assumption, which is vital for the proof. Usually, a hash function is implemented using the three methods `Initialize(), Update()`, and `Finalize()` [35]. Thereby, the method

Algorithm 3 Implementation of $\lambda-$BRH

```
Input: $c$ {Garlic}, $x$ {Value to Hash}, $\lambda$ {Depth}
Output: $x$ {Hash Value}

 1: $s \leftarrow$Initialize($H$)
 2: $s \leftarrow s$.Update($c \mid\mid 0^*$)
 3: $s \leftarrow s$.Update($x$)
 4: $t \leftarrow s$.copy()
 5: $v_0 \leftarrow t$.Finalize()
 6: for $i = 1, \ldots, 2^c - 1$ do
 7:     $s \leftarrow s$.Update($v_{i-1}$)
 8:     $t \leftarrow s$.copy()
 9:     $v_i \leftarrow t$.Finalize()
10: end for
10: for $k = 0, \ldots, \lambda - 1$ do
11:     $s \leftarrow s$.Update($v_0 \mid\mid v_{2^c-1}$)
12:     $t \leftarrow s$.copy()
13:     $r_0 \leftarrow t$.Finalize()
14:     for $i = 1, \ldots, 2^c - 1$ do
15:         $s \leftarrow s$.Update($r_{i-1} \mid\mid v_{tau(i)}$)
16:         $t \leftarrow s$.copy()
17:         $r_i \leftarrow t$.Finalize()
18:     end for
19:     $v \leftarrow r$
20: end for
21: return $r_{2^c-1}$
```

`Initialize()` generates the initial chaining value, the method `Update()` processes arbitrary large data updating the internal state, and the `Finalize()` method finalizes the message, e.g., padding, and computes the output hash. In our case, for each iteration of c, we proceed as shown in Algorithm 3. Thus, depending on the used compression function, for each invocation of the hash function, we reduce the effort by one compression function call. Moreover, one can precompute the values from Line 2 to save an additional compression function call. Note that we use the 0^*-padding to stretch the value c to the block size. Finally, the saved computational effort can be reinvested in the usage of higher security parameter.

6 Security of Catena-λ

In this section we provide the security analysis of a $\lambda-$BRG, since it is the basic structure of Catena-λ. Before we present our claims in Section 6.2, we introduce some preliminaries in Section 6.1.

6.1 Memory Hardness and the Pebble Game

Hellman presented in [20] 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) [40]. As the control flow of the $\lambda-$BRH operation can be represented as a DAG, i.e., the nodes of the graph represent the inputs and outputs of the hash function and an edge denotes a hash function invocation, we adapt the model from [40] to analyze our scheme. In the following we introduce this model, which is also called *pebble game*. It has been occasionally used in cryptographic context, see, e.g. [16] for a recent example.

Pebble Game. This 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 [29]), 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

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

Therefore, the interest of T is bounded to the range $N \leq T(S) \leq 2^N$, where $T(S)$ has to increase if S decreases. In general, a pebble game is a common model to derive and analyze TMTs as shown in [48,49,51,53,55].

6.2 Security Analysis

Obviously, a $\lambda-$BRG consists of a sequential structure, since each node within the graph is at least derived from its predecessor. We denote by $G = 2^g$ the number of vertices within one row of a $\lambda-$BRG. Let v_j^i denote the j-th vertex of the i-th row of a $\lambda-$BRG with $i \in \{0, \ldots, \lambda\}$ and $j \in \{0, \ldots, G-1\}$. An $\lambda-$BRG can be pebbled in time $T = (\lambda + 1) \cdot G$ using at least $S = G$ pebbles. Therefore, the G pebbles are placed sequentially on the input vertex $v_0^0, \ldots, v_{G-1}^0$. Then, the pebble placed on v_0^0 is moved to v_0^1 and the $G-1$ remaining pebbles of the first row are sequentially placed on the nodes $v_1^1, \ldots, v_{G-1}^1$. Now, the pebble on the vertex v_0^1 is moved to v_0^2 and the $G-1$ pebbles are sequentially placed on the nodes $v_1^2, \ldots, v_{G-1}^2$. This line of action is repeated for all $i \in \{0, \ldots, \lambda\}$, i.e., until the output of the $\lambda-$BRG is produced.

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

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

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

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

Proof.

Induction Hypothesis ($n = \lambda$):

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

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

$n = 0$**:**

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

$n = 1$**:**

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

Induction Step ($n \rightsquigarrow n+1$):

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

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

□

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

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

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

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

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

Since A' is in possession of $\lambda \cdot S + 1$ pebbles, we can repeat the procedure for all but the last row. Thus, the effort for pebbling λ rows $r^0, \ldots, r^{\lambda-1}$ generating S intervals in each row, and placing one additional pebble (runner pebble) on v^λ_0, is given by

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

Then, moving the runner pebble from v^λ_0 to v^λ_{G-1} in a sequential order requires

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

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

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

□

Conjecture 1 (Lower Bound) *If $(\lambda{+}1) \leq S \leq G/2^\lambda$, then, pebbling the $\lambda-$BRG $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input nodes with S pebbles, takes time*

$$T > \frac{G^{\lambda+1}}{S^\lambda}.$$

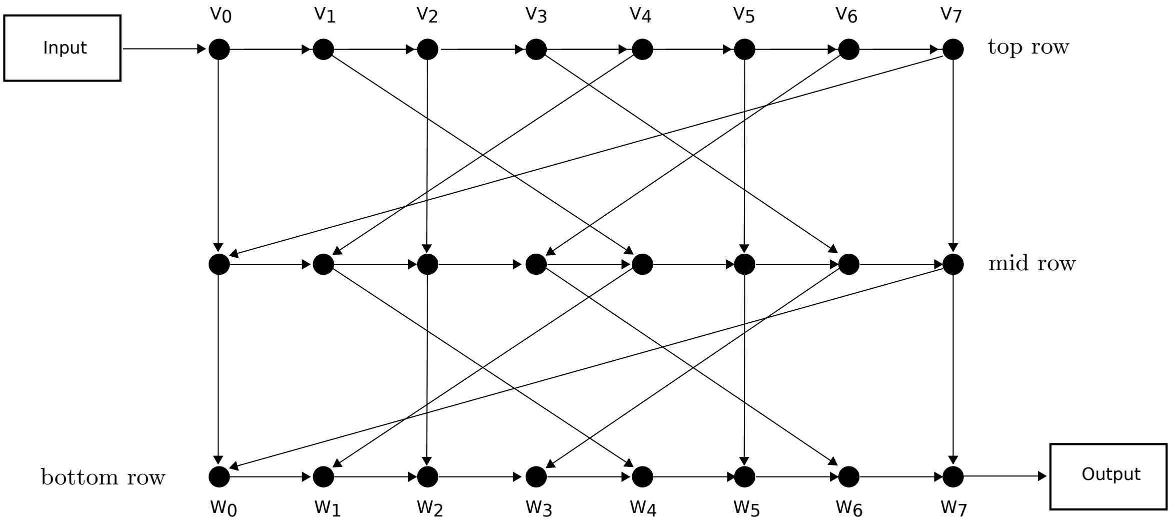

Fig. 2. A $\lambda-$bit-reversal graph with $\lambda = 2$ ($2-$BRG) and $g = 3$.

The case $\lambda = 1$ was already proven by Lengauer and Tarjan in [29]. For the sake of completeness, we show their proof again in Appendix B. Here, we present the proofs for $\lambda \in \{2, 3\}$ (see Lemma 2 and Lemma 3), i.e., for a 2-BRG (see Figure 2) and a 3-BRG (see Figure 3). Nevertheless, we leave the proof for a generic lower bound as an open research problem. Note that the cases for $\lambda \in \{1, 2, 3\}$ are sufficient for the practical instantiation of `Catena`. Even $\lambda = 4$ would be artificial strong for the practice.

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

$$T \geq G^3/128S^2.$$

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

Since we have at most $S-1$ pebbles left to pebble the mid row, at least $2^s - (S-1) \geq 3S$ intervals are pebble-free at z_{j-1}. In the following we denote all vertices in a pebble free interval as *unmarked*, otherwise as *marked*. Note that at most $1/4$ of the nodes are *marked*, whereas the majority $(3/4)$ of the nodes are *unmarked*.

Then, we divide the mid row again in 2^{g-s} intervals J^1_j of length 2^s to re-establish the same configuration as in the bottom row. We denote the interval J^1_j as *bad*, if more than half

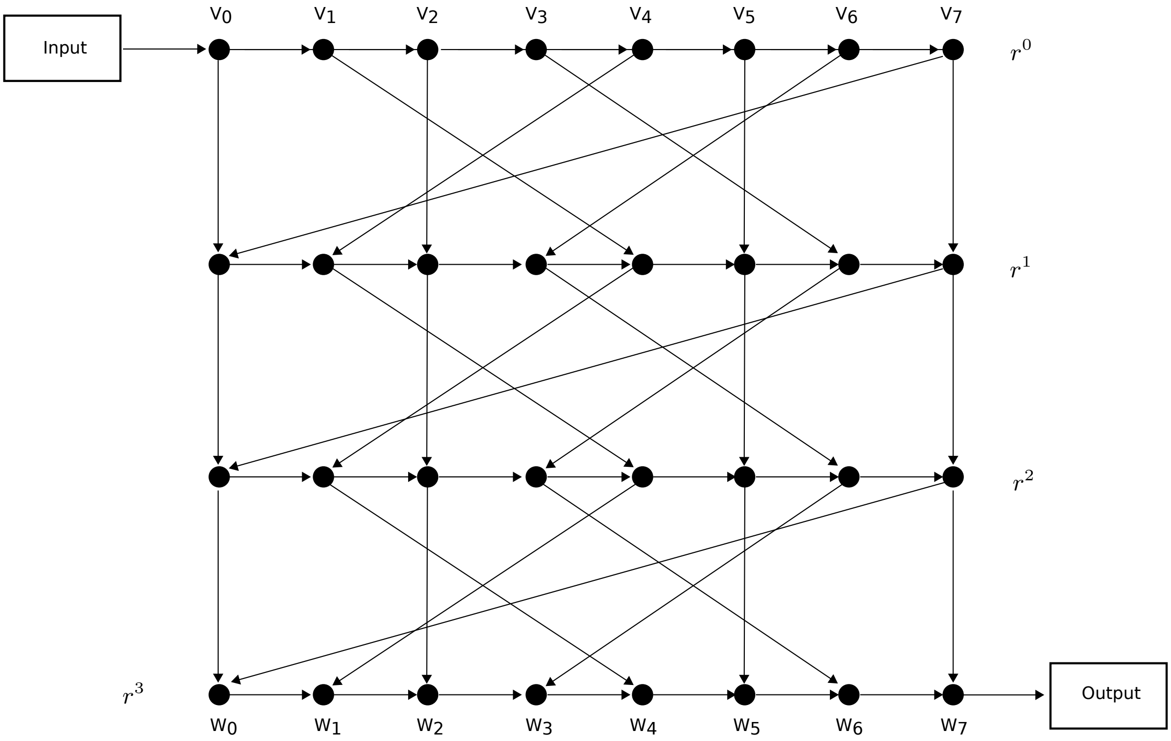

Fig. 3. A $\lambda-$bit-reversal graph with $\lambda = 3$ (3−BRG) and $g = 3$.

of its vertices ($> 2^{s-1}$) are marked (otherwise we denote it as a *good* interval). Since the mid row has at at most $G/4$ marked vertices, at least half of the intervals J_j^1 can be considered as good. Each of those mid row intervals divide the top naturally into 2^s intervals I_i^0 of length 2^{g-s} where each vertex of J_j^1 defines the high end of an interval I_i^0.

We denote a top level interval I_i^0 as good if the vertex of J_j^1 that defines its high end is *marked*. Thus, we have at least 2^{s-1} good intervals in the top row. Therefrom, at most $2^{s-1} - S \geq S$ intervals are pebble free. Thus, the cost for pebbling one mid row interval J_j^1 is at least $2^{g-s} \cdot S \geq G/4$. With this knowledge we can lower bound the cost of pebbling a bottom row interval by $2^{g-s-1} \cdot G/4 \geq G^2/32S$. Thus, the total cost of pebbling a 2−BRG is at least $2^{g-s} \cdot G^2/32S \geq G^3/128S^2$. □

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

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

Proof. Note that if an adversary is not in possession of $2^\sigma + 1$ pebbles, we give it as many pebbles as necessary for free until it has $2^\sigma + 1$ pebbles. Let the Row r^3 be divided into 2^{g-s} intervals of length 2^s, where $2^s \geq 2^{\sigma+3}$ and where the j-th interval I_j^3 consists of the vertices $v_{j2^s}^3, \ldots, v_{(j+1)2^s-1}^3$ with $0 \leq j < 2^{g-s}$. Let z_j be the first time (i.e., the number of the first move) that a pebble is placed on the last vertex of the interval I_j^3 (i.e., $v_{(j+1)2^s-1}^3$).

Let $z_{j-1} = 0$. Then, $z_j > z_{j-1}$ for $0 \leq j < 2^{g-s}$. In order to find a lower bound on $z_j - z_{j-1}$, we observe that at time z_{j-1} the interval I_j^3 is pebble-free and thus, all 2^s vertices in I_j^3 have to be pebbled between z_{j-1} and z_j. By definition of the bit-reversal permutation, the immediate predecessors on the input path of the vertices in I_j^3 divide the Row r^2 naturally into 2^s intervals I_i^2 of length 2^{g-s}, where each vertex of I_j^3 defines the high end of an interval I_i^2 with $0 \leq i < 2^s$.

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

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

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

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

□

Corollary 1. *For pebbling a $\lambda-$bit-reversal graph $\Pi_g^\lambda(\mathcal{V}, \mathcal{E})$ with $G = 2^g$ input nodes and S pebbles with $(\lambda+1) \leq S \leq G/2^\lambda$, it holds that*

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

6.3 Additional Security Features

Preimage Security. In this section we discuss the success probability of an adversary for guessing a valid password, given a salt value s, a hash value h, and an algorithm PS

(password scrambler). Then, the password-recovery advantage (see Definition 4, Section 3) for `Catena`-λ is given by

$$\mathbf{Adv}^{\text{REC}}_{\texttt{Catena-}\lambda}(q) \leq \frac{q}{2^e} + \mathbf{Adv}^{\text{PRE}}_{H}(q), \tag{1}$$

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

Proof (Sketch). We measure the quality of a password population by the min-entropy, $-\log_2(\max\{\Pr[pwd]\})$, the negative base-2 logarithm of the largest probability of any password from that population. Let e denote the min-entropy of passwords generated by `Catena`-λ. Then, an adversary can guess a password by trying out 2^e password candidates. For a maximum of q queries, it holds that the success probability is given by $q/2^e$. Instead of guessing 2^e password candidates, an adversary can also try to find a preimage for a given hash value h. It is easy to see from Algorithm 1 that an adversary thus has to find a preimage for H in Line 4. More detailed, for a given value h with $h \leftarrow H(c \mid\mid (\lambda+1)\cdot 2^c \mid\mid x)$, A has to find a valid value for x. We denote by $\mathbf{Adv}^{\text{PRE}}_{H}(q)$ the advantage of an adversary A to find a preimage for H using at most q queries. Equation 1 is then given by summing up the individual terms. □

Pseudorandomness. For proving the pseudorandomness of `Catena`-λ, we refer to the definition Random-Oracle Security, which was introduced in Section 3 (see Definition 5). Therefore, we model the internally used hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle.

Theorem 2. *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*

$$\mathbf{Adv}^{\$}_{Catena_H}(q) = \left|\Pr[A^{Catena_H} \Rightarrow 1] - \Pr[A^{\$} \Rightarrow 1]\right| \leq \frac{q^2(\lambda+1)}{2^{n-g}}.$$

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

Suppose that y^j denotes the output of λ−BRH(g, x, λ) of the j-th query (Line 3 of Algorithm 1, where F^{λ}_{H} is replaced by the bit-reversal hash operation shown in Algorithm 2). Then, $H(g \mid\mid (\lambda+1)\cdot 2^g \mid\mid y^j)$ is the output of `Catena`-$\lambda(a^j)$. In the case that $y^1, \ldots, y^q$ are pairwise distinct, A can not distinguish $H(g \mid\mid (\lambda+1)\cdot 2^g \mid\mid \cdot)$ from $\$(\cdot)$, since both functions are modeled as random oracles, returning a value chosen uniformly at random from the set $\{0,1\}^n$.

Therefore, we have to upper bound the probability of the event $y^i = y^j$ with $i \neq j$. Due to the assumption that $A's$ queries are pairwise distinct, there must be at least one collision for H, i.e., $z \neq z'$ with $H(z) = H(z')$. More precisely, we need a collision inside a specific domain represented by the counter-index tuple (c, b), i.e., $H(c \mid\mid b \mid\mid x) = H(c \mid\mid b \mid\mid x')$ for $x \neq x'$ (Line 4 of Algorithm 1 and lines 1, 3, 6, and 8 of Algorithm 2) with $c \in \{1, \ldots, g\}$ and $b \in \{0, \ldots, (\lambda+1)\cdot 2^g\}$. We call such a collision a `bad event`.

We can exploit our observations to define a new game `Catena`′-λ which works as follows. Firstly, it calls `Catena`-λ and stores all inputs and outputs of H in a query list $\mathcal{Q}$. Secondly, returns a random value R and finally, let an adversary win iff $\mathcal{Q}$ contains `bad event`. Remark, the advantage of winning the game `Catena`′-λ 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 values $y^1, \ldots, y^q$.

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}
$x \leftarrow \textbf{Catena}(pwd, t', s, g)$
$k \leftarrow \emptyset$
for $i = 1, \ldots, \lceil \ell/|n| \rceil$ do
    $k \leftarrow k \,||\, H(0 \,||\, i \,||\, \mathcal{I} \,||\, \ell \,||\, x)$
end for
return $\textbf{Truncate}(k, \ell)$ {truncate $k$ to the first $\ell$ bits}
```

Now, we upper bound the probability that $\mathcal{Q}$ contains `bad event`. Note that the probability that $\mathcal{Q}$ contains a collision at some position (c, b) is at most $q^2/2^{n+1}$. The probability that $\mathcal{Q}$ contains a bad event at any position (c, b), for the c-th iteration of the for loop (lines 2–5 of Algorithm 1) is at most $q^2 \frac{(\lambda+1)\cdot 2^c}{2^{n+1}}$. Thus, we can upper bound the probability that $\mathcal{Q}$ contains a bad event by

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

Our claim follows from the union bound. □

7 Further Application of the `Catena` Framework

7.1 The Key-Derivation Function `Catena-KG`

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

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

Then, the call of `Catena` is followed by an output transform that takes the output x of `Catena`, a one-byte *key identifier* $\mathcal{I}$, and a parameter ℓ for the key length as the input, and generates key material of the desired output size. `Catena-KG` is even able to handle the generation of extra-long keys (longer than the output size of H), by applying H in Counter Mode [15]. Note that longer keys do not imply improved security, in that context.

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

In contrast to the password hashing scenario, where a user want to log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [57], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Thus, we recommend to use $g = 20$ when Catena is used for key derivation.

Security Analysis. It is easy to see that Catena-KG inherits its $\lambda-$memory hardness from Catena (see Section 6, Corollary 1), since it invokes Catena (Line 1 of Algorithm 4). Next, we show that Catena-KG a good pseudorandom function (PRF) in the random oracle model.

Theorem 3. *In the random oracle model we have*

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

Proof. Suppose that H is modeled as random oracle. For the sake of simplification, we omit the truncation step and let the adversary always get access to the untruncated key k.

Note that all inputs to H from Algorithm 4 (cf. Line 4) contain zero as their first byte. Hence, they never occur as inputs in any invocations in $\mathtt{Catena}_H(pwd, t', s, g)$, because the first byte of the input therein is always a value in the interval $[1, g]$. Suppose x^i denotes the output of Catena of the i-th query. In the case $x^i \neq x^j$ for all values with $1 \leq i < j \leq q$, the output k is always a random value, since H is always invoked with a fresh input (see Line 4, Algorithm 4). The only chance for an adversary to distinguish $\mathtt{Catena\text{-}KG}_H(\cdot)$ from the random function $\$(\cdot)$ is a collision in $\mathtt{Catena}_H$. The probability for this event can be upper bounded by similar arguments as in the proof of Theorem 2. □

7.2 Catena for Proofs of Work

The concept of proofs of work was introduced by Dwork and Naor [12] 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* " [12]. 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. As an advancement to CPU-bound function, Abadi et al. [1], and Dwork et al. [11] 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 [13] a compact representation for this table by using a time-memory trade-off for its generation.

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.

Guessing Secret Bits (Pepper). Let pwd denote a fixed password chosen by S. Furthermore, we denote by s a maybe 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 $h = \mathtt{Catena}(s, pwd)$ and sends the tuple $(pwd, h, s_{[0,k-p-1]}, k)$ 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 $h' = \mathtt{Catena}(s', pwd)$ about 2^k times and comparing if $h = h'$. If so, C gains access to S. The effort of C is given by about 2^k computations of Catena (and about 2^k comparisons for $h = h'$). Hence, the effort of C is scalable by adapting k.

Guessing the Correct Password. In this scenario S chooses an m-bit password pwd and a salt s. Then, S computes $h = \texttt{Catena}(s, pwd)$ and sends h and s to C. The client C then has to guess the password by computing about 2^m times $h' = \texttt{Catena}(s, pwd')$ 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.

8 Conclusion and Outlook

In this paper we have presented `Catena`, a novel password-scrambling framework which is based on a $\lambda-$memory-hard function. Further, it is the first scheme 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 and g the memory consumption. Further, we introduced a provably-secure instantiation of `Catena` using a $\lambda-$bit-reversal graph ($\lambda-$BRG) – called `Catena`-λ. The motivation for the overall design is to thwart GPU-based attacks and to provide cache-time resistance. Thereby, we were inspired by the discovery of cache-timing attacks on `scrypt`. Nevertheless, the application of the `Catena` framework is not restricted to the generation of password hashes. It also fits very well for the usage as a key derivation, 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 instantiation `Catena`-λ) 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.*

9 Acknowledgement

Thanks to Eik List, Colin Percival, and Stephen Touset for their helpful hints and comments, as well as fruitful discussions.

References

1. Martin Abadi, Michael Burrows, and Ted Wobber. Moderately Hard, Memory-Bound Functions. In *NDSS*, 2003.
2. Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In *CHES*, number Generators, pages 309–318, 2001.
3. 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.
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. 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.
6. Daniel J. Bernstein. Cache-timing attacks on AES, 2005.
7. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.

8. 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/`.
9. Stephen A. Cook. An Observation on Time-Storage Trade Off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.
10. Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. `http://www.akkadia.org/drepper/SHA-crypt.txt`. Accessed May 16, 2013.
11. Cynthia Dwork, Andrew Goldberg, and Moni Naor. On Memory-Bound Functions for Fighting Spam. In *CRYPTO*, pages 426–444, 2003.
12. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *CRYPTO*, pages 139–147, 1992.
13. Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and Proofs of Work. In *CRYPTO*, pages 37–54, 2005.
14. Morris Dworkin. Recommendation for Block Cipher Modes of Operation - Methods and Techniques. NIST special publication 800-38a, National Institute of Standards and Technology (NIST), May 2001. See `http://csrc.nist.gov/publications/nistpubs/index.html`.
15. Morris Dworkin. *Special Publication 800-38A: Recommendation for block cipher modes of operation.* National Institute of Standards, U.S. Department of Commerce, December 2001.
16. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In *CRYPTO*, pages 335–353, 2011.
17. Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES*, pages 1–17, 2009.
18. Eric Glass. The NTLM Authentication Protocol and Security Support Provider. `http://davenport.sourceforge.net/ntlm.html`. Accessed May 16, 2013.
19. 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.
20. Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
21. John Hopcroft, Wolfgang Paul, and Leslie Valiant. On Time Versus Space. *J. ACM*, 24(2):332–337, April 1977.
22. B. Kaliski. RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000.
23. Poul-Henning Kamp. The history of md5crypt. `http://phk.freebsd.dk/sagas/md5crypt.html`. Accessed May 16, 2013.
24. Takumi Kasai, Akeo Adachi, and Shigeki Iwata. Classes of pebble games and complete problems. In *ACM Annual Conference (2)*, pages 914–918, 1978.
25. 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.
26. Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family. In *FSE*, pages 244–263, 2012.
27. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996.
28. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.
29. 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.
30. Andrzej Lingas. A PSPACE Complete Problem Related to a Pebble Game. In *ICALP*, pages 300–321, 1978.
31. T. Alexander Lystad. Leaked Password Lists and Dictionaries - The Password Project. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`. Accessed May 16, 2013.
32. Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996.
33. Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.
34. Robert Morris and Ken Thompson. Password Security - A Case History. *Commun. ACM*, 22(11):594–597, 1979.
35. National Institute of Standards and Technology. ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions, 2008.

36. Krishna Neelamraju. Facebook Pages: Usage Patterns | Recommend.ly. `http://blog.recommend.ly/facebook-pages-usage-patterns/`. Accessed May 16, 2013.
37. Nvidia. Nvidia GeForce GTX 680 - Technology Overview, 2012.
38. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology-CRYPTO 2003*, 3:617–630, 2003.
39. NIST National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. April 1995. See http://csrc.nist.gov.
40. 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.
41. Wolfgang J. Paul and Robert Endre Tarjan. Time-Space Trade-Offs in a Pebble Game. In *ICALP*, pages 365–369, 1977.
42. Colin Percival. Cache Missing for Fun and Profit. BDSCan 2004.
43. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
44. Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
45. A.G. Reinhold. HEKS: A family of key stretching algorithms, 1999.
46. Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992.
47. Semiocast SAS. Brazil becomes 2nd country on Twitter, Japan 3rd Netherlands most active country. `http://goo.gl/QOeaB`. Accessed May 16, 2013.
48. J. Savage and S. Swamy. Space-time trade-offs on the FFT algorithm. *Information Theory, IEEE Transactions on*, 24(5):563 – 568, sep 1978.
49. John E. Savage and Sowmitri Swamy. Space-Time Tradeoffs for Oblivious Interger Multiplications. In *ICALP*, pages 498–504, 1979.
50. Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *FSE*, pages 191–204, 1993.
51. Ravi Sethi. Complete Register Allocation Problems. *SIAM J. Comput.*, 4(3):226–248, 1975.
52. Jens Steube. oclHashcat-plus - Advanced Password Recovery. `http://hashcat.net/oclhashcat-plus/`. Accessed May 16, 2013.
53. Sowmitri Swamy and John E. Savage. Space-Time Tradeoffs for Linear Recursion. In *POPL*, pages 135–142, 1979.
54. Stefan Tillich, Christoph Herbst, and Stefan Mangard. Protecting AES Software Implementations on 32-Bit Processors Against Power Analysis. In *ACNS*, pages 141–157, 2007.
55. Martin Tompa. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. In *STOC*, pages 196–204, 1978.
56. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.
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` [22] 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 smaller output sizes, [43] 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

Algorithm 5 The `scrypt` algorithm and its core operation `ROMix` [43].

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

second phase (lines 24–27), `ROMix` updates x depending on v_j. The sequential memory hardness comes 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 is risky 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 $x \leftarrow \mathrm{PBKDF2}(x, s, 1, 1)$.

A.1 Brief Analysis of `ROMix`

In the following we introduce a way to run `ROMix` with less than G units of storage. 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}, …, $v_{(S-1)K}$ – 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, one can easily see 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 [43]). 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 $\{0, \dots, G-1\}$. With no way to anticipate the next j, the best approach 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 so 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 $\texttt{PBKDF2}(pwd, s, 1, 1)$, 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 $x \leftarrow H(x \oplus v_j)$. Including the work 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 Discussion

At the current point of time, our cache-timing attacks are theoretical. Even if one manages to run some spy process on a machine using `scrypt`, the requirement to interrupt `ROMix` twice at the right points of time is demanding. Nevertheless, even the theoretical ability of mounting such attacks should be seriously taken into account.

The idea of attacking cryptographic algorithms from hardware side (side-channel attacks) is not new [27], neither is the usage of a spy process for theoretical cache-timing attacks [42]. In [6], Bernstein demonstrated practically how to recover AES keys by using cache-timing information: *"The problem lies in AES itself: it is extremely difficult to write constant-time high-speed AES software [...]. Constant time low-speed AES software is fairly easy to write but is also unacceptable for many applications."* Meanwhile, this claim can be denoted as obsolete, since Käsper and Schwabe have shown in [17], that it is possible to write a fast and constant-time AES-GCM implementation, which is resistant against timing-attacks. Moreover, the AES New Instructions (AES-NI) can be a helpful tool to write constant-time implementations.

Nevertheless, we argue that there is a problem in `scrypt` itself. One can certainly implement `scrypt` such that cache-timings leak no information about the password. But, we believe this would drastically reduce the performance of `scrypt`. As a compensation – recall that password scramblers are intentionally slow, but must be "fast enough" for the user – one would have to set the cost parameter G to some smallish value, but this would only make regular attacks more efficient, since adversaries can use faster implementations. At the end of the day, this may defeat the entire point of using `scrypt` at all.

The core of the problem is the fact that `ROMix` reads a value v_j, where the index $j \leftarrow x \bmod G$ depends on x and thus, on the password. It would be very compelling to have a password scrambler which is at least memory-hard *and* computes j in some password-independent way, i.e., only depending on the loop index i. In this paper we actually presented such a password scrambler, `Catena` or resp. `Catena`-λ, which uses a variation of the bit-reversal hash operation to compute j from i.

B Lower Bound for Pebbling a BRG

Lemma 4 ([29]). *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*

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

Proof. The proof is trivial for $S > G/4$. Thus, assume that $S \leq G/4$. Choose the integer s such that $2S \leq 2^s < 4S$. Let the output path be divided into 2^{g-s} intervals of length 2^s. The j-th interval I_j $(0 \leq j < 2^{g-s})$ consists of the vertices $\tau_{j2^s}, \ldots, \tau_{(j+1)2^s-1}$. Let z_j be the first time (i.e., the number of the first move) that a pebble is placed on $\tau_{(j+1)2^s-1}$, that is, on the highest vertex in I_j. Let $z_{j-1} = 0$. Then, $z_j > z_{j-1}$ for $0 \leq j < 2^{g-s}$. In order to find a lower bound on $z_j - z_{j-1}$, we observe that at time z_{j-1} the interval I_j is pebble-free and thus, all 2^s vertices in I_j have to be pebbled between z_{j-1} and z_j. By definition of the bit-reversal permutation, the immediate predecessors on the input path of the vertices in I_j divide the input path naturally into $2^s - 1$ intervals of length 2^{g-s}. (The immediate predecessor of a vertex in I_j defines the high end of an interval. The intervals at the ends of the input path are disregarded.) At time z_{j-1} at most $S - 1$ pebbles are on the input path. Thus, at least $2^s - 1 - (S - 1) \geq S$ intervals are pebble-free at z_{j-1}. All of them have to be pebbles completely before z_j. This takes at least $S \cdot 2^{g-s} > G/4$ placements. Therefore, $z_j - z_{j-1} > G/4$ for $0 \leq j < 2^{g-s}$, and thus, before time z_{2^g-1} at least $2^{g-s}G/4 > G^2/16S$ placements have to occur.

□