Password Hashing Competition - Survey and Benchmark

George Hatzivasilis[1], Ioannis Papaefstathiou[1], and Charalampos Manifavas[2]

[1] Dept. of Electronic & Computer Engineering, Technical University of Crete, Chania, Crete, Greece
gchatzivasilis@isc.tuc.gr, ygp@mhl.tuc.gr
[2] Dept. of Informatics Engineering, Technological Educational Institute of Crete, Heraklion, Crete, Greece
harryman@ie.teicrete.gr

Abstract. Password hashing is the common approach for maintaining users' password-related information that is later used for authentication. A hash for each password is calculated and maintained at the service provider end. When a user logins the service, the hash of the given password is computed and contrasted with the stored hash. If the two hashes match, the authentication is successful. However, in many cases the passwords are just hashed by a cryptographic hash function or even stored in clear. These poor password protection practises have lead to efficient attacks that expose the users' passwords. PBKDF2 is the only standardized construction for password hashing. Other widely used primitives are bcrypt and scrypt. The low variety of methods derive the international cryptographic community to conduct the Password Hashing Competition (PHC). The competition aims to identify new password hashing schemes suitable for widespread adoption. It started in 2013 with 22 active submissions. Nine finalists are announced during 2014. In 2015, a small portfolio of schemes will be proposed. This paper provides the first survey and benchmark analysis of the 22 proposals. All proposals are evaluated on the same platform over a common benchmark suite. We measure the execution time, code size and memory consumption of PBKDF2, bcrypt, scrypt, and the 22 PHC schemes. The first round results are summarized along with a benchmark analysis that is focused on the nine finalists and contributes to the final selection of the winners.

1 Introduction

Poor password protection practices [1] have been exploited by attackers, with mounts of user passwords being exposed [2, 3]. Simple use of cryptographic functions is not enough. More advanced password hashing schemes (PHS) have been proposed, like the PBKDF2 [4], bcrypt [5] and scrypt [6].

However, the evolution of parallel computing and dedicated hardware devices empower attackers to perform more efficient attacks [7]. Password crackers on GPUs, FPGAs and ASICs try out several attempts in parallel, gaining a significant boost in disclosing the user information. The widely used PBKDF2 and bcrypt are vulnerable to such attacks.

The trend to defend these attacks are memory-hard PHSs. The memory is bounded on parallel platforms as every parallel component must have access to it. Moreover, the memory elements on dedicated hardware implementations are considered expensive. Thus, the number of parallel attempts that the password cracker can perform is significantly reduced when a PHS with high memory requirements is countered. The defender adjusts the hash iteration count and memory requirements to design secure schemes. The goal is that the password scrambling on parallel cores isn't much faster than it is on a single core. scrypt adopts this strategy. However, it is vulnerable to other types of attacks, like cache-timing [8] and garbage-collector attacks [9].

This limited set of available solutions lead the international cryptographic community to announce the Password Hashing Competition (PHC) [10] in 2013. The goal is to attract researchers in deploying modern and secure schemes for password hashing. An overview of the 22 initial candidates is presented in [11]. In 2014, 9 finalists are selected based on security, efficiency, and simplicity criteria and the extra features that they provide. A small portfolio of about 5 schemes will be announced in the second quarter of 2015 based on further performance and security analysis. They are expected to become "de facto" standards and be further examined by NIST [12] and other organizations for formal standardization.

This paper provides a survey and benchmark analysis of the 22 PHC submissions and the 3 current solutions for password hashing. Implementations of all 25 schemes are evaluated on the same platform and the results are summarized on an unitary benchmark table. At present, this is the only performance evaluation of the PHC proposals.

2 Background Theory

2.1 Passwords

Passwords are user-memorable secrets [13], consisting of several printable characters. They constitute the main mean for authentication in computer systems. For each active user account, a service provider maintains a pair of the user's identity and his secret password. The user inputs this information to login the service (e.g. [14–16]).

Passwords are also utilized in order to generate cryptographic keys. Key Deviation Functions (KDF) [17] parse a password to derive one or more keys that are related with this input password. The keys are then used on cryptographic operations, like the encryption of session communication [18].

A password of 8 characters (8 bytes) is the ordinary option. Such user-originated secrets may suffer from low entropy and be vulnerable to attacks. Attackers launch exhaustive search attacks by trying out all character combinations until they found the right password for a user account. Then, they own the account as the legitimate user does. Newer trends of user-drawn graphical passwords also exhibit low-entropy properties, offering an average security of 4-5 bytes [19].

Key stretching is the typical method for protecting such attacks. Usually, hash functions process the password and produce a fix-length output, which now acts as the password. The result is longer than the original password (e.g. 32 or 64 bytes), making the attacks less feasible. The hashed password is further fortified by iterating the hash function several times. Thus, the attacker is slow down by a factor of 2^{n+m}, where n is the number of the iterations and m is the number of the output bits. However, the user is also slowed down. The parameters of key stretching are bounded by the user's tolerance to compute a robust hash password.

As we aforementioned, the user-related information is maintained by service providers. In services where a high volume of users have to be verified simultaneously the load on the server may become unmanageable. Server relief (SR) protocols are established between the clients and the server to balance the total effort. The clients perform part of the PHS computations (e.g. some of the PHS iterations) while the server performs the final steps and the account verification.

The service provider could decide to increase security (e.g. increase the PHS iterations or the hash size). Hash password upgrade independent from the user (HUIU) is an imperative feature of a candidate PHS to enable the seamless operation of the service and the convenience of the user. The provider upgrades the security of the stored hash passwords without the user's involvement and knowledge of the password.

When two or more users have the same password, they result the same hashed password too. The disclosure of this information for one of these users could erase security issues for the rest ones. The problem is exponentially evolved as many users utilize the same password in different services. To prevent the correlation of hashed passwords that are created by the same password, a small parameter of random bytes, called salt (usually 8 bytes long), is utilized. The salt is commonly generated when the user account is created and is concatenated with the password during hashing. Thus, the same password produces different hashes for different users or services. Normally, it is stored in plaintext along with the hashed password. The authentication procedure uses the salt to validate the password of a login request. Figure 1, illustrate the generic PHS.

2.2 Applications

Password hashing and key deviation are applied in many domains. Different applications exhibit diverse features and properties that the candidate scheme must comply with. PHSs are mainly used in general applications on mainstream computers, web applications and embedded systems.

General password hashing applications on mainstream computers process infrequent and low volumes of authentication data. As computers, like PCs, have sufficient computational resources, the main goal is to achieve the highest level of security. Both PHS and KDF are important. The user can further improve the security by applying tools that detect low-entropy or widely-used passwords [20].

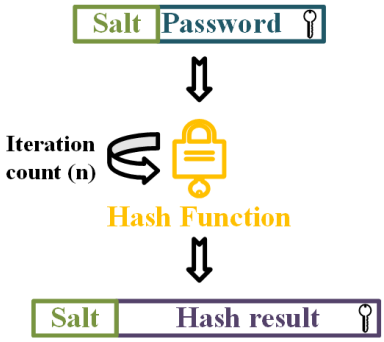

Fig. 1. PHS generic scheme.

Web applications serve thousands of users. The server maintains the authentication data for all users and must respond to high volumes of simultaneous login requests from clients. The authentication process must be fast enough to comply with the communication protocol requirements and the client's tolerance. In terms of security, the overall computational demands of handling a single login request must enable the uninterrupted service of many clients while avoiding DoS attacks. The SR functionality can provide a good tradeoff among the client and server, accomplishing the aforementioned performance and security features. Successful attacks on the web service infrastructure expose mounts of user-related data, including passwords. Usually, users own a few low-entropy passwords that are utilized in different web applications [21]. Online dictionary attacks with login histories are also common [22]. Thus, security concerns arise not only for the vulnerable services but for other applications which are used by exposed users as well. This cause motivates PHC and derive PHS as the main goal of a web application. The analysis of the stolen hashed passwords is further fortified by HUIU functionality, where the service provider periodically upgrades security to adapt to the technology evolution and the increasing computational capabilities of the attacker.

Embedded systems are deployed in a wide range of domains including pervasive and ubiquitous computing [23, 24]. The resource-constrained nature of the underlying devices and the persistent need for smaller size and lower production costs derived efficient lightweight implementations. Security is just a part of the whole functionality and becomes prominent issue. Security primitives must contribute with lightweight designs that consume low computational resources and memory. The most constrained devices, like sensors, devote only a few bytes or KBs of memory to provide moderate level of security [25, 26]. Typical embedded systems maintain a small amount of authentication-related data, in contrast to web applications. Device-to-device short-term communication is the most common interaction (e.g. in wireless sensor networks) [27], making session key deviation a desirable goal to enhance security. Garbage-collector attacks [9] are countered be build-in memory safety techniques for embedded applications [28].

3 Mainstream Password Hashing Schemes

Hash-based schemes constitute the most common solution for password protection. KDFs that produce one or more secret keys from a secret value (e.g. password) are usually implemented by secure cryptographic hash functions or HMACs. PBKDF2, bcrypt and scrypt are currently the widely-used PHSs and KDFs for mainstream applications.

3.1 PBKDF2

The Password-Based Key Derivation Function 2 (PBKDF2) [4], which is included in the RSA Laboratories' Public-Key Cryptography Standards (PKCS) series (PKCS #5 v2.0) [29] and the RFC 2898 [30], constitutes the only standardized scheme. PBKDF2 uses HMACs and takes as input the password and a salt. The salt hardens dictionary (try hundreds of likely possibilities to determine the secret) [8] and rainbow table attacks (ability to use tables of precomputed hashes) [9, 31] and it must be up to 8 bytes. The data is processed several times to produce the derived key. The standard was established in 2000 and recommended a minimum of 1000 iterations, but today this is not considered adequate. One drawback is its implementation as a small circuit with low RAM. This fact enables cheap brute-force attacks on ASICs and GPUs.

3.2 bcrypt

bcrypt [5] is a KDF and is based on the block cipher Blowfish [32]. It is the default PHS of the BSD operating system. The password is up to 56 bytes and the produced hash value is 24 bytes. The iteration count is a power of 2. It is increased to counter brute force attacks due to the increasing computation power of the attackers. The 16 byte salt defends against rainbow table attacks. bcrypt uses 4KB RAM and is slightly stronger than PBKDF2 in defending attacks on ASICs and GPUs. Still the memory requirements permit efficient attacks on FPGAs.

3.3 scrypt

In 2012, scrypt [6] was announced as an Internet Draft by the IETF with the intention to become an informational RFC [33]. PBKDF2 and the stream cipher Salsa [34] are internally utilized. scrypt uses arbitrarily large amounts of memory and is the most resistant KDF to such attacks. It is estimated that the cost of a hardware brute force attack is around 4000 and 20000 times larger than in bcrypt and PBKDF2 respectively. However, the huge memory requirements can be exploited by denial-of-service attacks on servers, while handle frequent login requests.

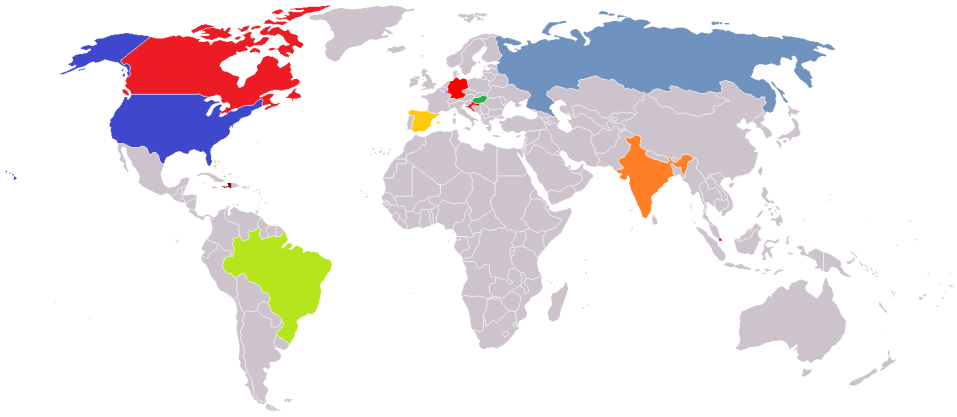

Fig. 2. PHC globe.

4 Password Hashing Competition

Password Hashing Competition (PHC) [10] evaluates 22 new PHSs [35] in terms of security, performance and flexibility. The candidates are: AntCrypt, Argon, battcrypt, Catena, Centrifuge, EARWORM, Gambit, Lanarea DF, Lyra2, MAKWA, MCS_PHS, ocrypt, Parallel, PolyPassHash, POMELO, Pufferfish, Rig, schvrch, Tortuga, TwoCats, Yarn and yescrypt. Two more PHSs (Catfish and M3lcrypt) were submitted but withdrawn before the evaluation process. All the 22 PHC candidates are detailed in the following subsections. Figure 2, illustrates the 11 countries with submissions in PHC (USA 8, Russia 3, Germany 2, Brazil 1, Canada 1, Croatia 1, Haiti 1, Hungary 1, India 1, Luxembourg 1, Singapore 1, Spain 1).

The competition advances our knowledge in designing secure and efficient PHSs. After the first round evaluation, 9 state-of-the-art finalists are announced for password protection.

4.1 Technical Guidilines and API Compliance

All PHC submissions are available worldwide on a royalty free basis with no patent or patent application covering their use or implementation. They take as input parameters at least:

- A password of any length between 0 and 128 bytes regardless of the encoding
- A salt of 16 bytes
- An output size of up to 32 bytes
- One or more cost parameters that tune time and/or space requirements

A reference implementation of each submission is provided in C/C++ that complies with the PHC API:

$$\begin{gathered} int\, PHS(void * out,\ size\_t\, outlen,\ const\, void * in,\ size\_t\, inlen, \\ const\, void * salt,\ size\_t\, saltlen, \\ unsigned\, int\, t\_cost,\ unsigned\, int\, m\_cost); \end{gathered} \tag{1}$$

where t_cost and m_cost parameterize time and memory usage. Optional implementations are submitted to other languages, like Python, PHP, or given CPU/GPU and micro-architectures. Also, a comprehensive set of test vectors is included in each code submission.

4.2 Evaluation Criteria

All candidates were evaluated in terms of security, simplicity and offered functionality.

A secure PHS should behave as a random one-way function providing features like random-looking output, collision resistance and immunity to length extension. It must be secure against known attacks and analysis on current PHS (e.g. cryptanalytic attacks and time-memory tradeoffs) and prevent speed-up or other improvement of optimized crackers on parallel platforms (e.g. multi-core CPUs, GPUs, ASIC, FPGA). No information is leaked on a password's length and resilience to side-channel attacks, like timing attacks and leakages, is guaranteed. Protection against fully or partially compromised servers should also be considered.

The elegance and clarity of the design, like modularity and symmetry, are induced to corroborate the security analysis and enhance admissibility. The scheme should be easy to understand and implement for coding, testing, debugging or integrating in existing systems. Simple and sound algorithms are appropriate. The encapsulation of other known primitives is desirable but should not be extensive. A decent submission should include qualitative documentation regarding the scheme and the reference implementation. Original and novel proposals are also expedient.

Except from the core password hashing functionality the submissions are evaluated in terms of additional functionality. The appropriate PHS should act as a secure KDF too. Operations like SR and HUIU are deliberated. The interactivity of the cost parameters for time and memory should be effective enough and the expected requirements could not be bypassed by attackers. Schemes that excel in specific applications, like web service authentication, client login, key deviation or embedded devices, are examined.

4.3 Finalists

The nine finalists that are selected among the 22 initial candidates are: Argon, battcrypt, Catena, Lyra2, MAKWA, Parallel, POMELO, Pufferfish and yescrypt. The high diversity among the algorithms is beneficial as the different

PHSs are suitable for a wide range of password hashing applications (e.g. KDF, client login). Four to five winners will be announced in the second quarter of 2015.

Argon. Argon is a memory-hard and secure hash function, optimized for security, clarity, and efficiency. It can be used for password hashing, key derivation and any other memory-hard operation. The main feature is the protection against tradeoff attacks by imposing a significant penalty on the running time of the algorithm for any reduction of the available memory by the attacker. Argon utilizes XORs, block permutation, and the round function of the block cipher AES [36] with a 5-round 128-bit fixed-key. It applies a t-byte permutation and the efficiency is linear to the memory and time parameters. Memory can occupy any number of kilobytes and is not bounded to power of 2 values only. t_cost represents the time that can be spent by the authentication application. Each building block is motivated by a certain goal and the overall design can be easily understood and analysed. On the server side, Argon offers SR and HUIU.

Argon follows the design strategy of building cryptographic hash functions and block ciphers. The memory is filled and a sequence of identical rounds processes the data. The permutation is data-dependent and is similar to the stream cipher RC4 [37] state permutation. These fast diffusion and data-dependent permutation operations provide a strong defence against memory-saving attacks. It is estimated that the penalty factor to tradeoff attacks is 50 and 150 when 1/2 and 1/4 of memory is used respectively.

The performance on modern CPUs can be improved when AES-NI assembly instruction set (NIS) [38] are used for the AES computations. Moreover, Argon can be parallelized in CPU with up to 32 threads by processing slice permutations and group transformations. However, it isn't efficient in GPUs as it extensively uses memory, which produces large latency on memory-unfriendly architectures.

battcrypt. battcrypt (Blowfish All The Things) is a simplified scrypt and targets server-side applications. It utilizes the block cipher Blowfish and the hash function SHA-512 [39] to achieve password hashing. Blowfish is used in CBC mode of operation and it is selected because it is well-studied, included in PHP, and is slow on GPUs. The overall design is considered secure even if the Blowfish is found to be broken.

The memory usage is determined only by the m_cost parameter and t_cost parameter affects only the execution time. This is a design choice which is suitable for web services where small amount of memory are used each time hashes are upgraded. battcrypt is considered faster on CPUs than GPUs. It can parallelize two or three Blowfish calculations on CPUs which would approximately double the speed. Moreover, battcrypt supports SR when the salt and setting are public.

In comparison to bcrypt that also utilizes Blowfish, battcrypt is faster as it performs about 0.752% to 1.726% less Blowfish blocks. In PHP, battcrypt processes the double amount of work done by bcrypt under equivalent settings.

Catena. Catena is designed for usage in multiple environments, like user-database backup, multi-core CPUs, and low-memory devices. It is simple and easy to analyse as it is a composed cryptographic operation based on a cryptographic hash function. Catena follows a graph-based structure called "Bit-Reversal Graph" and is instantiated by a cryptographic hash function. The defender can choose any strong hash function. The reference implementation suggests the hash functions SHA-512 and BLAKE2b [40]. SHA-512 is a standardized and widely supported hash function. BLAKE2b implements the Simple Instruction Multiple Data (SIMD) approach and provides protection against massively parallel attacks by GPUs.

The reference paper is well-documented and provides a thorough security analysis with proofs. The time-memory tradeoff analysis is based on the pebble-game approach [9]. Catena provides preimage security, indistinguishability from random, resistance against side-channel (e.g. cache-timing attacks), and lower bounds on the time-memory tradeoff. It produces a high computational cost for massively parallel crackers in GPUs, ASICs and FPGAs.

Catena supports HUIU by increasing the t_cost and m_cost parameters. SR is achieved by enabling the client to compute most of the iterations while the server computes only the last one. It can also operate in a keyed password hashing mode by XORing the output of the unkeyed Catena with the hash of the user ID, the m_cost and the secret key.

Lyra2. Lyra2 utilizes hash functions with sponge structure to enable password hashing. It improves the previous version of Lyra2 and is simple to implement in software. The design is strictly sequential to counter attacks by multi-core platforms. The reference version uses the hash function BLAKE2b in a duplex sponge structure. The memory is organized as a large matrix which must be stored in RAM during the whole password hashing process. Moreover, an optional parameter, called "basil", can be used as an additional salt to avoid collisions for trivial pairs of password and salt.

The reference paper provides a thorough security analysis. Lyra2 intends to make the derived key non-invertible, keep the memory matrix in RAM during the password hashing, and prevent attackers from parallelizing the algorithm.

The legitimate user can improve the performance by storing the memory matrix in volatile memory. Although Lyra2 is strictly sequential, the version Lyra2$_p$ is proposed to allow parallel execution on multiple cores for the defender, increasing the cost of an attack. The security properties of Lyra2$_p$ are also analysed.

MAKWA. MAKWA is a PHS and KDF. It uses modular squaring and operates on big numbers, similar with the public-key cryptosystem RSA [41]. The prime factors of the modulus aren't required to execute MAKWA. If the factors are known to a trusted entity, the expensive operations are avoided and the processing effort is reduced. The NIST standardized HMAC_DRBG over the SHA-256 [42] is utilized internally as a deterministic KDF. The cost parameter affects only the execution time while the memory requirements are constant.

Its main feature is the SR procedure that allows the bulk of processing cost to be offloaded to an external untrusted entity. This operation is called delegation and resembles the blind signatures and the RSA private key operation. Moreover, MAKWA supports offline hash upgrade and password escrow.

The security analysis on GPUs, FPGAs and ASICs resembles the feasibility of performing attacks on RSA. MAKWA is resistant to similar attacks and fulfils its design goals.

Parallel. Parallel password based key derivation function (PPBKDF) is design for applications with low memory. The design is compact with constant and low memory requirements. It isn't a memory-hard function and is optimized for parallel execution. It is simple and as collision resistant as the internal hash function. The reference implementation utilizes the SHA-512.The attacker-defender ratio is considered 1 with any advancements in cracking are advancements for the defender too.

POMELO. POMELO has simple design and is easy to implement. It operates on 8-byte words and uses three state update functions. The first function is a simple non-linear feedback function and the other two functions provide simple random memory access over data. These tree functions protect POMELO against preimage, low memory and SIMD attacks. The memory size and the computational complexity are adjusted by the m_cost and t_cost parameters respectively. POMELO is efficient even when large state is used and retains security against cache-timing attacks and attacks by GPUs and dedicated hardware. Moreover, it supports HUIU in a straightforward manner.

Pufferfish. The Pufferfish PHS, utilizes Blowfish and HMAC-SHA-512 and is based on bcrypt. The Blowfish cipher is extended in order to use 8-byte words and dynamic, arbitrarily-sized and password-dependent S-boxes. The extended cipher is applied on the bcrypt algorithm with minimal modifications. Moreover, it supports variable-length output and functions as a KDF.

The m_cost parameter increases the memory requirements by log_2 KB and the t_cost parameter increases complexity by log_2 iterations. On CPUs, Pufferfish is faster than bcrypt while on GPUs it is slower. Also, it inherits the GPU resistsnce of bcrypt.

The cryptographic properties of the extended Blowfish that is implemented by Pufferfish haven't been validated. However, it is considered that Pufferfish retains its security properties while HMAC and SHA-512 remain unbroken.

yescrypt. yescrypt is built upon scrypt and tweaks its security. The overall functionality is managed via bit flags. A set of these flags offers a mode of operation that is compatible with scrypt.

In contrast to scrypt, yescrypt supports more parallelism options at the thread- and instruction- level. It extensively uses SIMD and the reference im-

plementation include OpenMP support [43]. Moreover, a modified ROMix algorithm is proposed which can make use of an optional pre-filled read-only lookup table (ROM) along with the usual sequential-wite, random-read lookup table (RAM) of the original scrypt. Most of the Salsa20/8 computations that are used by scrypt can be replaced by the yescrypt's custom pwx-form algorithm, while the two primitives can be also inter-mixed.

The computational complexity can be increased while keeping the memory requirements constant. As with scrypt, the security is based on the HMAC, SHA-256 and PBKDF2 primitives. yescrypt enhances security against attacks on GPUs, FPGAs and ASICs.

4.4 Non-finalists

The 13 non-finalists are: AntCrypt, Centrifuge, EARWORM, Gambit, Lanarea DF, MCS_PHS, ocrypt, PolyPassHash, Rig, Schvrch, Tortuga, TwoCats and Yarn. Non-selection does not impose that these schemes are found less suitable than the finalists. Many of the non-finalists are less mature, analysed and understood, receiving less support by the panel. Some of them contribute original ideas and can be suitable for practical applications.

AntCrypt. AntCrypt takes its name from the anthill construction. It separates the computational effort from the cryptographic hardness and provides a clear and well-motivated design. The computational overhead is based on a set of functions with floating point arithmetic and the security on a secure hash function.

The computational functions are called once and the execution order is determined by the inner state. After all functions process the data, the results are XORed. They affect the branch divergence factor and harden the parallel execution for an attacker (e.g. on GPUs). The floating point arithmetic is chosen as its implementation is costly on FPGAs and ASICs. The selection of the proper functions depends on several factors, like the attacker's architecture, and the current set is likely to change in future versions.

The security of the current version is based on the SHA-512 hash function, which is extended in order to adapt a larger output size. SHA-512 is widely accepted and implemented by cryptographic libraries and its security is proven and tested over several years. The choice is not restricted and can be substituted by any hash function with sufficient large state and output size. In AntCrypt, the hash function hashes the inner state and determines the entropy.

AntCrypt supports the password and output sizes that are requested by PHC. The salt is fixed at 16 bytes. The m_cost parameter determines the rounds of the inner loop and the amount of memory that is consumed by the state. The t_cost parameter defines the iterations of the outer loop.

AntCrypt provides hash upgrade within certain constraints. If the intermediate hash is stored, the strength can be increased by increasing the t_cost parameter and resuming the computation from this point. The knowledge of the password isn't required while the m_cost parameter is fixed.

Moderate memory resources are consumed starting from 256 bytes, with 32KB suggested as a reasonable choice. AntCrypt restricts the parallel execution for an attacker on CPUs, GPUs, FPGAs and ASICs while it can provide low parallelism for legitimate users on GPUs (for the execution of the computational functions).

AntCrypt is not selected in the finalists list due to the usage of float-point arithmetic and data-dependent branching. Float-point arithmetic reduces portability and reproducibility. The data-dependent branching that is not sufficiently analysed in the current security information theory and the the overall design imposes more risks than benefits.

Centrifuge. Centrifuge consists of a block cipher acting as a pseudorandom number generator (PRNG), a hash function, a large table of pseudorandom sequences, and a substitution box. For the reference implementation, the AES-256 block cipher is utilized in CFB mode of operation. The initialization vector of the cipher is produced by the password and the salt. The SHA-512 hash function seeds the PRNG. The large table is combined to form the resulting password hash. The 8x8 S-box adds complexity and prevents parallelization. Centrifuge is configurable in byte increments and operates with variable-length password, salt and output.

For PHC, the t_cost and m_cost parameters must be smaller than 64 and are interpreted as the exponents of the relevant powers of 2. The original algorithm can use higher values than 64.

Centrifuge benefits from NIS on modern CPUs. The AES in CFB mode is selected in order to prevent the parallelization of the sequence generation in GPUs, ASICs or FPGAs.

However, Centrifuge is not included in the finalists. The byte-grained random memory accesses results in a too slow implementation for the concrete memory usage.

EARWORM. EARWORM intends to provide password hashing security at low time-cost and targets server-side applications. It utilizes the AES encryption round function to process the data and the scheme PBKDF2-HMAC-SHA-256 to produce the output. Arena, a large pre-initialized array, is taken as input, imposing arbitrarily high memory-cost with arbitrarily low time-cost. Arena's data are read-only and are initialized randomly. They are used as the round keys of AES. The security is based on this high demand of memory bandwidth, known as ROM-port hardness. On the contrary, the large ROM requirements makes EARWORM impractical for constraint platforms.

EARWORM satisfies the design goals of cryptographic one-way functions but it is not suitable for key derivation. Dictionary attacks should be the most efficient way for preimage attacks. It is not resistant to second-preimage or collision attacks and it is not a sequential memory-hard function.

On CPUs, EARWORM can be parallelized on multiple cores and can be benefited from NIS. On GPUs, it produces high latency due to the high memory

requirements. FPGAs lack of sufficient I/O bandwidth and aren't viable means of attacking EARWORM.

EARWORM is not a finalist. It is not second-preimage resistant and does not constitute a secure KDF. The security is based on a complex ROM-hardness approach. Other similar and better understood techniques that are affordable in the targeted server-side applications should be considered instead. The multiple external primitives that are utilized were taken as a drawback.

Gambit. Gambit utilizes hash functions with sponge structure and large state. It is not intended to win the competition and is an attempt to draw attention on some of its techniques and design principles. The reference implementation embodies the Keccak[1600] hash function [44]. Keccak is the winner of the SHA-3 competition and the new hash function standard. Gambit also incorporates a keyfile in ROM. The file increases the cost for an attacker (especially when specialized hardware is used) and can be utilized as a second factor of authentication, representing something that the client "has". Moreover, it supports generation of multiple keys and an easy SR procedure. The design conforms to the Crypto Coding Standard [45] recommendations and is considered resilient to cache timing attacks. Gambit is sequential in nature and only the internal sponge hash function can offer some degree of parallelization.

As it is presumed, Gambit did not pass to the first evaluation phase. It exhibits a similar but less mature design than the finalist Catena. Also, it is less resistant to attacks that exploit ASIC due to the usage of Keccak.

Lanarea DF. Lanarea Derivation Function (DF) is a heavily serialized KDF. The main design goal is to harden the parallel execution on GPGPUs or ASICs. The algorithm is initialized by a simple process and then a convoluted scheme is applied to rearrange data in the main loop. Lanarea utilizes a pseudorandom function (PRF) to produce the output key, which is multiple of 32 bytes length. The reference implementation uses the BLAKE2b as the PRF.

The security is mostly based on the PRF. BLAKE2b has been thoroughly verified and is considered safe. Lanarea is secure for server-client and system level authentication, and stream and long term encryption.

As with Centrifuge, it is rejected as it is too slow for the specific memory usage.

MCS_PHS. MCS_PHS is a PHS and KDF. The core of the scheme is the hash function MCSSHA-8 [46]. For deriving encryption keys, MCS_PHS uses the KDF PBKDF_MCS [47] – a simple algorithm based on PBKDF and MCSSHA-8.

The security of MCS_PHS is strongly determined by MCSSHA-8. The MCSSHA hash function family was candidate on the SHA-3 competition and independent security analysis was conducted. MCSSHA-8 is the latest version and the most secure one.

The required memory is about 1KB and the complexity is mainly depends on the t_cost parameter that specifies the iteration count. The performance of MCS_PHS is comperable to PBKDF2 with SHA-1 [48].

It is rejected as the design is similar with PBKDF2 but with no significant improvement. Also, the security of the underlying function MCSSHA-8 is not clear with early versions being broken.

Omega Crypt. Omega Crypt (ocrypt) is a memory-hard function for password hashing and key derivation. It realizes data-dependent branching and very wide SIMD architectures. The hash function CubeHash [49] hashes the input parameters and derives a key. The stream cipher ChaCha [50] is then instantiated with this key and initializes a large block of memory. In each iteration, the output of ChaCha derives one of several branches that manipulate the memory block. Cudehash processes the data in the final step and outputs the result.

CudeHash is simple and flexible while ChaCha is simple and fast. The overall structure of ocrypt provides more computational and memory difficulty than the two individual primitives.

The proposal also considers the attacks that can exploit the SIMD architecture, like side-channel attacks on GPUs, and avoids them. It is configured in execution time and memory usage providing many of the anti-ASIC and anti-FPGA properties of scrypt.

As with AntCrypt, data-dependent branching is considered a disadvantage along with partially predictable branches and memory addresses. The security and performance analysis are inadequate and not convincing.

PolyPassHash. PolyPassHash applies a PolyHashing scheme to provide protection above PHS. On systems that employ simple PHS, an attacker can obtain the stored hash passwords and analyse them independently. With PolyHashing the attacker must obtain a threshold of passwords before he becomes capable in verifying stolen hashes. Then, the PHS protection must be also overcame.

PolyPassHash is composed of two building blocks: a threshold scheme for PolyHashing and a salted hash function. The reference implementation suggests the Shamir Secret Sharing (SSS) [51] as the threshold scheme and the SHA-256 as the hash function. The computational complexity of SSS is based on the k degree polynomial over a finite field. The default k value for PolyPassHash is 3. The standardized SHA-256 is used to hash the password and the salt. The hashes are encrypted with the AES.

The overall scheme process the password file at the server-side when the system restarts. Then,a threshold of users must login before the passwords can be verified. After this step, the login requests are served with similar computational overhead as for PHS-only systems. The storage cost is one byte per user account. The memory overhead is about 1KB independent of the number of passwords. Moreover, PolyPassHash suggests an alternative partial verification process that allows users to login immediately after a restart without this need of verifying a threshold of users.

PolyPassHash can be applied on current applications without modification of the client applications or the login process. It is based only one software and the system administrator can change the threshold factor without affecting the users.

An attacker must guess 3 passwords simultaneously. On GPUs, this fact imposes about 23 orders of magnitude more effort than on PHS-only systems. On CPUs, even a threshold of 2 secrets would provide sufficient security.

PolyPassHash is actually a protocol that recovers a symmetric key used to encrypt passwords and does not constitute a PHS. The offered functionality is out of the competition's scope.

Rig. Rig is a memory-hard and strictly sequential PHS. The design is simple, flexible in adjusting the memory usage, and easy to implement in software. It functions on 8-byte words and performs bit-reversal permutation techniques independent of the password. Rig executes a cryptographic hash function repeatedly, with Blake2b being proposed for the reference implementation.

It supports HUIU and SR. For a fixed number of internal iterations, the hash upgrade is achieved by increasing the number of rounds. Each round takes as input the output of the previous round and consumes the double memory. For the SR protocol, the server sends the salt to the client, who then bears the maximum effort to calculate the hashed password. The server calculates the final hash by concatenating the salt, a counter value, and the result sent by the client.

The scheme mimics the Random Oracle Model for providing theoretical justification of the security. Rig counters cache-timing and memory-free attacks as well as DoS attacks on server.

There is little scope for parallelism as the update of each iteration depends on the calculations of the previous hash operation. This makes Rig inefficient in dedicated hardware platforms that try to exploit multiple computational units in parallel. Moreover, the high demands in memory is expensive in such platforms.

Although its design is similar to the finalist Catena, Rig received less attention by the selecting panel. The submission is less qualitative with errors and bugs being found in the specification and code.

Schvrch. Schvrch is a novel PHS and proposes a new class of functions for modelling hash functions. The scheme uses three functions, separating the slow and big computations with an extra round for more slow down. Schvrch operates as a Cellular Automata (CA). The security analysis of the software implementation is based on cyclomatic complexity special case [52] – a software testing metric.

However, Schvrch is cryptographically weak and ASIC-friendly. The total cost is based on the addition of time and memory features.

Tortuga. Tortuga is design for key stretching applications. It is based on a sponge function with a recursive Feistel network for permutation, called Turtle algorithm [53]. A block cipher with an enormous key is created on the fly. Tortuga

can also work as a memory-hard KDF. The basic principles that are proposed by Totruga have neither been tested in practice nor been subjected to independent analysis.

Therefore, it is rejected in the first round evaluation as it fails in basic statistical tests.

TwoCats. TwoCats is a compute-time and sequential memory-hard PHS. It uses a two-loop architecture, as scrypt, where the first loop performs password-independent memory addressing while the second loop performs password-dependent memory addressing at an early stage. Thus, TwoCats supports thread- and instruction-level parallelism. Moreover, the scheme utilizes integer multiplication which is fast in GPUs and enhances the overall multiplication chain hardening. The scheme can operate with the Blake2b, Blake2s, SHA-256 or the SHA-512 as the internal hash function. TwoCats can operates as a KDF and supports HUIU and SR.

Three APIs are implemented to enhance usability. The APIs take different input parameters to support users with basic to advanced password hashing knowledge. SkinnyCat is a forth API, which implements a simpler subset of the TwoCats.

The scheme is secure against cache-timing and time-memory tradeoff attacks. Also, it provides strong defence against attacks on GPUs, FPGAs and ASICs.

The scheme performs well both on 32- and 64-bit platforms. It is suitable for PC, web server, mobile and embedded applications. The defender can be also benefited for parallel execution both by the multi-threading and SIMD capabilities of TwoCats.

It is an interesting mix of ideas. However, the overall design is less understood than other competing schemes, provoking its rejection.

Yarn. Yarn is a memory-hard PHS designed for x86 processors. The design is simple and compact. The AES round encryption function from NIS is utilized in order to implement an AES-like component. Blake2b is also used for compressing data and producing hashes. Yarn consists of five phases where the AES-like component and the Blake2b hash function process the data to produce the final hash. The scheme performs repeatable and sequential memory lookups. The AES computations and the memory lookups can be performed in parallel. The salt is optional and an additional parameter can be provided to adjust the level of internal parallelism. It deliberately does not use multi-threaded parallelism, as it would inquire a higher level primitive for computing parallel Yarn functions and combining the results.

The security is based on Blake2b properties and the AES permutation. The sequential computations affect the performance which is determined by the memory latency. The computation demands of AES along with m_cost parameters that consume large memory, derive Yarn inefficient on GPUs, FPGAs, and ASICs.

Yarn's design is less mature and friendly-tuned than other relevant schemes and it is not including in the finalists.

5 Evaluation

5.1 General features

We analyse the general properties of the examined PHSs and discuss their main features. Table 1 summarizes the design details of PBKDF2, bcrypt, scrypt and the 22 PHC submissions based on their main cryptographic primitives, the internal memory requirements and the offered functionality.

Symmetric ciphers and hash functions constitute the main cryptographic primitives for a PHS. For block ciphers, AES and Blowfish are utilized by five and three schemes respectively. For stream ciphers, two schemes use Salsa and one ChaCha. For hash functions, SHA256 and SHA512 are the most common choices with seven PHSs for each function. Blake2b also gains significant attention with 5 schemes adopting the hash function while SHA1 is used by three schemes. Each of the functions Keccak, Blake2s, CudeHash and MCSSHA-8 is presented by one scheme. Other cryptographic primitives that are utilized by novel proposals are the Shamir Secret Sharing, the cellular automata and the turtle algorithm. The PHC finalists relay on the well-studied and understood AES, Blowfish, Salsa, SHA256, SHA512, Blake2b and SHA1.

Regarding the memory requirements, 7 schemes use less resources than bcrypt (4KB) and only 6 schemes use high volumes of memory as scrypt (more than 1GB). From the nine PHC finalists, MAKWA, Parallel and yescrypt require neglected memory, battcrypt, Catena and Pufferfish take a few KBs-MBs of memory to operate, and Argon, Lyra2 and POMELO can use GBs of memory.

PHS and KDF constitute the main goal of the competition. Although a finalist, POMELO does not operate as KDF. All other finalists operate both as PHS and KDF, providing SR functionality as well. Argon, Catena, Lyra2, Parallel and POMELO implement HUIU. The finalist MAKWA provides additionally offline hash upgrade and password escrow. Other notable functionality that is offered by non-finalists include the multiple keys deviation of Gambit and the poly-hashing scheme of PolyPassHash.

The one third of the finalists is originated from USA. Luxembourg, Germany, Brazil, Canada, Singapore and Russia hold from one active finalist each.

PHS	**Cryptographic Primitives**	**Internal Memory**	**PHC Functionality**	**Additional Functionality/Features**	**Country**
PBKDF2	HMAC–SHA1	negl.	PHS, KDF	-	USA
bcrypt	Blowfish	4KB	PHS, KDF	-	USA
scrypt	PBKDF2, Salsa20/8	1GB	PHS, KDF	-	Canada

Finalists					
Argon	AES128	1KB - 1GB	PHS, KDF	SR, HUIU	Luxembourg
battcrypt	Blowfish-CBC, SHA512	18KB - 128MB	PHS, KDF	SR	USA
Catena	Blake2b / SHA512	8MB	PHS, KDF	SR, HUIU	Germany
Lyra2	Blake2b	400MB - 1GB	PHS, KDF	SR, HUIU	Brazil
MAKWA	HMAC_DRBG, SHA256	335KB	PHS, KDF	SR, offline hash upgrade, password escrow	Canada
Parallel	SHA512	negl.	PHS, KDF	SR, HUIU	USA
POMELO	-	1KB - 8GB	PHS	HUIU	Singapore
Pufferfish	Blowfish, HMAC-SHA512	4KB - 16KB	PHS, KDF	SR	USA
yescrypt	scrypt	44KB - 3MB (RAM), 3GB (ROM)	PHS, KDF	SR	Russia
Non-finalists					
AntCrypt	SHA512	32KB	PHS, KDF	HUIU, floating point operations	Germany
Centrifuge	AES256-CFB, SHA512	56KB - 2MB	PHS	-	Spain
EARWORM	AES, PBKDF2-HMAC-SHA256	2GB (ROM)	PHS	SR	USA
Gambit	Keccak[1600]	50MB	PHS, KDF	SR, Multiple keys	Hungary
Lanarea DF	Blake2b	256 byte	PHS, KDF	SR	USA
MCS_PHS	MCSSHA-8, PBKDF_MCS	1KB	PHS, KDF	SR	Russia
ocrypt	ChaCha, CubeHash	1MB - 1GB	PHS, KDF	-	USA
PolyPassHash	AES, SHA256, SSS	1KB	PHS	HUIU, PolyHashing	USA
Rig	Blake2b	15MB	PHS, KDF	SR, HUIU	India
schvrch	Cellular Automata (CA)	4KB - 8MB	PHS	-	Croatia
Tortuga	Turtle alg.	32KB	PHS, KDF	-	Haiti

TwoCats	Blake2b / Blake2s / SHA256 / SHA512	1KB - 192GB	PHS, KDF	SR, HUIU	USA
Yarn	AES, Blake2b	16 byte - 192GB	PHS	SR	Russia

Table 1: PHS details

5.2 Software implementation benchmark

PBKDF2, bcrypt, scrypt and 22 PHC proposals are evaluated under an Intel Core i7 at 2.10GHz CPU with 8GB RAM, running 64-bit operating systems. The C implementations of PBKDF2 by CyaSSL [54], bcrypt by openwall [55] and scrypt-1.1.6 by Tarsnap [56] are utilized. For the PHC proposals, the submitted reference and optimized implementations [35] are evaluated. battcrypt is installed on Ubuntu 15.04. The rest schemes are installed on Windows 8.1 Pro and are executed on cygwin. All implementations are assessed over a common benchmark suite. We measure the code size, memory consumption and execution time of each scheme. Table 2 summarizes the software evaluation of each PHS based on the default sizes for output, password and salt, and the indicative t_cost and m_cost parameters as reported by each submission.

The password is defined by the user and affects the entropy of the overall result. AntCrypt uses the shortest password of 2 bytes. In most of the PHC candidates (18 schemes), the password is set to 8 bytes as in scrypt. MCS_PHS uses 11 byte, bcrypt 12 byte, PolyPassHash 16 byte and PBKDF2 24 byte password. Argon has the longest password of 32 bytes.

The randomly generated salt prevents the correlation of hashed passwords that are originated from the same password. battcrypt has a short salt of 4 bytes. PBKDF2 doubles the size to 8 bytes. Most of the candidates (20 schemes) use 16 byte salts as bcrypt. scrypt and Argon have the longest salt of 32 bytes.

Each PHS stretches the initial password input. The larger output protects the final outcome from attacks on short and low-entropy passwords. However, the defender is also slowed down for large output sizes. The shortest output of 32 bytes is produced by the non-finalists Argon, PolyPassHash and TwoCats. bcrypt outputs 54 bytes hashed passwords. The rest 19 candidates, including all finalists, produce 64 bytes output as PBKDF2 and scrypt.

The t_cost and m_cost parameters are introduced by PHC in order to parametrize the different proposals in terms of computational and memory requirements. There is no norm in determining specific values for each parameter. The range values vary as every scheme adopts its own convention based on the inner structure.

The executable code size determines the space of consistent memory that is occupied by the software implementation and reflects to the implementation size. PBKDF2, bcrypt, battcrypt, Catena and yescrypt produce the most compact implementations making them suitable for constrained devices (less than 36KB ROM with low RAM consumption). AntCrypt, Argon, centrifuge, the optimized EARWORM, Gambit, Lanarea, Lyra2, MAKWA, ocrypt, Parallel, PolyPassHash, POMELO, Pufferfish, Rig, schvrch, Tortuga and Yarn have moderate requirements in code size (67-103KB). scrypt, the reference EARWORM, MCS_PHS, the optimized Pufferfish and TwoCats result large code sizes.

The runtime requirements of a software implementation are defined by the RAM space that is needed to operate. PBKDF2, MAKWA, Parallel and yescrypt are not RAM-hard PHSs. PBKDF2 and Parallel consume neglected RAM. MAKWA has low RAM requirements as bcrypt. yescrypt consumes low to moderate RAM (44 KB - 16.46MB) in cooperation with ROM-hard structures. From the rest RAM-hard PHSs, Gambit, Lanarea, MCS_PHS, Tortuga, TwoCats have low (less than 1MB), PolyPassHass moderate (less than 64MB) and scrypt high memory requirements. AntCrypt, battcrypt, Lyra2, Pufferfish and Yarn require low to moderate memory. Catena, EARWORM and Rig takes moderate to high RAM resources to operate. centrifuge, POMELO, ocrypt and schvrch support the full range from low to high memory usage.

Speed is a core parameter in selecting a software implementation and is strongly affected by the computational capabilities of the system. The time that it takes to compute the hashed password must comply with the communication protocol's constraints and the user's tolerance. Gambit, Lanarea, MAKWA, MCS_PHS, Parallel, PolyPassHash, Rig, Tortuga, TwoCats and yescrypt takes a few ms to operate. bcrypt and scrypt requires moderate time (a few secs). The rest PHSs, offer higher degree of adaptability. AntCrypt, battcrypt, Catena, the optimized EARWORM, Lyra2, POMELO, schvrch and Yarn requires low to moderate time. Argon, centrifuge, the reference EARWORM, PBKDF2, Pufferfish and ocrypt cover the full range from low to high execution time.

PHS	**Password (bytes)**	**Salt (bytes)**	**Output (bytes)**	**t_cost**	**m_cost**	**ROM (KB)**	**RAM (KB)**	**CPU(secs)**
PBKDF2	24	8	64	1000	0	30	0	0,002024
PBKDF2	24	8	64	2048	0	30	0	0,004150
PBKDF2	24	8	64	4096	0	30	0	0,008141
PBKDF2	24	8	64	10000	0	30	0	0,019386
PBKDF2	24	8	64	1000000	0	30	0	1,908592
PBKDF2	24	8	64	16777216	0	30	0	32,969576
bcrypt	12	16	54	12	0	27	492	2,668653
scrypt	8	32	64	5	0	182	450656	2,837654
Finalists								
Argon	32	32	32	3	2	82	192	0,008917
Argon	32	32	32	254	1	82	144	0,037167
Argon	32	32	32	236	10	82	172	0,285078

Argon	32	32	32	56	100	82	244	0,674987
Argon	32	32	32	3	1000	82	1216	0,551141
Argon	32	32	32	3	10000	82	10172	5,520845
Argon	32	32	32	3	100000	82	100168	55,457062
Argon	32	32	32	3	1000000	82	1000192	577,022082
battcrypt	8	4	64	0	0	27	18	0,000312
battcrypt	8	4	64	1	0	27	18	0,000395
battcrypt	8	4	64	1	1	27	30	0,000758
battcrypt	8	4	64	1	2	27	42	0,001984
battcrypt	8	4	64	1	3	27	74	0,004231
battcrypt	8	4	64	1	4	27	138	0,016379
battcrypt	8	4	64	1	5	27	266	0,022217
battcrypt	8	4	64	1	6	27	520	0,045737
battcrypt	8	4	64	1	7	27	1000	0,091217
battcrypt	8	4	64	1	8	27	2000	0,149676
battcrypt	8	4	64	1	9	27	4000	0,254784
battcrypt	8	4	64	1	10	27	8000	0,418699
battcrypt	8	4	64	1	11	27	16000	0,766960
battcrypt	8	4	64	1	12	27	32000	1,472554
battcrypt	8	4	64	1	13	27	64000	2,853051
battcrypt	8	4	64	2	1	27	25	0,001088
battcrypt	8	4	64	3	1	27	25	0,001617
battcrypt	8	4	64	4	1	27	25	0,002175
Catena–blake2b	8	16	64	3	18	25	16384	0,353742
Catena–blake2b	8	16	64	3	20	25	65596	2,619238
Catena–blake2b	8	16	64	3	21	25	128484	5,461030
Catena–sha512	8	16	64	3	18	13	16496	0,783590
Catena–sha512	8	16	64	3	20	13	65720	5,389355
Catena–sha512	8	16	64	3	21	13	131240	11,664960
Lyra2	8	16	64	5	5	98	44	0,000084
Lyra2	8	16	64	5	100	98	696	0,001463
Lyra2	8	16	64	5	1000	98	6104	0,015104
Lyra2	8	16	64	5	10000	98	60128	0,159651
Lyra2	8	16	64	5	131071	98	787416	2,916398
MAKWA	8	16	64	0	0	95	335	0,000096
MAKWA	8	16	64	10	0	95	335	0,000132
MAKWA	8	16	64	100	0	95	335	0,000273
MAKWA	8	16	64	1000	0	95	335	0,002035
MAKWA	8	16	64	4000	0	95	335	0,007838
MAKWA	8	16	64	8192	0	95	335	0,015621
Parallel	8	16	64	0	0	71	0	0,001000
Parallel	8	16	64	0	10	71	0	0,001018
Parallel	8	16	64	10	0	71	0	0,047020
Parallel	8	16	64	10	10	71	0	0,047051
POMELO	8	16	64	0	0	67	12	0,000031

POMELO	8	16	64	0	7	67	1064	0,003537
POMELO	8	16	64	0	17	67	1048648	7,951508
POMELO	8	16	64	7	0	67	12	0,001270
POMELO	8	16	64	7	7	67	1064	0,171375
POMELO	8	16	64	20	0	67	12	8,504152
Pufferfish–ref	8	16	64	0	0	103	156	0,000057
Pufferfish–ref	8	16	64	6	2	103	120	0,003359
Pufferfish–ref	8	16	64	6	10	103	1192	2,225718
Pufferfish–ref	8	16	64	10	10	103	1188	38,341005
Pufferfish–ref	8	16	64	6	11	103	2236	19,884300
Pufferfish–opt	8	16	64	0	0	398	220	0,000055
Pufferfish–opt	8	16	64	6	2	398	180	0,002980
Pufferfish–opt	8	16	64	6	10	398	1140	2,183917
Pufferfish–opt	8	16	64	10	10	398	1132	37,668099
yescrypt-ref	8	16	64	0	0	36	44	0,000094
yescrypt-ref	8	16	64	0	7	36	1516	0,004173
yescrypt-ref	8	16	64	0	8	36	2112	0,007209
yescrypt-ref	8	16	64	1	8	36	2112	0,008901
yescrypt-ref	8	16	64	2	8	36	2112	0,012088
yescrypt-ref	8	16	64	3	8	36	2112	0,015313
yescrypt-ref	8	16	64	0	11	36	16448	0,058253
yescrypt-opt	8	16	64	0	0	44	72	0,000116
yescrypt-opt	8	16	64	0	7	44	1148	0,003296
yescrypt-opt	8	16	64	0	8	44	2124	0,005796
yescrypt-opt	8	16	64	1	8	44	2124	0,006885
yescrypt-opt	8	16	64	2	8	44	2124	0,009448
yescrypt-opt	8	16	64	3	8	44	2124	0,011544
yescrypt-opt	8	16	64	0	11	44	16460	0,046733
Non-finalists								
AntCrypt	2	16	64	15	1	70	76	0,001850
AntCrypt	2	16	64	16	1	70	76	0,001507
AntCrypt	2	16	64	17	1	70	76	0,001621
AntCrypt	2	16	64	10	5	70	132	0,015143
AntCrypt	2	16	64	11	5	70	132	0,016656
AntCrypt	2	16	64	12	5	70	132	0,018127
AntCrypt	2	16	64	5	10	70	416	0,243313
AntCrypt	2	16	64	6	10	70	416	0,291065
AntCrypt	2	16	64	7	10	70	416	0,339220
AntCrypt	2	16	64	1	14	70	4256	0,834133
AntCrypt	2	16	64	2	14	70	4256	1,596931
AntCrypt	2	16	64	3	14	70	4256	2,368481
AntCrypt	2	16	64	1	17	70	32972	7,598387
centrifuge	8	16	64	0	0	69	56	0,000013
centrifuge	8	16	64	8	16	69	3764	0,810746
centrifuge	8	16	64	8	20	69	65672	13,173753

EARWORM–ref	8	16	64	1	12	122	16312	0,001873
EARWORM–ref	8	16	64	10000	16	122	262132	19,585622
EARWORM–opt	8	16	64	1	12	94	16324	0,000423
EARWORM–opt	8	16	64	10000	16	94	262140	1,098950
Gambit	8	16	64	16	25	81	8	0,000043
Gambit	8	16	64	128	511	81	12	0,000322
Lanarea	8	16	64	1	1	77	60	0,068024
Lanarea	8	16	64	10	10	77	188	0,113400
MCS_PHS	11	16	64	0	32	122	1000	0,001
MCS_PHS	11	16	64	1000	32	122	1000	0,029000
MCS_PHS	11	16	64	0	256	122	1000	0,001
MCS_PHS	11	16	64	1000	256	122	1000	0,026000
ocrypt	8	16	64	0	0	76	1008	0,034912
ocrypt	8	16	64	0	8	76	262184	4,845470
ocrypt	8	16	64	8	0	76	1056	4,086190
ocrypt	8	16	64	8	8	76	262184	11,992220
PolyPassHash	16	16	32	1	0	78	3412	0,000054
PolyPassHash	16	16	32	2	0	78	3412	0,000055
PolyPassHash	16	16	32	4	0	78	3412	0,000054
PolyPassHash	16	16	32	8	0	78	3412	0,000058
PolyPassHash	16	16	32	16	0	78	3412	0,000057
PolyPassHash	16	16	32	32	0	78	3412	0,000062
PolyPassHash	16	16	32	64	0	78	3412	0,000072
PolyPassHash	16	16	32	128	0	78	3412	0,000092
PolyPassHash	16	16	32	253	0	78	3412	0,000135
Rig	8	16	64	1	21	82	245776	0,531948
Rig	8	16	64	2	17	82	15416	0,044597
Rig	8	16	64	3	21	82	245824	1,156860
Rig	8	16	64	5	15	82	3896	0,011732
schvrch	8	16	64	0	0	68	4	0,007409
schvrch	8	16	64	100000	0	68	4	0,472748
schvrch	8	16	64	0	50000	68	99992	2,350343
schvrch	8	16	64	100000	50000	68	99992	2,828699
Tortuga	8	16	64	2	3	72	28	0,000052
Tortuga	8	16	64	610	987	72	32	0,001214
Tortuga	8	16	64	2584	2584	72	32	0,004607
Tortuga	8	16	64	17711	6765	72	32	0,024750
Tortuga	8	16	64	28657	89	72	32	0,032184
TwoCats	8	16	32	0	0	154	32	0,000002
TwoCats	8	16	32	0	10	154	1132	0,001221

TwoCats	8	16	32	5	5	154	40	0,000002
TwoCats	8	16	32	10	0	154	32	0,000002
TwoCats	8	16	32	10	10	154	1140	0,191255
Yarn	8	16	64	0	0	74	8	0,000031
Yarn	8	16	64	0	5	74	8	0,000138
Yarn	8	16	64	62	5	74	8	0,000167
Yarn	8	16	64	62	10	74	24	0,003398
Yarn	8	16	64	30	19	74	8040	1,719149
Yarn	8	16	64	62	20	74	16156	3,361276
Yarn	8	16	64	30	21	74	31904	6,971758

Table 2: Software implementations of PBKDF2, bcrypt, scrypt and the 22 PHC candidates

5.3 Finalists comparison

We analyze the features of the widely-used schemes PBKDF2, bcrypt and scrypt and the nine finalists. PHC intends to propose a set of PHSs, covering a wide range of diverse designs and different applications. We compare the different schemes based on the offered functionality and efficiency. Then, we mark the less competitive proposals and propose the most suitable solutions for different types of applications, like general applications on mainstream computers, web authentication and embedded systems.

Regarding design diversion, Argon, battcrypt, Catena, Lyra2, POMELO and Pufferfish are based on RAM-hardness. Among them only Argon, Lyra2 and POMELO consume high amounts of memory as scrypt. MAKWA, Parallel and yescrypt are not RAM-hard PHSs. Parallel is based on computational hardness and performs better than the relevant PBKDF2. yescrypt is based on scrypt and tweaks its security. MAKWA has constant memory requirements and depends on big number computations as RSA.

Argon, Catena, Lyra2 and Parallel support the full functionality of PHS, KDF, SR and HUIU. battcrypt, MAKWA, Pufferfish and yescrypt provide PHS, KDF with SR. POMELO is the only finalist that does not operate as a secure KDF and functions only as PHS with HUIU.

Catena and Lyra2 are the most well-analysed schemes in terms of security. Argon and MAKWA are also documented. battcrypt, Parallel and yescrypt do not provide proofs for the claimed properties. Pufferfish's security is not fully validated and POMELO produces lower randomness properties as it doesn't work as KDF.

Figure 3, illustrates the code size of the examined PHSs. PBKDF2, bcrypt, battcrypt, Catena and yescrypt produce compact implementations (less than 36KB ROM with less than 18KB RAM) making them suitable for constrained embedded devices, like sensors. Argon, Lyra2, MAKWA, Parallel, POMELO and Pufferfish have moderate requirements in code size. scrypt takes about the

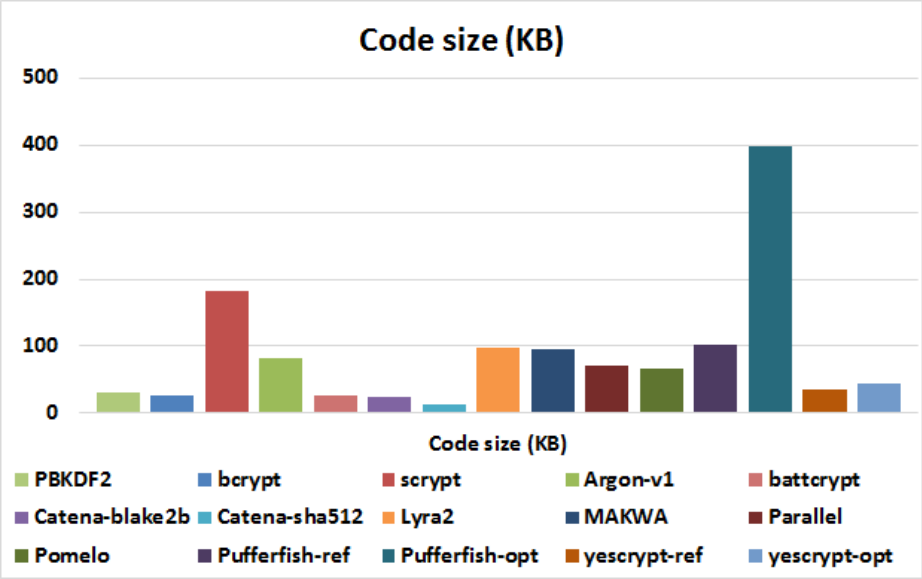

Fig. 3. The code size of PBKDF2, bcrypt, scrypt and the nine PHC finalists.

double code resources. The optimized implementation of Pufferfish produces the largest code in return of slightly better speed than the reference implementation.

The most efficient implementations are estimated based on the execution time that is needed for similar amounts of memory. Figure 4, illustrates the speed to RAM-consumption measurements of the finalists for different levels of memory usage. From the memory-hard schemes, Lyra2 and Catena are the most efficient ones, followed by POMELO, battcrypt, Argon and Pufferfish. From the non RAM-hard schemes, Parallel and yescrypt are the most efficient PHSs, followed by MAKWA.

Here we summarize the main disadvantages for some of the finalists based on the aforementioned analysis. POMELO is the only finalist that does not operate as a secure KDF due to lower randomness properties – a main evaluation criterion of PHC – which may obstruct its final selection. Similarly, security issues of Pufferfish could arise as its security properties are not fully validated. Moreover, Pufferfish and Argon are the least efficient schemes from all finalists, based on memory consumption and speed. battcrypt is a simplified scrypt and is designed for server-side applications. It provides the main functionality but does not support HUIU, an imperative property for long-term security in this domain. Other schemes that provide the full functionality are more efficient.

For typical RAM-hard schemes, Lyra2 is the best choice, followed by Catena. Both schemes offer the full functionality, are well-documented and analysed, and are the most efficient in terms of execution time and memory usage. Lyra2 is considered better for general applications as it can operates on high amounts of memory, like scrypt. Catena produces low code size and memory requirements,

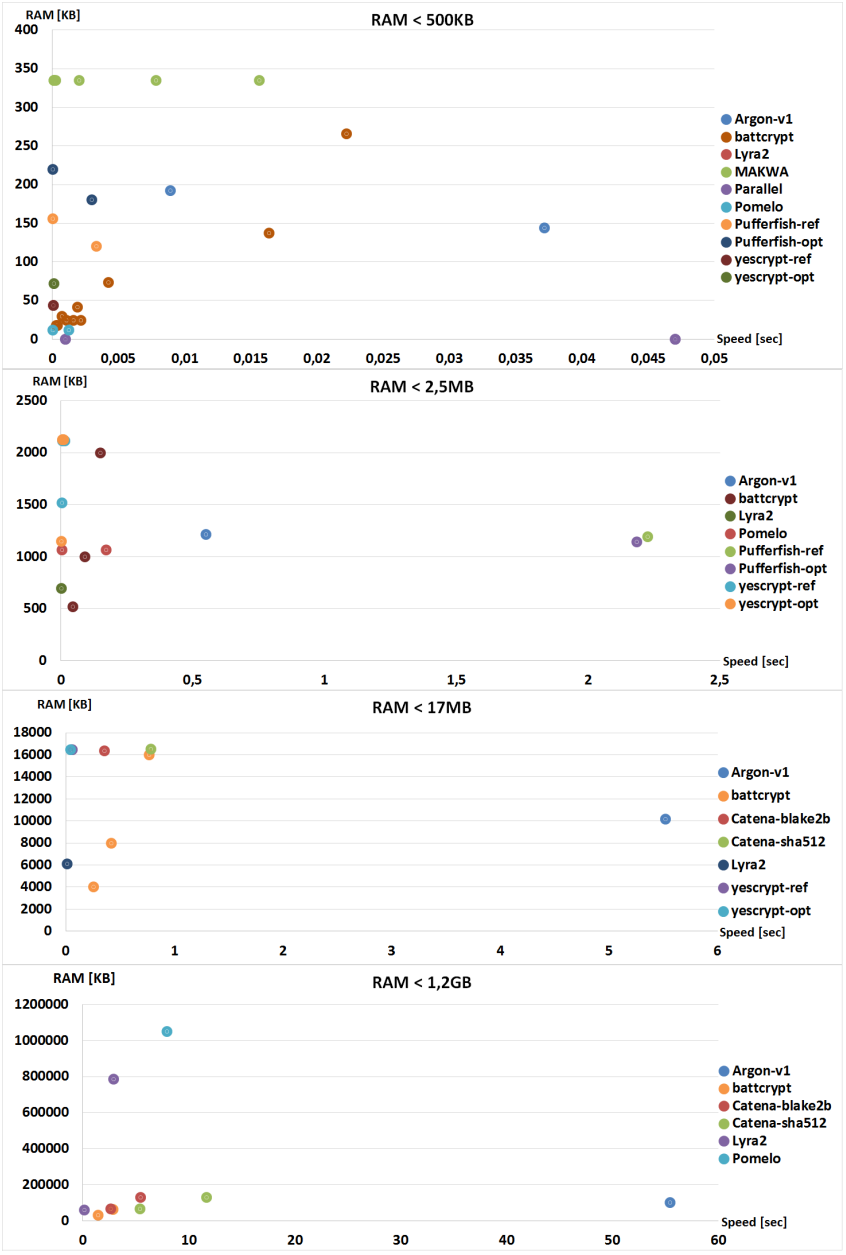

Fig. 4. PHC Finalists - Speed to RAM.

making it suitable for constraint environments and embedded systems. Both schemes function well on the web domain.

For non RAM-hard schemes, Parallel is one of the fastest finalists and the most suitable scheme for general and web applications. It is optimized for parallel execution, exploiting the computational capabilities of modern CPUs and implements the full functionality. yescrypt is also efficient and produces lightweight implementations that can be applied in embedded systems. MAKWA consumes similar amounts of RAM as bcrypt and it is two to four magnitudes faster. Its security properties are better documented than Parallel and yescrypt and can substitute bcrypt in general password hashing applications.

6 Conclusions

The maintenance of user passwords constitutes a significant factor related to the provided security of a service. Security breaches on famous applications have reveal massive amounts of user data, harming the reliability of their providers. The poor password hashing techniques and the limited available solutions lead the international cryptographic community to organize the Password Hashing Competition (PHC). The competition intends to delivery a small portfolio of modern and secure schemes for password hashing and key deviation. This paper provides a comparative analysis among the currently available schemes and the ones proposed in the PHC. We survey the first round results of the competition and evaluate reference and optimized software implementations of totally 25 schemes on the same platform. The general features of each scheme are analysed and a benchmark analysis is held, focusing on the nine finalists of the first PHC round. We contribute to the final selection of the winners by highlighting the efficiency of each finalist in terms of execution time, memory consumption and code size. We notify five of the finalists that excel based on their overall performance and documentation and map them in different types of applications. For RAM-hard schemes, Lyra2 can be used for general and web password hashing applications while Catena on web applications and embedded systems. For non RAM-hard schemes, Parallel, yescrypt and MAKWA are suggested. Parallel can exploit the computational capabilities of modern CPUs, yescrypt can be applied in general applications and embedded systems, and MAKWA can constitute an efficient replacement of the widely-used bcrypt.

Acknowledgement

This work was funded by the General Secretarial Research and Technology (G.S.R.T.), Hellas under the Artemis JU research program nSHIELD (new embedded Systems arcHItecturE for multi-Layer Dependable solutions) project. Call: ARTEMIS-2010-1, Grand Agreement No: 269317.

References

1. Florencio, D., Herley, C., Van Oorschot, P. C.: An administrator's guide to internet password research, 28^{th} USENIX conference on Large Installation System Administration (LISA'14), Seattle, WA, November 9-14, 2014, pp. 35-52 (2014)

2. Richmond, S., Williams, C.: Millions of internet users hit by massive Sony PlayStation data theft, The Telegraph, London, April 26, 2011. http://www.telegraph.co.uk/technology/news/8475728/Millions-of-internet-users-hit-by-massive-Sony-PlayStation-data-theft.html (2011)
3. Finkle, J., Saba J.: LinkedIn suffers data breach - security experts, Reuters, June 2012. http://in.reuters.com/article/2012/06/06/linkedin-breach-idINDEE8550EN20120606 (2012)
4. Kaliski, B.: RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000 (2000)
5. Provos, N., Mazires, D.: A Future-Adaptable Password Scheme, USENIX Annual Technical Conference, pp. 8192 (1999)
6. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan09, May 2009 (2009)
7. Orman, H: Twelve Random Characters: Passwords in the Era of Massive Parallelism, IEEE Internet Computing, vol. 17, issue 5, September-October, 2013, pp. 91-94 (2013)
8. Forler, C., Lucks, S., Wenzel, J.: Catena: A memory-consuming password scrambler, Cryptology ePrint Archive, Report 2013/525 (2013)
9. Forler, C., Lucks, S., Wenzel, J.: The Catena Password Scrambler, PHC submission, May 15, 2014. https://password-hashing.net/submissions/specs/Catena-v3.pdf (2014)
10. Cryptographic competition: Password Hashing Competition (PHC), April 25, 2013. https://password-hashing.net/ (2013)
11. Forler, C., List, E., Lucks, S., Wenzel, J.: Overview of the Candidates for the Password Hashing Competition And their Resistance against Garbage-Collector Attacks, Cryptology ePrint Archive, Report 2014/881 (2014)
12. U.S. Department of Commerce, National Institute of Standards and Technology (NIST). http://www.nist.gov/
13. Bonneau, J., Schechter, S.: Towards reliable storage of 56-bit secrets in human memory, 23^{rd} USENIX conference on Security Symposium (SEC'14), San Diego, CA, USA, 2014, pp. 607-623 (2014)
14. Chen, L., Lim, H. W., Yang, G.: Cross-Domain Password-Based Authenticated Key Exchange Revisited, ACM Transactions on Information and System Security (TISSEC), vol. 16, issue 4, April 2014, article No. 15 (2014)
15. Acar, T., Belenkiy, M., Kupcu, A.: Single password authentication, Computer Networks: The International Journal of Computer and Telecommunications Networking, vol. 57, issue 13, September, 2013, pp. 2597-2614 (2013)
16. Sun, H.-M., Chen, Y.-H., Lin, Y.-H.: oPass: A User Authentication Protocol Resistant to Password Stealing and Password Reuse Attacks, IEEE Transactions on Information Forensics and Security, vol. 7, issue 2, 29 September, 2011, pp. 651-663 (2011)
17. NIST: Recommendation for Password-Based Key Derivation, NIST Special Publication 800-132, December 2010. http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf (2010)
18. Gennaro, R., Lindell, Y.: A framework for password-based authenticated key exchange, ACM Transactions on Information and System Security (TISSEC), vol. 9, issue 2, May 2006, pp. 181-234 (2006)
19. Van Oorschot, P. C., Thorpe, J.: On predictive models and user-drawn graphical passwords, ACM Transactions on Information and System Security (TISSEC), vol. 10, issue 4, January 2008, Article No. 5 (2008)

20. Duffy, N., Jagota, A.: Connectionist password quality tester, IEEE Transactions on Knowledge and Data Engineering, vol. 14, issue 4, July/August 2002, pp. 920-922 (2002)
21. Florencio, D., Herley, C.: A large scale study of web password habits, 16^{th} International Conference on World Wide Web (WWW 2007), Banff, Alberta, Canada, May 2007, pp. 657-666 (2007)
22. Van Oorschot, P. C., Stubblebine, S.: On countering online dictionary attacks with login histories and humans-in-the-loop, ACM Transactions on Information and System Security (TISSEC), vol. 9, issue 3, August 2006, pp. 235-258 (2006)
23. Seo, S.-H., Kim, J.-H., Hwang, S.-H., Kwon, K. H., Jeon, J. W.: A reliable gateway for in-vehicle networks based on LIN, CAN, and FlexRay, ACM Transactions on Embedded Computing Systems (TECS), vol. 11, issue 1, March 2012, Article No. 7 (2012)
24. Mannan, M., Van Oorschot, P. C.: Leveraging personal devices for stronger password authentication from untrusted computers, Journal of Computer Security, IOS Press Amsterdam, The Netherlands, vol. 19, issue 4, December 2011, pp. 703-750 (2011)
25. Manifavas, C., Hatzivasilis, G., Fysarakis, K., Rantos, K.: Lightweight cryptography for embedded systems a comparative analysis, 6^{th} International Workshop on Autonomous and Spontaneous Security (SETOP 2013), in conjunction with the 18^{th} annual European research event in Computer Security (ESORICS 2013) symposium, Springer, LNCS, vol. 8247, 12-13 September, 2013, pp. 333-349. (2013)
26. Dini, G., Savino, I. M.: LARK: A Lightweight Authenticated ReKeying Scheme for Clustered Wireless Sensor Networks, ACM Transactions on Embedded Computing Systems (TECS), vol. 10, issue 4, November 2011, Article No. 41 (2011)
27. Dong, Q., Liu, D.: Using Auxiliary Sensors for Pairwise Key Establishment in WSN, ACM Transactions on Embedded Computing Systems (TECS), vol. 11, issue 3, September 2012, Article No. 59 (2012)
28. Dhurjati, D., Kowshik, S., Adve, V., Lattner, C.: Memory safety without garbage collection for embedded applications, ACM Transactions on Embedded Computing Systems (TECS), vol. 4, issue 1, February 2005, pp. 73-111 (2005)
29. RSA Laboratories: PKCS #5: Password-Based Cryptographic Standard, version 2.0, 2000. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-5-password-based-cryptography-standard.htm (2000)
30. B. Kaliski, RSA Laboratories: PKCS #5: Password-Based Cryptography Specification Version 2.0, IETF, RFC2898, September 2000. http://tools.ietf.org/html/rfc2898 (2000)
31. Avoine, G., Junod, P., Oechslin, P.: Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables, ACM Transactions on Information and System Security (TISSEC), vol. 11, issue 4, July 2008, Article No. 17 (2008)
32. Schneier, B.: Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), Fast Software Encryption, Springer, LNCS, vol. 809, 1994, pp. 191-204 (1994)
33. Percival, C., Josefsson, S.: The scrypt Password-Based Key Derivation Function, IETF, January 26, 2015. https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-02 (2015)
34. Bernstein, D.J.: The Salsa20 family of stream ciphers, eSTREAM project, 2007. http://cr.yp.to/papers.html#salsafamily (2007)
35. Password Hashing Competition (PHC): Candidates, March 31, 2014. https://password-hashing.net/candidates.html (2014)

36. NIST: Advanced Encryption Standard, FIPS-197, November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (2001)
37. Paul, G., Maitra, S.: RC4 stream cipher and its variants, Discrete Mathematics and Its Applications, CRC press, 2011 (2011)
38. Intel Software Network Rev 3.01, AES Instructions Set, Intel, 2008, https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, Accessed 20 Mar 2015.
39. NIST: Secure Hash Standard, FIPS 180-2, April 1995. http://csrc.nist.gov (1995).
40. Aumasson, J.-P., Neves, S., Wilcox-OHearn, Z., Winnerlein, C.: BLAKE2: Simpler, Smaller, Fast as MD5, ACNS, Springer, LNCS, vol. 7954, 2013, pp 119135 (2013)
41. RSA Laboratories: PKCS #1: RSA Cryptography Standard Version 2.2, 2012. http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf (2012)
42. NIST: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), NIST Special Publication 800-90A, January 2012. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf (2012)
43. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP, Kaufmann, M., 2000, ISBN 1-55860-671-8, http://lib.mdp.ac.id/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf (2000)
44. NIST: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Draft FIPS PUB 202, May 2014. http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf (2014)
45. cryptocoding.net: Crypto Coding Standard, January 23, 2013. https://cryptocoding.net/index.php/Coding_rules (2013)
46. Maslennikov, M.: Secure Hash Algorithm MCSSHA-8, August 18, 2014. http://crypto.systema.ru/mcssha/MCSSHA-8%20(eng).pdf (2014)
47. Maslennikov, M.: Password Hashing Scheme MCS_PHS, PHC submission, February 16, 2014. https://password-hashing.net/submissions/specs/MCS_PHS-v2.pdf (2014)
48. NIST: Secure Hash Standard (SHS), FIPS PUB 180-4, March 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf (2012)
49. Bernstein, D. J.: CubeHash specification, September 14, 2009. http://cubehash.cr.yp.to/submission2/spec.pdf (2009)
50. Bernstein, D. J.: ChaCha, a variant of Salsa20, January 20, 2008. http://cr.yp.to/chacha/chacha-20080128.pdf (2008)
51. Shamir, A: How to share a secret. Communications of the ACM vol. 22, issue 11, pp. 612613 (1979)
52. McCabe, T. J.: A complexity measure, IEEE Transactions on Software Engineering, vol. SE-2, no 4, December, 1976. http://www.literateprogramming.com/mccabe.pdf (1976)
53. Blaze, M.: Efficient Symmetric-Key Ciphers Based on an NP Complete Subproblem. http://www.crypto.com/papers/turtle.pdf (1996)
54. CyaSSL: pwdbased.c. wolfSSL, CyaSSL 3.2.0, 2014. http://www.yassl.com/yaSSL/Source/output/ctaocrypt/src/pwdbased.c.html (2014)
55. Garcia, R.: bcrypt. Openwall, Solar Designer. https://github.com/rg3/bcrypt (2014)
56. Tarsnap: scrypt-1.1.6. Tarsnap online backups for the truly paranoid, January 2010. http://www.tarsnap.com/scrypt/scrypt-1.1.6.tgz (2010).