Catena: A Memory-Consuming Password Scrambler

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 better store the one-way hash of their clients' passwords, rather than storing the password in the clear. This paper introduces Catena, a new one-way function for that purpose. Catena is memory-hard, which can hinder massively parallel attacks on cheap memory-constrained hardware, such as recent "graphical processing units", GPUs. Furthermore, Catena has been designed to resist cache-timing attacks. This distinguishes Catena from scrypt, which may be sequentially memory-hard, but which we show to be vulnerable to cache-timing attacks.
Additionally, 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 are user-memorizable secrets, commonly used for user authentication and cryptographic key derivation.[1] 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., [1] 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 take some time for both kinds of users legitimate ones and attackers. 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).

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

(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. 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 [23], 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` [28]. To the best of our knowledge, it is also the only one – up to now.

A memory-consuming password scrambler may suffer from a new problem, though. 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 first show that this is actually an issue for `scrypt`, and then present a fix by introducing `Catena`, a new password scrambler which consumes lots of memory (like `scrypt`), but does not have a password-defined memory-access pattern. We formally analyze the security of `Catena` and its memory consumption using the "pebble game" approach, that dates back to the early days of Theoretical Computer Science [5,11,26,17,13].

Background. As observed by Wilkes in the late 1960s [40], storing plain authentication passwords is insecure. About 10 years later, the UNIX system integrated some of Wilkes ideas [21] 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 [18].

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 as 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 defends against attacks where password hashes from many different users are known to the attacker, e.g., against the usage of rainbow tables [24].

There are different ways to perform key stretching. One is to keep p bits of the salt secret, turning them into *pepper* [19]. Both attackers 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. A better approach would be to start at a random value and wrap around at 2^p. Kelsey et al. [14] analyzed another key stretching approach where a cryptographic operation is iterated n times. Boyen proposed in [4] a user-defined implicit choice for n by iterating until the user presses a "halt" button.

According to Moore's Law [20], 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 amount of user accounts is inactive or rarely used, e.g., 70.1% of all Facebook accounts experience zero updates per month [22] and 73% of all Twitter accounts do not have at least one tweet per month [31]. 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 or otherwise having to know the password. We call this feature a *client-independent update* of the password hash.

When using pepper for key-stretching, client-independent updates are straightforward. When the 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.

Shifting the Burden. 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 H and (2) an efficient one-

way function h. By default, the server computes the password hash $h(H(pwd \mid\mid s))$ from the password pwd and a salt s. Alternatively, the server sends s to the client who responds $x = H(pwd \mid\mid s)$. Finally, the server just computes $h(x)$. We denote this as *server relief*. 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.

Key Derivation. Beyond authentication, passwords are also used to derive symmetric keys. Obviously, one can just use the ouput 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 or has to derive more than one key. Thus, it is prudent to consider a *key derivation function (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.

Contribution. Our primary contribution is a novel password scrambler called `Catena` (Latin for "chain", due to its sequential structure). In the random oracle model, we prove `Catena` to be memory-hard (see Section 2.2, Definition 1). `Catena` thus thwarts back massively-parallelized attacks using GPUs and similar hardware. Furthermore, `Catena` has been designed to be resistant against cache-timing attacks.

The need to address this issue has been inspired by the cache-timing attacks we present in Section 3 against `scrypt` [28]. To the best of our knowledge, such attacks for `scrypt` have not been known before.

We enhance `Catena` by integrating several useful and novel features for password scramblers. `Catena` supports client-independent updates by increasing the *garlic* or by turning *salt* bits into *pepper*. The notion of *garlic* reflects the property that an incrimination of this parameter by '1' doubles the memory usage and at least doubles the computational time. Furthermore, `Catena` has built-in support for server relief.

Finally, we extend `Catena` towards a password-based key derivation function `Catena-KG`.

Outline. Section 2 briefly overviews modern password scramblers and their properties. Section 3 describes `scrypt` and cache-timing attacks against it. Section 4 introduces `Catena`, a new memory-demanding password scrambler. Section 5 analyses and formally proves the main security parameters of `Catena`. Section 6 presents `Catena-KG`, an adaption of `Catena` especially designed to derive a secret key from a password. Finally, Section 7 concludes the paper.

2 Practical Password Scramblers and their Properties

Table 2.1 provides an overview of password scramblers that are or have been in frequent use, compared with `Catena`. It indicates whether the password scrambler supports *salt*, *server relief*, and *client-independent updates*. Furthermore, the table

lists all possible values of cost factor (security parameter) including the default values, the memory usage, and issues from which the considered password scrambler may suffer from.

2.1 Frequently used Password Scramblers

Hash Function Based Password Scramblers. Not long ago, `md5crypt` has been used in nearly all Free-BSD and Linux-based systems to scramble user passwords. It is based on the well-known MD5 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 [36]. Additionally, its own author does not consider `md5crypt` secure any more [12]. Common Linux distributions nowadays employ `sha512crypt`, e.g., Debian, Ubuntu, or Fedora. It provides similar features as `md5crypt`, but uses SHA-512 instead of MD5. Furthermore, the number of iterations can be chosen by the user. `NTLMv1` [9] 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 [36]. 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) [39]. 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, where c is a user-chosen value which is given by default with $c = 1,000$.

`bcrypt`. The `bcrypt` algorithm is built upon the Blowfish block cipher [34]. 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` has not been designed with the intention to thwart parallelized attackers 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 a AMD HD 7970 graphic card [36]. 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`. (There was HEKS [30], but it has been broken by the author of `scrypt` [28].) As its core, `scrypt` uses the sequentially memory-hard function `ROMix`, which can take G units of memory and performs $2G$ operations. With only G/K units of memory, the number of operations

Algorithm	Cost Factor (default)	Memory	Server Relief	Client-Indep. Updates	Issues
`crypt` [21]	25	small	-	-	"too fast"
`md5crypt` [12]	1,000	small	-	-	"too fast"
`sha512crypt` [6]	1,000–999,999 (5,000)	small	-	-	(small memory)
`NTLMv1` [9]	1	small	-	-	"too fast"
`PBKDF2` [39]	1–∞ (1,000)	small	-	-	(small memory)
`bcrypt` [29]	2^4–2^{99} (2^6, 2^8)	4,168 bytes	-	-	(constant memory)
`scrypt` [28]	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.

goes up to $2G \cdot K$. In [28] Percival recommends $G = 2^{14}$ and $G = 2^{20}$ for password hashing and key derivation, respectively. We will describe and analyze `scrypt` and `ROMix` in the next chapter.

2.2 Memory-Related Properties

Memory-Hardness. To describe memory requirements, we adopt and slightly changed the notion from [28]. The intuition is that for any parallelized attack, using c cores, the required memory per core is decreased by a factor of $1/c$, 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 $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\alpha})$.

Thus, for $S \cdot T = N^2$, using c cores, we have

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

Client-Independent Update. Modifying a password scrambler to allow client-independent updates to increased security parameters (i.e., cost factors) may sometimes be quite straightforward: turn some of the *salt* bits into *pepper*, or take the current password hash and apply some additional iterations. If we have *garlic*, i.e., if the required memory is a adjustable security parameter, this is a little bit tricky. When the garlic factor is increased from g to, say, $g + 1$, our password scrambler follows the following approach: suppose $H_g(x)$ denotes an invocation of the main function which uses 2^g units of storage (and time), and h denotes the invocation of a memory- and time-efficient one-way hash function. Then we implement the password scrambler `Catena` as

$$\texttt{Catena}(pwd, s) = h(H_{g+1}(h(H_g(h(\ldots h(H_1(pwd, s)))))))),$$

where pwd and s denote password and salt, respectively. This sequential structure does not only enable `Catena` to provide client-independent updates, but also support server-relief by computing all steps except the last call of h on client side.

Cache-Timing Attacks. Consider the implementation of a password scrambler, where data are 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 the adversary could use this 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.

3 The `scrypt` Password Scrambler

Algorithm 1 describes the `scrypt` password scrambler and its core operation `ROMix`. For pre- and post-processing, `scrypt` invokes the one-way function `PBKDF2` 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, often 512 bit). To support hash functions with smaller output sizes, [28] proposes to instantiate H by a function called BlockMix, which we will not elaborate on. For our security analysis of `ROMix`, we modelled 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, namely the array variables $v_0, v_1 \ldots, v_{G-1}$ are set to $x, H(x), \ldots H(\ldots(H(x)))$, respectively. In the second phase (lines 24–27), `ROMix` updates x depending on v_j. The sequential memory hardness 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,

Algorithm 1 The `scrypt` algorithm and its core operation `ROMix` [28].

`scrypt`	ROMix
Input:	**Input:** x {Initial State} , G {Cost Parameter}
pwd {Password}	**Output:** x {Hash value}
s {Salt}	20: **for** $i = 0, \ldots, G-1$ **do**
G {Cost Parameter}	21: $v_i \leftarrow x$
Output: x {Password Hash}	22: $x \leftarrow H(x)$
10: $x \leftarrow \text{PBKDF2}(pwd, s, 1, 1)$	23: **end for**
11: $x \leftarrow \text{ROMix}(x, G)$	24: **for** $i = 0, \ldots, G-1$ **do**
12: $x \leftarrow \text{PBKDF2}(pwd, x, 1, 1)$	25: $j \leftarrow x \bmod G$
13: **return** x	26: $x \leftarrow H(x \oplus v_j)$
	27: **end for**
	28: **return** x

Algorithm 2 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

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 1 by $x \leftarrow \text{PBKDF2}(x, s, 1, 1)$.

3.1 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` (cf. Algorithm 2) 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. That means that the second phase of `ROMixMC` is about $\Theta(K)$ times slower than the second phase of `ROMix`, and this dominates the workload for `ROMixMC`.

Next, we briefly discuss why `ROMix` is sequentially memory-hard (for the full proof see [28]). The intuition is as follows. The indices j are determined by the output of the random oracle H and thus, essentially, uniformly distributed random values over $\{0, \ldots, G-1\}$. With no way to anticipate the next j, the 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.

3.2 Cache-Timing Attacks

Algorithm 1 (`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 2nd phase (Line 24), 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 have 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.

The Preliminary Cache-Timing Attack. Let x be the output of `PBKDF2`(p',*salt*,1,1), where p' denotes the current password candidate. 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 1).
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 p' in a list.
4. Else conclude that p' is a wrong password.

This sieve can run in parallel on any number of cores, where each core tests another password candidate p'. 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.

The 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 the 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 2, 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.

3.3 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 taken into account seriously.

The idea of attacking cryptographic algorithms from hardware side (side-channel attacks) is not new [15], neither is the usage of a spy process for theoretical cache-timing attacks [27]. In [2] 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 [8], 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.

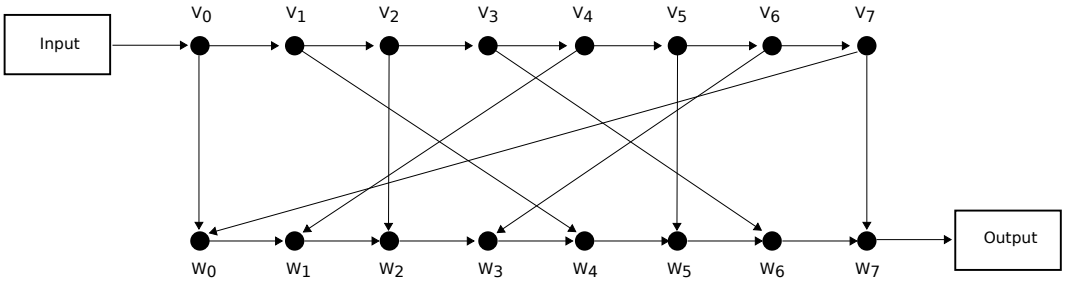

Fig. 1. A sequential bit-reversal graph (SBRG) with $g = 3$ (eight input and eight output nodes).

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 attackers 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 the next section we actually present such a password scrambler, `Catena`, which uses a minor variation of the bit-reversal function to compute j from i – a very simple function, indeed. Note that `Catena`'s proof of memory hardness is very different from the proof in [28] for `ROMix`.

4 The `Catena` Password Scrambler

In this section we introduce `Catena`, our new and sustainable password scrambling algorithm with high resilience against cache-timing attacks.

4.1 Preliminaries

The core of `Catena` is the *Sequential Bit-Reversal Hashing* (SBRH) operation. The origin of this operation is the sequential bit-reversal graph which is then again deduced form the bit-reversal permutation which is defined next.

Definition 2 (Bit-Reversal Permutation τ). *Fix a natural number g and represent $i \in \mathbb{Z}_{2^g}$ as a binary g-bit number, $(i_0, i_1, \ldots, i_{g-1})$, i.e., $i = \sum_{j=0}^{g-1} 2^j i_j$. The bit-reversal permutation $\tau : \mathbb{Z}_{2^g} \to \mathbb{Z}_{2^g}$ is defined by*

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

Algorithm 3 Catena

Input: pwd {Password}, t {Tweak} s {Salt}, g {Garlic} $\geq g_0$ {Initial Garlic Value}, H {Hash Function}
Output: x {hash of the password}
1: $x \leftarrow t \mid\mid pwd \mid\mid s$
2: **for** $c = g_0, \ldots, g$ **do**
3: $\quad x \leftarrow \mathbf{SBRH}(c, x, H)$
4: $\quad x \leftarrow H(c \mid\mid 2^{c+1} \mid\mid x)$
5: **end for**
6: **return** x

Algorithm 4 SBRH (Sequential Bit-Reversal Hashing)

Input: c {Garlic}, x {Value to Hash}, H {Hash Function}
Output: x {Hash Value}
1: $v_0 \leftarrow H(c \mid\mid 0 \mid\mid x)$
2: **for** $i = 1, \ldots, 2^c - 1$ **do**
3: $\quad v_i \leftarrow H(c \mid\mid i \mid\mid v_{i-1})$
4: **end for**
5: $x \leftarrow H(c \mid\mid 2^c \mid\mid v_0 \mid\mid v_{2^c-1})$
6: **for** $i = 1, \ldots, 2^c - 1$ **do**
7: $\quad j \leftarrow \tau(i)$ {set j to the reverse bit order of i}
8: $\quad x \leftarrow H(c \mid\mid 2^c + i \mid\mid x \mid\mid v_j)$
9: **end for**
10: **return** x

With the help of the bit-reversal permutation τ, we introduce the Sequential Bit-Reversal Graph (SBRG), which is a Directed Acyclic Graph (DAG) and almost identical to the Bit-Reversal Graph (BRG) presented by Lengauer and Tarjan in [16].

Definition 3 (BRG and SBRG). *Fix a natural number g. The BRG has 2^{g+1} vertices, namely 2^g input vertices $v_0, \ldots v_{2^g-1}$ and 2^g output vertices $w_0, \ldots w_{2^g-1}$, and the following edges:*

- *$2^g - 1$ edges from v_{i-1} to v_i for $i \in \{1, \ldots, 2^g - 1\}$,*
- *$2^g - 1$ edges from w_{i-1} to w_i for $i \in \{1, \ldots, 2^g - 1\}$,*
- *2^g edges from v_i to $w_{\tau(i)}$ for $i \in \{0, \ldots, 2^g - 1\}$, where τ is the bit-reversal permutation.*

The SBRG is the BRG with one additional edge from v_{2^g-1} to w_0.

The edges of the SBRG define the information flow of the "Sequential Bit-Reversal Hashing" operation (cf. Algorithm 4). An illustration of the sequential bit-reversal graph for $g = 3$ is given in Figure 1.

4.2 Catena

Algorithm 3 describes the Catena password scrambler and Algorithm 4 its core operation SBRH. Both algorithms employ a cryptographic hash function H. In both algorithms, the password-dependent input of H is appended to a prefix tuple (c,i), where c denotes the iteration counter or the actual *garlic* factor and i denotes the hash function invocation counter. After each iteration (lines 2–5 of Algorithm 3) the invocation counter is reset to zero (cf. Line 1 of Algorithm 4). This position encoding guarantees the uniqueness of inputs for H within a run of Catena. This becomes handy in the security analysis of Catena. We recommend to encode the values c and i as 8-bit values and 32-bit values, respectively.

For our security analysis, we will model H as a random oracle. For practical purposes, we recommend Keccak-512 [3] – the winner of the SHA-3 competition[2]. The 512-bit output length suits Catena quite well, 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.

When the output size of H is equal to the size of a cache line (or a multiple) each time a value is written to (or read from) a location v_i, the time to access v_i is the same.

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 just cannot gain much from such an optimization. If the values are stored in their natural order: v_0, v_1, ..., v_{2^g-1}, then, the adversary drastically reduces its cache misses in the first phase (lines 2–5 of Algorithm 4) to $2^g/k$ cache misses, but in the second phase (lines 6–8 of Algorithm 4) , the number of cache misses is 2^g. If the 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 phase 1 and phase 2. If k is not really huge, the benefit from such an optimization would remain small.

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

$$t \leftarrow \texttt{0x00} \mid\mid n \mid\mid |s| \mid\mid H(AD),$$

where the first byte denotes the mode (0x00 for password hashing and 0x01 when Catena is used as a key derivation function). The second 16-bit value n denotes the output length of the underlying hash function H in bits, and the third 16-bit value $|s|$ denotes the total length of the salt in bits. The last 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

[2] http://csrc.nist.gov/groups/ST/hash/sha-3/winner_sha-3.html

the password hashes. The tweak is processed together with the salt and the secret password (see Line 1 of Algorithm 3). 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.

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 5 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, e.g., we tested that $g = 16$ and $g = 17$ are suitable values on a Lenovo Thinkpad T430 (Intel Core i5-3210M) for `Catena`-Keccak-512. 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. For the initial deployment of `Catena`, we recommend to set the initial garlic value g_0 to g to achieve the best ratio between running time and memory usage.

Server Relief. In the last iteration of the **for**-loop in Algorithm 3, the client has to omit the last invocation of the hash function H (see Line 4) and transmit the output of `Catena` to the server. Then, 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 side, exonerating 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.

5 Security Analysis of `Catena`

We denote a password scrambling algorithm to be secure if it provides at least memory-hardness and preimage security. Furthermore, it should be resistant against cache-timing attacks. In the following we show that the memory-hardness is inherited from the underlying SBRG function, and the preimage security is guaranteed by the usage of the cryptographic hash function H. Since the memory access pattern of `Catena` is static and therefore, independent from the password, it provides resistance against cache-timing attacks. Finally, we show that `Catena` behaves like a good random function, which is useful for enhancing `Catena` for secure key derivation.

5.1 Memory-Hardness

Hellman presented in [10] 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 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 time-memory trade-off. In 1970, Hewitt and Paterson introduced a method for analyzing time-memory trade-offs on directed acyclic graphs (DAG, see Definition 4) [25]. As the workflow of our password scrambling scheme `Catena` 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 [25] 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. [7] for a recent example.

Definition 4 (Directed Acyclic Graph (DAG)). *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 $v_i \leq v_j$.*

Pebble Game. This model is restricted to DAGs with bounded in-degree and can be seen as a one player game. Let $\Pi(\mathcal{V}, \mathcal{E})$ be a DAG with bounded in-degree and let G be the number of nodes within $\Pi(\mathcal{V}, \mathcal{E})$. For our scheme, we restrict the in-degree to g, where $g = log_2(G)$. In the setup phase of the game, the player gets S pebbles (tokens) with $S \leq G$. 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 pebbled, 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 pebble all nodes of a graph $\Pi(\mathcal{V}, \mathcal{E})$ at least once. The time-memory trade-off 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 [16]), 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 G can be pebbled with G pebbles in time G (in topological order). On the other hand, if a graph $\Pi(\mathcal{V}, \mathcal{E})$ of size G can be pebbled with S pebbles at all, it can be pebbled with S pebbles in time

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

Therefore, the interest of T is bounded to the range $G \leq T(S) \leq 2^G$, where $T(S)$ has to increase if S decreases. In general, a pebble game is a common model to derive and analyze time-memory trade-offs as shown in [32,33,35,37,38].

Sequential Bit-Reversal Graph. Obviously, an SBRG consists of a sequential structure, since each node within the graph is at least derived from its predecessor. An SBRG can be pebbled in time $T = 2G$ using at least $S = G$ pebbles. Therefore, the first G pebbles are placed sequentially on the input vertex $v_0, \ldots, v_{G-1}$ (see Figure 1, where $G = 8$). Then, moving the pebble placed on v_0 to w_0, and finally moving the pebble sequentially from w_0 to w_{G-1}. As the memory requirement for $S = G$ pebbles is usually too large, we also consider scenarios with $S < G$ pebbles. As the smallest number of pebbles an SBRG can be pebbled with is $S = 2$, we consider this case first. An SBRG can be pebbled with only two pebbles in time $T \geq G^2/2$. Therefore, we denote one of the pebbles as input pebble y, and the other one as output pebble y. We start by placing x on v_0 and using y to pebble $v_1, \ldots, v_{G-1}$ in sequential order, and then move y from v_{G-1} to w_0. Lets denote p as the current position of x, starting with $p = 0$. Now, we have to repeat $G-1$ time the following step to pebble the SBRG: Use x to pebble $v_{\tau(p)}$, move y from w_p to w_{p+1}, and increase p by one. Note that in average we have to move x about $G/2$ times to reach $v_{\tau(p)}$. Hence, we need at least $G^2/2$ pebble movements to pebble the SBRG. Now, we analyze the non-trivial case of $3 \leq S \leq G-1$.

Theorem 1. *For pebbling a sequential bit reversal graph $\Pi_g(\mathcal{V}, \mathcal{E})$ with $G = 2^g$ input nodes and S pebbles with $3 \leq S \leq G-1$ we have*

$$ST = O(G^2).$$

The proof follows from Lemma 1 and 2 (see Appendix A).

Lemma 1. *The sequential bit reversal graph $\Pi_g(\mathcal{V}, \mathcal{E})$ can be pebbled with $3 \leq S \leq G-1$ pebbles in time*

$$T \leq \left\lfloor \frac{(S-3)G}{(S-2)} \right\rfloor + 1 + \frac{G^2}{2(S-2)} + G + \frac{G}{(S-2)} \leq \frac{G^2}{2(S-2)} + 3G.$$

Proof. We divide the input nodes $(v_0, \ldots, v_{G-1})$ into $(S-2)$ disjunct intervals $I_0, \ldots I_{S-3}$ of size $\lfloor G/(S-2) \rfloor$ with $I_i = [v_x, v_y]$ with $x = \lfloor iG/(S-2) \rfloor$ and $y = \lfloor (i+1)G/(S-2) \rfloor - 1$. Then, we place a pebble on the first element (v_x) of each interval. This can be done in time $\lfloor (S-3)G/(S-2) \rfloor + 1$. We denote the set of all points v_x as $\mathcal{U}$. Note that since we use $(S-2)$ pebbles to allocate the first points of each interval, we have two pebbles left. We denote one of them as *runner* pebble and the other one as *jumper* pebble. Lets denote v_z with $z = \frac{(S-3)G}{(S-2)}$ the first element within the last interval I_{S-3}. We set the runner pebble to the element v_{z+1}. Now, we can move the runner pebble in a sequential order from v_{z+1} to v_{G-1} using

at most $G/(S-2)$ steps. As the element v_0 is already pebbled, we can move the runner pebble from v_{G-1} to w_0. After placing the runner pebble on w_0, we set $p = 1$ and repeat the following (final) step $G-1$ times: Let $v_{\tau(p)}$ be an element of I_i. If $v_{\tau(p)} \notin \mathcal{U}$, the node $v_{\tau(p)}$ is pebbled using the jumper pebble starting at $v_{\lfloor iG/(S-2) \rfloor}$. If $v_{\tau(p)} \in \mathcal{U}$, the jumper pebble does not have to be moved and the corresponding output node can be pebbled using the pebble which was already placed on $v_{\tau(p)}$ in the first step, i.e., when creating the intervals. Then move the runner pebble to w_p and increase p by one.

Note that we have to move the runner pebble $G + G/(S-2)$ times, and the jumper pebble at least $\frac{G^2}{2(S-2)}$ times. By adding up the summands of the individual steps, we proof our claim. □

Note: space and time to compute the SBRH operation (cf. Algorithm 4) are *similar* to pebble the SBRG, since the SBRH operation computes the last output vertex of an SBRG using a hash function H.

Corollary 1. *Assuming H to be an atomic function, `Catena` is a memory-hard function.*

5.2 Pseudorandomness

Obviously, one can break `Catena` by trying out all possible (or likely) password-candidates. In the following we show that there is essentially no faster way to attack `Catena`, even when it comes to distinguishing `Catena` from a random function. We measure the quality of a password population by the min-entropy, $\log_2(\max\{\Pr[pwd]\})$, the base-2 logarithm of the largest probability of any password from that population. As above, we model the hash function $H : \{0,1\}^* \to \{0,1\}^n$ as a random oracle. In the following we show that there exists no adversary which can distinguish $\texttt{Catena}_H(\cdot)$ from a random function $\$(\cdot)$ significantly better than by trying out password candidates in likelihood order.

Theorem 2. *Let m denote the min-entropy of passwords, q the number of queries made by the adversary, h the given hash value of a password, and model H as a random oracle. Then we have*

$$\mathbf{Adv}^{\$}_{\mathtt{Catena}_H}(A) = \left|\Pr[A^{\mathtt{Catena}_H,h} \Rightarrow 1] - \Pr[A^{\$,h} \Rightarrow 1]\right| \leq \frac{q}{2^m} + \frac{q^2}{2^{n-g-1}}.$$

Proof. It is easy to see that the success probability for trying out likely password candidates can be upper bounded by $\frac{q}{2^m}$. Now suppose that $a^i = (p^i||s^i||t^i)$ represents the i-th query, where p^i denotes the password, s^i denotes the salt, and t^i the tweak of the i-th query. 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 $\texttt{SBRH}(g, x, H)$ of the j-th query (Line 3 of Algorithm 3). Then, $H(g \;||\; 2^{g+1} \;||\; y^j)$ is the output of $\texttt{Catena}_H(a^j)$. In the

case that $y^1, \ldots, y^q$ are pairwise distinct, A can not distinguish $\texttt{Catena}_H(.)$ from $\$(.)$, 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 $x \neq x'$ (lines 3,5, and 8 of Algorithm 4 and Line 3 of Algorithm 3). We call such a collision a `bad event`.

We can exploit our observations to define a new game $\texttt{Catena}'_H$ which works as follows. Firstly, it calls $\texttt{Catena}_H$ 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 $\texttt{Catena}'_H$ is higher than the advantage to distinguish $\texttt{Catena}_H(.)$ from $\$(.)$, since there are `bad events` that most likely lead to a list of unique values $y^1, \ldots, y^q$, e.g., a collision for w_{2^g-2} without another collision in v_{2^g-1} will only cause a collision for w_{2^g-1} with a probability of $1/2^n$ (cf. Figure 1).

Below 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 3) is at most $q^2 \frac{2^{c+1}}{2^{n+1}}$. Thus, we can upper bound the probability that $\mathcal{Q}$ contains a bad event by

$$\sum_{c=g_0}^{g} q^2 \frac{2^{c+1}}{2^{n+1}} < \sum_{c=0}^{g} q^2 \frac{2^{c+1}}{2^{n+1}} < \frac{q^2}{2^{n-g-1}}.$$

Our claim follows from the union bound. □

6 The Catena-KG Key Derivation Function

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

Algorithm 5 Catena-KG

Input: pwd {Password}, t' {Tweak}, s {Salt}, g {Garlic} $\geq g_0$ {Initial Garlic Value}, ℓ {Length of the Output Key}, $\mathcal{I}$ {Key Identifier}
Output: k {ℓ-bit key derived from the password}
1: $x \leftarrow \textbf{Catena}(t', p, s, g, g_0)$
2: $k \leftarrow \emptyset$
3: **for** $i = 1, \ldots, \lceil \ell / |n| \rceil - 1$ **do**
4: $\quad k \leftarrow k \mid\mid H(0 \mid\mid i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$
5: **end for**
6: **return** $\textbf{Truncate}(k, \ell)$ {truncate k to the first ℓ bits}

natural output size of Catena), see Algorithm 5. The core idea is to apply Catena using a slightly different tweak:

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

followed by an output transform, that takes the output of Catena, a *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. Note that longer keys don't 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 the *key identifier* should also become a part of the associated data. But actually, this would be a bad move. Setting up the connection would require legitimate users to run Catena several (e.g., four) 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 on single call to Catena with larger security parameters and then run the output transform for each key.

In contrast to the password hashing scenario, where a user want to log-in without noticeable delay, users may tolerate a delay of several seconds to derive an encryption key from a password process [39], e.g., when setting up a secure connection, or when mounting a cryptographic file system. Based on tests on a Lenovo Thinkpad T430 (Intel Core i5-3210M) for Catena-KG-Keccak-512, we consider $g = 20$ and $g = 21$ as suitable values for Catena-KG-Keccak-512.

Brief Security Analysis. For any reasonable choice of keys and key sizes, the time for the output transform is negligible, compared to the time for running Catena. Furthermore, the output transform is just calling H, which we model as a random oracle. Thus, Catena-KG inherits its security features from Catena.

Consider a password *pwd* which's min-entropy is at least m, a "real" case, where the adversary is given a couple of keys of any length, all derived from *pwd*, and also the password-hash of *pwd* using the original Catena. And consider a "random" case, where the adversary is given uniformly distributed random strings of the same length as in the "real" case. Distinguishing "real" from "random" takes at least time 2^m. For the proof it is useful that the first byte of the input for the output transform is 0 (cf. Line 4 in Alg. 5), while, when calling the hash function H in Alg. 3, the first byte is a value $c \neq 0$. A detailed proof will be given in the full version of the paper.

7 Conclusion and Outlook

In this paper we have presented Catena, a novel, provable secure, and memory-demanding password scrambler which is resistant to cache-timing attacks. To the

best of our knowledge, `Catena` is the first password scrambler that naturally supports (1) *client-independent updates*, to ease switching to stronger security parameters if required and (2) *server relief*, where almost all the effort for password scrambling is left to the client. Furthermore, we introduced the new notion of the security parameter *garlic*, which reflects the memory usage and computation cost of a password scrambler.

The research of this work and the design of `Catena` have been inspired by the discovery of the cache-timing attacks on `scrypt`, which we presented in Section 3. In contrast to `scrypt`, our design is based on a sequential bit-reversal graph (SBRG), which we proved to be memory-hard, which implies a quadratic time-memory trade-off (i.e., for time T and space S, we have $S \cdot T = \Theta(2^{2g})$ for the *garlic* security parameter g). The proof is based on the once common *pebble game*. Furthermore, we prove the resistance of `Catena` against indistinguishably-attacks under chosen passwords in the random oracle model.

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

References

1. 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.
2. Daniel J. Bernstein. Cache-timing attacks on AES, 2005.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.
4. 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/`.
5. Stephen A. Cook. An Observation on Time-Storage Trade Off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.
6. Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. `http://www.akkadia.org/drepper/SHA-crypt.txt`. Accessed May 16, 2013.
7. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In *CRYPTO*, pages 335–353, 2011.
8. Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES*, pages 1–17, 2009.
9. Eric Glass. The NTLM Authentication Protocol and Security Support Provider. `http://davenport.sourceforge.net/ntlm.html`. Accessed May 16, 2013.
10. Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
11. John Hopcroft, Wolfgang Paul, and Leslie Valiant. On Time Versus Space. *J. ACM*, 24(2):332–337, April 1977.
12. Poul-Henning Kamp. The history of md5crypt. `http://phk.freebsd.dk/sagas/md5crypt.html`. Accessed May 16, 2013.
13. Takumi Kasai, Akeo Adachi, and Shigeki Iwata. Classes of pebble games and complete problems. In *ACM Annual Conference (2)*, pages 914–918, 1978.

14. 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.
15. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996.
16. 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.
17. Andrzej Lingas. A PSPACE Complete Problem Related to a Pebble Game. In *ICALP*, pages 300–321, 1978.
18. T. Alexander Lystad. Leaked Password Lists and Dictionaries - The Password Project. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`. Accessed May 16, 2013.
19. Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996.
20. Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.
21. Robert Morris and Ken Thompson. Password Security - A Case History. *Commun. ACM*, 22(11):594–597, 1979.
22. Krishna Neelamraju. Facebook Pages: Usage Patterns | Recommend.ly. `http://blog.recommend.ly/facebook-pages-usage-patterns/`. Accessed May 16, 2013.
23. Nvidia. Nvidia GeForce GTX 680 - Technology Overview, 2012.
24. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology—CRYPTO 2003*, 3:617–630, 2003.
25. 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.
26. Wolfgang J. Paul and Robert Endre Tarjan. Time-Space Trade-Offs in a Pebble Game. In *ICALP*, pages 365–369, 1977.
27. Colin Percival. Cache Missing for Fun and Profit. BDSCan 2004.
28. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
29. Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
30. A.G. Reinhold. HEKS: A family of key stretching algorithms, 1999.
31. Semiocast SAS. Brazil becomes 2nd country on Twitter, Japan 3rd Netherlands most active country. `http://goo.gl/QOeaB`. Accessed May 16, 2013.
32. J. Savage and S. Swamy. Space-time trade-offs on the FFT algorithm. *Information Theory, IEEE Transactions on*, 24(5):563 – 568, sep 1978.
33. John E. Savage and Sowmitri Swamy. Space-Time Tradeoffs for Oblivious Interger Multiplications. In *ICALP*, pages 498–504, 1979.
34. Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *FSE*, pages 191–204, 1993.
35. Ravi Sethi. Complete Register Allocation Problems. *SIAM J. Comput.*, 4(3):226–248, 1975.
36. Jens Steube. oclHashcat-plus - Advanced Password Recovery. `http://hashcat.net/oclhashcat-plus/`. Accessed May 16, 2013.
37. Sowmitri Swamy and John E. Savage. Space-Time Tradeoffs for Linear Recursion. In *POPL*, pages 135–142, 1979.
38. Martin Tompa. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. In *STOC*, pages 196–204, 1978.
39. Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen. SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications. Technical report, NIST, Gaithersburg, MD, United States, 2010.
40. M.V. Wilkes. *Time-Sharing Computer Systems*. MacDonald computer monographs. American Elsevier Publishing Company, 1968.

A Lower Bound for Pebbling a BRG

Lemma 2 ([16]). *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.

□