Argon2: The Better Password Hashing Function Than Bcrypt

Daniel Ryan Levyson 13516132
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516132@std.stei.itb.ac.id

***Abstract*—Password is a string to secure access into certain resources from any unauthorized parties. Anybody who is authorized to access the resources should be able to keep the password as a secret prevent resources stealing or misuse by those who do not have permission. The system which acts as the store of resources and also resposibles for giving access to the resources should be secure enough to manage the access verification. In order to authenticate person by using password, computer should have mechanism to verify the password. Saving plain password somewhere to be used for verification produces security problem. Someone who can somehow find the place where the password is stored and read the password can gain access to any resources in the system. The authentication system should have secure way to store and verify password. Bcrypt is hashing function which can be used to store and verify password securely. The development of computer hardware leads to increasing chance of cracking the security of hashing algorithm, including Bcrypt. Certain hardwares is specifically designed to run certain cracking algorithm optimally. More secure hashing algorithm is needed to overcome the possibility of cracking using high-end computer hardware. Argon2 is another hashing function which has the ability to overcome the level of computation power of current hardwares.**

***Keywords*—password, hash, encryption, security.**

I. Introduction

Most of information systems use authentication system to limit access of certain information. The authentication system is resposible to identify the accessor by asking credentials. Every users of the system would be asked to register credentials in the first time they use the system. One of the credential is password. System would store the password somewhere in certain way. To identify the accessor later, system would access the stored password and in certain way verify the given password.

The most easiest way for the system to implement the authentication system is storing the user's password to file as a plain text. Attacker is defined as someone who tries to gain access into restricted resources. In this case, attacker works by finding any possible ways in the system to retrieve the file which stores all the password used for verification. So, this kind of authentication system is not reliable, when the attacker succeed, authentication become useless.

The more sophisticated way to store the password is using encryption technique which will be explained more in the next chapter. Instead of storing plain password, the password will be stored after it is encrypted. When the attacker gained access to the stored passwords, the attacker can not know the real passwords. It seems good, but to encrypt password, the authentication system needs secret key which is used to change password to encrypted form and reverse. After the attacker got the encrypted passwords, the attacker only needs to search for the secret key somewhere in the system. When the attacker found the secret key, all encrypted passwords can be decrypted and the real passwords would be revealed.

Encryption is not reliable to be used for password storage, because the real password definitely can be revealed from the encrypted password by knowing the secret key. In the other hand, hash function produces possibility to store password securely, because hash function is invented to be one way. That implies the input of hash function has no way to be retrieved by knowing its output.

Creating own implementation of hash function to protect password is a bad idea. Hash function can be considered as secure if it is tested and analysed well enough by attempting to break its security. In the other hand, we cannot rely on hiding the algorithm to ensure the security, because hiding the algorithm will produce another concern of security.

SHA-1, HAVAL-128, MD4, MD5, and RIPEMD are known as cryptographic hash function which are also known suffering from collision attack. SHA-2 is a general purpose hash function. it needs a short time to compute. With today's computation power of computer hardware, SHA-2 vulnerable to bruteforce attack. There is SHA-3 as the latest SHA hash function. It is faster than SHA-2. That implies SHA-3 is worse in overcoming bruteforce attack.

In 1999 Bcrypt was invented as secure password hashing algortihm. Bcrypt has been tested and chosen for a long time for protecting password. But problem has been arisen along with the higher ability of computation hardware. A Hybrid system of ARM/FPGA SOCs can be used to attack Bcrpyt. In 2015, Argon2 won *Password Hashing Competition*. It is designed to overcome the weakness of Bcrypt in mitigating current computational power for cracking password hashing algorithm.

II. Encryption And Hash Function

Encryption and hash function are important concept used in cryptography. The main difference between the two is whether

or not the input can be retrieved later after it is transformed.

A. Encryption

Encryption is the process of hiding a message by changing its form in such way so that unauthorized parties cannot read the hidden message. The hidden message is called plaintext, and the result of encryption is ciphertext. Authorized parties should have the secret key to hide and read the message. The process of returning the plaintext from ciphertext is known as decryption.

We can write encryption as a function which map plaintext P to chipertext C.

$$E(P) = C$$

Decryption is the inverse of encryption. We can write decryption as a function which map chipertext C to plaintext P.

$$D(C) = P$$

We can also substitute C to E(P).

$$D(E(P)) = P$$

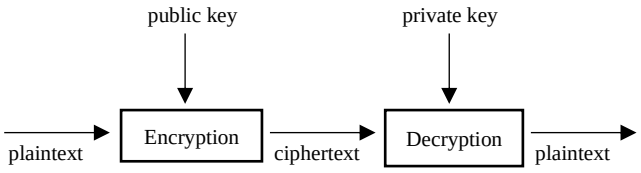
In history, encryption had been used since the age of roman empire. Julius Caesar, a roman emperor, use encryption to hide message that is sent to his governors. The technique of encryption is called Caesar Cipher. Caesar Chiper works by substituting every alphabetical characters in the plaintext with the next three character in alphabetical order.

Caesar Cipher can be represented more general with “three” substituted by variable K. So mathematically, we can write Caesar Cipher in the expression below.

$$E(P) = (P + K) \bmod 26 = C$$
$$D(C) = (C - K) \bmod 26 = P$$

In the example above, K is the cipher key. Because the cipher key used in encryption and decryption is the same, Caesar Chiper is called symmetric-key encryption. If the key used for encryption and decryption is different, the encryption is called asymmetric-key encryption or also known as public-key encryption.

In asymmetric-key encryption, there are public key and private key. As the name suggest, public key is for public use, it is not secret for everyone. Public key is used to encrypt the message. Private key is used to decrypt the ciphertext. Therefore, in asymmetric-key encryption, everyone can encrypt the message because the key used to encrypt is public. But only authorized parties can read the hidden message.



RSA algorithm is known as one of asymmetric encryption implementation. RSA algorithm has three parts: generating public and private key pair, message encryption, and message decryption. DES is known as one of symmetric encryption algorithm. DES is no longer considered as secure, but it becomes the fundamental understanding of block cipher.

Block cipher is a function which takes two input: k-bit string and n-bit string, and then returns n-bit string. k-bit string is a symmetric key for block cipher. Block cipher is known as powerful technique behind the strong encryption algorithm. In block cipher, there is basic component used to obscure the relationship between the key and the ciphertext which is called S-Box. Besides DES, another encryption algorithms that use the concept of block cipher are AES and Blowfish. Both are considered secure encryption algorithm until today.

B. Hash Function

Hash function is a function which takes random size input *k* and map *k* to value *v* which has fixed size. The very simple hash function use modulo operation, it has following form:

$$h(k) = k \bmod m = v$$

In the example above, the size of v depends on m. Because v has fixed size, there are cases when different input k gives the same value v as the output. For example, for m = 10, h(1) and h(11) has the same output v = 1. That condition is called collision. When using hash function for any purpose besides security, policy can be defined to handle the collision. For security purpose, collision in hash function is a security hole.

Good algorithm for cryptographic hash function should have following properties:

1. Pre-Image Resistance
   Input value k should be hard to find from known hash value v.

2. Second Pre-Image Resistance
   For input value k which has hash value v, it should be hard to find another input value which also output the same hash value v.

3. Collision Resistance
   It should be hard to find pair of input value which has the same output value.

In order to proof whether a hash function secure or not, we need to proof that the hash function has three of above

resistances criteria as good as possible.

If we look closer on encryption mechanism, we can actually use the idea of encryption to create hash function. Suppose we have a fixed plaintext and given another text as an input, we can use the given text as a secret key to transform the fixed plaintext into ciphertext. The secret key which is the given input is thrown away. By doing that, we create a hash function which the input k of hash function behaves as a secret key in encryption and the output v of hash function behaves as the chipertext.

C. Password Hashing Function

We already talked about two way function that is not good to be used in creating password storage. Using two way function for protecting password requires the system to store the secret key to be used in verifying password. Problem occurs when attacker can gain access to passwords and the secret key. All the encrypted passwords can be decrypted using the secret key.

Instead of using two way function, we can use one way function to protect password. The authentication system does not need to know the plain password, the system only needs to enter password into the function and compare the result with the one stored in password storage. It means we can actually use hash function to protect password.

There are several known attacks to break password hashing function besides brute force such as preimage attack, collision attack, dictionary attack, rainbow table attack and also side-channel attack. Any hash function that is weak to preimage attack and collision attack should be avoided for further usage in security. Dictionary attack is faster version of brute force, because it narrows the space of guessing by registering known words in dictionary to be used in brute force.

Because human always needs to remember the password, the password should be not too different from any meaningful and familiar words for human. Human's password is said to have high entropy. Even though human combines the password with number and non-alphabetical character, the entropy is still relatively high. Suppose someone's password is "?/Qu1cKbR0wNf0X/?". The password is hard enough but we can still see that it is created by modifying the words "quick brown fox". This is why dictionary attack exists.

Dictionary attack is faster than brute force but it still needs time. Rainbow attack is actually dictionary attack which decreases the processing time significantly but in the same time increases the required disk space. Rainbow attack precomputes the dictionary and it makes the process of knowing the plain password from its hash becomes much faster, because the process only compare the precomputed hash and the password hash, when any hash is matched, than the attacker knows the plain password before the password is precomputed. Even though this technique requires the attacker to gain access to all the hashed password, but the technique to secure the access to retrieve the hashed password is another security concern which is not discussed here.

To overcome the weakness of human's password, an attempt is made to lower human's password entropy. The idea is generating another string to be combined with human's password so that the combination will produce more rainbow table attack resistant hash. The generated string should be random for each password and long enough. The longer the string, the lower the entropy of password. This generated string is called Salt. Salt can be put before or after the password. Salt is also useful to make same password to have different hash. The authentication system should also store the Salt to be used when verifying password and it is considered as safe, not compromising security.

III. BCRYPT PASSWORD HASHING

As stated in the introduction, SHA-2 is actually good general hash function, but it is not secure enough to be used for hashing password. Hardware computational power is increasing over time. There might be possibility for attacker to brute force password by using combined high-end hardware computational power. Moreover, quantum computer exists nowadays which significantly faster than classical computer when doing brute force. In order to make brute force almost impossible, we should use slow hash function to hash password.

Bcrypt algorithm is hash function which has expensive key setup phase. Bcrypt can follow the increasing computational power of hardware, because it can be configured to be slower by increasing the number of iteration. Bcrypt utilizes Blowfish to setup the key. Blowfish is notable as complex symmetric-key block cipher. Following parts will discuss Blowfish and Bcrypt in more detail.

A. Blowfish as Password Hash Function

Blowfish is symmetric-key block cipher, designed by Bruce Schneier, which is known by its large key-independent S-Boxes and highly complex key schedule. Blowfish has 16 rounds Feistel network. A round-specific data derived from the cipher key is called a round key. A key schedule is an algorithm that calculates all the round keys from the key. Blowfish has 64-bit block size and 32 bits up to 448 bits key length.

For every round, Blowfish algorithm does the following actions:

1. XOR the left half of the data with the r-th P-array entry.
2. Use the XORed data as input for Blowfish's F-function.
3. XOR the F-function's output with right half of the data.
4. Swap L and R.

Blowfish's F-function will split 32-bit input into four eight bit quarters. S-boxes will transform the quarters which is 8-bit length to 32-bit output. Then, the output is added module 2^{32} and XORed to produce the final 32-bit output.

Following are the steps of key schedule algorithm in Blowfish:

1. Initialization of P-array and S-boxes with values derived from hexadecimal digits of pi (first 12 digits of pi in hexadecimal 3.243F6A8885A3…..).
2. Byte by byte, secret key is XORed with all the P-entries in order.
3. A 64-bit all-zero block is then encrypted with the algorithm as it stands.

4. The resulting ciphertext then replaces P1 and P2.
5. The same ciphertext is encrypted again with new subkeys, and the new ciphertext replaces P3 and P4.
6. This process will continue to replace the entire P-array and all the S-boxes.

Blowfish algorithm will run 521 times to generate all the subkeys. Blowfish only needs about 4 KB space of RAM. Its small usage of RAM makes Blowfish possible to be used in embedded systems.

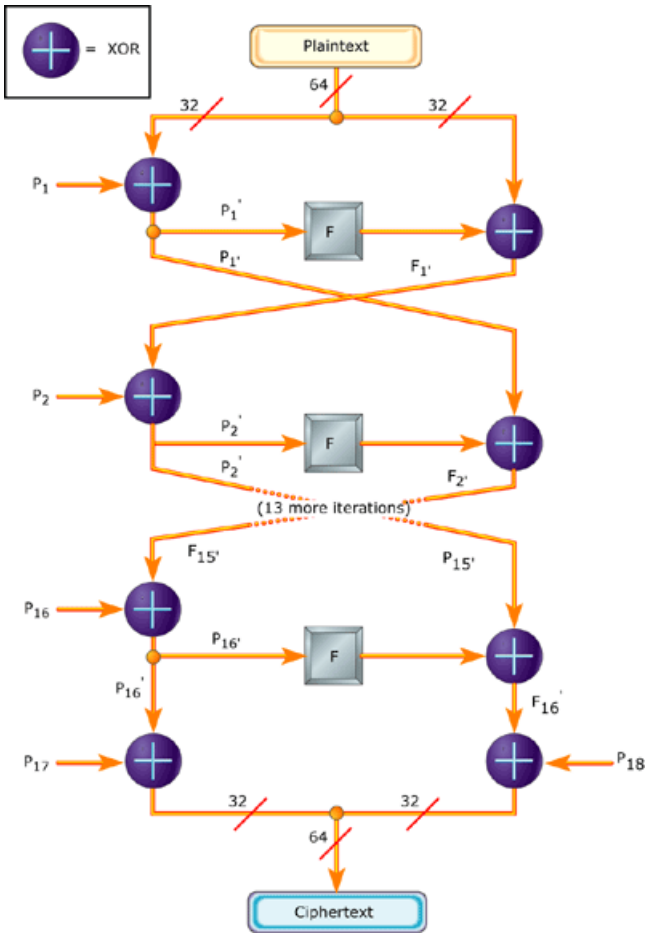


Figure 1 Blowfish algorithm.
Source: http://wiki.cas.mcmaster.ca/index.php/Blowfish

B. Bcrypt

Bcrypt is a hash function spesifically designed for password hashing by Niels Provos. It was developed to harden password storage of Unix authentication system. Bcrypt is adaptive to computational power of hardware. Bcrypt receives several parameters which one of the parameter is an iteration-count. The iteration-count can be configured to certain value which is slowing down cracking process for security but also fast enough for verifiying given password.

Bcrypt take advantages of the expensive key setup in Eksblowfish. Eksblowfish refers to expensive key schedule blowfish, it is a cost parameterizable and salted variation of the Blowfish block cipher. Eksblowfish takes three parameters: cost, salt, and key. The cost parameter is what makes this algorithm adaptive to computational power. Increasing the value of cost creates more expensive key schedule to be computed. The key parameter is user-chosen password. Eksblowfish returns set of subkeys and S-boxes.

Following are steps of Eksblowfish algorithm:

1. Copying the digits of number π first into subkeys, and also copying them into S-boxes.
2. Expanding the key by modifying the P-array and S-boxes based on the value of the 128-bit salt and the variable length key. It XORs all the subkeys in the P-array with encryption key. The i-th 32 bits of key are XORed with i-th of P.
3. Encrypting the key using Blowfish encryption algorithm for 2^{cost} times.

After getting set of subkeys and S-boxes from Eksblowfish, Bcrypt encrypts the text "OrpheanBeholderScryDoubt" repeteadly for 64 times in a mode called Electronic Codebook. Electronic Codebook mode is one of block cipher mode of operation. The result of encryption is then concatenated with the cost and salt to provide information for later verification process.

Bcrypt has following scheme:

$2b$[iteration]$[salt][hash value]

Following Bcrypt's password string has cost parameter equal to 12 which indicates 2^{12} key expansion rounds, salt "ZMqo8uLOikgx2eNcRZoMy9", and resulting hash "xad68L7lJZdL1ZAgcfl7p92hWyIjldG":

```
$2b$12$ZMqo8uLOikgx2eNcRZoMy9xad68L7lJZdL1ZAgcfl7p92hWyIjldG
```

The length of salt used in Bcrypt is 128 bits with Radix-64 encoding, so it is 22 characters. The length of hash value of Bcrypt is 184 bits, it is 31 characters length with Radix-64 encoding. So in total, authentication system will store string with length 58 plus digit of iteration number. For the input, user-chosen password length should not be longer than 72 bytes or the password will be truncated.

Bcrypt has been implemented in many programming languages such as C, C++, C#, Go, Java, Javascript, Perl, PHP, Python, Ruby, and other languages. It was originally used for OpenBSD authentication system, but nowadays it has been widely used to securely store password in many web applications.

IV. ARGON2 AS PASSWORD HASH FUNCTION

Other alternatives besides Bcrypt, which has been explained before, are HMAC, PBKDF2, and Scrypt. Bcrypt is better for password hashing than HMAC and PBKDF2 due to its flexibility in specifying computation cost. Bcrypt is invented to focus on computational cost. Nowadays, specialized computer chips such as FPGA, ASICs, and GPU can be obtained buy an attacker easier than huge amounts of memory. This fact open the

weakness of Bcrypt which focus on compuation cost. Scrypt is newer than Bcrypt. Scrypt aims for high memory, but the existence of a trivial time-memory tradeoff allows compact implementations with the same energy cost. Scrypt is not flexible in separating time and memory costs. Better hash function should be able to flexible enough to to trade large memory for fast computation and low memory for slow computation.

Argon2 is made by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 is invented to achieve memory-hard trait of a hash function. It is made to fill memory at fast rate and use multiple computing units effectively, but in the same time still providing defense against tradeoff attacks. Argon2 is able to fill 1 GB of RAM in a fraction of second. Argon2 is optimally designed for hardware with x86 architecture. It takes advantage of cache and memory organization of recent AMD and Intel processors., and smaller amounts even faster. It can scale to the arbitrary number of parallel computing units.

There are three available versions of Argon2: Argon2d, Argon2i, and Argon2id. Argon2d focuses on fast computation and uses data-depending memory acces. Argon2d is suitable for applications with no threats from side-channel timing attacks such as cryptocurrencies. Argon2i uses data-independent memory access. Argon2i is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks. Argon2id behaves like Argon2i for the first half of the first iteration over the memory and works as Argon2d for the rest, thus providing both bruteforce cost savings and side-channel attack, such as meltdown and spectre, protection due to time-memory tradeoffs.

There are two type inputs for Argon2: primary and secondary inputs. Primary inputs are message P, the password, and nonce S, the salt. Message P may have any length from 0 to $2^{32} - 1$ bytes. Nonce S may have any length from 8 to $2^{32} - 1$ bytes. Primary inputs are required to use Argon2.

Secondary inputs are optional and consist of several parameters as described below:

- Degree of parallelism p is any number from 1 to 2^{24}-1.
- Tag length τ is any integer number of bytes from 4 to 2^{32}--1.
- Memory size m is any integer number of kilobytes from 8p to 2^{32}-1.
- Number of iterations t is any integer number from 1 to $2^{32} - 1$.
- Version number v is one byte 0x13;
- Secret value K is any number from 0 to 2^{32}-1 bytes.
- Associated data X may have any length from 0 to 2^{32}-1 bytes.
- Type y of Argon2: 0 for Argon2d, 1 for Argon2i, 2 for Argon2id.

Argon2 uses internal hash function H and internal compression function G with two 1024-byte inputs and a 1024-byte output. Function G is based on its internal permutation. Hash function H is the Blake2b hash function. Function G is iterated m times.

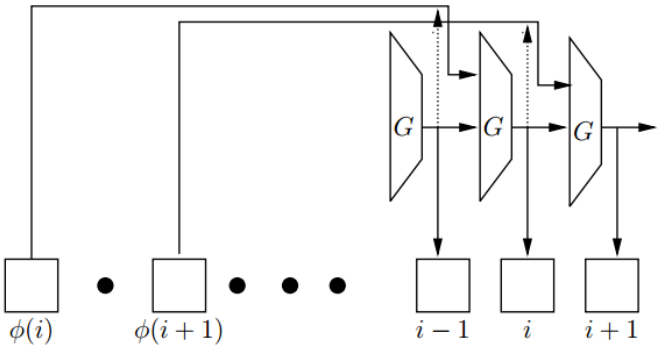


Figure 2 Argon2 mode of operation with no parallelism.
Source: https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf

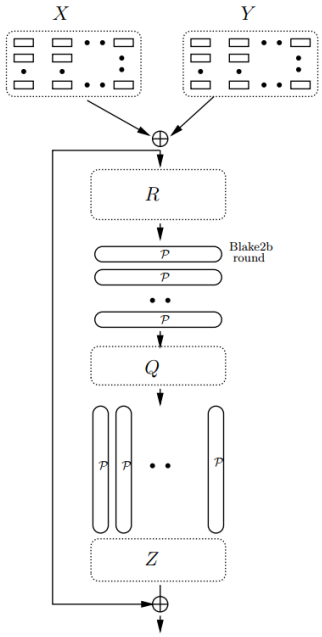


Figure 3 Argon2 compression function G.
Source: https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf

Suppose we want to hash a password “test”, the result of Argon2 function is:

```
$argon2i$v=19$m=1024,t=2,p=2$TmxLemFoVnZFaEJuT1NyYg$4j2ZFDn1fVS70ZExmlJ33rXOinafcBXrp6A6grHEPkI
```

We can see there are several parts on the resulting hash divided by dollar sign.

```
argon2i
v=19
m=1024,t=2,p=2
TmxLemFoVnZFaEJuT1NyYg
4j2ZFDn1fVS70ZExmlJ33rXOinafcBXrp6A6grHEPkI
```

The first part is the algorithm name which is argon2i for this example. The second part is the version number v. The third part is some secondary inputs for the algorithm, there are memory size m, time cost t, and degree of parallelism p. The fourth part is nonce S which is a random string and encoded in Base64. The size of nonce S is 16 bytes. The last part contains the hash value, encoded in Base64. The hash size is 32 bytes.

V. Conclusion

Although there are many ways to do authentication, password is still chosen to be the main way for authentication. Because of its extreme significance in securing access to resources, a system should have reliable authentication gate. Password hashing is a secure method to store password in order to hide the form of plain password from the attacker. Good hash function should be resistance to preimage attack, second preimage attack, and also collision attack. The threat of brute force, dictionary, and rainbow table attack demands an ideal hash function to be slow enough and should be able to scale to follow increasing computational power.

Even though Bcrypt has the ability to adapt with increasing computational power, it only focuses on computation time. Scrypt as the newer hash function lacks the ability of separating time and memory costs. Argon2 provides an improvement from the previous hash functions to mitigate general attacks in cryptography by considering today's computation power and specific hardwares.

VII. Acknowledgment

The author would like to thank all lecturers in Discrete Mathematics class who provide the basic knowledge needed to write this article and also give inspiration to write about this topic.

References

[1] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich. (2017). Argon2: the memory-hard function for password hashing and other applications. PHC Release.

[2] Zimuel, Enrico. (2017). Protecting passwords with Argon2 in PHP 7.2. https://framework.zend.com/blog/2017-08-17-php72-argon2-hash-password.html. Accessed on December 4, 2019.

[3] Preziuso, Michele. (2019). Password Hashing: Scrypt, Bcrypt and ARGON2. https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2-e25aaf41598e. Accessed on December 4, 2019.

[4] Munir, R. (2016). Matematika Diskrit. Bandung: INFORMATIKA.

[5] Niels Provos, David Mazières. (1999). A Future Adaptable Password Scheme. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference.

[6] Schneier, Bruce. (1994). Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html. Accessed on December 3, 2019.

[7] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive Report 2004/199, 16 Aug 2004, revised 17 Aug 2004. Accessed on December 3, 2019.

[8] Kennedy, David. (2015). Of History & Hashes: A Brief History of Password Storage, Transmission, & Cracking. https://www.trustedsec.com/2015/05/passwordstorage. Accessed on December 3, 2019.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 6 Desember 2019

Daniel Ryan Levyson 13516132