# Memory-Demanding Password Scrambling

Christian Forler, Stefan Lucks, and Jakob Wenzel

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

**Abstract.** Most of the common password scramblers hinder password-guessing attacks by "key stretching", e.g., by iterating a cryptographic hash function many times. With the increasing availability of cheap and massively parallel off-the-shelf hardware, iterating a hash function becomes less and less useful. To defend against attacks based on such hardware, one can exploit their limitations regarding to the amount of fast memory for each single core. The first password scrambler taking this into account was `scrypt`. In this paper we mount a cache-timing attack on `scrypt` by exploiting its password-dependent memory-access pattern. Furthermore, we show that it is possible to apply an efficient password filter for `scrypt` based on a malicious garbage collector. As a remedy, we present a novel password scrambler called Catena which provides both a password-independent memory-access pattern and resistance against garbage-collector attacks. Furthermore, Catena instantiated with the here introduced $(G, \lambda)$-DBH operation satisfies a certain time-memory tradeoff called $\lambda$-memory-hardness, i.e., using only $1/b$ the amount of memory, the time necessary to compute the password hash is increased by a factor of $b^{\lambda}$. Finally, we introduce a more efficient instantiation of Catena based on a bit-reversal graph.

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

## 1 Introduction

Passwords[1] are user-memorizable secrets for user authentication and cryptographic key derivation. Typical (user-chosen) passwords suffer from low entropy and can be attacked by trying out all possible candidates in order of likelihood. In some cases, the security of interactive password-based authentication and key derivation can be enhanced by dedicated cryptographic protocols defeating "offline" password guessing where an adversary is in possession of the password hashes (see [3] for an early example). Otherwise, the best protection are cryptographic password scramblers, performing *key stretching*. This means to hash the password with an intentionally slow one-way hash function to delay the adversary, without inconveniencing the user.

[1] We do not distinguish "passwords" from"passphrases" or "PINs".

Traditional password scramblers, e.g., `md5crypt` [10] or `sha512crypt` [7], iterate a cryptographic primitive (a block cipher or a hash function) many times. An adversary who has $b$ computing units (*cores*) can easily try out $b$ different passwords in parallel. With recent technological trends, such as the availability of graphical processing units (GPUs) with hundreds of cores [13], the question of how to slow down such adversaries becomes pressing. Fast memory is expensive. Thus, each core of a GPU (or any other cheap and massively parallel machine) possesses a very limited amount of fast memory ("cache"). Therefore, a defense against massively-parallel attacks on cheap hardware is to consume plenty of memory to cause a large amount of cache misses, up to the limit available to the user. Modern password scramblers allow to adjust the required memory by a logarithmic security parameter $g$, called *garlic* (memory-cost factor). A required property for such memory-consuming algorithms is *memory-hardness*, i.e., assume that a password scrambler uses $S = 2^g$ units of memory and an adversary has less than $S$ units of memory for each core. Then, the attack must slow down greatly. The first password scrambler that took this into account was `scrypt` [16], which inherited these features from its underlying function called `ROMix`. However, a memory-consuming password scrambler might suffer from a new problem. If the memory-access pattern depends on the password, this pattern may leak during a *cache-timing attack*.

***Contribution.*** In this paper we present two side-channel attacks against `ROMix` (1) a cache-timing attack exploiting its password-dependent memory-access pattern. This attack requires a spy process that runs on the defender's machine, without access to the internal memory of `ROMix` and (2) we show that `ROMix` is vulnerable to garbage-collector attacks. Both attacks should be considered severe since they might put the usage of memory-consuming password scramblers at high risk.

As a remedy, we introduce Catena, a memory-consuming password-scrambling framework which is resistant against the mentioned side-channel attacks. Furthermore, we present two instantiations whose workflow can be represented as directed acyclic graphs with bounded indegree. One is based on an adapted variant of the bit-reversal graph and the other one is based on an adapted variant of the double-butterfly graph (which is on the other hand constructed by putting two fast Fourier transformations (FFT) back-to-back).

***Outline.*** Section 2 introduces two notions of memory-hardness. Section 3 describes our side-channel attacks against `scrypt`. In Section 4, we introduce our novel password-scrambling framework Catena. Section 5 describes two instantiations, namely Catena-BRG and Catena-DBG, and Section 6 discusses their security. Section 7 concludes our work. Finally, in Appendix A we state the main difference between this version and our original submission.

## 2 Memory-Hardness

To describe memory requirements, we adopt and slightly change the notion from [16].

**Definition 1 (Memory-Hard Function [16]).**
*Let $g$ denote the memory-cost factor. For all $\alpha > 0$, a memory-hard function $f$ can be computed on a Random Access Machine using $S(g)$ space and $T(g)$ operations, where $S(g) \in \Omega(T(g)^{1-\alpha})$.*

Thus, for $S \cdot T = G^2$ with $G = 2^g$, using $b$ cores, we have $(1/b \cdot S) \cdot (b \cdot T) = G^2$. A formal generalization of this notion is given in the following.

**Definition 2 ($\lambda$-Memory-Hard Function).**
*Let $g$ denote the memory-cost factor. For a $\lambda$-memory-hard function $f$, which is computed on a Random Access Machine using $S(g)$ space and $T(g)$ operations with $G = 2^g$, it holds that $T(g) = \Omega(G^{\lambda+1}/S(g)^\lambda)$.*

Thus, we have $(1/b \cdot S^\lambda) \cdot (b \cdot T) = G^{\lambda+1}$.

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

***$\lambda$-Memory-Hard vs. Sequential Memory-Hard.*** In [16], Percival introduced the notion of sequential memory-hardness (SMH), which is satisfied by his introduced password scrambler `scrypt`. An algorithm is sequential memory-hard if an adversary has no computational advantage from the use of multiple CPUs. This means that one does not gain any advantage from parallelism, i.e., running such an algorithm on $b$ cores, one needs $b$ times the memory required for one core. On the other hand, a $\lambda$-memory-hard function satisfies a certain time-memory tradeoff. Thus, if only $\mathcal{O}(1/b)$ times the memory is available, one needs $\mathcal{O}(b^\lambda)$ times the computational effort, independent of the number of cores.

## 3 Side-Channel Attacks on scrypt

Algorithm 1 describes the `scrypt` password scrambler and its core operation `ROMix`. For pre- and post-processing, `scrypt` invokes the one-way function `PBKDF2` [9] 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, [16] 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.

**Algorithm 1** The `scrypt` algorithm and its core operation `ROMix` [16].

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

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

`ROMix` takes two inputs: an initial state $x$ that depends on both password and salt, and the array size $G$ that defines the required storage. One can interpret $\log_2(G)$ as the garlic of `scrypt`. In the *expand phase* (Lines 20–23), `ROMix` initializes an array $v$. More detailed, the array cells $v_0, \dots, v_{G-1}$ are set to $x, H(x), \dots, H(\dots(H(x)))$, respectively. In the *mix 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 of `scrypt` is its use of 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 while `scrypt` is running. Compromising the memory should not happen, anyway, but this issue could easily be fixed without any known bad effect on the security of `scrypt`, e.g., one could replace Line 12 of Algorithm 1 by $x \leftarrow \texttt{PBKDF2}(x, s, 1, 1)$.

**Algorithm 2** `ROMixMC`, performing `ROMix` with $G/K$ storage.

```
Input: $x$ {Initial State},
       $G$ {1st Cost Parameter},
       $K$ {2nd Cost Parameter}
Output: $x$ {Hash Value}
 1: for $i = 0, \dots, G-1$ 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 $i = 0, \dots, G-1$ do
 8:     $j \leftarrow x \bmod G$
 9:     $\ell \leftarrow K \cdot \lfloor j/K \rfloor$
10:     $y \leftarrow v_\ell$
11:     for $m = \ell + 1, \dots, j$ 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 $x$
```

Below, we will attack `scrypt` from the hardware side. The general idea of side-channel attacks against cryptographic algorithms is not new [11], neither is the usage of a spy process for cache-timing attacks [15]. But, to the best of our knowledge, we are the first to apply this approach to `scrypt`, or to password-hashing in general.

### 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 $K \leftarrow G/S$. The memory-constrained algorithm `ROMixMC` (see 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 $v_0$, $v_K$, $v_{2K}$, $\dots$, $v_{(S-1)K}$ – using all the $S$ available memory units. At Line 9, the variable $\ell$ is assigned to the highest multiple of $K$ less or equal to $j$. 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 $v_\ell$. These computations call $H$ $(K-1)/2$ times on average. Thus, the mix phase of `ROMixMC` is about $\Theta(K)$ times slower than the mix phase of `ROMix`, which dominates the workload for `ROMixMC`.

Next, we briefly discuss why `ROMixMC` is sequentially memory-hard (for the full proof see [16]). The intuition is as follows: the indices $j$ are determined by the output of the random oracle $H$ and thus, 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.

### 3.2 Cache-Timing Attacks

***The Spy Process.*** The idea to compute a “random” index $j$ and then ask for the value $v_j$, which is so useful for sequential memory-hardness, may be exploited to mount a cache-timing attack against `scrypt`. Consider a spy process that runs on the same machine as `scrypt`. This spy process cannot read the internal memory of `scrypt`, but shares its cache memory with `ROMix`:

1. The spy process interrupts `ROMix`, just before entering the mix phase (Line 24 of Algorithm 1) and overwrites the (entire) cache with arbitrary values $w_i$ to flush out all `ROMix`’ values $v_j$.
2. The spy process allows `ROMix` to perform a few more iterations of the mix loop (Line 24–27).
3. The spy process interrupts `ROMix` again. Now it reads the $w_i$, measuring precisely how long each read operation takes. If the corresponding $v_j$ has been used by `ROMix` in the second step, a “cache-miss” occurs, wich makes reading $w_i$ slow. Else, $w_i$ is likely to be still cached, and reading it is likely to be fast.

So, the spy process can tell an adversary the indices $j$ for which $v_j$ has been read during the first few iterations of the mix loop (Lines 24–27 of Algorithm 1). Given this information, we can attack `scrypt`.

***First Cache-Timing Attack.*** Let $x$ be the output of the operation `PBKDF2`$(pwd, s, 1, 1)$, where $pwd$ denotes the current password candidate and $s$ the salt. Then, we can sieve the password candidates as follows:

1. Run the expand phase of `ROMix`, without storing the values $v_i$, i.e., skip Line 21 of Algorithm 1.
2. Compute the index $j \leftarrow x \bmod G$.
3. If $j$ is one of the indices were read by `ROMix`, then store $pwd$ in a list. Otherwise, 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, i.e., the data structure to decide if $j$ is one of the indices being read with $v_j$, which can be shared among all cores. Thus, we can use exactly the kind of hardware, that `scrypt` was designed to hinder.

Let $r$ denote the number of iterations the loop in Lines 24–27 of `ROMix` performed, before the second interrupt from the spy process. Thus, we have a list of $r$ indices $j$ used by `ROMix`. The probability for a false password to survive is $r/G$.

We can further improve the attack. Assuming $r \ll G$, we may have space to store the $r$ values $v_j$ that were actually used by `ROMix` on each core. This allows us to simulate the first $r$ iterations of the loop in Lines 24–27. We can discard a password candidate immediately if the simulation tries to read any $v_j$ which is not on our short list. The probability for a false password candidate to survive is now down to $(r/G)^r$.

***Second Cache-Timing Attack.*** It may be more realistic to assume the second interrupt to be late, perhaps after the *mix phase* of `ROMix`. So, the loop in Lines 24–27 of `ROMix` was run $r = G$ times and, on average, each $v_i$ has been read once. Actually, some values have been read several times, and we expect about $(1/e)G \approx 0.37\,G$ array elements $v_i$ not to have been read at all. At a first look, we can eliminate about 37 % of the false password candidates – a 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 second 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 $v_0$. Like `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 information from our spy process). Thus, if only 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 $v_i$ was not 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 $(G - 1)/2$ calls of $H$. Additionally, each iteration needs one call to $H$ for computing $x \leftarrow H(x \oplus v_j)$. Including the work for the expand phase, with $G$ calls to $H$, the expected number of calls to reject a wrong password is about

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

As it turns out, rejecting a wrong password with constant memory is faster than 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.

### 3.3 The Garbage-Collector Attack

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

But, memory-demanding password scrambling may also open the gates for new attack opportunities for the adversary. If we allocate a huge block of memory for password scrambling, holding $v_0$, $v_1$, …, $v_{G-1}$, this memory becomes "garbage" after the password scrambler has terminated, and will be collected for reuse, eventually. One usually assumes that the adversary learns the hash of the secret. The *garbage-collector attack* assumes that the adversary additionally learns the memory content, i.e., the values $v_i$, after termination of the password scrambler.

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

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

### 3.4 Discussion

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

However, in an idealized "laboratory" setting, the applicability of cache-timing attacks against ROMix has been demonstrated [2]. The idealized conditions included execution rights on the system.

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

1. the adversary can find out when the password has been changed,

2. and the adversary can mount a password-guessing attack,

just from knowing the memory-access pattern.

Note that severe security issues can be caused by the second point. Consider any offline attack on the password. When passwords are hashed using an old-style password hash function, e.g., `md5crypt` [10], the adversary needs to first read the file containing the password hash. Without the password hash, mounting an offline attack is not possible. Even plaintext passwords are safe from offline adversaries which are not capable of reading the file containing the plaintext passwords. But, using the seemingly strong password hash function `scrypt` may enable offline password cracking, even when the adversary fails to ever learn the password hash.

## 4 Catena – A Memory-Hard Password Scrambler

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

### 4.1 Specification

A formal definition is shown in Algorithm 3, where the function $F_\lambda$ (see Line 3) is a placeholder for a certain instantiation. The password-dependent input of $H$ is appended to a prefix $c$, which denotes the iteration counter (garlic factor). Note that a secure password scrambler must satisfy preimage security. Catena inherits the preimage security from the underlying hash function $H$. Next, we discuss the tweak and two further novel features of Catena.

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

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

The first byte $\lambda$ defines together with the value $g$ (see above) the security parameters for Catena. The 32-bit value $|s|$ denotes the total length of

**Algorithm 3** Catena

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

the salt in bits. Finally, the n-bit value $H(AD)$ is the hash of the associated data AD, which can contain additional information like hostname, user-ID, name of the company, or the IP of the host, with the goal to customize the password hashes. Note that the order of the values does not matter as long as tey are fixed for a certain application.

The tweak is processed together with the secret password and the salt (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 different applications of Catena, and can depend on all possible input data. Note that one can easily provide unique tweak values (per user) when including the user-ID in the associated data.

### 4.2 Properties

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

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

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

***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). 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.

## 5 Instantiations

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

### 5.1 Catena-BRG

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

**Definition 3 (Bit-Reversal Permutation $\tau$).** *Fix a number $k \in \mathbb{G}$ and represent $i \in \mathbb{Z}_{2^k}$ as a binary $k$-bit number, $(i_0, i_1, \ldots, i_{k-1})$. The bit-reversal permutation $\tau : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ is defined by*

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

The bit-reversal permutation $\tau$ defines the $(G,\lambda)$-Bit-Reversal Graph.

**Definition 4 ($(G,\lambda)$-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 $(G,\lambda)$-bit-reversal graph $\Pi_g^\lambda(\mathcal{V},\mathcal{E})$ consists of $(\lambda+1)\cdot 2^g$ vertices*

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

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

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

For example, Figure 1 illustrates an (8,1)-BRG. Note that this graph is almost identical – except for one additional edge $e = (v_7^0, v_0^1)$ – to the bit-reversal graph presented by Lengauer and Tarjan in [12].

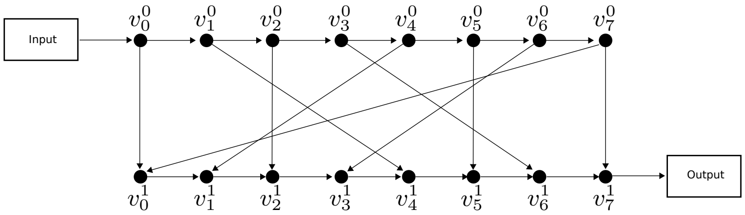

**Fig. 1.** An $(8, 1)$-BRG.

**Algorithm 4** $(G, \lambda)$-Bit-Reversal Hashing ($(G, \lambda)$-BRH)

**Input:** $g$ {Garlic}, $x$ {Value to Hash}, $\lambda$ {Depth}, $H$ {Hash Function}
**Output:** $x$ {Password Hash}

```
$v_0 \leftarrow H(x)$
for $i = 1, \ldots, 2^g - 1$ do
    $v_i \leftarrow H(v_{i-1})$
end for
for $k = 1, \ldots, \lambda$ do
    $r_0 \leftarrow H(v_0 \mid\mid v_{2^g-1})$
    for $i = 1, \ldots, 2^g - 1$ do
        $r_i \leftarrow H(r_{i-1} \mid\mid v_{\tau(i)})$
    end for
    $v \leftarrow r$
end for
return $r_{2^g-1}$
```

***Bit-Reversal Hashing.*** The $(G, \lambda)$-Bit-Reversal Hashing function is defined in Algorithm 4. It requires $\mathcal{O}(2^g)$ invocations of a given hash function $H$ for a fixed value of $x$. The three inputs ($g$, $x$, $\lambda$) of $(G, \lambda)$-BRH represent the garlic $g = \log_2(G)$, the value to process, and the depth, respectively. Thus, $g$ specifies the required units of memory. Moreover, incrementing $g$ by one doubles the time and memory effort for computing the password hash.

## 5.2 Catena-DBG

Note that a $(G, \lambda)$-Double-Butterfly Graph is based on a stack of $\lambda$ $G$-superconcentrators. The following definition of a $G$-superconcentrator is a slightly adapted version of that introduced in [12].

**Definition 5 ($G$-Superconcentrator).** *A directed acyclic graph $\Pi(\mathcal{V}, \mathcal{E})$ with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$, a bounded indegree, $G$ inputs, and $G$ outputs is called a $G$-superconcentrator if for every $k$ such that $1 \leq k \leq G$ and for every pair of subsets $V_1 \subset \mathcal{V}$ of $k$ inputs and*

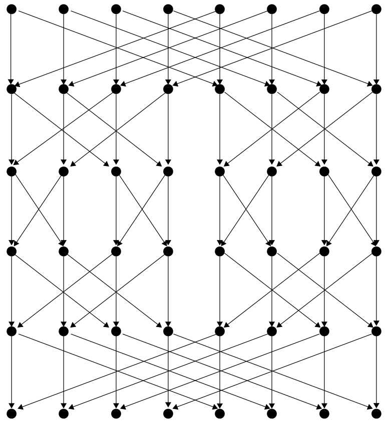

**Fig. 2.** A Cooley-Tukey FFT graph with eight input and output vertices.

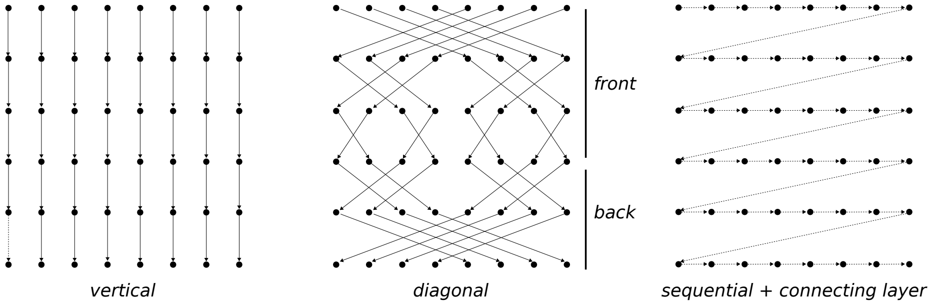

**Fig. 3.** Types of edges as we use them in our definitions.

*$V_2 \subset \mathcal{V}$ of $k$ outputs, there are $k$ vertex-disjoint paths connecting the vertices in $V_1$ to the vertices in $V_2$.*

A double-butterfly graph (DBG) is a special form of a $G$-superconcentrator which is defined by the graph representation of two back-to-back placed Fast Fourier Transformations [5]. More detailed, it is a representation of twice the Cooley-Tukey FFT algorithm [6] omitting one row in the middle (see Figure 2 for an example where $G = 8$). Therefore, a DBG consists of $2g$ rows.

Based on the double-butterfly graph, we define the sequential and stacked $(G, \lambda)$-double-butterfly graph. In the following, we denote $v_{i,j}^k$ as the $j$-th vertex in the $i$-th row of the $k$-th double-butterfly graph.

**Definition 6 ($(G, \lambda)$-Double-Butterfly Graph).** *Fix a natural number $g \geq 1$ and let $G = 2^g$. Then, the $(G, \lambda)$-Double-Butterfly Graph $\Pi(\mathcal{V}, \mathcal{E})$ consists of $2^g \cdot (\lambda \cdot (2g - 1) + 1)$ vertices*

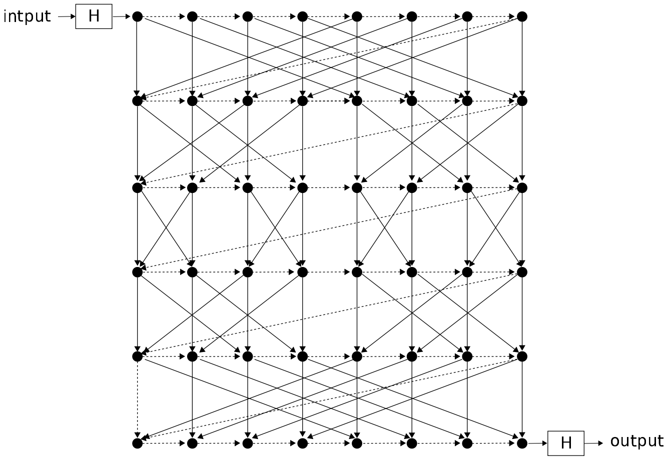

**Fig. 4.** An $(8,1)$-double-butterfly graph.

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

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

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

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

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

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

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

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

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

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

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

**Algorithm 5** $(G, \lambda)$-Double-Butterfly Hashing

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

***Double-Butterfly Hashing.*** The $(G, \lambda)$-double-butterfly hashing operation is defined in Algorithm 5. The structure is based on a $(G, \lambda)$-double-butterfly graph. Note that the function $\sigma$ (see Lines 7 and 9) is given by

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

Thus, $\sigma$ determines the indices of the vertices of the diagonal edges.

Since the security of Catena in terms of password hashing is based on a time-memory tradeoff, it is desired to implement it in an efficient way, making it possible to increase the required memory. We recommend to use BLAKE2b [1] as the underlying hash function, implying a block size of 1024 bits with 512 bits of output. Thus, it can process two input blocks within one compression function call. This is suitable for Catena-BRG since a bit-reversal graph satisfies a fixed indegree of at most 2. When considering Catena-DBG, we cannot simply concatenate the inputs to $H$ while keeping the same performance per hash function call, i.e., three inputs to $H$ require two compression function calls, which is a strong slow-down in comparison to $(G, \lambda)$-BRG. Therefore, we compute $H(X, Y, Z) = H(X \oplus Y \mid\mid Z)$ instead of $H(X, Y, Z) = H(X \mid\mid Y \mid\mid Z)$ obtaining the same performance as Catena-BRG per hash function call. Obviously, this doubles the probability of input collision. Nevertheless, for a 512-bit hash function the advantage for an adversary is still negligible.

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

## 6 Security

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

### 6.1 Resistance Against Side-Channel Attacks

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

Considering a malicious garbage collector, each of Algorithms 4 and 5 exposes the arrays $v$ and $r$. Both arrays are overwitten multiple times. Therefore, Catena-DBG is resistant against garbage-collector attacks. *Thus, any variant of* Catena *with some fixed* $\lambda \geq 2$ *is at least as resistant to garbage-collector attacks as the same variant with* $\lambda - 1$ *in the absence of a malicious garbage collector.*

### 6.2 Memory-Hardness

In 1970, Hewitt and Paterson introduced a method for analyzing time-memory tradeoffs (TMTOs) on directed acyclic graphs [14], called *pebble game*. While their method has been known for decades, it was recently used in a cryptographic context, e.g., [8]. In general, a pebble game is a common model to derive and analyze TMTOs as shown in [17,18,19,20,21].

The pebble-game model is restricted to DAGs with bounded in-degree and can be seen as a single-player game. Let $\Pi(\mathcal{V}, \mathcal{E})$ be a DAG and let $G = |\mathcal{V}|$ be the number of vertices within $\Pi(\mathcal{V}, \mathcal{E})$. In the setup phase of the game, the player gets $S$ pebbles (tokens) with $S \leq G$. A pebble can be placed (*pebble*) or be removed (*unpebble*) from a vertex $v \in \mathcal{V}$ under certain requirements:

1. A pebble may be removed from a vertex $v$ at any time.
2. A pebble can be placed on a vertex $v$ if all predecessors of the vertex $v$ are marked.
3. If all immediate predecessors of an unpebbled vertex $v$ 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 pebble $\Pi$, i.e., to mark all vertices of the graph $\Pi$ at least once. The total amount of moves represent the computational costs.

**Catena-BRG.** In [12], Lengauer and Tarjan have already proven the lower bound of pebble movements for a $(G, 1)$-bit-reversal graph.

**Theorem 1 (Lower Bound for a $(G, 1)$-BRG [12]).** *If $S \geq 2$, then, pebbling the bit-reversal graph $\Pi_g(\mathcal{V}, \mathcal{E})$ consisting of $G = 2^g$ input nodes with $S$ pebbles takes time*

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

Biryukov and Khovratovich have shown in [4] that stacking more than one bit-reversal graph only adds some linear factor to the quadratic time-memory tradeoff. Hence, a $(G, \lambda)$-BRG with $\lambda > 1$ does not achieve the properties of a $\lambda$-memory-hard function.

**Catena-DBG.** Likewise, the authors of [12] analyzed the time-memory tradeoff for a stack of $\lambda$ $G$-superconcentrators. Since the double-butterfly is a special form of a $G$-superconcentrators their bound also holds for $(G, \lambda)$-DBG.

**Theorem 2 (Lower Bound for a $(G, \lambda)$-Superconcentrator [12]).** *Pebbling a $(G, \lambda)$-superconcentrator using $S \leq G/20$ black and white pebbles requires $T$ placements such that*

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

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

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

## 7 Conclusion

In this paper we introduced a new class of side-channel attacks, called garbage-collector attack, which bases on a malicious garbage collector. We showed that the common password scrambler `scrypt` is vulnerable to this kind of attacks. Furthermore, we presented a (theoretical) cache-timing attack on `scrypt` that exploits its password-dependent memory-access pattern. Both attacks enable an adversary to construct a *memoryless* password filter that enables massively-parallel password-guessing attacks. Moreover, we show that our attacks work even without knowledge of the password hash. All regular implementations, i.e., implementations that are not hardened against side-channel attacks, of password scramblers with a password-dependent memory access pattern appear to be vulnerable to these attacks.

As a remedy, we introduced a novel password-scrambling framework Catena. We presented two instantiations with a password-independent memory-access pattern: Catena-BRG and Catena-DBG. The former is more efficient and 1-memory-hard, whereas the latter is less efficient but offers a higher level of security, i.e., $\lambda$-memory-hardness. Finally, we want to emphasize that Catena-BRG and Catena-DBG inherit their security from well-analyzed structures, namely bit-reversal and double-butterfly graphs.

## Acknowledgement

Thanks to B. Cox, E. List, C. Percival, A. Peslyak, S. Thomas, S. Touset, and the reviewers of the ASIACRYPT'14 for their helpful hints, comments, and fruitful discussions. Finally, we thank A. Biryukov and D. Khovratovich for pointing out a flaw in our proof of the time-memory tradeoff for the $(G, \lambda)$-BRH operation by providing a tradeoff cryptanalysis [4].

## References

1. 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.
2. Anne Barsuhn. Cache-Timing Attack on scrypt. Bauhaus-Universität Weimar, Bachelor Dissertation, December 2013.
3. 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.
4. Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of Catena. PHC mailing list: discussions@password-hashing.net.
5. William F. Bradley. Superconcentration on a Pair of Butterflies. *CoRR*, abs/1401.7263, 2014.
6. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.
7. Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. `http://www.akkadia.org/drepper/SHA-crypt.txt`. Accessed May 16, 2013.
8. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-Evolution Schemes Resilient to Space-Bounded Leakage. In *CRYPTO*, pages 335–353, 2011.
9. B. Kaliski. RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000.
10. Poul-Henning Kamp. The history of md5crypt. `http://phk.freebsd.dk/sagas/md5crypt.html`. Accessed May 16, 2013.
11. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996.
12. 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.
13. Nvidia. Nvidia GeForce GTX 680 - Technology Overview, 2012.
14. 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.
15. Colin Percival. Cache Missing for Fun and Profit. BDSCan 2004.
16. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
17. J. Savage and S. Swamy. Space-time trade-offs on the FFT algorithm. *Information Theory, IEEE Transactions on*, 24(5):563 – 568, sep 1978.

18. John E. Savage and Sowmitri Swamy. Space-Time Tradeoffs for Oblivious Interger Multiplications. In *ICALP*, pages 498–504, 1979.
19. Ravi Sethi. Complete Register Allocation Problems. *SIAM J. Comput.*, 4(3):226–248, 1975.
20. Sowmitri Swamy and John E. Savage. Space-Time Tradeoffs for Linear Recursion. In *POPL*, pages 135–142, 1979.
21. Martin Tompa. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. In *STOC*, pages 196–204, 1978.

## A Changelog

Based on the cryptanalysis provided by Biryukov and Khovratovich in [4], we decided to provide a slightly changed version in comparison to our submitted version. The major changes are (1) removing the flawed proof for $\lambda$-memory-hardness of a $(G, \lambda)$-BRG and (2) providing a new instance called Catena-DBG based on a $(G, \lambda)$-double-butterfly graph (variant of a stack of $\lambda$ Double-Butterfly Graphs (DBG)).