A multi-round bilinear-map-based secure password hashing scheme

1st Csanád Bertók
Faculty of Informatics
University of Debrecen
Debrecen, Hungary
bertok.csanad@inf.unideb.hu

2nd Andrea Huszti
Faculty of Informatics
University of Debrecen
Debrecen, Hungary
huszti.andrea@inf.unideb.hu

3rd Tamás Kádek
Faculty of Informatics
University of Debrecen
Debrecen, Hungary
kadek.tamas@inf.unideb.hu

4th Zsanett Jámbor
Faculty of Informatics
University of Debrecen
Debrecen, Hungary
jamborzs96@gmail.com

***Abstract*—We construct a multi-round, secure password hashing scheme that is designed to be resistant against off-line attacks, such as brute force, dictionary and rainbow table attacks. We compare our scheme to the password hashing algorithms used in practice from the point of view of the technical requirements of the Password Hashing Competition. We provide a security analysis, which shows that the proposed algorithm is also collision, hence second pre-image resistant.**

***Index Terms*—password hashing scheme, bilinear map, multi-round, off-line attacks**

I. Introduction

Many entity authentication systems depend on password-related information that is stored at the service provider. When a user logins the service, the hash of the given password is computed and compared with the stored hash. If the two hashes match, the authentication is successful. Users may choose "weak" passwords or not change the default passwords, which are easy for attackers to guess. Besides the password, usually, a salt value is also used. Salt is a short (12 - 48 bits) random piece of data that is concatenated with the password before hashing. It is then stored with the hash of password information. An attacker who succeeds in stealing the password file or database is forced to run an exhaustive, computationally expensive off-line attack to find the users' passwords from the salted hashes, *i.e.* testing each possible password with each possible salt value. In the scientific literature, several one- and two-factor authentication schemes are proposed, where passwords play an essential role ([1], [10]–[12]).

Passwords are also utilized in order to generate secret cryptographic keys. Password-authenticated key exchanges (PAKE [3]) and password-based key derivation functions (PBKDF [21],[17],[22], [6]) are typical examples.

There have been several reports of vulnerabilities being discovered in password management applications that show that passwords are not stored securely. In Tesla SolarCity Solar Monitoring Gateway, Digi ConnectPort X2e applied a .pyc file to store the cleartext password for the Python user account ([7]). In 2019, a study by Independent Security Evaluators (ISE) found vulnerabilities in password management software [16]. In version 4 of 1Password, it was noticed that the master password in plain text is available in RAM. Then in 2021, Passwordstate suffered a supply chain attack ([20]). Attackers sent a compressed file to the victim device in which malicious code was able to obtain users' master passwords. Numerous studies have discoursed the security vulnerabilities of IoT ecosystems ([2], [14]). The MyQ system ([2]) did not require strong passwords, which were easy targets for brute-force attacks. Hence strong password protection or two-factor authentication is needed.

Several password hashing schemes (PHS) have been proposed. The Password-Based Key Derivation Function 2 (PBKDF2) ([13]) is included in the RSA Laboratories' Public-Key Cryptography Standards (PKCS) series (PKCS #5 v2.0) and the RFC 2898. There are hashing algorithms that are deliberately designed to be slow in order to hobble attackers conducting an off-line attack. For example, the bcrypt hashing scheme ([18]) can be configured with a cost factor that exponentially increases its execution time by requiring a sequential series of computations. The winner of the Password Hashing Competition ([15]) is the Argon, which can be used for password hashing and key derivation. Its main feature is the protection against time-memory trade-off attacks, where the adversary can trade memory for time and do his job on fast hardware with low memory.

II. Our contribution

We construct a multi-round, secure password hashing scheme and provide a security and efficiency analysis. In [4] a password registration scheme is introduced, where the passwords are stored with bilinear maps in a single round. One-wayness property of the bilinear map is showed. Here we consider the solution as a stand-alone password hashing scheme designed to be multi-round, hence by increasing the number of rounds, we can slow down the algorithm to be resistant against brute force and dictionary attacks. We provide a security analysis, which shows that the proposed algorithm is also second pre-image and collision resistant. By applying

The presented research has been partially supported by the SETIT Project (no. 2018-1.2.1-NKP-2018-00004) and TKP Project (no. TKP2020-NKA-04). SETIT Project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme. Project no. TKP2020-NKA-04 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2020-4.1.1-TKP2020 funding scheme. Research was supported by Eötvös Loránd Research Network.

salt values, the scheme is secure against rainbow table attacks. We compare our scheme - in terms of efficiency - against some common password hashing algorithms. The proposed algorithm is evaluated on three technical aspects, runtime performance, memory usage and code size, which are the requirements of the Password Hashing Competition. We show that our algorithm forms a suitable alternative for password hashing.

A. Outline of the article

In Section III, we describe off-line attacks, in Section IV the preliminaries, basic definitions and the way of mapping to elliptic curve groups are given. In Section V, we present the proposed scheme, which includes the implementation issues. In Section VI, the security analysis is given. Section VII contains an efficiency analysis that is followed by our conclusion in Section VIII.

III. Off-line attacks

One security challenge where one-way functions became widely used is password storage. Storing only their hash value keeps the users' passwords hidden from the attackers and even the verifier while providing the necessary information to carry out the user authentication. In this section, the off-line attacks, *i.e.* brute force, dictionary and rainbow table attacks, are reviewed.

A. Brute force and dictionary attacks

In the case of password-based authentication, supposing that creating the inverse of the one-way hash function is technically impossible, determining a hash domain element for a hashed value can be only done by testing the hash value of all possible input strings – plain text passwords – in a given domain, trying to find hash value matches. If one of the input strings has the same hash value as the one stored to verify the password, it can be used for successful authentication even if it does not match the original password precisely.

In dictionary attacks, the adversary uses a predefined list of words and phrases, a dictionary. It is generated from the most often used passwords collected from multiple resources. There are forensic applications that index all the words stored on the victim's hard drive. The adversary tests each word in the dictionary, whether its hash is in the database or password file.

It is assumed that during off-line attacks, the adversary gains access to not only the hashes but also the salt values. Salt values make the attack slower, hence for each possible password, each salt value stored in the database should be tested. On the other hand, finding the inverse of a given hash value by applying brute force or dictionary attacks can be fast and successful, the salt value does not influence the attack. In such cases, the hash function calculation is slowed down by making them multi-round. The the number of rounds, the slower the hash calculation is.

B. Rainbow tables

The TMTO (time-memory trade-off) cryptanalysis is widely used against one-way functions. The challenge is to give an efficient way to find an inverse of a given hash value. Like the rainbow table attack, several methods are borne based on Hellman's original idea published in 1980 ([9]). His approach was to reduce the time of cryptanalysis with the help of precalculated data.

To calculate the value of a hash function on every attempt is a high time cost. Rainbow tables realize time-memory trade-off attacks to crack passwords at high speed using pre-computation tables.

Pre-generated key-value mappings can be stored to fasten the process. This mapping uses the hashed values of passwords as keys, so a successful hash-value lookup results in a string that can replace the password during the authentication process. The available storage space limits the size of the mapping.

Following Hellman's idea, not only hash value and input pairs but hash value chains can be pre-generated. Each chain starts with a randomly generated password, and the next possible password is calculated from the previous by using the hash function. However, the range of the hash function and the domain of the password texts usually differ, so the hash values must be transformed using a reduction function to the given password domain. Afterwards, the next hash value member of the chain is generated. The ith chain with k hash value members using the hash function H and the reduction function R can be written as

$$p_{i,1} \xrightarrow{H} c_{i,1} \xrightarrow{R} p_{i,2} \xrightarrow{H} c_{i,2} \xrightarrow{R} p_{i,3} \rightarrow \cdots \rightarrow p_{i,k} \xrightarrow{H} c_{i,k},$$

where only the first and the last element of each chain ($p_{i,1}$,$c_{i,k}$) must be stored. To find a password domain element for a given hash value $\tilde{c}_1$ first, the following chain needs to be calculated:

$$\tilde{c}_1 \xrightarrow{R} \tilde{p}_2 \xrightarrow{H} \tilde{c}_2 \xrightarrow{R} \tilde{p}_3 \rightarrow \cdots \rightarrow \tilde{p}_j \xrightarrow{H} \tilde{c}_j.$$

If there is a chain ending by $c_{i,k}$, where $\tilde{c}_j = c_{i,k}$ for some i, then

$$h^{-1}(\tilde{c}_1) = p_{i,k+1-j}.$$

The rainbow table is a space/time trade-off, and the purpose of the salt is to make that more costly. For each possible password all possible salt values should be considered, hence one table should generated for each possible salt. With 12 bits of salt (used by early Unix systems) that would mean 4096 different tables. That is feasible and achievable on modern storage devices. Increasing the size of the salt makes the rainbow table generation computationally infeasible, although if we only apply a single salt value which is built in the source code for all user passwords then rainbow table generation becomes feasible. Thus choosing different salt values for different client passwords is essential.

IV. Preliminaries

Our scheme is based on the bilinear map. Let us review the definition of the admissible bilinear map [5].

Definition 1. Let $\mathbb{G}$ additive and $\mathbb{G}_{\mathbb{T}}$ multiplicative be two groups of order p for some large prime p. A map $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{T}}$ is an admissible bilinear map if satisfies the following properties:

1) Bilinear: We say that a map $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{T}}$ is bilinear if $\hat{e}(aP, bQ) = \hat{e}(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and all $a, b \in \mathbb{Z}$.
2) Non-degenerate: The map does not send all pairs in $\mathbb{G} \times \mathbb{G}$ to the identity in $\mathbb{G}_{\mathbb{T}}$. Since $\mathbb{G}$, $\mathbb{G}_{\mathbb{T}}$ are groups of prime order, if P is a generator of $\mathbb{G}$ then $\hat{e}(P, P)$ is a generator of $\mathbb{G}_{\mathbb{T}}$.
3) Computable: There is an efficient algorithm to compute $\hat{e}(P, Q)$ for any $P, Q \in \mathbb{G}$.

The Weil and Tate pairings prove the existence of such constructions. Typically, $\mathbb{G}$ is an elliptic-curve group and $\mathbb{G}_{\mathbb{T}}$ is the multiplicative group of a finite field.

One of the well-known problems assumed to be infeasible in elliptic curve cryptography is the Computational Diffie-Hellman Problem (CDHP) ([19]). We review the definition for elliptic curve groups.

Definition 2. Let $\mathbb{G}$ be a cyclic group with generator $G \in \mathbb{G}$ and let $xG, yG \in \mathbb{G}$. The Computational Diffie-Hellman Problem is to compute xyG.

A. Mapping into Elliptic Curve groups

In the proposed password hashing algorithm, the initial password (and then the "intermediate" passwords after each round) is mapped to a point on an elliptic curve. For this we follow the technique in [4] and [8]. We only highlight the main steps here, for the exact technical details, we refer to [8].

Let q be a prime congruent to 3 modulo 4. Consider the supersingular elliptic curve $E : y^2 = x^3 + ax$ over $\mathbb{Z}_q$. The chosen transformation is as follows

$$tr : \mathbb{Z}_q \longrightarrow E(\mathbb{Z}_q)$$
$$x \mapsto \left(\varepsilon(x) \cdot x, \varepsilon(x)\sqrt{\varepsilon(x) \cdot (x^3 + ax)}\right),$$

where $\sqrt{\cdot}$ denotes the square root over $\mathbb{Z}_q$ and $\varepsilon(x) = \left(\frac{x^3+ax}{q}\right)$, where $\left(\frac{\cdot}{q}\right)$ is the Legendre symbol.

This encoding induces a bijection in the form of $\mathbb{Z}_q \setminus T \longrightarrow E(\mathbb{Z}_q) \setminus W$, where T is the set of roots of $f(x) = x^3 + ax$ in $\mathbb{Z}_q$ and W is the set of Weierstrass points of $E(\mathbb{Z}_q)$.

We point out that the encoding is effectively computable. Mapping tr can be calculated by using a single exponentiation and several multiplications, no need to calculate either the Legendre symbol or the square root. For the full algorithm we refer to [4] and [8].

V. The proposed scheme

We propose an efficient password storage algorithm that hashes passwords using bilinear mapping. We investigate the feasibility of using bilinear mapping for password storage while considering the requirements of the Password Hashing Competition (PHC). Password Hashing Competition ([15]) was based on the following minimum technical requirements:

- Be able to handle passwords between 0 and 128 bytes, regardless of the encoding.
- The salt length of 16 bytes.
- The output length of 32 bytes.
- One or more parameters to configure time (t_cost) and/or memory demands (m_cost).

These criteria are met by our algorithm. As a first step, we map the password to a prime field of a given size depending on the possible length of the passwords. In our implementation, we have chosen a 512 bit-long prime. The salt we use is an elliptic curve point that exceeds 16 bytes. Finally, the output has also met the criteria, our output length is 512 bits. In our case, the configurable parameter is the time (t_cost), which we can be set by the number of rounds.

In addition, we have designed a system that fits the most common password-based authentication protocols. The password is stored on the server-side as a salted hash value.

Besides, our goal is to design a password storage algorithm that can be slowed down by increasing the number of rounds. We also examine the memory usage and measure the size of the code of the algorithm, and compare it with algorithms that are popular nowadays. The results show that the memory usage and the code size are less than the compared algorithms.

A. The password hashing scheme

The proposed algorithm generates a hash value for a given password and salt. It consists of two main parts: *(a)* a mapping into the elliptic curve group and *(b)* a bilinear mapping.

Let E denote an elliptic curve defined over a finite field $\mathbb{Z}_q$ and let $G \in E(\mathbb{Z}_q)$ be a generator point. Elliptic curve parameters are chosen in a way that the system resists all known attacks on the elliptic curve discrete logarithm problem in $\langle G \rangle$.

Precisely, a supersingular elliptic curve of the form $E : y^2 = x^3 + ax$ over $\mathbb{Z}_q$, where q is a prime congruent to 3 modulo 4 is chosen. The sets T and W are calculated, where T is the set of roots of $f(x) = x^3 + ax$ in $\mathbb{Z}_q$ and W is the set of Weierstrass points of $E(\mathbb{Z}_q)$.

The password is mapped into $\mathbb{Z}_q \setminus T$. This map should be constructed to be bijective. An ASCII-based solution, where the maximum length of the password bitstring is the size of q can be designed to be suitable.

Afterwards, we apply the mapping to the elliptic curve group (tr), hence an elliptic curve point is generated from the password. Mapping $tr : \mathbb{Z}_q \setminus T \mapsto E(\mathbb{Z}_q) \setminus W$ is bijective, consequently we still have a sufficiently large password-space. For $\mathbb{G}$ we use a cyclic subgroup of the r-torsion group of the curve. After multiplying the elliptic curve point with the cofactor we get an element of $\mathbb{G}$. Then the bilinear paring $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{T}}$ is computed with $\mathbb{G}_T$ being the group of the rth roots of unity inside the finite field $\mathbb{Z}_{q^k}$, where k is the so-called embedding degree, which in our case is 2. An

elliptic curve point $S \in \mathbb{G}$ is also generated randomly as a salt value.

For slowing down our password hashing algorithm, we construct rounds. The larger the number of rounds, the slower the algorithm is. One round consists of the transformation tr, the bilinear mapping $\hat{e}$ and a conversion of the result of $\hat{e}$ to an element of $\mathbb{Z}_q$ with the technique mentioned before. Then in the next round we can use the map tr to get an elliptic curve point from the output of the previous round.

In the first round, the bilinear mapping is calculated with the elliptic curve points in $\mathbb{G}$ generated from the password and the salt as input. In the following rounds, the inputs of the bilinear map are two points. One of them is the addition of the salt and iG, where i denotes the number of the round and G is the generator point in $\mathbb{G}$. The other point is the elliptic curve point $tr(m)$, where m is the converted result of the previous bilinear mapping. Finally, the converted output of the bilinear map of the last round is stored as a password hash value.

The proposed password hashing scheme is designed to be resistant against brute force and dictionary attacks, since the number of rounds can be set as large as necessary. Moreover, applying a 512 bit-long salt value that is different for each password makes the scheme secure against rainbow table attacks.

B. Our implementation

In the implementation, we use the reduced Tate pairing and the hashToCurve method, which executes the mapping tr. The input of our algorithm is a password, and it outputs the password hash value with the salt. One of the points is the salt added to iG, and the other is the curve point generated by the hashToCurve method. After each round the Convert method is applied to generate an element of $\mathbb{Z}_q$ from the output of the bilinear map. The pseudo-code of the algorithm is given as follows.

Algorithm The proposed algorithm

INPUT: password
OUTPUT: PswStore, S
1: Initialize $E(\mathbb{Z}_q)$
2: Initialize S
3: $PswStore \leftarrow Convert(password)$
4: **for** $i = 0$ up to number of rounds **do**
5: $R \leftarrow hashToCurve(PswStore)$
6: $PswStore \leftarrow TatePairing(R, S + iG)$
7: $PswStore \leftarrow Convert(PswStore)$
return $(PswStore, S)$

We use the following supersingular elliptic curve: $y^2 = x^3 + x$. We have chosen the size of q to be 512 bits. The prime we use:

$$q = 731329576256467855322039941411215536384068289627312830254310277821058411810144462486413246228592183502383911176278505421042514024101864935444574549103938 7$$

and the order of $\mathbb{G}$ and $\mathbb{G}_{\mathbb{T}}$ for the bilinear pairing:

$$r = 730750818665451459101842416358141509827966402561.$$

In [4] it is given for our setting that $|T| = 1$ and $|W| \leq 4$.

VI. SECURITY ANALYSIS

The following security requirements are considered during the design phase:

1) Pre-image resistance
2) Second pre-image resistance
3) Collision resistance

It is sufficient to show that the bilinear map is a one-way pairing to prove the pre-image resistance of the proposed password hashing algorithm. In [4] the authors prove that the one-wayness of the bilinear pairing and the infeasibility of the Computational Diffie-Hellman (CDH) problem are related. More precisely, the following theorem is given.

Theorem A. *Let $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be a bilinear map. Let $\langle G \rangle = \mathbb{G}$ and $\langle g \rangle = \mathbb{G}_T$ be any elements such that $\hat{e}(G, G) = g$. If the CDH problem is infeasible for $g, g^a, g^b \in \mathbb{G}_T$ with any $a, b \in \mathbb{Z}_q$, then $\hat{e}$ is a one-way pairing.*

Based on this theorem, pre-image resistance of the algorithm is proved, provided that the CDH problem is hard.

The bilinear pairing is considered over torsion groups of the elliptic curve, the order of these groups is smaller than q. The possibility of the event that two passwords (whereby password we mean a point on E) yield the same hash value is calculated. From Section V-A we know that the size of $\mathbb{G}$ is r and since $E[r] \simeq \mathbb{Z}_r \times \mathbb{Z}_r$, the size of the r-torsion is r^2 with $r + 1$ cyclic groups of order r. This means that every point of E will be mapped into one of these $r + 1$ groups. For every point (except $\mathcal{O}$) in the same subgroup, the bilinear pairing gives the same result. If r is sufficiently large, then the probability of a collision is negligible. By using combinatorics, it can be computed that for our chosen curve, this possibility is 10^{-48}. Since this probability is negligible, we can say that our algorithm (with the chosen curve) is collision and thus second pre-image resistant.

VII. EFFICIENCY AND MEMORY USAGE ANALYSIS

In this section, we provide a detailed efficiency analysis for our scheme. In case of off-line attacks the password storage algorithm should be as slow as needed. Assuming that the adversary knows the salt, the slower the hash algorithm, the harder it is to find the password.

For the comparison, we ran the algorithms with a different number of rounds and examined their runtime performance. The algorithms hash the password `"hky6"`. We have compared the runtime performance of Bcrypt to our algorithm. Moreover, we have also chosen an asymmetric algorithm, the RSA, to contrast with our one.

The table below shows how many seconds the algorithms need to hash the password considering the same number of

rounds. At first sight, it is obvious that our construction is much slower than the Bcrypt and RSA.

Number of rounds	Bcrypt	Our algorithm	RSA
16	0,0030458	0,5346845	0,009764
32	0,0037977	0,8726219	0,0090346
64	0,0069453	1,7379774	0,0251674
128	0,0130193	1,5386831	0,0334561
256	0,023243	3,5953085	0,0638214
512	0,0431535	5,3515215	0,1371731
1024	0,087049	10,3966082	0,2013071
2048	0,167253	20,9222832	0,452279
4096	0,3439718	46,5067361	0,7515071
8192	0,6667411	86,7408044	1,3365767

If we want to keep the hash calculation under 1 second, in the case of Bcrypt, we need to run 8192 rounds. Compared to the efficiency of an asymmetric scheme, we can see that RSA requires 4096 rounds, while in our algorithm, 32 rounds are sufficient.

In the case of the Argon2 algorithm, other parameters can be set beside the number of rounds, so we test some of the recommendations for password hashing.

Type:	Argon2id	Argon2i	Argon2d
Memory size (KiB):	1048576	65536	4096
Number of iterations:	2	2	128
Degree of paralellism:	8	4	1
salt length:	16	24	16
Hash length:	32	24	32
Time (sec):	2,0267844	0,2813827	0,6441149

We can see that we can slow down the algorithms with different parameter settings as much as it is needed. Our algorithm is a suitable alternative since the runtime performance can be controlled.

Another aspect on which we compare our algorithm to the others is memory usage. Since our code is written in Python, we used the memory profiler module for the analysis.

The Memory Profiler is a Python module that monitors the memory consumption of a process as well as line-by-line analysis of memory consumption for python programs. It is important to note that the Memory Profiler itself consumes a significant amount of memory.

We compared our code, Bcrypt and Argon2 implemented in Python, while hashing the given password under 1 second.

First, we look at the memory consumption of the Bcrypt algorithm, where the number of rounds is 2^{13}.

Fig. 1. Bcrypt memory usage

```
  # Mem usage MiB    Increment OCC Line Contents
=== ============ ========== === ============================================
 18   20.12890625 20.12890625   1 @profile(precision=10,stream = fp1)
 19                             def bcryptTest():
 20   20.19140625  0.06250000   1   salt = bcrypt.gensalt(rounds=13)
 21   20.19921875  0.00781250   1   bcrypt.hashpw(b'1234',salt)
```

We parameterized the Argon2 algorithm according to the C implementation, except for the number of rounds, which we set to 128.

Fig. 2. Argon2 memory usage

```
  # Mem usage MiB    Increment OCC Line Contents
=== ============ ========== === ============================================
 23   20.08593750 20.08593750   1 @profile(precision=10,stream = fp)
 24                             def argonTest():
 25   20.09375000  0.00781250   2   argon2Hasher = argon2.PasswordHosher(
 26   20.08593750  0.00000000   1     type=argon2.Type.D,time_cost=128,
 27   20.08593750  0.00000000   1     memory_cost=2 ** 12,parallelism=1,
 28   20.08593750  0.00000000   1     hash_len=32,salt_len=16)
 29   20.17578125  0.08203125   1   argon2Hasher.hash("1234")
```

In our implementation we set the number of rounds to 32.

Fig. 3. Our algorithm memory usage

```
  # Mem usage MiB    Increment OCC Line Contents
=== ============ ========== === ============================================
795   22.03125000 22.03125000   1 @profile(precision=10,stream = fp)
796                             def pwHashTest():
797   22.03125000  0.00000000   1   toHash = '1234'
798   22.05859375  0.00781250  33   for i in range(0,32):
799   22.05859375  0.01953125  32     generatedPoint = ec.hashToCurve(toHash)
800   22.05859375  0.00000000  64     toHash = TatePairing\
801   22.05859375  0.00000000  32       .compute(TatePairing,P,generatedPoint,ec)\
802                                       .toString()
```

In the results, the first column is the line number of the code, and the next column (Mem usage) is the memory usage of the Python interpreter. The third column (Increment) shows the difference in memory of the current line with respect to the last one. The last column (Line Contents) shows the profiled code contained in the current line.

Finally, we examined the size of our code and compared it to the code size of Bcrypt and Argon2. Based on the data available for us, Bcrypt is about 20-30 kB, and Argon2 is about 80-110 kB. Our code is about between the code size of the Bcrypt and the Argon2.

VIII. Conclusion

There are several cases when attackers exploit password protection practices and reveal user passwords. We have constructed a multi-round bilinear-map-based solution for secure password storage. First, we showed that our algorithm had met all the minimum technical requirements given by the Password Hashing Competition and possessed the necessary security requirements. Then, we compared it with the password hashing algorithms nowadays popular and used in practice. After creating a detailed efficiency analysis, we can conclude that our algorithm forms a suitable alternative.

IX. Acknowledgement

We would like to thank the anonymous reviewers for their valuable remarks.

References

[1] S. Agrawal, P. Miao, P. Mohassel, P. Mukherjee, *PASTA: password-based threshold authentication.* In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. (2018), 2042–2059.

[2] T. Alladi, V. Chamola, B. Sikdar, *Consumer IoT: Security vulnerability case studies and solutions.*, IEEE Consumer Electronics Magazine, **9.2**, (2020), 17–25.

[3] S. M. Bellovin, M. Merritt, *Encrypted key exchange: Password-based protocols secure against dictionary attacks,* Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy. IEEE, (1992), 72–84.

[4] Cs. Bertók, A. Huszti, Sz. Kovács, N. Oláh, *Provably Secure Identity-Based Remote Password Registration*, Publ. Math. Deb., accepted.

[5] D. Boneh, M. Franklin, *Identity based encryption from the Weil pairing.* SIAM J. on Computing, **32**, (2003), 586–615.

[6] M. Brož, V. Matyáš *Selecting a New Key Derivation Function for Disk Encryption*. Foresti S. (eds) Security and Trust Management. STM 2015. Lecture Notes in Computer Science, **9331**, (2015), 185–199.

[7] *CVE-2020-9306*, https://www.cvedetails.com/cve/CVE-2020-9306/.

[8] P.-A. Fouque, M. Tibouchi, *Deterministic Encoding and Hashing to Odd Hyperelliptic Curves.* Pairing-Based Cryptography - Pairing 2010. (2010), 265–277.

[9] M. E. Hellman. *A cryptanalytic time-memory trade off.* IEEE Transactions on Information Theory, **26**, (1980), 401-–406.

[10] A. Huszti, N. Oláh, *A simple authentication scheme for clouds*, Proceedings of IEEE Conference on Communications and Network Security (CNS), (2016), 565–569.

[11] A. Huszti, N. Oláh, *Security analysis of a cloud authentication protocol using applied pi calculus.*, International Journal of Internet Protocol Technology, **12**, (2019), 16–25.

[12] A. Huszti, N. Oláh, *Scalable, password-based and threshold authentication for smart homes.*, International Journal of Information Security, (2022), 1–17.

[13] B. Kaliski., *RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0.*, Technical report, IETF, (2000).

[14] M. A. Khan, K. Salah, *IoT security: Review, blockchain solutions, and open challenges*, Future Generation Computer Systems, **82**, (2018), 395–411.

[15] *Password Hashing Competition*, https://www.password-hashing.net/.

[16] *Password Managers: Under the Hood of Secrets Management*, https://www.ise.io/casestudies/password-manager-hacking/.

[17] C. Percival, S. Josefsson. *The scrypt password-based key derivation function.* RFC, **7914**, (2016), 1–16.

[18] N. Provos, D. Mazieres, *A Future-Adaptable Password Scheme* Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, (1999), 1–13.

[19] I. Shparlinski *Computational Diffie-Hellman Problem.* In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security. Springer, Boston, MA. (2011)

[20] *Supply chain attack on the password manager Clickstudios*, https://csis.com/the-hub/peter-kruse-security-blog/supply-chain-attack-on-the-password-manager-clickstudios-passwordstate/.

[21] M. S. Turan, E. Barker, W. Burr, L. Chen, *Recommendation for password-based key derivation.* NIST special publication 800-132.

[22] F. F. Yao, Y. L. Yin, *Design and analysis of password-based key derivation functions*, IEEE Transactions on Information Theory, **51**, (2005), 3292–3297.