Pleco and Plectron – Two Provably Secure Password Hashing Algorithms

Bo Zhu, Xinxin Fan, and Guang Gong
Department of Electrical and Computer Engineering, University of Waterloo, Canada
{bo.zhu,x5fan,ggong}@uwaterloo.ca

ABSTRACT

Password-based authentication has been widely deployed in practice due to its simplicity and efficiency. Storing passwords and deriving cryptographic keys from passwords in a secure manner are crucial for many security systems and services. However, choices of well-studied password hashing algorithms are extremely limited, as their security requirements and design principles are different from common cryptographic algorithms. In this paper, we propose two practical password hashing algorithms, PLECO and PLECTRON. They are built upon well-understood cryptographic algorithms, and combine advantages of symmetric and asymmetric primitives. By employing the Rabin cryptosystem, we prove that the one-wayness of PLECO is at least as strong as the hard problem of integer factorization. In addition, both password hashing algorithms are designed to be sequential memory-hard, in order to thwart large-scale password cracking by parallel hardware, such as GPUs, FPGAs, and ASICs. Moreover, total computation and memory consumptions of PLECO and PLECTRON are tunable through their cost parameters.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*; D.4.6 [**Operating Systems**]: Security and Protection—*authentication, cryptographic controls*

General Terms

Security, Algorithm

Keywords

Password, hashing, login, authentication

1. INTRODUCTION

Password-based authentication is probably the most widely deployed security mechanism across all information systems thanks to its cost-effectiveness and efficiency. However, there are two fundamental limitations of password-based authentication: 1) Users routinely pick poor passwords which are particularly subject to dictionary attacks or brute-force search; and 2) A device or server storing a large number of passwords is consistently a juicy target for attackers, and how to store passwords securely and minimize damages if the device or server has been breached is non-trivial. As an effective countermeasure, all passwords should be obscured together with user-specific, random and high-entropy salts by applying a one-way function, namely *password hashing*, before storing them in the device or server. During authentication, the user's input is processed in the same way and then the result is compared with the one stored in the device or server.

Password hashing is one of the most basic security considerations for setting up a password-based authentication system, and there are several requirements that a good password hashing algorithm should fulfill:

- Similar as most cryptographic primitives, the password hashing algorithm should behave as a random function that ensures *one-wayness* and *collision resistance*, and is resistant to side-channel attacks as well as known cryptanalytic technologies such as time-memory tradeoff [10, 14], differential/linear cryptanalysis [7, 18];
- Different from most cryptographic primitives, the password hashing algorithm should be *heavyweight* in computation and memory usage to slow down brute-force attacks to a certain degree and make large-scale attacks economically difficult. Note that the desired heavyweightness is expected to be roughly consistent for all platforms, no matter software or hardware;
- *Server-specific shortcut* is an optional but very attractive feature for a password hashing scheme. Once this feature is enabled and certain private information is known, legitimate servers or devices can obscure passwords by using less computation (*server-specific computational shortcut*) and/or less memory (*server-specific memory shortcut*).

Although password hashing is the foundation of many real-world security systems and services, there are only limited proposals well studied and adopted, due to the aforementioned uncommon and demanding requirements. PBKDF2, which is a key derivation function designed by RSA Laboratories and standardized in RFC 2898 [15], has been the sub-

ject of extensive research and still remains the best conservative choice. PBKDF2 is a conventional design that mainly relies on iterating a pseudorandom function (usually HMAC-SHA1) a certain number of times. However, the iterative design leads to quite unaggressive usage of memory, which makes large-scale and parallel cracking possible [11]. bcrypt is designed by Provos and Mazières [25], based on the Blowfish cipher [29] with a purposefully expensive key schedule. Thanks to the adaptive iteration count that can be increased to make it slower, bcrypt could remain resistant to brute-force search attacks even with vast increases in computing power. Like PBKDF2, bcrypt works in-place in memory and performs poorly towards thwarting attacks using dedicated hardware. scrypt, designed by Percival [23], is a proposal which not only offers stronger security from a theoretical point of view than the other two but also allows one to configure the amount of space in memory required to efficiently complete the algorithm. The customizable memory requirement makes it difficult for attackers to build large-scale cracking circuits for scrypt, since these dedicated hardware usually have very constrained memory. The design of scrypt has been extensively evaluated when scrypt was selected as the underlying proof-of-work function for many cryptocurrencies, e.g., Litecoin and Dogecoin.

1.1 Our Contributions

In this paper, we propose two novel password hashing algorithms, named PLECO and PLECTRON, respectively, based upon several well-studied cryptographic structures and primitives. The novelty in the designs of PLECO and PLECTRON is the combination of asymmetric and symmetric components that offers a twofold benefit: 1) Since the tools to cryptanalyzing asymmetric algorithms are quite different from those for symmetric ones, the composition of asymmetric and symmetric components will make cryptanalysis much harder. This is analogous to the designs of ARX based cryptographic primitives [1, 4, 20] and the block cipher IDEA [16] where mixed operations are used; 2) The asymmetric component not only makes our scheme provably secure (the security of PLECO is as strong as the hard problem of integer factorization), but also enables server-specific computational shortcuts as a result of faster exponentiation via the Chinese Remainder Theorem when factors of moduli are known. In addition to describing the PLECO and PLECTRON designs in great detail, we also theoretically prove their security with respect to one-wayness and collision resistance.

1.2 Organization

The organization of the paper is as follows. Section 2 describes the cryptographic primitives that PLECO and PLECTRON employ. The designs of PLECO and PLECTRON are specified in Section 3. We discuss the design rationale and provide the security analysis of PLECO and PLECTRON in Section 4. We also propose several extensions of the new hashing algorithms in Section 5, followed by the efficiency analysis in Section 6. We discuss the related work in Section 7 and finally conclude this paper in Section 8.

2. INGREDIENTS OF PLECO/PLECTRON

This section briefly describes several cryptographic primitives, which are the core components in the designs of PLECO and PLECTRON.

2.1 Provably One-Way Function

It is proven that the security of the Rabin public-key encryption scheme is equivalent to the hard problem of integer factorization [26]. More theoretically, let us define

$$\mathsf{Rabin}_n(x) = x^2 \pmod{n},$$

where x is a positive integer in the multiplicative group of integers modulo n. Then computing the square roots, i.e., reversing the function $\mathsf{Rabin}_n(x)$, is proven to be equivalent to factorizing the integer n.

To obtain a hard-to-factor modulus n, one can utilize the same approach as generating moduli for the RSA algorithm, i.e., randomly generating two large prime numbers p and q, and using their product $n = p \cdot q$ as a modulus. The other approach is to choose certain large composite numbers with unknown factorization, e.g., the Mersenne composite number used in Shamir's SQUASH construction [31]. More about publicly auditable moduli will be discussed in Section 5.2.

2.2 Sponge-Based Hash Function

KECCAK, which is designed by Bertoni, Daemen, Peeters, and Van Assche [5], is the winner of the SHA-3 cryptographic hash function competition held by NIST [21]. KECCAK is based on a unique construction, namely *sponge construction*, which can *absorb* an arbitrary-length binary string as input, and then *squeeze* out a binary string of any required length as output.

In our password hashing designs, KECCAK is adopted to:

- Fully mix password and salt strings;
- Expand short input strings to the large space of the Rabin encryption scheme;
- Alternately apply to intermediate states with the Rabin encryption scheme (or other public-key schemes) to gain more cryptanalytic strength.
- May process the final state to produce hash tags of required lengths.

If not specified, default parameters of KECCAK should be used, i.e., $r = 1024$ and $c = 576$.

2.3 Sequential Memory-Hard Construction

The password-based key derivation function scrypt was proposed by Percival in order to thwart parallel brute-force attacks using GPUs, FPGAs or ASIC chips on passwords, and has been widely used by cryptocurrencies. One of core components of scrypt, namely ROMix, is proven to be *sequential memory-hard*. One important feature of being sequential memory-hard is that parallel algorithms cannot asymptotically achieve efficiency advantage than non-parallel ones [23]. In other words, brute-force password search by using dedicated hardware with constrained memory, such as GPUs, FPGAs, and ASICs, would not be significantly faster than a single-core desktop computer. For a more detailed definition of sequential memory-hard, the reader is referred to [23].

We list ROMix in Algorithm 1 since it is highly relevant to our designs of PLECO and PLECTRON. In Algorithm 1, H

is a cryptographic hash function, *bstr* is a binary string, *cost* is called the CPU/memory cost parameter that must be larger than one, and Integerify is a bijective function that maps binary strings to integers.

Algorithm 1 ROMix($bstr$, $cost$)

1: $x \leftarrow bstr$
2: **for** $i \leftarrow 0$ to $cost - 1$ **do**
3: $\quad v_i \leftarrow x$
4: $\quad x \leftarrow H(x)$
5: **end for**
6: **for** $i \leftarrow 0$ to $cost - 1$ **do**
7: $\quad j \leftarrow \mathsf{Integerify}(x) \pmod{cost}$
8: $\quad x \leftarrow H(x \oplus v_j)$
9: **end for**
10: **return** x

3. DESIGN OF PLECO AND PLECTRON

The design rationale of PLECO and PLECTRON is to inherit the existing structure of scrypt that is proved to be sequential memory-hard, and to improve its inner components for providing better security and asymmetry in computation as desired.

The following notations are used in this section:

- $||$ concatenates two binary strings;
- $\mathsf{int}(\cdot)$ converts a binary string into a non-negative integer, where the little-endian convention is used, i.e., the left-most (lowest address) bit is the least significant bit of the integer[1];
- $\mathsf{str}_b(\cdot)$ converts a non-negative integer back to a binary string by using the same bit ordering convention as $\mathsf{int}(\cdot)$, and may append zeros to the string in order to achieve a total length of b bits;
- 0^t denotes a t-bit all-zero binary string, e.g., $0^t = \mathsf{str}_t(0)$ for $t > 0$, and 0^0 means an empty string;
- $\mathsf{len}(\cdot)$ denotes the bit-length of a binary string;
- $\mathsf{size}(\cdot)$ denotes the number of bits in the binary representation of a given non-negative integer, e.g., $\mathsf{size}(256) = 9$ and $\mathsf{size}(255) = 8$;
- KECCAK$_b$ denotes a KECCAK instance that produces exactly b bits as output.

Given a modulus n, we define a new hash function

$$\mathcal{H}_n(x) = \mathsf{str}_N(\mathsf{Rabin}_n(1 + \mathsf{int}(\text{KECCAK}_{N-1}(x)))),$$

where $N = \mathsf{size}(n)$. To be secure, N should be at least 1024, or preferably larger than 3072, according to [2]. As we mentioned before, the modulus n can be obtained using the same approach for generating the RSA modulus $n = p \cdot q$ or chosen from a public composite number with unknown factorization as proposed in the design of SQUASH [31].

[1]For software implementations, we recommend using the following convention: The 8 least significant bits are stored in the byte with the lowest address, and within a byte the least significant bit is the coefficient of 2^0. This follows the internal implementation convention of KECCAK [6].

Our new password hashing algorithm PLECO is defined by Algorithm 2, which takes as input

- a modulus n,
- a 128-bit binary string *salt* as a unique or randomly generated salt,
- a variable-length (≤ 128 bytes) binary string *pass* as a user password,
- a positive integer *tcost* as the time cost parameter,
- and a positive integer *mcost* as the memory cost parameter.

Algorithm 2 PLECO(n, $salt$, $pass$, $tcost$, $mcost$)

1: $L \leftarrow 8 \cdot \lceil \mathsf{size}(n)/8 \rceil - \mathsf{size}(n)$
2: $x \leftarrow salt||\mathsf{str}_{16}(\mathsf{len}(pass))||pass||0^{1024-\mathsf{len}(pass)}$
3: $ctr \leftarrow 0$
4: $x \leftarrow \mathcal{H}_n(\mathsf{str}_{128}(ctr)||x)$
5: **for** $i \leftarrow 0$ to $tcost - 1$ **do**
6: $\quad$ **for** $j \leftarrow 0$ to $mcost - 1$ **do**
7: $\quad\quad v_j \leftarrow x$
8: $\quad\quad ctr \leftarrow ctr + 1$
9: $\quad\quad x \leftarrow \mathcal{H}_n(\mathsf{str}_{128}(ctr)||x)$
10: $\quad$ **end for**
11: $\quad$ **for** $j \leftarrow 0$ to $mcost - 1$ **do**
12: $\quad\quad k \leftarrow \mathsf{int}(x) \pmod{mcost}$
13: $\quad\quad ctr \leftarrow ctr + 1$
14: $\quad\quad x \leftarrow \mathcal{H}_n(\mathsf{str}_{128}(ctr)||x||0^L||v_k)$
15: $\quad$ **end for**
16: $\quad ctr \leftarrow ctr + 1$
17: $\quad x \leftarrow \mathcal{H}_n(\mathsf{str}_{128}(ctr)||x)$
18: **end for**
19: **return** x

Lines 6-15 of Algorithm 2 are essentially the same as ROMix, except that:

- An incremental counter *ctr* is always prepended to the intermediate variable x in each step;
- Instead of XORing v_k with x as the design of ROMix, we concatenate them and input into $\mathcal{H}_n$.

Sections 4.2 and 4.4 will give the detailed reasons why we introduce these changes.

PLECO will produce an N-bit hash tag, but sometimes applications need to flexibly choose tag sizes, e.g., generating cryptographic keys from passphrases entered by users. We recommend applying KECCAK to the output of PLECO again to produce tags of required lengths. We name this modified algorithm as PLECTRON and specifies its design in Algorithm 3, where

- *hsize* denotes the desired bit-length of the hash tag.

Algorithm 3 PLECTRON(n, *salt*, *pass*, *tcost*, *mcost*, *hsize*)

1: $t \leftarrow$ PLECO(n, *salt*, *pass*, *tcost*, *mcost*)
2: **return** KECCAK$_{hsize}(t)$

4. SECURITY ANALYSIS

The designs of PLECO and PLECTRON combine public-key and symmetric-key algorithms and alter the operation sequence to make cryptanalysis harder. This is analogous to the designs of several ARX ciphers and the block cipher IDEA, where mixed operations are used. In what follows, we discuss security properties of PLECO and PLECTRON in detail.

4.1 One-Wayness

One of the most important security goals of designing a password hashing scheme is one-wayness, i.e., attackers should not be able to devise any methods faster than brute-force search for reversing the hashing algorithm in order to obtain original passwords.

In our designs, the cryptographic hash function KECCAK and the provably one-way function Rabin_n are applied to the intermediate state x alternatively. To the best of our knowledge, no weaknesses have been reported when combining these two algorithms. In order to reverse $\mathcal{H}_n$ the attackers may have to analyze KECCAK and Rabin_n separately. On one hand, even if the one-wayness of KECCAK is completely broken, say replacing KECCAK by an identity function, the one-wayness of $\mathcal{H}$ is still guaranteed by Rabin_n, i.e., the hardness of integer factorization. On the other hand, if any weakness of iterating KECCAK is found, the weakness is highly likely to be covered up by the computations of Rabin_n.

More formally, we give the following definition.

Definition 1. For a given function f and a pre-specified set Y containing certain outputs of f, we define the advantage of an adversary $\mathcal{A}$ finding preimages of the elements in Y (i.e., reversing f) as

$$\mathbf{Adv}_f^{\mathrm{Pre}(Y)}(\mathcal{A}) \stackrel{\mathrm{def}}{=} \Pr[y \stackrel{\$}{\leftarrow} Y, x \leftarrow \mathcal{A}^{f,y} : f(x) = y],$$

where $y \stackrel{\$}{\leftarrow} Y$ means randomly assigning one element of Y to y.

Then we can show the preimage security of $\mathcal{H}_n$ is guaranteed by Rabin_n, as described in the following lemma.

LEMMA 1 (ONE-WAYNESS OF $\mathcal{H}_n$). *For any adversary $\mathcal{A}$, we have*

$$\mathbf{Adv}_{\mathcal{H}_n}^{\mathrm{Pre}(S)}(\mathcal{A}) \le \mathbf{Adv}_{\mathsf{Rabin}_n}^{\mathrm{Pre}(S)}(\mathcal{A}),$$

where S is a set consisting of certain outputs of $\mathcal{H}_n$,

PROOF. Assume that n is an N-bit modulus. Once a preimage of $\mathcal{H}_n$ is found, e.g., $y = \mathcal{H}_n(x)$, we let $x' = 1 + \mathsf{int}(\text{KECCAK}_{N-1}(x))$ and $y' = \mathsf{int}(y)$, and then x' is a preimage of y' of Rabin_n. □

For a reasonably large set S, computing preimages of Rabin_n regarding S is still as hard as factoring the integer n, since the factorization will be known after obtaining a constant number of preimages on average. For example, for RSA-like moduli, the expected number of preimages required is 2 [19].

Please note that Lemma 1 presents a simplified bound only for the case that n is not factored by adversaries. If the factorization of n is known to adversaries, the one-wayness of $\mathcal{H}$ is still guaranteed by KECCAK.

Based on Lemma 1, we can investigate the one-wayness of the whole design of PLECO.

THEOREM 1 (ONE-WAYNESS OF PLECO). *If* PLECO *and $\mathcal{H}_n$ use a same modulus n, then we have*

$$\mathbf{Adv}_{\text{PLECO}}^{\mathrm{Pre}(S)}(\mathcal{A}) \le \mathbf{Adv}_{\mathsf{Rabin}_n}^{\mathrm{Pre}(S)}(\mathcal{A}),$$

where S is a set containing all possible outputs of PLECO.

PROOF. Assume that a preimage of PLECO is found, then the steps of PLECO before the last $\mathcal{H}_n$ can be recomputed, so the preimage of $\mathcal{H}$ is obtained. Thus, we have

$$\mathbf{Adv}_{\text{PLECO}}^{\mathrm{Pre}(S)}(\mathcal{A}) \le \mathbf{Adv}_{\mathcal{H}_n}^{\mathrm{Pre}(S)}(\mathcal{A}),$$

which implies

$$\mathbf{Adv}_{\text{PLECO}}^{\mathrm{Pre}(S)}(\mathcal{A}) \le \mathbf{Adv}_{\mathsf{Rabin}_n}^{\mathrm{Pre}(S)}(\mathcal{A}),$$

due to Lemma 1. □

For the preimage in the above theorem, we do not differentiate the two cases: 1) a preimage containing both *salt* and *pass*, or 2) a preimage including only *pass* for a pre-specified *salt*. For the second case, the first KECCAK$_{N-1}$ in PLECO can be seen as a specialized KECCAK instance, e.g., the design of KECCAK supports simply prepending a message with a key to construct a Message Authentication Code (MAC) scheme [5]. Therefore, an adversary's advantage of recovering *pass* still satisfies the bound in Theorem 1, even if *salt* is public or known to adversaries.

Next, let us consider the one-wayness of PLECTRON. If we assume that, in order to reverse PLECTRON, any adversary has to first reverse KECCAK$_{hsize}$ and then reverse PLECO, we can simply get a bound like

$$\begin{aligned}\mathbf{Adv}_{\text{PLECTRON}}^{\mathrm{Pre}(S)}(\mathcal{A}) &\le \mathbf{Adv}_{\text{KECCAK}_{hsize}}^{\mathrm{Pre}(S)}(\mathcal{A}) \cdot \mathbf{Adv}_{\text{PLECO}}^{\mathrm{Pre}(S)}(\mathcal{A}) \\ &\le \mathbf{Adv}_{\text{KECCAK}_{hsize}}^{\mathrm{Pre}(S)}(\mathcal{A}) \cdot \mathbf{Adv}_{\mathsf{Rabin}_n}^{\mathrm{Pre}(S)}(\mathcal{A}).\end{aligned}$$

However, it cannot be guaranteed that adversaries will always try to obtain the intermediate value between PLECO and KECCAK$_{hszie}$. Consider the case where *hsize* is very small, say two bits. After trying random passwords for four times, there will be one producing a 2-bit pre-specified hash tag. Therefore, in theory, we can only give the following theorem on one-wayness of PLECTRON.

THEOREM 2 (ONE-WAYNESS OF PLECTRON). *If $\mathcal{H}_n$ and* PLECTRON *use a same modulus n, then we have*

$$\mathbf{Adv}_{\text{PLECTRON}}^{\mathrm{Pre}(S)}(\mathcal{A}) \le \mathbf{Adv}_{\text{KECCAK}_{hsize}}^{\mathrm{Pre}(S)}(\mathcal{A}),$$

where S is a set containing all possible outputs of PLECTRON.

PROOF. Once a primage of PLECTRON is found, e.g.,

$$y = \text{PLECTRON}(n, s, p, tc, mc, hsize),$$

we compute

$$x = \text{PLECO}(n, s, p, tc, mc).$$

Then x is a preimage of y of KECCAK_{hsize}. □

As a cryptographic hash function, KECCAK is designed to be preimage-resistant, which means that for essentially all outputs, finding any input hashing to a pre-specified output should be computationally infeasible [19, 27].

4.2 Collision Resistance

Collision and second-preimage resistances are also desirable when designing a password hashing scheme. In this context, an occurrence of collision may result in two passwords being hashed to the same tag, whereas a second-preimage implies that given a password *pass1* one may find the second one *pass2* producing the same tag. It is easy to see that if there exists an algorithm for constructing second-preimages, then it can also be used to generate collisions, so collision resistance implies second-preimage resistance.

It is easy to see that once a collision of KECCAK_{N-1} is found, then it will result in a collision of $\mathcal{H}_n$. Furthermore, if the outputs of KECCAK contain two roots of Rabin_n, then it will also produce a collision of $\mathcal{H}_n$. Therefore, the collision resistance of $\mathcal{H}_n$ is bounded by properties of Rabin_n and KECCAK together.

Formally, we give the following security definition.

Definition 2. For a given function f, we define the advantage of an adversary $\mathcal{A}$ to find a collision of f as

$$\mathbf{Adv}_f^{\text{Coll}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[x_1, x_2 \leftarrow \mathcal{A}^f : f(x_1) = f(x_2)].$$

To better analyze the collision resistance of $\mathcal{H}_n$, we give the following the definitions.

Definition 3. For a given function f, we define the advantage of an adversary $\mathcal{A}$ to obtain an output difference d as

$$\mathbf{Adv}_f^{\text{Diff}(d)}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[x_1, x_2 \leftarrow \mathcal{A}^{f,d} : f(x_1) = d - f(x_2)].$$

Definition 4. For a given positive composite integer m, we define the advantage of an adversary $\mathcal{A}$ to obtain a non-trivial factor of m as

$$\mathbf{Adv}_m^{\text{Fact}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[x \leftarrow \mathcal{A}^m, 1 < x < m : x | m].$$

This advantage $\mathbf{Adv}_f^{\text{Diff}(d)}(\mathcal{A})$ should be negligible for any secure cryptographic hash function f, since these hash functions are designed to be indistinguishable from pseudorandom functions.

Then we have the following lemma about collisions of $\mathcal{H}_n$.

LEMMA 2 (COLLISION RESISTANCE OF $\mathcal{H}_n$). *For any adversary $\mathcal{A}$, we have*

$$\begin{aligned}\mathbf{Adv}_{\mathcal{H}_n}^{\text{Coll}}(\mathcal{A}) \;&\leq\; \mathbf{Adv}_{\text{KECCAK}_{N-1}}^{\text{Coll}}(\mathcal{A}) + \mathbf{Adv}_{\text{KECCAK}_{N-1}}^{\text{Diff}(n)}(\mathcal{A}) \\ &+\; \mathbf{Adv}_n^{\text{Fact}}(\mathcal{A}),\end{aligned}$$

where $N = \mathsf{size}(n)$.

PROOF. Suppose a colliding pair x_1 and x_2 of $\mathcal{H}_n$ are found, i.e., $\mathcal{H}_n(x_1) = \mathcal{H}_n(x_2)$. Let $r_1 = \text{KECCAK}_{N-1}(x_1)$ and $r_2 = \text{KECCAK}_{N-1}(x_2)$. Then we have three cases:

- If $r_1 = r_2$, then a collision of KECCAK_{N-1} is found;
- If $r_1 \neq r_2$, let $s_1 = 1 + \mathsf{int}(r_1)$ and $s_2 = 1 + \mathsf{int}(r_2)$, and then:
  - If $s_1 = n - s_2$, then a pair producing the output difference n of KECCAK_{N-1} is found;
  - If $s_1 \neq n - s_2$, then $\gcd(s_1 - s_2, n)$ is a non-trivial factor of n.

Therefore, the lemma holds. □

As the previous discussion on Lemma 1, Lemma 2 also gives a simplified bound on collisions that satisfies our purpose of showing $\mathcal{H}_n$ to be secure. Even if n is factored, it should still be hard to construct collisions of the whole $\mathcal{H}_n$, since adversaries need to control outputs of KECCAK_{N-1} to be among roots corresponding to a same squaring value. For example, for RSA-like moduli, there are only four roots mapping to one output.

Based on Lemma 2, we give the following theorem to characterize adversaries' collision advantage on PLECO.

THEOREM 3 (COLLISION RESISTANCE OF PLECO). *If the cost parameters, mcost and tcost, of* PLECO *keep unchanged, and $\mathcal{H}_n$ and* PLECO *use a same modulus n, then we have*

$$\begin{aligned}\mathbf{Adv}_{\text{PLECO}}^{\text{Coll}}(\mathcal{A}) \;&\leq\; \mathbf{Adv}_{\text{KECCAK}_{N-1}}^{\text{Coll}}(\mathcal{A}) \\ &+\; \mathbf{Adv}_{\text{KECCAK}_{N-1}}^{\text{Diff}(n)}(\mathcal{A}) \\ &+\; \mathbf{Adv}_n^{\text{Fact}}(\mathcal{A}).\end{aligned}$$

PROOF. Once a collision of PLECO is found, then there must exist a collision of the internal hash function $\mathcal{H}_n$. Thus, we have

$$\mathbf{Adv}_{\text{PLECO}}^{\text{Coll}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathcal{H}_n}^{\text{Coll}}(\mathcal{A}).$$

Therefore, the theorem holds. □

Please note that if we use the original design of ROMix, i.e., XORing x and v_k instead of concatenating them together as input, the bound for collisions will be much more difficult to be discovered and proven, because different intermediate values x_1 and x_2 may still yield an identical input to the internal hash function $\mathcal{H}_n$, i.e., $x_1 \oplus v_{k_1} = x_2 \oplus v_{k_2}$. However,

in the current design of PLECO, different x_1 and x_2 will never generate identical inputs to $\mathcal{H}_n$.

For PLECTRON, we have the following theorem.

THEOREM 4 (COLLISION RESISTANCE OF PLECTRON). *If the cost parameters and output hash length, mcost, tcost and hsize, of* PLECTRON *keep unchanged, and* $\mathcal{H}_n$ *and* PLECTRON *use a same modulus* n*, then we have*

$$\begin{aligned}\mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{PLECTRON}_n}(\mathcal{A}) &\leq \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{KECCAK}_{N-1}}(\mathcal{A}) \\ &+ \mathbf{Adv}^{\mathrm{Diff}(n)}_{\mathrm{KECCAK}_{N-1}}(\mathcal{A}) \\ &+ \mathbf{Adv}^{\mathrm{Fact}}_{n}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{KECCAK}_{hsize}}(\mathcal{A}).\end{aligned}$$

PROOF. If a collision of PLECTRON is found, e.g.,

$$\begin{aligned}&\mathrm{PLECTRON}(n, s_1, p_1, tc, mc, hsize) \\ =\ &\mathrm{PLECTRON}(n, s_2, p_2, tc, mc, hsize),\end{aligned}$$

then we let

$$\begin{cases} t_1 = \mathrm{PLECO}(n, s_1, p_1, tc, mc) \\ t_2 = \mathrm{PLECO}(n, s_2, p_2, tc, mc) \end{cases}.$$

We have the following two cases:

- If $t_1 = t_2$, then a collision of PLECO is found.
- If $t_1 \neq t_2$, then a collision of KECCAK_{hsize} is found.

Therefore, we have

$$\mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{PLECTRON}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{PLECO}}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{KECCAK}_{hsize}}(\mathcal{A}).$$

Thus, the theorem holds. □

4.3 Thwarting Parallel Brute-Force Attacks

Although the designs of PLECO and PLECTRON may be secure for random inputs in theory, users' passwords are usually weak and easily crackable by using parallel search based on dedicated or custom-designed hardware, such as GPUs, FPGAs, and ASICs. Thus password hashing designs should thwart such attacks as much as possible.

The hardware such as GPUs, FPGAs, and ASICs can feature thousands of cores for parallel computation, but in return each core possesses very restrained memory space. By using the structure of ROMix, the internal iteration of PLECO (Lines 6-15 in Algorithm 2) inherits scrypt's security property of being sequential memory-hard. PLECO and PLECTRON also provide a tunable memory parameter *mcost* to increase their memory cost as desired. Although the design of PLECO is slightly different from ROMix, the security proofs of ROMix can be easily transferred to here, since in the original proofs the internal hash function is treated as a Random Oracle. Recommendations for cost parameter choices in practice will be given in Section 6.1.

Please note that in scrypt, a structure called BlockMix is used to build an internal hash function with wide input/output from a small function Salsa20 core [4]. However, BlockMix may not be necessary for PLECO since the input/output lengths of $\mathcal{H}_n$ are relatively large. As a side benefit of omitting BlockMix, our scheme is simpler and easier for analysis, when compared to scrypt.

4.4 Preventing Self-Similarity Attacks

An incremental counter *ctr* is prepended to the intermediate states of PLECO before each invocation of $\mathcal{H}_n$, which enables us to protect PLECO and PLECTRON from certain potential self-similarity attacks, such as fixed points or iterative patterns of $\mathcal{H}_n$. The similar technique is used in many other cryptographic designs, such as KECCAK, PRESENT [8] and PRINCE [9].

5. OTHER EXTENTIONS

In this section, we propose a number of extensions of PLECO and PLECTRON.

5.1 Discrete Logarithm Based Hash Function

Gibson has proved that if factoring n is hard, the following discrete logarithm based hash function

$$\mathcal{G}_n(x) = g^x \pmod n$$

is one-way and collision-free [13][2], where x is a positive integer and g is a generator of the multiplicative group of integers modulo n. The security of this hash function is guaranteed by the hardness of integer factorization, since a collision will lead to the factorization of n.

We define a new hash function

$$\mathcal{GH}_n(x) = \mathsf{str}_N(\mathcal{G}_n(1 + \mathsf{int}(\mathrm{KECCAK}_N(x)))),$$

for a given positive integer n, where $N = \mathsf{size}(n)$. If $\mathcal{H}_n(x)$ in PLECO is replaced by $\mathcal{GH}_n(x)$, the security, especially the collision resistance, of PLECO and PLECTRON would be further enhanced.

LEMMA 3. *For any adversary* $\mathcal{A}$*, we have*

$$\mathbf{Adv}^{\mathrm{Pre}(S)}_{\mathcal{GH}_n}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{Fact}}_{n}(\mathcal{A}),$$

and

$$\mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{GH}_n}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{KECCAK}_N}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{G}_n}(\mathcal{A}).$$

PROOF. The proofs are similar to the ones for Lemmas 1 and 2, so they are omitted here. □

It is easy to see that $\mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{GH}_n}(\mathcal{A})$ is smaller than $\mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{H}_n}(\mathcal{A})$, because $\mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{G}_n}(\mathcal{A})$ here should be equivalent to the term $\mathbf{Adv}^{\mathrm{Pre}(S)}_{\mathsf{Rabin}_n}(\mathcal{A})$ in Lemma 1, but the second term $\mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{H}_n}(\mathcal{A})$ has an extra $\mathbf{Adv}^{\mathrm{Diff}(n)}_{\mathrm{KECCAK}_{N-1}}(\mathcal{A})$ (see Lemma 2).

We have the following theorem about using $\mathcal{GH}_n$ in PLECO.

THEOREM 5. *Suppose the cost parameters, mcost and tcost, of* PLECO *keep unchanged, and* $\mathcal{GH}_n$ *and* PLECO *use a same*

[2]It is claimed in [30] that this hash function was proposed by Shamir, and a simple proof was given by Rivest.

N-bit modulus n. If replacing $\mathcal{H}_n$ in PLECO *by $\mathcal{GH}_n$, namely* PLECO′*, we will have*

$$\mathbf{Adv}^{\mathrm{Pre}(S)}_{\mathrm{PLECO}'}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{Pre}(S)}_{\mathcal{G}_n}(\mathcal{A}),$$

and

$$\mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{PLECO}'}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{Coll}}_{\mathrm{KECCAK}_N}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{Coll}}_{\mathcal{G}_n}(\mathcal{A}),$$

where S is a set containing all possible outputs of PLECO′*.*

PROOF. As the proofs for Theorems 1 and 3, the one-wayness and collision resistance of PLECO′ are guaranteed by the security properties of $\mathcal{GH}_n$. Thus, the theorem holds. □

For PLECTRON, if $\mathcal{H}_n$ is replaced by $\mathcal{GH}_n$, its one-wayness (Theorem 2) does not change, but the bound for its collisions will be improved as PLECO.

Another benefit of using $\mathcal{GH}_n$ instead of $\mathcal{H}_n$ is that $\mathcal{G}_n$ is proven to be secure for any positive integer as input. Compared to Rabin_n, whose security properties usually only consider the inputs within the multiplicative group of integers modulo n, $\mathcal{G}_n$ allows much more flexibility when we adopt it to design security schemes. For example, in the design of $\mathcal{GH}_n$, the output length of KECCAK may be equal to or larger than $\mathsf{size}(n)$; while in $\mathcal{H}_n$, KECCAK is set to generate less than $\mathsf{size}(n)$ bits.

Although the discrete logarithm based hash function $\mathcal{G}_n$ is more secure and flexible, it is much less efficient than Rabin_n due to the slow modular exponentiation computations.

5.2 Using Publicly Auditable Modulus

As observed by Shamir in [31], the Rabin scheme cannot be efficiently inverted for any modulus n with unknown factorization. As a result, large composite Mersenne numbers with unknown factorization and of the form $n = 2^k - 1$ can be used as the modulus, which enables efficient software implementation of PLECO and PLECTRON (see Section 6.2 for performance comparison). A table summarizing the factorization of Mersenne numbers of the form $M_k = 2^k - 1$ is maintained by Leyland [17]. Certain interesting Mersenne numbers that might be used as the moduli in PLECO and PLECTRON for different security levels are $2^{1277}-1$, $2^{2137}-1$, and $2^{3049} - 1$.

Furthermore, RSA-like moduli might not be suitable if PLECO or PLECTRON are used in cryptocurrencies for proofs of work, because RSA-like moduli must be generated by someone. With the private knowledge of the factors of n, one may compute the hash functions more efficiently than others (which will be discussed in Section 6.3). By using public composite numbers with unknown factors, we can eliminate potential trapdoors in cryptocurrency systems.

One potential method to publicly generate moduli with unknown factorization is constructing the numbers called RSA-UFOs proposed by Sander in [28]. But RSA-UFOs may be too large for practical applications, e.g., in order to be used as a 1024-bit modulus it may require more than 40000 bits. Moreover, RSA-UFOs will inevitably contain many known factors if they are generated by the method in [28]. Therefore, the performance and security of RSA-UFOs still need more investigations.

5.3 Transforming Existing Hashes to Larger Cost Settings

For PLECO, its final output can re-enter the algorithm from Line 6 (Algorithm 2), which is equivalent to increasing the time cost parameter *tcost* by one. During the additional computations, we can also choose a larger memory parameter *mcost*. Under such circumstance, hash tags can be updated according to new cost settings without the knowledge of original passwords.

5.4 Variants with More Efficient Software Implementations

In order to be easily and efficiently implemented in software, it is better for the modulus n to have a size that is a multiple of word sizes. But in certain circumstances, we cannot choose the size of n freely, e.g., when using Mersenne composite numbers, so we may need to make small changes for the original algorithms of $\mathcal{H}_n$ and PLECO to achieve a better efficiency.

Let

$$\begin{cases} UB = w \cdot \lceil N/w \rceil \\ LB = w \cdot (\lceil N/w \rceil - 1) \end{cases},$$

where w is the desired word size and N is the size of the modulus n. Then we define the following modified version of $\mathcal{H}_n$.

$$\mathcal{RH}_n(x) = \mathsf{str}_{UB}(\mathsf{Rabin}_n(1 + \mathsf{int}(\mathrm{KECCAK}_{LB}(x))))$$

By replacing $\mathcal{H}_n$ by $\mathcal{RH}_n$, the software performance may be improved, because operations are applied to a multiple of words. But if the inputs into Rabin_n are so small that their squaring results do not need to be modulo n, then adversaries can easily compute the original inputs. The smaller LB is, the higher the probability of Rabin_n getting such inputs will be. Therefore, LB should not be too small.

To unify lengths of internal variables, we may simply substitute LB with UB, and get the following hash function.

$$\mathcal{RH}'_n(x) = \mathsf{str}_{UB}(\mathsf{Rabin}_n(1 + \mathsf{int}(\mathrm{KECCAK}_{UB}(x))))$$

The collision probability of $\mathcal{RH}'_n$ will be higher, as there are inputs larger than n that cause collisions, e.g., $\mathsf{Rabin}_n(x + n) = \mathsf{Rabin}_n(x)$. However, the overall security of the password hashing scheme should still be fine, since it will be difficult to construct inputs with such additional differences through KECCAK.

It is also possible to remove the operation of adding one in $\mathcal{H}_n$, i.e., defining the following function to replace $\mathcal{H}_n$.

$$\mathcal{SH}_n(x) = \mathsf{str}_N(\mathsf{Rabin}_n(\mathsf{int}(\mathrm{KECCAK}_{N-1}(x))))$$

There is a negligible chance that $\mathrm{KECCAK}_{N-1}(x)$ outputs zero, and the result of $\mathcal{SH}_n$ will be an all-zero string. If we treat KECCAK as a pseudorandom function, this probability will be $1/2^{N-1}$. Even if this incident happens, it will likely disappear when $\mathcal{SH}_n$ is iterated for multiple times with an

incremental counter. Henceforth, the security level of the entire design of PLECO will not be influenced.

6. PERFORMANCE ANALYSIS

In this section, we discuss the time and memory costs of PLECO and PLECTRON.

6.1 Tunable Time and Memory Costs

The designs of PLECO and PLECTRON provide two parameters, *tcost* and *mcost*, for applications to tune their time and memory consumptions.

The parameter *mcost* adjusts the amount of memory that needs to be present during the computations of PLECO and PLECTRON. The memory usage is expected to be around

$$\mathsf{size}(n) \cdot mcost$$

bits. Due to the sequential memory-hardness property inherited from ROMix, without having such amount of memory, the computation time of PLECO and PLECTRON will increase dramatically.

The parameter *tcost* has limited ability to adjust the time usage of PLECO and PLECTRON, since the total time cost also relies on the memory usage. To complete a full computation of PLECO, it requires

$$2 \cdot mcost \cdot tcost + tcost + 1$$

invocations of $\mathcal{H}_n$. PLECTRON needs one more invocation of KECCAK than PLECO.

Although our work aims to provide flexible solutions that can be tuned by users or developers for different applications, we would like to discuss a little bit about how to choose cost parameters properly in practice. At the time of writing, the graphic cards (GPUs) on the market have up to thousands of cores and several gigabytes of memory, so each core may have couples of megabytes of memory on average. FPGA or ASIC based circuits usually have less memory per core than GPUs. Therefore, in order to effectively thwart capital-rich attackers for building large-scale cracking circuits, the memory usage of password-hashing or proof-of-work algorithms should require tens of megabytes of memory at minimum, e.g., $tcost \geq 2^{16}$ for PLECO and PLECTRON. Consider the design of Litecoin, in which scrypt is configured to consume only 128 kilobytes of memory. In our opinion, 128 kilobytes are too conservative, and thus cannot fully remove the advantages of attackers equipped with dedicated hardware.

For the choices of the time cost parameter *tcost*, it should be fine to stay with the minimum numbers, say 1 or 2, unless certain memory-constrained application scenarios want more control on computation time.

6.2 Efficiency of Software Implementations

PLECO and PLECTRON are built upon well-established cryptographic primitives, and their implementations have been studied for years. The modular squaring operation is the basis for efficient implementations of RSA encryption/signature widely used in TLS/SSL, and KECCAK is designed to be efficient in both software and hardware.

We have tested our initial implementations of PLECO and PLECTRON on a 2.6 GHz Intel Core i7 processor for 80-, 112-, and 128-bit security levels. For each security level, an RSA-like modulus $n = p \cdot q$ as well as a Mersenne number with similar bit-length (see Table 1) are chosen as the moduli in PLECO and PLECTRON, in order for performance comparison (Table 2). We set $mcost = 2^{16}$ when profiling the software performance, which means the programs will consume $2^{16}\mathsf{size}(n)$-bit memory, i.e., around 17 MB for PLECO/PLECTRON using the modulus $2^{2137} - 1$. We have also tested scrypt on the same machine, using the configuration $(N = 2^{14}, r = 8, p = 1)$ that yields a similar memory usage as PLECO/PLECTRON with 2048-bit moduli, and it takes 35 ms to compute scrypt.

Table 1: Parameters for Different Security Strengths

Security Strength	Size of RSA-Like Modulus (in bits)	Mersenne Number
80-bit	1024	$2^{1277} - 1$
112-bit	2048	$2^{2137} - 1$
128-bit	3072	$2^{3049} - 1$

The performance data in Table 2 shows that as the modulus size grows, the running time of the algorithms PLECO/PLECTRON will increase (along with the memory usage), and computation of the Rabin_n part will gradually dominate the running time. Moreover, using Mersenne numbers as moduli will yield much more efficient computations than choosing RSA-like ones.

Although slowness is somehow desirable in password hashing designs for thwarting large-scale password cracking, we should consistently improve the time efficiency of the implementations of PLECO and PLECTRON. For example, by reducing the computation time of $\mathcal{H}_n$, we will have more flexibility for the time parameter *tcost*. On the other hand, attackers are always trying to speed up their cracking methods, so there is no reason why we should stick to under-optimized implementations.

6.3 Shortcut with Private Information

It would be very attractive if password hashing algorithms could support private parameters or keys to speed up hashing computations. For example, legitimate servers with certain private information may compute or verify hash tags faster than attackers who have obtained only salts and hash tags. In this way, servers will save time and hardware costs without risking too much about the overall security.

Due to the nature of the modular exponentiation operation, if we know the factorization of the modulus n, e.g., knowing p and q for $n = pq$, the computation can be finished with less time, by using the Chinese Remainder Theorem (CRT). Such performance gain might not be obvious for Rabin_n, as its operations are simple. But if the discrete logarithm based hash function $\mathcal{G}_n(x) = g^x \pmod n$ is used in $\mathcal{H}_n$, the computation will be greatly accelerated if factors of n are known. But p and q should be kept securely as always, e.g., being encrypted or stored in a hardware security module.

Note that even if p and q are leaked to attackers, the over-

Table 2: Software performance of Pleco/Plectron with $tcost = 1$ and $mcost = 2^{16}$ (in s)

Modulus Size (in bits)	Pleco		Plectron	
	RSA-like Modulus	Mersenne Number	RSA-like Modulus	Mersenne Number
1024 / 1277	0.684	0.538	0.686	0.540
2048 / 2137	2.215	1.185	2.235	1.203
3072 / 3049	4.355	2.135	4.358	2.146

all security of Pleco and Plectron still has Keccak as a "fail-safe". With the private information, attackers can compute Pleco or Plectron as efficient as legitimate servers, but the brute-force search for original passwords may still be a must.

7. RELATED WORK

In this section, we will compare Pleco and Plectron with several other related algorithms. Especially, many new designs have been proposed recently, since an open competition about password hashing (a.k.a. PHC) is currently ongoing[3]. We only choose couples of designs in PHC to be discussed here, which may be the most typical or relevant to ours.

scrypt

As we mentioned in Section 2.3, scrypt presents the idea along with the first concrete design of sequential memory-hard algorithms. The internal structure of our designs Pleco and Plectron are based on ROMix of scrypt.

However, the overall design of scrypt is complicated. It uses BlockMix and the Salsa20/8 core [4] to construct an internal hash function to be used in ROMix, and adopts PBKDF2 with HMAC and SHA256 to process first and final messages. Therefore, it might be error-prone for developers to implement scrypt due to the involvement of multiple cryptographic primitives and its complicated structure.

Moreover, although the internal structure ROMix is proven to be sequential memory-hard, there are no security proofs for the overall design of scrypt. Especially, the Salsa20/8 core is not collision-resistant, so it appears that scrypt can hardly be proven to be collision-resistant, which might leave scrypt certain weaknesses in some application scenarios.

Makwa

Makwa is a password hashing function designed by Pornin [24], and to the best of our knowledge it is the only design proposed in the PHC that adopts asymmetric cryptographic operations. Makwa uses a RSA/Rabin-like operation that the intermediate value x is raised to the degree of $2w + 1$, i.e., $y = x^{2w+1}$, where w is a time/work cost parameter and n is a Blum integer serving as a modulus.

Makwa is not designed to be memory-hard, and thus has very limited ability to thwart brute-force password cracking based on special hardware.

Catena

Catena is designed by Forler, Lucks and Wenzel, as a provably secure password scrambler that can be used for key derivation or proof of work/space [12].

The one-wayness of Catena is guaranteed by its underlying hash function; while the security of Pleco is assured by both Keccak and the hard problem of integer factorization.

In order to avoid the random memory access pattern that makes cache-timing attacks possible [3], Catena does not employ sequential memory-hard structures like ROMix. Instead, Catena provides a new memory-hard property called λ-*memory-hard*, which focuses more on single-core settings and does not provide much resistance to parallel attacks with dedicated circuits. However, in cache-timing attacks, adversaries may need to fully or partially control victims' host machines in order to accurately measure timings, which is a difficult requirement. Thus, in our view, being sequential memory-hard is more desirable than avoiding cache-timing attacks, if these two goals are not achievable in one design of password hashing.

SQUASH

SQUASH is a challenge-response protocol for RFIDs designed by Shamir [31], and aims to provide provable security based on the Rabin cryptosystem. In SQUASH, a challenge is first mixed with a secret, and then processed by an optimized implementation of Rabin encryption scheme with Mersenne numbers as moduli. Our idea of combining symmetric and asymmetric cryptographic algorithms originates from the design of SQUASH.

Ouafi and Vaudenay has shown that SQUASH is insecure if the mixing function is linear [22]. Pleco and Plectron should not suffer from the same weakness, since the cryptographic hash function Keccak is employed as a mixing function.

8. CONCLUDING REMARKS

We propose two provably secure password hashing algorithms, Pleco and Plectron, which are built upon well-studied cryptographic primitives, such as 1) a provably one-way function Rabin_n based on the hard problem of integer factorization, 2) the SHA-3 hash competition winner Keccak, and 3) the sequential memory-hard construction ROMix. We prove that Pleco and Plectron inherit the security properties of Rabin_n, Keccak and ROMix, i.e., one-wayness, collision resistance and sequential memory-hardness. The designs of Pleco and Plectron provide two parameters *tcost* and *mcost* that can be tuned for different application scenarios.

In order to fully utilize the memory-hardness property of

[3]Its official website is at `https://password-hashing.net`.

ROMix, the internal hash function $\mathcal{H}_n$ should be as fast as possible, since during a fixed time period the total amount of memory that can be consumed is limited by the computational speed of $\mathcal{H}_n$. The current design of $\mathcal{H}_n$ is based on modular squaring of big integers, so it may not be fast enough for certain devices with constrained CPU power. We have proposed several modified designs of $\mathcal{H}_n$ that have better software efficiency in Section 5.4. However, the security of these potential solutions needs further investigations.

It is encouraging to design password hashing algorithms by combining asymmetric and symmetric premitives, since the combined algorithms may offer provable security and possibilities of server-specific computational shortcuts. We expect more password hashing designs consisting of both asymmetric and symmetric components to appear.

Acknowledgments

The authors would like to thank Samuel Neves for bringing up the paper about RSA-UFOs.

9. REFERENCES

[1] J. P. Aumasson and D. J. Bernstein, SipHash: a fast short-input PRF, Progress in Cryptology, INDOCRYPT 2012, LNCS 7668, pp. 489–508, 2012.

[2] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, Recommendation for key management, part 1: general (revision 3), *NIST Special Publication 800-57*, 2012.

[3] D. J. Bernstein, Cache-timing attacks on AES, *available at* `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[4] D. J. Bernstein, The Salsa20 family of stream ciphers, *New Stream Cipher Designs*, LNCS 4986, pp. 8–97, 2008.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, The Keccak SHA-3 submission. *Submission to NIST (Round 3)*, 2011.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, Keccak implementation overview, version 3.2, *available at* `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`, 2012.

[7] E. Biham and A. Shamir, Differential cryptanalysis of DES-like cryptosystems, *Journal of CRYPTOLOGY*, vol. 4, no. 1, pp. 3–72, 1991.

[8] A. Bogdanov, L. R. Knudsen, G. Leander, and C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, PRESENT: an ultra-lightweight block cipher, *Cryptographic Hardware and Embedded Systems, CHES 2007*, LNCS 4727, pp 450–466, 2007.

[9] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, *et al.*, PRINCE – A low-latency block cipher for pervasive computing applications, *Advances in Cryptology - ASIACRYPT 2012*, LNCS 7658, pp. 208–225, 2012.

[10] W. Diffie, Exhaustive cryptanalysis of the NBS data encryption standard, vol. 10, no. 6, pp. 74-84, *IEEE Computer*, 1977.

[11] M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalcin and R. Zimmermann, Evaluation of standardized password-based key derivation against parallel processing platforms, *Computer Security, ESORICS 2012*, LNCS 7417, pp. 716–733, 2012.

[12] C. Forler, S. Lucks and J. Wenzel, Catena: a memory-consuming password scrambler, *Cryptology ePrint Archive, Report 2013/525*, 2013.

[13] J. K. Gibson, Discrete logarithm hash function that is collision free and one way, *IEE Proceedings E (Computers and Digital Techniques)*, 138, no. 6, pp 407–410, 1991.

[14] M. E. Hellman, A cryptanalytic time-memory trade-off, *Information Theory, IEEE Transactions on*, vol. 26, no. 4, pp. 401–406, 1980.

[15] B. Kaliski, PKCS# 5: Password-based cryptography specification version 2.0, *RFC 2898, available at* `http://www.ietf.org/rfc/rfc2898.txt`, 2000.

[16] X. Lai and J. L. Massey, A Proposal for a New Block Encryption Standard, LNCS 473, pp. 389–404, 1991.

[17] P. Leyland, Factorization of Mersenne numbers, $M_n = 2^n - 1$, *available at* `http://www.leyland.vispa.com/numth/factorization/factors/mersenne.txt`, 2008.

[18] M. Matsui, Linear cryptanalysis method for DES cipher, *Advances in Cryptology, EUROCRYPT 1993*, pp. 386–397, 1994.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL, 1997.

[20] R. M. Needham and D. J. Wheeler, TEA extensions, *available at* `http://www.movable-type.co.uk/scripts/xtea.pdf`, 1997.

[21] NIST Computer Security Division, The SHA-3 cryptographic hash algorithm competition, *available at* `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[22] K. Ouafi and S. Vaudenay, Smashing squash-0. *Advances in Cryptology, EUROCRYPT 2009*, LNCS 5479, pp. 300–312, 2009.

[23] C. Percival, Stronger key derivation via sequential memory-hard functions, *BSDCan*, 2009.

[24] T. Pornin, The MAKWA password hashing function – specifications v1.0, *available at* `https://password-hashing.net/submissions/specs/Makwa-v0.pdf`, 2014.

[25] N. Provos and D. Mazieres, A future-adaptable password scheme, *USENIX Annual Technical Conference, USENIX 1999*, pp.81–91, 1999.

[26] M. O. Rabin, Digitalized signatures and public-Key functions as intractable as factorization, *Technical Report*, MIT, 1979.

[27] P. Rogaway and T. Shrimpton, Cryptographic hash-function basics: definitions, implications and separations for preimage resistance, second-preimage resistance, and collision resistance, *Fast Software Encryption, FSE 2004*, LNCS 3017, pp. 371-388, 2004.

[28] T. Sander, Efficient accumulators without trapdoor, *Second International Conference on Information and Communication Security, ICICS 99*, LNCS 1726, pp. 252-262, 1999.

[29] B. Schneier, Description of a new variable-length key, 64-bit block cipher (Blowfish), *Fast Software Encryption, FSE 1994*, LNCS 809, pp. 191–204, 1994.

[30] R. Senderek, A discrete logarithm hash function for RSA signatures, *available at* `http://senderek.com/SDLH/discrete-logarithm-hash-for-RSA-signatures.ps`, 2003.

[31] A. Shamir, SQUASH – A new MAC with provable security properties for highly constrained devices such as RFID tags, *Fast Software Encryption, FSE 2008*, LNCS 5086, pp 144-157, 2008

APPENDIX

A. TEST VECTORS

For testing and reference, we give the following input parameters and their corresponding hash output of PLECTRON, using the Mersenne number $2^{2137} - 1$ as the modulus. Please note that the hexadecimal numbers for the entries *salt*, *pass* and *tag* represent byte strings, e.g., `546865` means a string *The*, and long strings are written in multiple lines.

salt	4c880aa553669c3869f62b389c2c3499
pass	54686520717569636b2062726f776e20
	666f78206a756d7073206f7665722074
	6865206c617a7920646f67
tcost	2
mcost	1024
hsize	256
tag	7969ad4aae09ba48e61cc5e348f1de39
	c15475d69eee42cffe8770a88f2f3e93

B. THE ALGORITHM NAMES

Pleco or Plecostomus is a kind of fishes that is very popular among aquarists, as Pleco fishes help keeping water clean. The word Plecostomus itself means folded mouth.

Plectron is a small piece of metal or plastic that is used to plunk musical instruments.