Overview of the Candidates for the Password Hashing Competition

And their Resistance against Garbage-Collector Attacks

Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel
`<first name>.<last name>@uni-weimar.de`

Bauhaus-Universität Weimar

Abstract. In this work we provide an overview of the candidates of the Password Hashing Competition (PHC) regarding to their functionality, e.g., client-independent update and server relief, their security, e.g., memory-hardness and side-channel resistance, and its general properties, e.g., memory usage and underlying primitives. Furthermore, we formally introduce two kinds of attacks, called Garbage-Collector and Weak Garbage-Collector Attack, exploiting the memory handling of a candidate. The following overview considers all candidates which are not yet withdrawn from the competition.

Keywords: Password Hashing Competition, Overview, Garbage-Collector Attacks

1 Introduction

Typical adversaries against password-hashing algorithms (also called password scramblers) try plenty of password candidates in parallel, which becomes a lot more costly if they need a huge amount of memory for each candidate. The defender (the honest party), on the other hand, will only compute a single hash, and the memory-cost parameters should be chosen such that the required amount of memory is easily available to the defender.

Up to now, there exist two basic strategies of how to design a memory-demanding password scrambler:

Type-A: Allocating a huge amount of memory which is rarely overwritten.
Type-B: Allocating a reasonable amount of memory which is overwritten multiple times.

The primary goal of the former type of algorithms is to increase the cost of dedicated password-cracking hardware, i.e., FPGAs and ASICs. However, algorithms following this approach do not provide high resistance against garbage-collector attacks, which are formally introduced in this work. The main goal of the second approach is to thwart GPU-based attacks by forcing a high amount of cache misses during the computation of the password hash. Naturally, algorithms following this approach provide some kind of built-in robustness against garbage-collector attacks.

But, memory-demanding password scrambling may also provide a completely new attack opportunity for an adversary, exploiting the handling of the target's machine memory. We introduce the two following attack models: (1) Garbage-Collector Attacks, where an adversary has access to the internal memory of the target's machine **after** the password scrambler terminated; and (2) Weak Garbage-Collector Attacks, where the password itself (or a value derived from the password using an efficient function) is written to the internal memory and almost never overwritten during the runtime of the password scrambler. If a password scrambler is vulnerable in either one of the attack models, it may be possible to significantly reduce the effort for testing a password candidate.

Remark 1. For our theoretic consideration of the proposed attacks, we assume a natural implementation of the algorithms, e.g., that some possible mentioned overwriting of the internal state **after** the invocation of an algorithm is neglected due to optimization.

2 (Weak) Garbage-Collector Attacks and their Application to ROMix and `scrypt`

In this section we first provide a definition of our attack models, i.e., the Garbage-Collector (GC) attack and the Weak Garbage-Collector (WGC) attack. For illustration, we first show that ROMix (the core of `scrypt` [19]) is vulnerable against a GC attack (this was already shown in [11], but without a formal definition of the GC attack), and second, we show that `scrypt` is also vulnerable against a WGC attack.

2.1 The (Weak) Garbage-Collector Attack

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

Definition 1 (Garbage-Collector Attack). *Let* $PS_G(\cdot)$ *be a memory-demanding password scrambler that depends on a memory-cost parameter* G *and let* Q *be a positive constant. Furthermore, let* v *denote the internal state of* $PS_G(\cdot)$ *after its termination. Let* $\mathcal{A}$ *be a computationally unbounded but always halting adversary conducting a garbage-collector attack. We say that* $\mathcal{A}$ *is successful if some knowledge about* v *reduces the runtime of* $\mathcal{A}$ *for testing a password candidate* x *from* $\mathcal{O}(PS_G(x))$ *to* $\mathcal{O}(f(x))$ *with* $\mathcal{O}(f(x)) \lll \mathcal{O}(PS_G(x))/Q, \forall x \in \{0,1\}^*$.

In the following we define the Weak Garbage-Collector Attack (WGCA), which exploits the fact that the password *pwd* or the hash of the password $H(pwd)$ must be in memory (or at least recomputed) in the last step of a password scrambler.

Definition 2 (Weak Garbage-Collector Attack). *Let* $PS_G(\cdot)$ *be a password scrambler that depends on a memory-cost parameter* G*, and let* $F(\cdot)$ *be an underlying function of* $PS_G(\cdot)$ *that can be efficiently computed. We say that an adversary* $\mathcal{A}$ *is successful in terms of a weak garbage-collector attack if a value* $y = F(pwd)$ *remains in memory during (almost) the entire runtime of* $PS_G(pwd)$*, where pwd denotes the secret input.*

Thus, an algorithm is vulnerable to a WGC attack, if either the secret input x or a value directly derived from x, using an efficient function F, has to be in memory during the invocation of a password scrambler. An adversary that is capable of reading the internal memory of a password scrambler during its invocation, gains knowledge about v. Thus, it can reduce the effort for filtering invalid password candidates by just computing $v' = F(x)$ and checking whether $v = v'$, where x denotes the current password candidate. Note that the function F can also be given by the identity function. Then, the plain password remains in memory, rendering WGC attacks trivial.

2.2 (Weak) Garbage-Collector Attacks on `scrypt`

Garbage-Collector Attack on ROMix. Algorithm 1 describes the necessary details of the `scrypt` password scrambler together with its core function ROMix. The pre- and post-whitening steps are given by one call (each) of the standardized key-derivation function PBKDF2 [15], which we consider as a single call to a cryptographically secure hash function. The ROMix function takes the initial state x and the memory-cost parameter G as inputs. First, ROMix initializes an array v of size $G \cdot n$ by iteratively applying a cryptographic hash function H (see Lines 20-23), where n denotes the output size of H in bits. Second, ROMix accesses the internal state at randomly computed points j to update the password hash (see Lines 24-27).

It is easy to see that the value v_0 is a plain hash (using PBKDF2) of the original secret *pwd* (see Line 10). Further, from the overall structure of `scrypt` and ROMix it follows that the internal memory is written once but never overwritten (Lines 20-23). Thus, all values $v_0, \ldots, v_{G-1}$ can be accessed by a garbage-collector adversary $\mathcal{A}$ after the termination of `scrypt`. For each password candidates pwd', $\mathcal{A}$ can now simply compute $x' \leftarrow \text{PBKDF2}(pwd')$ and check whether $x' = v_0$. If so, pwd' is a valid preimage. Thus,

Algorithm 1 The algorithm `scrypt` [19] and its core operation ROMix.

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

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

$\mathcal{A}$ can test each possible candidate in $\mathcal{O}(1)$, rendering an attack against `scrypt` (or especially ROMix) practical (and even memory-less).
As a possible countermeasure, one can simply overwrite $v_0, \ldots, v_{G-1}$ after running ROMix. Nevertheless, this step might be removed by a compiler due to optimization, since it is algorithmically ineffective.

Weak Garbage-Collector Attack on ***`scrypt`*.** In Line 12 of Algorithm 1, `scrypt` invokes the key-derivation function PBKDF2 the second time using again the password *pwd* as input again. Thus, *pwd* has to be stored in memory during the entire invocation of `scrypt`, which implies that `scrypt` is vulnerable to WGC attacks.

3 Overview

Before we present the tables containing the comparison of the candidates for the Password Hashing Competition (PHC), we introduce the necessary notions (see Table 1) to understand the tables.

Identifier	**Description**
	Primitives/Structures
BC	Block cipher
SC	Stream cipher
PERM	Keyless permutation
HF	Hash function
BRG	Bit-Reversal Graph
DBG	Double-Butterfly Graph
	General Properties
CIU	Supports client-independent update
SR	Supports server relief
KDF	Usable as Key-Derivation Function (requires outputs to be pseudorandom)
FPO	Using floating-point operations
Flexible	Underlying primitive can be replaced
Iteration	Algorithm is based on iterations/rounds
	Security Properties
GCA Res.	Resistant against garbage-collector attacks (see Definition 1)
WGCA Res.	Resistant against weak garbage-collector attacks (see Definition 2)
SCA Res.	Resistant against side-channel attacks.
ROM-port	Special form of memory hardness [8].
Shortcut	Is it possible to bypass the main (memory and time) effort of an algorithm by knowing additional parameters, e.g., the Blum integers p and q for MAKWA which are used to compute the modulo n.

Table 1. Notations used in Tables 2, 3, and 4.

Comments for Table 2. The values in the column ”Memory“ come from the authors recommendation for password hashing or are marked as ‘○‘ if no recommendation exists. The entry “A (CF)” denotes that only the compression function of algorithm A is used. An entry $A(XR)$ denotes that an algorithm A is reduced to X rounds. The `scrypt` password scrambler is just added for comparison. If an algorithm can only be partially be computed in parallel, we marked the corresponding entry with ’part.’. Note that POMELO and *schvrch* do not depend on an existing underlying primitive but on an own construction.

Algorithm	**Based On**	**Iteration**	**Memory Usage**	**Parallel**	**Underlying Primitive**		**Underlying Mode**
					BC/SC/PERM	**HF**	
AntCrypt		✓	32 kB	part.	-	SHA-512	-
ARGON	AES	✓	1 kB - 1 GB	✓	AES (5R)	-	-
battcrypt		✓	128 kB - 128 MB	part.	Blowfish-CBC	SHA-512	-
CATENA	BRG/DBG	✓	8 MB	part.	-	BLAKE2b	-
CENTRIFUGE		✓	2 MB	-	AES-256	SHA-512	-
EARWORM		✓	2 GB (ROM)	✓	AES (1R)	SHA-256	PBKDF2$_{\text{HMAC}}$
Gambit	Sponge	✓	50 MB	-	Keccak$_f$	-	-
Lanarea DF		✓	256 B	-	-	BLAKE2b	-
Lyra2	Sponge	✓	400 MB - 1 GB	-	BLAKE2b (CF)	-	
MAKWA	Squarings	✓	negl.	✓	-	SHA-256	HMAC
MCS_PHS		✓	negl.	-	-	MCSSHA-8	-
ocrypt	`scrypt`	✓	1 MB - 1 GB	-	ChaCha	CubeHash	-
Parallel		✓	negl.	✓	-	SHA-512	-
PolyPassHash	Shamir Sec. Sharing	-	negl.	-	AES	SHA-256	-
POMELO		✓	(8 KB, 8 GB)	part.	-	-	-
Pufferfish	Blowfish/`bcrypt`	✓	4 - 16 kB	-	Blowfish	SHA-512	HMAC
Rig	BRG	✓	15 MB	part.	-	BLAKE2b	-
`scrypt`		✓	1 GB	-	Salsa20/8	-	PBKDF2
schvrch		✓	8 MB	part.	-	-	-
`Tortuga`	Sponge & rec. Feistel	✓	○		Turtle	-	-
SkinnyCat	BRG	✓	○	-	-	SHA-*/BLAKE2*	-
TwoCats	BRG	✓	○	✓	-	SHA-*/BLAKE2*	-
Yarn		✓	○	part.	BLAKE2b (CF), AES	-	-
yescrypt	`scrypt`	✓	3 MB (RAM)/3 GB (ROM)	part.	Salsa20/8	SHA-256	PBKDF2$_{\text{HMAC}}$

Table 2. Overview of PHC Candidates and their general properties (Part 1).

Comments for Table 3. Even if the authors of a scheme do not claim to support client-independent update (CIU) or server relief, we checked for the possibility and marked the corresponding entry in the table with ’✓’ or ’part.’ if possible or possible under certain requirements, respectively. Note that we say that an algorithm does not support SR when it requires the whole state to be transmitted. Moreover, we say that an algorithm does not support CIU if any additional information to the password hash itself is required. Note that Catena refers to both instantiations, i.e., Catena-BRG and Catena-DBG.

Algorithm	**CIU**	**SR**	**FPO**	**Flexible**
AntCrypt	✓	-	✓	part.
ARGON	✓	✓	-	✓
battcrypt	✓	-	-	part.
Catena	✓	✓	-	✓
CENTRIFUGE	-	-	-	✓
EARWORM	-	✓	-	-
Gambit	-	✓	opt.	part.
Lanarea DF	-	✓	-	✓
Lyra2	✓	✓	-	part.
Makwa	part.	-	-	✓
MCS_PHS	-	✓	-	part.
ocrypt	-	-	-	✓
Parallel	✓	✓	-	✓
PolyPassHash	✓	-	-	✓
POMELO	✓	-	-	-
Pufferfish	-	✓	-	part.
Rig	✓	✓	-	✓
scrypt	-	-	-	✓
schvrch	-	-	-	-
Tortuga	-	-	-	-
SkinnyCat	-	✓	-	✓
TwoCats	✓	✓	-	✓
Yarn	-	✓	-	-
yescrypt	-	✓	-	✓

Table 3. Overview of PHC Candidates and their general properties (Part 2).

Comments for Table 4. The column “Type” specifies which type of a memory-demanding design a certain algorithm satisfies. The types “A” and “B” are as described in Section 1 and marking an algorithm by “-” denotes that it is not designed to be memory-demanding. An entry supplemented by ’*’ (as for Memory-Hard. and Security Analysis), denotes that there exists not sophisticated analysis or proofs for the given claim/assumption. For GCA/WGCA Res., ’✓*’ denotes that this kind of resistance is only given under certain conditions. For SCA Res., ’part.’ (partial) means that only one or more parts (but not all) provide resistance against side-channel attacks.

Remark 2. Note that we do not claim completeness for Table 4. For example, we defined a scheme not to be resistant against side-channel attacks if it maintains a password-dependent memory-access pattern. Nevertheless, there exist several other types of side-channel attacks such as those based on power or acoustic analysis.

				Resistance				
Algorithm	**Type**	**Memory-Hardness**	**KDF**	**GCA**	**WGCA**	**SCA**	**Security Analysis**	**Shortcut**
AntCrypt	B	✓	✓	✓	✓	✓*	✓*	-
ARGON	B	✓	✓	✓	✓	-	✓	-
battcrypt	B	✓	✓	✓	-	✓	✓*	-
CATENA-BRG	B	✓	✓	-	✓	✓	✓	-
CATENA-DBG	B	λ	✓	✓	✓	✓	✓	-
CENTRIFUGE	A	✓*	-	-	-	✓	✓*	-
EARWORM	B	ROM-port	-	✓	-	✓	✓	-
Gambit	B	✓*	✓	✓	✓	✓	✓*	-
Lanarea DF	B	✓*	✓	✓	✓	part.	✓*	-
Lyra2	B	✓	✓	✓	✓	part.	✓	-
MAKWA	-	-	✓	✓	✓	part.	✓	✓
MCS_PHS	-	-	✓	-	✓	✓	-	-
ocrypt	B	✓*	✓	✓	✓	-	✓*	-
Parallel	-	-	✓	✓	✓	✓	✓*	-
PolyPassHash	-	-	-	-	-	-	✓	✓
POMELO	B	-	-	✓	✓	part.	✓*	-
Pufferfish	B	✓*	✓	✓	✓	-	✓*	-
Rig	B	λ	✓	✓	✓	✓	✓	-
`scrypt`	A	sequential	✓	-	-	-	✓	-
schvrch	B	-	-	✓	✓	✓	✓*	-
`Tortuga`	B	✓*	✓	✓	✓	✓	✓*	-
SkinnyCat	A	sequential	✓	-	-	part.	✓	-
TwoCats	B	sequential	✓	✓	-	part.	✓	-
Yarn	B	✓*	-	✓	-	-	✓*	-
yescrypt	A	ROM-port, sequential	✓	-	-	-	✓*	-

Table 4. Overview over the security properties of PHC candidates.

4 Resistance of PHC Candidates against (W)GC Attacks

In this section we briefly discuss potential weaknesses of each PHC candidate regarding to garbage-collector (GC) and weak-garbage-collector (WGC) attacks or argue why it provides resistance against such attacks. Note that we assume the reader to be familiar with the internals of the candidates since we only concentrate on those parts of the candidates that are relevant regarding to GC/WGC attacks.

***AntCrypt** [9].* The internal state of AntCrypt is initialized with the secret *pwd*. During the hashing process, the state is overwritten `outer_rounds`×`inner_rounds` times, which thwarts GC attacks. Moreover, since *pwd* is used only to initialize the internal state, WGC attacks are not applicable.

ARGON *[3].* First, the internal state derived from *pwd* is the input to the padding phase. After the padding phase, the internal state is overwritten by applying the functions `ShuffleSlices` and `SubGroups` at least L times. Based on this structure, and since *pwd* is used only to initialize the state, ARGON is not vulnerable against GC/WGC attacks.

***battcrypt** [24].* Within battcrypt, the plain password is used only once, namely to generate a value $key = \text{SHA-512}(\text{SHA-512}(salt \mid\mid pwd))$. The value *key* is then used to initialize the internal state, which is expanded afterwards. In the *Work* phase, the internal state is overwritten `t_cost`×`m_size` times using password-dependent indices. Thus, GC attacks are not applicable.

Note that the value *key* is used in the three phases *Initialize blowfish*, *Initialize data*, and *Finish*, whereas it is overwritten in the phase *Finish* the first time. Thus, *key* must remain in memory until the final phase, rendering the following WGC attack possible: For each password candidates x and the known value *salt*, compute $key' = \text{SHA512}(\text{SHA512}(salt \mid\mid x))$ and check whether $key' = key$. If so, mark x as a valid password candidate.

Catena *[11].* CATENA has two instantiations CATENA-BRG and CATENA-DBG, which are based on a (G, λ)-Bit-Reversal Graph and a (G, λ)-Double-Butterfly Graph, respectively. Both instantiations use an array of G elements each as their internal state. This state is overwritten $\lambda - 1$ times for CATENA-BRG and $(2\log_2(G) - 1) \cdot \lambda + 2\log_2(G) - 2$ times for CATENA-DBG. Hence, when considering CATENA-BRG, a GC adversary with access to the state can reduce the effort for testing a password candidate by a factor of $1/\lambda$. When considering CATENA-DBG, the reduction of the computational cost of an adversary is negligible. The authors mention this fact by recommending CATENA-DBG when considering GC attacks.

For CATENA-BRG as well as CATENA-DBG, the password *pwd* is used only to initialize the internal state. Thus, both instantiations provide resistance against WGC attacks.

CENTRIFUGE* *[1]. The internal state M of size `p_mem`×`outlen` byte is initialized with a seed S derived from the password and the salt as follows: $S = H(s_L \mid\mid s_R)$, where $s_L \leftarrow H(pwd \mid\mid len(pwd))$ and $s_R \leftarrow H(\text{salt} \mid\mid len(\text{salt}))$. Furthermore, S is used as the initialization vector (IV) and the key for the CFB encryption. The internal M is written once and later only accessed in a password-dependent manner. Thus, a GC adversary can launch the following attack:

1. receive the internal state M (or at least $M[1]$) from memory
2. for each password candidate x:
   - (a) initialization (seeding and S-box)
   - (b) compute the first table entry $M'[1]$ (during the *build table* step)
   - (c) check whether $M'[1] = M[1]$

The final step of CENTRIFUGE is to encrypt the internal state, requiring the key and the IV, which therefore must remain in memory during the invocation of CENTRIFUGE. Thus, the following WGC attack is applicable:

1. Compute $s_R \leftarrow H(salt \mid\mid len(salt))$
2. For every password candidate x:
   - (a) Compute $s'_L \leftarrow H(x \mid\mid len(x))$ and $S' = H(s'_L \mid\mid s_R)$, and compare if $S' = IV$
   - (b) If yes: mark x as a valid password candidate
   - (c) If no: go to Step 2

EARWORM *[12].* EARWORM maintains an array called *arena* of size $2^{m\_cost} \times L \times W$ 128-bit blocks, where $W = 4$ and $L = 64$ are recommended by the authors. This read-only array is randomly initialized (using an additional secret input which has to be constant within a given system) and used as AES round keys. Since the values within this array do not depend on the secret *pwd*, knowledge about *arena* does not help any malicious garbage collector. Within the main function of EARWORM (WORKUNIT), an internal state *scratchpad* is updated multiple times using password-dependent accesses to *arena*. Thus, a GC adversary cannot profit from knowledge about *scratchpad*, rendering GC attacks not applicable.

Within the function WORKUNIT, the value *scratchpad_tmpbuf* is derived directly from the password as follows:

$$scratchpad\_tmpbuf \leftarrow \text{EWPRF}(pwd, 01 \mid\mid salt, 16W),$$

where EWPRF denotes $\text{PBKDF2}_{\text{HMAC-SHA256}}$ with the first input denoting the secret key. This value is updated only at the end of WORKUNIT using the internal state. Thus, it has to be in memory during almost the whole invocation of EARWORM, rendering the following WGC attack possible: For each password candidate x and the known value *salt*, compute $y = \text{EWPRF}(x, 01 \mid\mid salt, 16W)$ and check whether $scratchpad\_tmpbuf = y$. If so, mark x as a valid password candidate.

Gambit* *[21]. Gambit bases on a duplex-sponge construction [2] maintaining two internal states S and Mem, where S is used to subsequently update Mem. First, password and salt are absorbed into the sponge and after one call to the underlying permutation, the squeezed value is written to the internal state Mem and processed r times (number of words in the ratio of S). The output after the r steps is optionally XORed with an array lying in the ROM. After that, Mem is absorbed into S again. This step is executed t times, where t denotes the time-cost parameter. The size of Mem is given by m, the memory-cost parameter. Continuously updating the states Mem and S thwarts GC attacks. Moreover, since *pwd* is used only to initialize the state within the sponge construction, WGC attacks are not applicable.

Lanarea DF *[18].* Lanarea DF maintains a matrix (internal state) consisting of $16 \cdot 16 \cdot m\_cost$ byte values, where m_cost denotes the memory-cost parameter. After the password-independent setup phase, the password is processed by the internal pseudorandom function producing the array $(h_0, \ldots, h_{31})$, which determines the positions on which the internal state is accessed during the core phase (thus, allowing cache-timing attacks). In the core phase, the internal state is overwritten $t\_cost \times m\_cost \times 16$ times, rendering GC attacks impossible. Moreover, the array $(h_0, \ldots, h_{31})$ is overwritten $t\_cost \times m\_cost$ times which thwarts WGC attacks.

Lyra2 *[14].* The Lyra2 password scrambler (and KDF) is based on a duplex sponge construction maintaining a state H, which is initialized with the password, the salt, and some tweak in the first step of its algorithm. The authors indicate that the password can be overwritten from this point on, rendering WGC attacks impossible. Moreover, Lyra2 maintains an internal state M, which is overwritten (updated using values from the sponge state H) multiple times. Thus, GC attacks are not applicable for Lyra2.

Makwa *[22].* Makwa has not been designed to be a memory-demanding password scrambler. Its strength is based on a high number of squarings modulo a composite (Blum) integer n. The plain (or hashed) password is used twice to initialize the internal state, which is then processed by squarings modulo n. Thus, neither GC nor WGC attacks are applicable for Makwa.

MCS_PHS *[17].* Depending on the size of the output, MCS_PHS applies iterated hashing operations, reducing the output size of the hash function by one byte in each iteration – starting from 64 bytes. Note that the memory-cost parameter m_cost is used only to increase the size of the initial chaining value T_0. The secret input *pwd* is used once, namely when computing the value T_0 and can be deleted afterwards, rendering WGC attacks not applicable. Furthermore, since the output of MCS_PHS is computed by iteratively applying the underlying hash function (without handling an internal state which has to be placed in memory), GC attacks are not possible.

ocrypt *[10].* The basic idea of ocrypt is similar to that of `scrypt`, besides the fact that the random memory accesses are determined by the output of a stream cipher (ChaCha) instead of a hash function cascade. The output of the stream cipher determines which element of the internal state is updated, which consists of $2^{17+m_{cost}}$ 64-bit words. During the invocation of ocrypt, the password is used only twice: (1) as input to CubeHash, generating the key for the stream cipher and (2) to initialize the internal state. Neither the password nor the output of CubeHash are used again after the initialization. Thus, ocrypt is not vulnerable to WGC attacks.
The internal state is processed $2^{17+t_{cost}}$ times, where in each step one word of the state is updated. Since the indices of the array elements accessed depend only on the password and not on the content, GC attacks are not possible by observing the internal state after the invocation of ocrypt.

Remark 3. Note that the authors of ocrypt claim side-channel resistance since the indices of the array elements are chosen in a password-independent way. But, as the password (beyond other inputs) is used to derive the key of the underlying stream cipher, this assumption does not hold, i.e., the output of the stream cipher depends on the password, rendering (theoretical) cache-timing attacks possible.

Parallel *[25].* Parallel has not been designed to be a memory-demanding password scrambler. Instead, it is highly optimized to be comuted in parallel. First, a value *key* is derived from the secret input *pwd* and the salt by

$$key = \text{SHA-512}(\text{SHA-512}(salt) \mid\mid pwd).$$

The value *key* is used (without being changed) during the Clear work phase of Parallel. Since this phase defines the main effort for computing the password hash, it is highly likely that a WGC adversary can gain knowledge about *key*. Then, the following WGC attack is possible: For each password candidate x and the known value *salt*, compute $y = \text{SHA-512}(\text{SHA-512}(salt) \mid\mid x)$ and check whether $key = y$. If so, mark x as a valid password candidate. Since the internal state is only given by the subsequently updated output of SHA-512, GC attacks are not applicable for Parallel.

PolyPassHash* *[5]. PolyPassHash denotes a threshold system with the goal to protect an individual password (hash) until a certain number of correct passwords (and their corresponding hashes) are known. Thus, it aims at protecting an individual password hash within a file containing a lot of password hashes, rendering PolyPassHash not to be a password scrambler itself. The protection lies in the fact that one cannot easily verify a target hash without knowing a minimum number of hashes (this technical approach is referred to as PolyHashing). In the PolyHashing construction, one maintains a (k, n)-threshold cryptosystem, e.g., Shamir Secret Sharing. Each password hash $h(pwd_i)$ is blinded by a share $s(i)$ for $1 \leq i \leq k \leq n$. The value $z_i = h(pwd_i) \oplus s(i)$ is stored in a so-called PolyHashing store at index i. The shares $s(i)$ are not stored on disk. But, to be efficient, a legal party, e.g., a server of a social networking system, has to store at least k shares in the RAM to on-the-fly compare incoming requests on-the-fly. Thus, this system only provides security against adversaries which are only able to read the hard disk but not the volatile memory (RAM).

Since the secret (of the threshold cryptosystem) or at least the k shares have to be in memory, GC attacks are possible by just reading the corresponding memory. The password itself is only hashed and blinded by $s(i)$. Thus, if an adversary is able to read the shares or the secret from memory, it can easily filter wrong password candidates, i.e., makeing PolyPassHash vulnerable against WGC attacks.

POMELO* *[27]. POMELO contains three update functions $F(S, i), G(S, i, j)$, and $H(S, i)$, where S denotes the internal state and i and j the indices at which the state is accessed. Those functions update at most two state words per invocation. The functions F and G provide deterministic random-memory accesses (determined by the cost parameter t_cost and m_cost), whereas the function H provides random-memory accesses determined by the password, rendering POMELO at least partially vulnerable to cache-time attacks. Since the password is used only to initialize the state, which itself is overwritten about $2^{2 \cdot t\_cost} + 2$ times, POMELO provides resistance against GC and WGC attacks.

Pufferfish* *[13]. The main memory used within Pufferfish is given by a two-dimensional array consisting of 2^{5+m_cost} 512-bit values, which is regularly accessed during the password hash generation. The first steps of Pufferfish are given by hashing the password. The result is then overwritten $2^{5+m\_cost} + 3$ times, rendering WGC attacks not possible. The state word containing the hash of the password ($S[0][0]$) is overwritten 2^{t_cost} times. Thus, there does not exist a shortcut for an adversary, rendering GC attacks impossible.

Rig* *[6]. Rig maintains two arrays a (sequential access) and k (bit-reversal access). Both arrays are iteratively overwritten $r \cdot n$ times, where r denotes the round parameter and n the iteration parameter. Thus, rendering Rig resistant against GC attacks. Note that within the setup phase, a value α is computed by

$$\alpha = H_1(x) \quad \text{with} \quad x = pwd \mid\mid len(pwd) \mid\mid \ldots,$$

Since the first α (which is directly derived from the password) is only used during the initialization phase, WGC attacks are not applicable.

schvrch ***[26].*** The password scrambler *schvrch* maintains an internal state of $256 \cdot 64$-bit words (2 kB), which is initialized with the password, salt and their corresponding lengths, and the final output length. After this step, the password can be overwritten in memory. This state is processed t_cost times by a function $revolve()$, which affects in each invocation all state words. Next, after applying a function $stir()$ (again, changing all state entries), it expands the state to m_cost times the state length. Each part (of size state length) is then processed to update the internal state, producing the hash after each part was processed. Thus, the state word initially containing the password is overwritten $t\_cost \cdot m\_cost$ times, rendering GC attacks impossible. Further, neither the password nor a value directly derived from it is required during the invocation of *schvrch*, which thwarts WGC attacks.

`Tortuga`* *[23]. GC and WGC attacks are not possible for `Tortuga` since the password is absorbed to the underlying sponge structure, which is then processed at least two times by the underlying keyed permutation (Turtle block cipher [4]), and neither the password nor a value derived from it has to be in memory.

***SkinnyCat and TwoCats* [7].** SkinnyCat is a subset of the TwoCats scheme optimized for implementation. Both algorithms maintain a 256-bit state *state* and an array of 2^{m_cost+8} 32-bit values (*mem*). During the initialization, a value PRK is computed as follows:

$$PRK = Hash(len(pwd), len(salt), \dots, pwd, salt).$$

The value PRK is used in the initialization phase and first overwritten in the forelast step of SkinnyCat (when the function *addIntoHash()* is invoked). Thus, an adversary that gains knowledge about the value PRK is able to launch the following WGC attack: For each password candidates x and the known value *salt*, compute $PRK' = Hash(len(x), len(salt), \dots, x, salt)$ and check whether $PRK = PRK'$. If so, mark x as a valid password candidate.

Within TwoCats, the value PRK is overwritten at an early state of the hash value generation. TwoCats maintains a garlic application loop from $startMemCost = 0$ to $stopMemCost$, where $stopMemCost$ is a user-defined value. In each iteration, the value PRK is overwritten, rendering WGC attacks for TwoCats not possible.

Both SkinnyCat and TwoCats consist of two phases each. The first phase updates the first half of the memory (early memory) $mem[0, \dots, memlen/(2 \cdot blocklen) - 1]$, where the memory is accessed in a password-independent manner. The second phase updates the second half of the memory $mem[memlen/(2 \cdot blocklen), \dots, memlen/blocklen - 1]$, where the memory is accessed in a password-dependent manner. Thus, both schemes provide only partial resistance against cache-timing attacks. For SkinnyCat, the early memory is never overwritten, rendering the following GC attack possible:

1. Obtain $mem[0, \dots, memlen/(2 \cdot blocklen) - 1]$ and PRK from memory
2. Create a state $state'$ and an array mem' of the same size as *state* and *mem*, respectively
3. Set $fromAddr = slidingReverse(1) \cdot blocklen$, $prevAddr = 0$, and $toAddr = blocklen$
4. For each password candidate x:
   (a) Compute PRK' as described using the password candidate x
   (b) Initialize $state'$ and mem' as prescribed using PRK'
   (c) Compute $state'[0] = (state'[0] + mem'[1]) \oplus mem'[fromAddr + +]$
   (d) Compute $state'[0] = \text{ROTATE\_LEFT}(state'[0], 8)$
   (e) Compute $mem'[blocklen + 1] = state'[0]$
   (f) Check whether $mem'[blocklen + 1] = mem[blocklen + 1]$
   (g) If yes: mark x as a valid password candidate
   (h) If no: go to Step 4.

Note that this attack does not work for TwoCats since an additional feature in comparison to SkinnyCat is that the early memory is overwritten.

***Yarn* [16].** Yarn maintains two arrays *state* and *memory*, consisting of *par* and 2^{m_cost} 16-byte blocks, respectively. The array *state* is initialized using the salt. Afterwards, *state* is processed using the BLAKE2b compression function with the password *pwd* as message, resulting in an updated array *state1*. This array has to be stored in memory since it is used as input to the final phse of Yarn. The array *state* is expanded afterwards and further, it is used to initialize the array *memory*. Next, *memory* is updated continuously. Both *memory* and *state* are overwritten continuously. The array *state1* is overwritten at the lastest in the final phase of Yarn. Thus, GC attacks are not possible for Yarn. Nevertheless, the array *state1* is directly derived from *pwd* and stored until the final phase occurs. Thus, the following WGC attack is possible:

1. Compute $h \leftarrow \textsc{Blake2b\_GenerateInitialState}(outlen, salt, pers)$ as in the first phase of Yarn
2. For each password candidate x:
   (a) Compute $h' \leftarrow \textsc{Blake2b\_ConsumeInput}(h, x)$
   (b) Compute *state1'* $\leftarrow \textsc{Truncate}(h', outlen)$ and check whether *state1'* $=$ *state1*

***yescrypt* [20].** The yescrypt password scrambler maintains two lookup tables V and $VROM$, where V is located in the RAM and $VROM$ in the ROM. Depending on the flag `YESCRYPT_RW`, the behaviour of yescrypt for the usage of the memory in RAM can be switched from "write once, read many" to "read-write". Nevertheless, yescrypt does not completey overwrite the memory in RAM, rendering similar GC attacks as for `scrypt` possible (see Section 2.2). But, such an attack would require a higher effort in comparison the attack on `scrypt` since yescrypt overwrites the RAM locations at least partially. Thus, an adversary must first search for the memory parts which where not overwritten during the invocation of yescrypt.

When considering WGC attacks, one has to differ between two variants of yescrypt depending whether it runs in the `scrypt` compatibility mode or not. In `scrypt` compatibility mode, obviously the same WGC as for `scrypt` is applicable (see Section 2.2). If not running in `scrypt` compatibility mode, yescrypt uses the results of the initial call to PBKDF2 in the last step. Thus, the value which has to remain in memory is given by HMAC-SHA-256(SHA-256(pwd), $salt$). Since it is also possible to compute HMAC and SHA-256 efficiently, yescrypt does not provide resistance against WGC attacks.

5 Conclusion

In this work we provided an overview of the first-round candidates of the Password Hashing Competition, which are not yet withdrawn. Further, we analyzed each algorithm regarding to its vulnerability against garbage-collector and weak garbage-collector attacks. Even if both attacks require access to the memory on the target's machine, they show a potential weakness, which should be taken into consideration. As a results, we have shown GC attacks on CENTRIFUGE, Catena-BRG, MCS_PHS, PolyPassHash, `scrypt`, SkinnyCat, TwoCats, and yescrypt, whereas the attacks on SkinnyCat, TwoCats and yescrypt work only under certain requirements. Additionally, we have shown that WGC attacks are possible for battcrypt, CENTRIFUGE, EARWORM, PolyPassHash, `scrypt`, SkinnyCat, TwoCats, Yarn, and yescrypt.

6 Acknowledgement

Thanks to A. Peslyak, J. M. Gosney, H. Wu, B. Cox, D. Khovratovich, and all contributors to the PHC mailing list for providing us with valuable comments and fruitful discussions.

References

1. Rafael Alvarez. CENTRIFUGE – A password hashing algorithm. https://password-hashing.net/submissions/specs/Centrifuge-v0.pdf, 2014.
2. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.
3. Alex Biryukov and Dmitry Khovratovich. ARGON v1: Password Hashing Scheme. https://password-hashing.net/submissions/specs/Argon-v1.pdf, 2014.
4. Matt Blaze. Efficient Symmetric-Key Ciphers Based on an NP-Complete Subproblem, 1996.
5. Justin Cappos. PolyPassHash: Protecting Passwords In The Event Of A Password File Disclosure. https://password-hashing.net/submissions/specs/PolyPassHash-v0.pdf, 2014.
6. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for Password Hashing. https://password-hashing.net/submissions/specs/RIG-v2.pdf, 2014.
7. Bill Cox. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. https://password-hashing.net/submissions/specs/TwoCats-v0.pdf, 2014.
8. Solar Designer. New developments in password hashing: ROM-port-hard functions. http://distro.ibiblio.org/openwall/presentations/New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf, 2012.
9. Markus Dürmuth and Ralf Zimmermann. AntCrypt. https://password-hashing.net/submissions/AntCrypt-v0.pdf, 2014.
10. Brandon Enright. Omega Crypt (ocrypt). https://password-hashing.net/submissions/specs/OmegaCrypt-v0.pdf, 2014.

11. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. https://password-hashing.net/submissions/specs/Catena-v2.pdf, 2014.
12. Daniel Franke. The EARWORM Password Hashing Algorithm. https://password-hashing.net/submissions/specs/EARWORM-v0.pdf, 2014.
13. Jeremi M. Gosney. The Pufferfish Password Hashing Scheme. https://password-hashing.net/submissions/specs/Pufferfish-v0.pdf, 2014.
14. Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide. https://password-hashing.net/submissions/specs/Lyra2-v1.pdf, 2014.
15. B. Kaliski. RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000.
16. Evgeny Kapun. Yarn password hashing function. https://password-hashing.net/submissions/specs/Yarn-v2.pdf, 2014.
17. Mikhail Maslennikov. PASSWORD HASHING SCHEME MCS_PHS. https://password-hashing.net/submissions/specs/MCS_PHS-v2.pdf, 2014.
18. Haneef Mubarak. Lanarea DF. https://password-hashing.net/submissions/specs/Lanarea-v0.pdf, 2014.
19. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
20. Alexander Peslyak. yescrypt - a Password Hashing Competition submission. https://password-hashing.net/submissions/specs/yescrypt-v0.pdf, 2014.
21. Krisztián Pintér. Gambit – A sponge based, memory hard key derivation function. https://password-hashing.net/submissions/specs/Gambit-v1.pdf, 2014.
22. Thomas Pornin. The MAKWA Password Hashing Function. https://password-hashing.net/submissions/specs/Makwa-v0.pdf, 2014.
23. Teath Sch. Tortuga – Password hashing based on the Turtle algorithm. https://password-hashing.net/submissions/specs/Tortuga-v0.pdf, 2014.
24. Steve Thomas. battcrypt (Blowfish All The Things). https://password-hashing.net/submissions/specs/battcrypt-v0.pdf, 2014.
25. Steve Thomas. Parallel. https://password-hashing.net/submissions/specs/Parallel-v0.pdf, 2014.
26. Rade Vuckovac. *schvrch*. https://password-hashing.net/submissions/specs/Schvrch-v0.pdf, 2014.
27. Hongjun Wu. POMELO: A Password Hashing Algorithm. https://password-hashing.net/submissions/specs/POMELO-v1.pdf, 2014.