


Article

Efficient Implementation of a Crypto Library Using Web Assembly

BoSun Park ¹, JinGyo Song ² and Seog Chung Seo ^{1,*} 

¹ Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, Korea; 20175206@kookmin.ac.kr

² Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; sjk9304@kookmin.ac.kr

* Correspondence: scseo@kookmin.ac.kr; Tel.: +82-02-910-4742

Received: 16 September 2020; Accepted: 26 October 2020; Published: 3 November 2020



Abstract: We implement a cryptographic library using Web Assembly. Web Assembly is expected to show better performance than Javascript. The proposed library provides comprehensive algorithm sets including revised CHAM, Hash Message Authentication Code (HMAC), and ECDH using the NIST P-256 curve to provide confidentiality, data authentication, and key agreement functions. To optimize the performance of revised CHAM in the proposed library, we apply an existing method that is a four-round combining method and additionally propose the precomputation method to CHAM-64/128. The proposed revised CHAM showed an approximate 2.06 times (CHAM-64/128), approximate 2.13 times (CHAM-128/128), and approximate 2.63 times (CHAM-128/256) performance improvement in Web Assembly compared to JavaScript. In addition, CHAM-64/128 applying the precomputation method showed an improved performance by approximately 1.2 times more than the existing CHAM-64/128. For the ECDH using P-256 curve, the naive implementation of ECDH is vulnerable to side-channel attacks (SCA), e.g., simple power analysis (SPA), and timing analysis (TA). Thus, we apply an SPA and TA resistant scalar multiplication method, which is a core operation in ECDH. We present atomic block-based scalar multiplication by revising the previous work. Existing atomic blocks show a performance overhead of 55%, 23%, and 37%, but atomic blocks proposed to use only $P = (X, Y, Z)$ show 18%, 6%, and 11% performance overhead. The proposed Web Assembly-based crypto library provides enhanced performance and resistance against SCA thus, it can be used in various web-based applications.

Keywords: web; JavaScript; web assembly; crypto library; side channel analysis; fast implementation

1. Introduction

Recently, various types of Internet technology services, e.g., personal and business services, are provided to users via web-based applications due to the accessibility of the web. Typically, web-based applications comprised of servers and clients, and private information, e.g., private user data and passwords, are exchanged between clients and servers. Data transmitted in plaintext form are vulnerable to attackers thus, it is necessary to provide cryptographic operations to protect private data and build secure web-based services. In other words, data confidentiality, data authentication, and key establishment functions must be provided to develop secure web-based services [1].

JavaScript is a cross-platform script programming language that is used in various fields, e.g., server-side network programming, databases, and the Internet of Things (IoT) [2]. JavaScript is used in web browsers to display web sites and can be accessed from another application's built-in objects. However, JavaScript is an interpreted language and is relatively slower than native languages such as, e.g., C. In addition, it does not support the mathematical operations required for cryptographic

operations, which incurs heavy overhead when executing such cryptographic operations. Therefore, various web browser development companies have developed Web Assembly, which is a low-level language for web environments that provides performance that is similar to native languages (Web Assembly is being continuously extended) [3,4]. To date, several studies have investigated implementing cryptographic algorithms using JavaScript. Since they are based on low-performance JavaScript language, they do not provide sufficient performance. Furthermore, previous methods implemented a limited number of algorithms rather than forming a complete crypto library. To build secure communication between servers and clients in web applications, a crypto library that provides confidentiality, data authentication, and key establishment functions is required.

Thus, we propose an efficient Web Assembly-based crypto library for secure communication in various web applications. The proposed crypto library comprises of a block cipher, a message authentication code, and a key exchange algorithm. We selected the revised CHAM [5], Hash Message Authentication (HMAC) [6], and Elliptic-curve Diffie-Hellman (ECDH) using the National Institute of Standards and Technology (NIST) [7] recommended P-256 curve [8], as a block cipher, message authentication code, and key agreement method. The proposed Web Assembly-based crypto library provides much improved performance compared to JavaScript-based implementations. We apply several optimization techniques to further improve the performance of cryptographic operations in the proposed library. We apply various methods to implement a safe and fast CHAM algorithm. The original CHAM family algorithm is vulnerable to differential attacks, so the revised CHAM algorithm is used. The revised CHAM algorithm [5] is an algorithm configured to be safe from differential attacks by increasing the number of rounds from 80 to 88, 80 to 112, and 96 to 120 for CHAM-64/128, CHAM-128/128, and CHAM-128/256 respectively. In the revised CHAM algorithm, there is a process of changing the place of the word constituting the input value every round. We apply existing 4-round combining method [9] to improve the performance of revised CHAM. The 4-round combining method works faster by eliminating the unnecessary process of changing places by using the word values used in each round flexibly. We propose an additional pre-computation method for a faster operation in CHAM-64/128. The pre-calculation method is applied to the internal functions ROL1, ROL8, and Keyschedule functions of CHAM-64/128 [10]. ROL1 and ROL8 are functions that rotate one word used as an input value by 1-bit and 8-bit, respectively, and Keyschedule is a function that creates a round key. The three functions use 16-bit input values, and we apply the method of pre-computation ROL1, ROL8, and Keyschedule from $0 \times 0 \sim 0 \times ffff$. To a secure and efficient implementation of the ECDH key agreement method, we implement scalar multiplication, which is a core operation in the ECDH with the simple power analysis (SPA)-resistant and *wNAF* [11] method. The naive implementation of scalar multiplication is vulnerable to side-channel attacks (SCA) (e.g., simple power analysis (SPA) and timing analysis (TA)) [12,13]. In the case of scalar multiplication, if 1-bit of a scalar integer is 1, *ECADD* and *ECDBL* are performed, and when 0, *ECDBL* is performed, so the process is different. Therefore, it is divided into 1-bit units during analysis and eventually scalar integer values, which are important information, can be attacked. Since scalar multiplication is computationally intensive, a windowing method is used to compute it. Even though the *wNAF* method is a representative windowing method for computing scalar multiplication efficiently, it is vulnerable to SPA and TA. As an efficient countermeasure against SPA and TA, the concept of the atomic block was presented previously [14]. An atomic block consists of $*, +, -, +$ processes as one block. A fake operation, which is an unnecessary operation, is added to the *ECDBL* and *ECADD* operation process of scalar multiplication, and the structure is made so that it is calculated in the order of $*, +, -, +$. Thus, it is safe for SPA and TA because 1-bit of a scalar integer value is calculated in the order of $*, +, -, +$ regardless of 1 or 0. We improved the atomic block assuming that scalar multiplication is performed using only the basis point $P = (X, Y, Z)$. Change is completed from the existing atomic blocks $*, +, -, +$ to $*, +, -, +$. Thus, 10 and 16 addition operations were reduced in *ECDBL* and *ECADD*, respectively, compared to the existing atomic block. We apply *wNAF* and improved atomic block to the ECDH algorithm of the proposed crypto library.

Web Assembly and JavaScript are implemented algorithms, executed in Web browsers such as Chrome, Firefox, and Microsoft Edge respectively to measure performance. Following performance improvements that have been achieved in the order of Chrome, Firefox, and Microsoft Edge. In case of block cipher, 2.1, 2.1, and 2 times for CHAM-64/128, 3, 1.6, and 1.8 times for CHAM-128/128, and 3, 2.1, and 2.8 times for CHAM-128/256 shows performance improvement. CHAM-64/128 with applied pre-computation method shows a performance improvement of 1.2 times than not applied to the algorithm in three web browsers. For the key exchange algorithm, *wNAF* was applied to P-256. The atomic block method, which is an algorithm corresponding against SPA and TA, was also applied. When applying the existing atomic block and the proposed atomic block to *wNAF*, we check how much performance overhead appears than the original *wNAF* due to the increased number of operations, and how much the proposed atomic block is improved over the existing atomic block. For this purpose, each algorithm implemented in Web Assembly and JavaScript was measured in Chrome, Firefox, and Microsoft Edge. As a result, Web Assembly improved more than JavaScript, for the original *wNAF* by respectively 11, 12, and 11 times, the existing atomic block *wNAF* by respectively 10, 10, and 14 times, and the proposed *wNAF* by respectively 11, 12, and 14 times. Existing atomic block *wNAF* shows a performance overhead of 55%, 23%, and 37% compared to the original *wNAF*. However, the atomic block *wNAF* proposed to use only $P = (X, Y, Z)$, showing performance overheads of 18%, 6%, and 11%. The message authentication code is HMAC that uses SHA-256 to create a MAC. As a result of measurement, Web Assembly showed a higher performance over JavaScript by 7.5, 10.8, and 11 times for SHA-256, and 7.5, 24.8, and 7.1 times for HMAC.

Contribution

In this section, we propose the contributions of this paper.

1. First implementation of a crypto library using Web Assembly

Recently, web-based applications with various functions are being made in the cross-platform language JavaScript. Web-based applications require confidentiality, integrity, and key exchange algorithms to send and receive data. Cryptographic algorithms are made in JavaScript for use in web-based applications. However, JavaScript is a heavy language and the nature of JavaScript operations has disadvantages in implementing cryptographic algorithms that require many mathematical operations. Therefore, Web Assembly was created due to the need for a performance similar to that of low-level languages in the web environment. In this paper, we propose to build a crypto library with cryptographic algorithms implemented using Web Assembly to implement data security and faster cryptographic algorithms in web-based applications. The proposed crypto library includes the block cipher CHAM family, the message authentication code HMAC, and the key exchange algorithm ECDH. For each cryptographic algorithm, the code implemented by Web Assembly shows a better performance than JavaScript. Our implementations are measured in currently popular Web browsers such as Chrome, Firefox, and Microsoft Edge. As a result of the measurement, on average, the CHAM family improved in speed by about 2.2 times, HMAC by about 7.1 times, and ECDH scalar multiplication improved by 12.3 times.

2. Optimized implementations of a crypto library on Web Assembly

Since web-based applications exchange data with various environments, encryption is an essential function to send data confidentially. However, due to the advancement of technology and various environments and communication, the amount of data exchanged has also increased. Since the data to be communicated is encrypted in order, it is necessary to optimize for the environment in which the algorithm is used in order to encrypt quickly. The block cipher, a component of our proposed cipher library, is chosen as belonging to the CHAM family. However, the original CHAM algorithm is vulnerable to differential attacks. Therefore, CHAM-64/128, CHAM-128/128, and CHAM-128/256 use the revised CHAM algorithm which increases the number of rounds from 80 to 88, 80 to 112, and 96 to 120, respectively. In the revised CHAM algorithm, there is

a process of changing the place of the word constituting the input value for each round. For a faster encryption operation, we apply a 4-round combining method, which is an existing method, to eliminate the process of changing the word position to perform a flexible operation. Additionally, we propose a pre-computation method for faster operation in CHAM-64/128. The method we propose applies to the internal functions ROL1, ROL8, and Keyschedule functions of CHAM-64/128. ROL1 and ROL8 are operations that shift the input value by 1, 8-bit Rotation Left Shift, and KeySchedule is a round key generation function. The input values of the three functions are 16-bit, which is a method of storing and using the result values from 0×0 to $0 \times ffff$ after pre-calculation. Thus, in the encryption process, the previously calculated values are simply taken and used. As a result, the performance was improved about 1.2 times compared to when the pre-computation method was not applied in Chrome, Firefox, and Microsoft Edge.

3. Providing improved method that resists side channel attacks

Until now, there have not been many studies of side-channel analysis on the web environment. In particular, a secure key exchange protocol should be applied to provide a secure communication protocol in a web environment. ECDH is used as the key exchange algorithm. There is scalar multiplication, which is the main operation of ECDH. However, since the scalar multiplication process performs the *ECDBL*, *ECADD* operation when the value of 1-bit of the scalar integer is 1, and the *ECDBL* operation when it is 0, it is possible to attack the scalar value because each bit is classified during an attack. We propose a secure key exchange protocol that is applied by improving the previously studied atomic block to cope with TA and SPA, which are vulnerable to side channel analysis attacks in the web environment. Existing atomic blocks consist of *, +, −, and + in one block. Fake operations are added to the main operations of scalar multiplication, *ECDBL* and *ECADD*, and are configured to operate in the order of *, +, −, +. Therefore, it becomes difficult to distinguish because 1-bit values are calculated in the order of *, +, −, + regardless of 1 or 0. We change the existing atomic block to *, +, − and make it into one block. Thus, we reduced 10 and 16 addition operations in *ECDBL* and *ECADD*, respectively. The method we are suggesting is a method used only with $P = (X, Y, Z)$. In addition, we calculate by applying *wNAF* and a proposed atomic block to P-256 for efficient scalar multiplication. The implemented algorithms measured the results in web browsers Chrome, Firefox, and Microsoft Edge. As a result of the measurement, compared to the original *wNAF*, *wNAF* applied with an existing atomic block shows a performance overhead of about 33%, and *wNAF* with the proposed atomic block shows a performance overhead of about 11%. As a result, the proposed atomic block, compared to the existing method, reduced the performance overhead by $\frac{1}{3}$.

The remainder of this paper is organized as follows. Section 2 provides a basic overview of the web environment, Web Assembly's description and conversion process, and the need for a crypto library. Section 3 describes the architecture of the proposed crypto library and target cryptographic algorithm. Section 4 describes related work. Section 5 describes the construction of a crypto library using the proposed cryptographic algorithm. Section 6 describes the performance measurement results. Finally, Section 7 concludes the paper.

2. Background

2.1. Overview of Web Environment

Users frequently make use of web applications and access web services for a long time. There is a variety of web browsers, e.g., Chrome, Firefox and Microsoft Edge, to access the web. Web browsers are created using HTML, CSS, and JavaScript. A web browser uses a rendering engine that works on the content and data of a web page and a JavaScript engine to execute JavaScript code to drive the web browser. Each web browser uses a different rendering engine and a JavaScript engine. For example, Chrome uses Blink as the rendering engine and V8 as the JavaScript engine. Microsoft Edge uses

EdgeHTML and Chakra, Firefox uses Gecko and Rhino. Web-based applications view the same content on all devices, e.g., PCs and smartphones. Unlike native applications, web-based applications do not communicate directly with the operating system but run within the browser. Web-based applications can always keep up to date without downloading or upgrading, and operating systems do not require a separate platform, so a standard web language is made. Thus, users can easily access their choice of web using mobile devices, e.g., smartphones. The code in one web page does not affect the code in other pages. No matter which function is executed by the JavaScript code on a web page, other web pages are irrelevant to the result obtained from the previous web page. Due to the development of the web environment and the need for various functions, various libraries are created continuously to enable various functions in the web environment using JavaScript. In addition, web developers can use these JavaScript libraries easily and such libraries can be further modified. This is why JavaScript libraries and the web are constantly evolving.

The web page executes the HTML, CSS, and JavaScript code that makes up the web page, as shown in Figure 1, the rendering engine reads the code, parses the code, and then creates a Document Object Model (DOM) and CSS Object Model (CSSOM) tree. These trees create a render tree, which renders the web page to a web browser. The JavaScript engine handles the operation and program codes. The rendering engine stops working when it encounters JavaScript code. The JavaScript engine reads JavaScript code and creates a tree by parsing. After processing all of the JavaScript code, the rendering engine performs its own tasks again from the process where it stopped and processes the process.

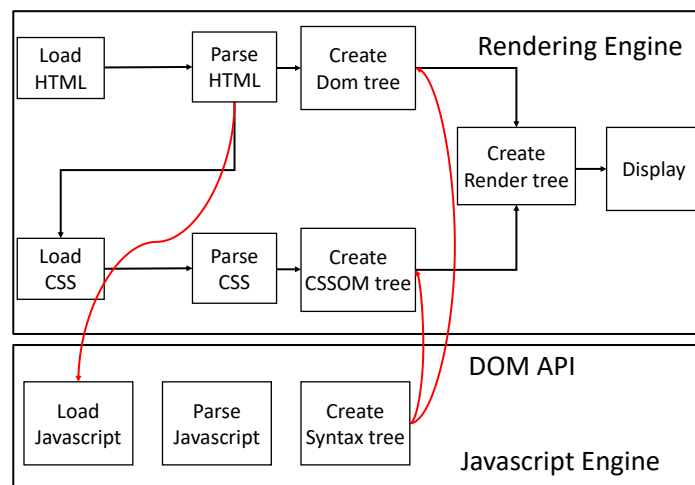


Figure 1. JavaScript working process.

2.1.1. Overview of Web Assembly

JavaScript is primarily used in web-based applications however, the operation speed of JavaScript is significantly slower than that of other native languages. Web-based applications cannot use native languages, e.g., C/C++. With the various content available on the web, the computation of content has become complicated or heavy, and implementing such operations in JavaScript is a disadvantage from a performance perspective. A language is required for the web that can be implemented and operate at a similar level of performance as a native language. Initially, Mozilla announced asm.js however, it has not received much attention due to its performance inefficiency. In addition, asm.js is difficult to use. The need for native language-level performance in web environments continued, and Web Assembly was created based on asm.js. Web Assembly is in constant development and web browser companies, e.g., Google, Microsoft, and Mozilla, are involved in its development. Web Assembly is not intended to replace JavaScript, but is designed to operate web-based applications efficiently with JavaScript. Web Assembly implements code using languages that can identify existing variable types, e.g., C/C++, Rust, Typescript, Assemblyscript, and Go, and then converts them to Web Assembly using Emscripten.

Figure 2 shows the process of converting C/C++ code to Web Assembly code. After writing an algorithm in C/C++, Emscripten enters the C/C++ code into the Clang + LLVM and receives the compilation results to generate the Web Assembly extension (i.e., a WASM file). The WASM file is not immediately accessible to the DOM thus, Emscripten can help print the results of the wasm execution in HTML documents through JavaScript glue code to access the DOM.

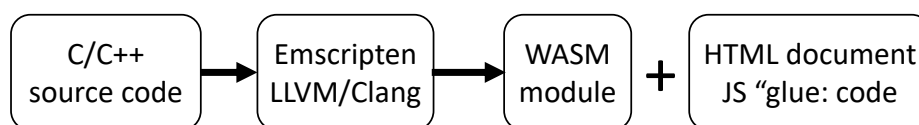


Figure 2. WebAssembly conversion process.

2.1.2. Necessity of Crypto Library for Secure Web Application

Web-based applications can easily be accessed by users through various devices, e.g., PCs and smartphones therefore, various users, e.g., companies, institutions, and individuals, are using web-based applications. Many users use web-based applications for information provision, collection, search, or personal work. Web-based applications must show the same data on different platforms thus, web-based applications are created using JavaScript (a cross-platform language). Therefore, users obtain the same information on different platforms. JavaScript is also used in server-side network programming, databases, and the IoT. Due to convenience and various features, web-based applications communicate with various other environments and platforms. This is why web-based application send and receive various data and store them on a server. To ensure the continuous development of web-based applications and data security, a crypto library comprising of cryptographic and authentication algorithms is required. For security, encryption is performed when data are stored on a server and decryption is needed when data are used. In addition, authentication is required to determine whether data transmitted and received during communication are intact. Therefore, to securely communicate with other environments in web-based applications, ensuring confidentiality and integrity is essential.

3. Secure Crypto Library Design

3.1. Design Motivation and Library Architecture

Crypto libraries created using JavaScript make it easy for users of other web environments to obtain and use cipher algorithms, e.g., block ciphers, key agreement, key exchange algorithms, and message authentication. Web-based application developers that use JavaScript enable users to safely use applications by using a crypto library to protect user information, encrypt, and safely store data created by the web-based application, and verify data integrity through message authentication. Even if a 1-bit error occurs, users cannot obtain the correct data thus, when implementing an encryption algorithm, it must be implemented carefully in the operation process.

In the case of JavaScript, data types are not divided into char, short, and int according to bit size like C/C++, and there are no dividing negative and positive numbers, e.g., unsigned and signed. With C/C++, the bit size of the value that can be stored for each data type is determined thus, parts that exceed the bit size are cut automatically when calculating integers, which is useful for parts that require subtraction after computation, e.g., modular addition, in the computation of cryptographic algorithms. It can express negative and positive numbers as unsigned and signed and there are many useful parts in the finite field operation of cryptographic algorithms. However, JavaScript is not divided into data type, unsigned, and signed, so each cryptographic algorithm has different word sizes, and additional operations must be used to obtain the desired result. JavaScript is a heavy language, and it is slower because it requires additional operations when performing the same operations as C/C++. Thus, JavaScript is less efficient when implementing cryptographic algorithms.

Converting existing programming languages, such as C/C++, Rust, etc., to Web Assembly is used via Emscripten to allow them to operate in a web environment. Data types can be divided and operated for each size, and positive and negative numbers can be distinguished, such as unsigned and signed, so that a user can get the desired value without additional operations, unlike JavaScript. For cryptographic algorithms with many mathematical operations, Web Assembly can be implemented and operated faster and more efficiently in a web environment. If users use a Web Assembly-based crypto library when communicating with the web environment and other environments, the web-based application can perform faster computations and encryptions than when using a JavaScript-based crypto library.

3.2. Target Block Ciphers

Revised CHAM

In ICISC 2017, National Security Research Institute Koo et al. proposed the lightweight CHAM crypto family [10], which is divided into CHAM-64/128, CHAM-128/128, and CHAM-128/256 depending on the parameters. Table 1 shows the CHAM parameters. It also features a stateless on-the-fly key schedule, which reduces key storage space and provides lightweight cryptography with the ARX structure, which is suitable for limited environments. The key scheduling process in CHAM is shown in Figure 3. The *ROL1*, *ROL11*, *ROL8*, and *XOR* operations generate $n/k \times 2$ round keys. Then, it encrypts all rounds with an $n/k \times 2$ round key. The encryption process comprises of odd and even rounds, and each round function comprises of *ROL1*, *ROL8*, and *XOR* operations, as well as modular addition. After each round, a cyclic left shift is performed in the word unit. The odd and even round encryption process of CHAM is shown in Figure 4.

In ICISC 2019 [5], it was suggested that the original CHAM was vulnerable to differential attacks by discovering the differential characteristics in the reduced round. CHAM-64/128, CHAM-128/128, and CHAM-128/256 found some differential characteristics in rounds 56, 72, and 78, respectively. Thus, for the revised CHAM the numbers of rounds are increased to defend against differential attacks. The revised CHAM-64/128 increases the number of rounds from 80 to 88, the revised CHAM-128/128 increases the number of rounds from 80 to 112, and the revised CHAM-128/256 increases the number of rounds from 96 to 120 rounds. Despite increasing the number of rounds, the revised CHAM showed efficient performance in both software and hardware, and was faster and safer against differential attacks than the lightweight SIMON and SPECK.

Table 1. Parameters of CHAM family (n : Block size, k : Key size, r : Round number, and w : Word size).

Cipher	n	k	r	w	k/w
CHAM-64/128	64	128	80	16	8
CHAM-128/128	128	128	80	32	4
CHAM-128/256	128	256	96	32	8

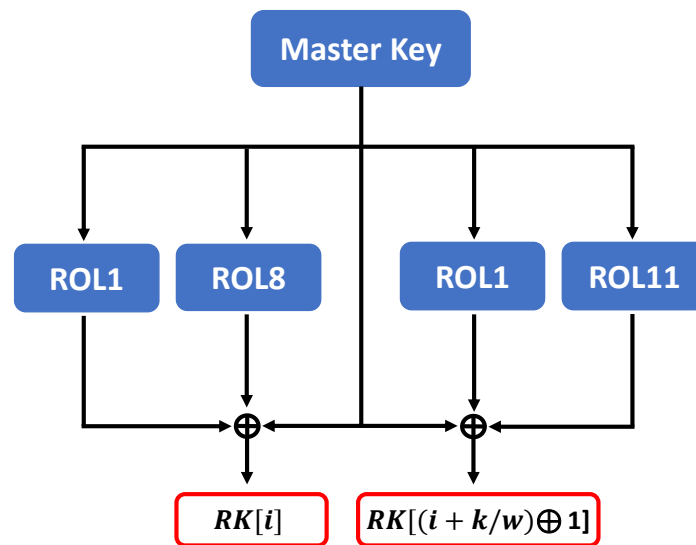


Figure 3. CHAM keyschedule.

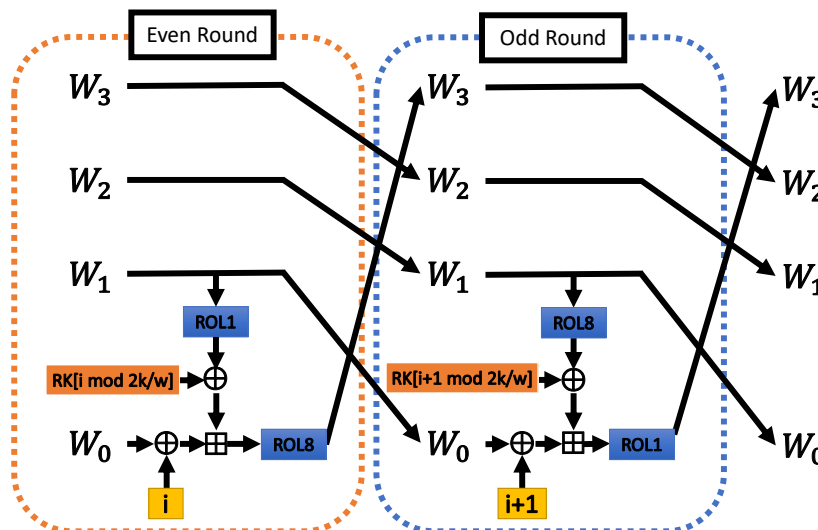


Figure 4. CHAM round function.

3.3. Target Message Authentication Code (MAC) Algorithm

3.3.1. Overview of HMAC

Web-based applications send and receive a lot of data in real time. To establish a secure communication environment, it is necessary to authenticate whether a message has been tampered with due to an intermediate attack, or whether the data have been transmitted from the correct user. MAC is used to confirm this and provides message integrity and authentication by generating a MAC by inputting a key shared with each other between the message sender and receiver. Various MAC, e.g., GCM, CCM, and HMAC have been proposed to provide message integrity and authentication. HMAC is classified into HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 according to the SHA-2 family used in the message compression process [6]. We use HMAC-SHA-256 as the target message authentication code by using SHA-256, which is the most frequently used in the message compression process. The overall process of HMAC-SHA-256 is shown in Figure 5. The MAC value is generated through two SHA-256 processes. IPAD and OPAD repeat 0x36 for IPAD and 0x5c for OPAD as much as the block length of the hash function. First, if the key length is greater than 512-bit, the key value is hashed. The remaining space is padded with zeros to adjust the length of the

key to 512-bit. If the length of the key is less than 512 bits, the remaining space is padded with zeros to adjust the length of the key to 512-bit. Then, the input of the hash function is set by applying the XOR operation to each 512-bit IPAD and OPAD and then, the message value for authentication is added after the IPAD and padded XOR result value to form a single message, and a 256-bit hash value is generated through the SHA-256 process. Finally, the generated hash value is pasted after the OPAD and padded K-value XOR result to form a single message, and then set as the input data for SHA-256. Finally, the generated hash value becomes the MAC value for message authentication.

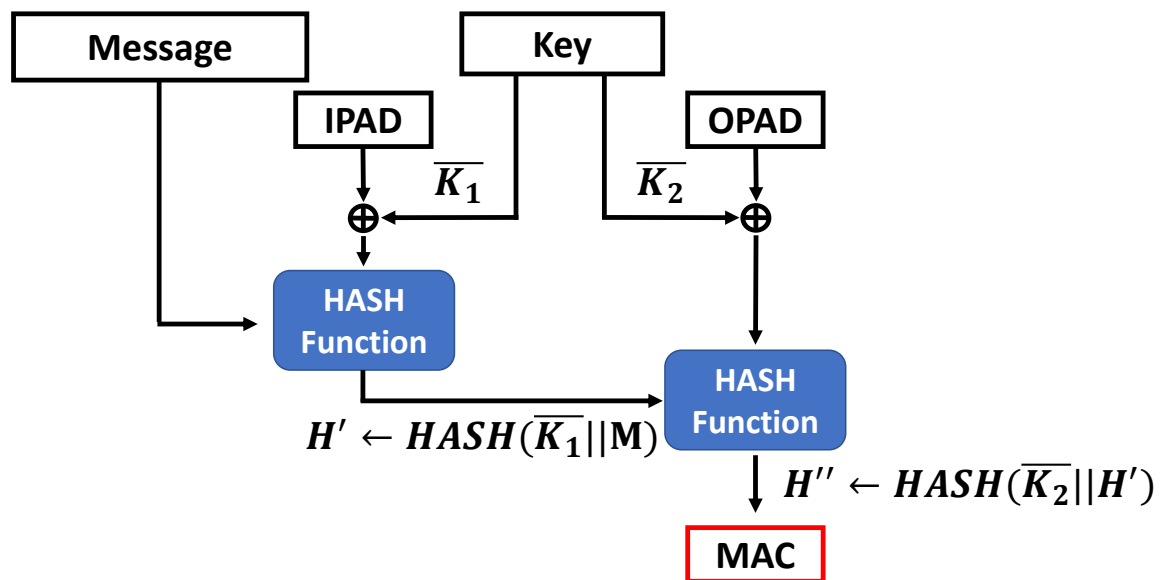


Figure 5. Hash Message Authentication Code (HMAC) process.

3.3.2. Overview of SHA-256

SHA-256 Internal functions: SHA-256 use six logical functions, where each function operates on 32-bit words, which are represented as x , y , and z . The result of each function is a new 32-bit word. The six logical functions are expressed as follows [15].

Definition 1. SHA-256 logical function:

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{1}$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{2}$$

$$\sum_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \tag{3}$$

$$\sum_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \tag{4}$$

$$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus (x \gg 3) \tag{5}$$

$$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus (x \gg 10) \tag{6}$$

SHA-256 Padding the Message: The SHA-256 block has a 512-bit size, and the block operation is performed in 32-bit units. The SHA-256 function stores the length of the input data in the last block 64-bit. Therefore, the padding process must be included in the SHA-2 family for storing the message length, the padding process is summarized as follows.

- Padding process

Step 0 Let l is the length of the message;

- Step 1** Append the bit “1” to the end of the message;
- Step 2** Followed by k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k = 448 \bmod 512$;
- Step 3** Then append the 64-bit that is equal to the message length l expressed using a binary representation.

Padding can be inserted before hash computation begins on a message or any other time during the hash computation prior to processing the block(s) that will contain the padding [15].

SHA-256 Message Compression: The block operation in SHA-256 repeats the same process for 64 rounds. In the block operation, each round uses padded message data. Thus, SHA-256 must expand the data using message padding, this process is the message expansion process. Algorithm 1 shows the pseudocode of the SHA-256 message expansion process.

Algorithm 1 SHA-256 Message expansion process

Require: 32-bit word Message $M = (M[0], M[1], \dots, M[15])$
Ensure: Expansion Data $W = (W[0], W[1], \dots, W[63])$

- 1: **for** $i = 0$ to 15 **do**
- 2: $W[i] = M[i]$;
- 3: **end for**
- 4: **for** $i = 16$ to 63 **do**
- 5: $W[i] = \sigma_1^{256}(W[i - 2]) \boxplus W[i - 7] \boxplus \sigma_0^{256}(W[i - 15]) \boxplus W[i - 16]$
- 6: **end for**
- 7: **return** W

In Algorithm 1, blocks the size of 512 bits are labeled M . An M block is divided into 16 32-bit words, each divided data are labeled $M[i]$, and the output of the message expansion process is labeled $W(0 < i < 16)$. Message compression updates the digest value through the extended W and eight initialized 32-bit working variables. The eight working values are a, b, c, d, e, f, g , and h , respectively. Algorithm 2 shows the pseudocode for the SHA-256 message compression process. In Algorithm 2, K_t^{256} is the round constant defined in the literature [15]. Then, Algorithm 2 is executed, the digest is updated using the eight working values. In SHA-256, the digest comprises eight 32-bit words. When the SHA-256 algorithm is called, the digest is initialized to a defined value [15]. After the message compression process, the digest is updated with the eight working values. The digest updates the 32-bit word and working value with 2^{32} modular addition (\boxplus). When message compression uses the last padding block, the SHA-256 digest is updated through a working value. Finally, SHA-256 returns a 256-bit digest.

Algorithm 2 SHA-256 Message Compression

Require: Expansion Data $W = (W[0], \dots, W[63])$
Require: Working variables (a, b, c, d, e, f, g, h) in hash state
Ensure: Updated working variables (a, b, c, d, e, f, g, h) in hash state

- 1: **for** $t = 0$ to 63 **do**
- 2: $T_1 = h + \sigma_1^{256}(e) \boxplus Ch(e, f, g) \boxplus K_t^{256} \boxplus W[t]$
- 3: $T_2 = \sigma_0^{256}(a) \boxplus Maj(a, b, c)$
- 4: $h = g, g = f, f = e, e = d \boxplus T_1, d = c, c = b, b = a, a = T_1 \boxplus T_2$
- 5: **end for**
- 6: **return Hash value** (a, b, c, d, e, f, g, h)

3.4. Target Key Agreement Algorithm

ECDH with P-256 Curve

P-256 is a NIST curve amongst the 15 elliptic curves recommended by NIST [8]. It is an elliptic curve defined over a 256-bit prime field that offers approximately 128-bit security. This elliptic curve is defined by the following equation:

$$y^2 = x^3 - 3x + b \tag{7}$$

where b is a constant in a finite field F_p . The prime p is a 256-bit prime selected for easy modular reduction. This elliptic curve has an Abelian group structure with identity element O called the point of infinity. Scalar multiplication calculates kP using the 256-bit scalar value integer k and base point $P = (X_1, Y_1)$ to obtain $Q = (X_3, Y_3)$ values. Here, the algorithms used for scalar multiplication are *ECADD* and *ECDBL*. The input value points used for *ECDBL* and *ECADD* are affine coordinate systems $P = (X_1, Y_1)$, $Q = (X_2, Y_2)$. *ECDBL* calculates $P + Q = 2P$ when $P = Q$ and *ECADD* performs $P + Q$ when $P \neq Q$. The security of ECC is based on the difficulty of computing the elliptic curve discrete logarithm problem (ECDLP), i.e., it is very difficult to find scalar value k when Q and k are given by $Q = kP$.

The prime curve's equation is $y^2 = x^3 + ax + b$. The prime curve is divided into P-256, P-384, and P-521 for each parameter. Here, scalar multiplication is performed using the affine coordinate system. *ECADD* is performed whenever the 1-bit value of scalar k , i.e., the input value of scalar multiplication, is 1. *ECADD* includes inverse circle arithmetic. Among the finite field operations (addition, subtraction, multiplication, and inverse), inverse operations are the heaviest. Therefore, rather than performing inverse calculation through *ECADD* whenever the 1-bit value is 1 by extending to the projective coordinate system, the load on the inverse calculation is reduced by performing the inverse calculation once after the scalar multiplication operation. This method calculates scalar multiplication quickly using a more optimized method than projective coordinate by implementing scalar multiplication with a Jacobian coordinate system fixed at $a = -3$. After converting the affine coordinate system to the Jacobian coordinate system, the *ECDBL* and *ECADD* operations are performed as shown in Table 2. After the scalar multiplication operation is completed, the value of kP can be obtained by converting the Jacobian coordinate system to the affine coordinate system [8].

Table 2. Jacobian *ECDBL*, *ECADD*.

<i>ECDBL</i>	<i>ECADD</i>
$P = (X_1, Y_1, Z_1)$	$P = (X_1, Y_1, Z_1)$
$P + Q = 2P = (X_3, Y_3, Z_3)$	$Q = (X_2, Y_2, Z_2)$
	$P + Q = (X_3, Y_3, Z_3)$
$M = 3X_1^2 + aZ_1^4$	$U_1 = X_1Z_2^2$
$S = 4X_1Y_1^2$	$U_2 = X_2Z_1^2$
$T = 8Y_1^4$	$S_1 = Y_1Z_2^3$
	$S_2 = Y_2Z_1^3$
	$H = U_2 - U_1$
	$R = S_2 - S_1$
$X_3 = M^2 - 2S$	$X_3 = R^2 - H^3 - 2U_1H^2$
$Y_3 = M(S - X_3) - T$	$Y_3 = R(U_1H^2 - X_3) - S_1H^3$
$Z_3 = 2Y_1Z_1$	$Z_3 = HZ_1Z_2$

ECDH is a Diffie–Hellman key exchange protocol that uses elliptic curve-based operations [7]. Elliptic curve cryptography is a public key method based on an elliptic curve and security in the discrete logarithm problem. In addition, as an alternative to RSA, it provides security with a much shorter key length than RSA. The elliptic curve-based operation comprises *ECADD* and *ECDBL*. *ECADD* is an operation that adds two points, and *ECDBL* is an operation that doubles a point.

The dP for scalar d , i.e., a point on the elliptic curve at the point at base point P , is calculated as scalar multiplication using two elliptic curve operations.

The Diffie–Hellman key exchange is security with the difficulty of the discrete logarithm problem. Here, Alice and Bob calculate $g^a \bmod p$ and $g^b \bmod p$ with the private keys a and b , respectively, in the cyclic group $\langle g \rangle$ with order p . Then, after sending $g^a \bmod p$ and $g^b \bmod p$ to Bob and Alice respectively, by exponentially multiplying each private key to the transmitted value, private keys as $g^{ab} \bmod p$ can be exchanged safely without revealing key information to an attacker. In DH, the key lengths of a and b are long, which is a disadvantage however, ECDH, which combines elliptic curve cryptography and DH, provides efficient security with a short key length using elliptic curve cryptography. The entire process of ECDH is shown in Figure 6. Here, Alice and Bob generate private keys a and b , respectively, and, after generating private keys, Alice and Bob set the base point G on the elliptic curve to calculate public keys aG and bG , respectively, and send the public keys aG and bG to each other. Finally, Alice and Bob calculate point abG on the elliptic curve through scalar multiplication of their private key values on the transmitted public key.

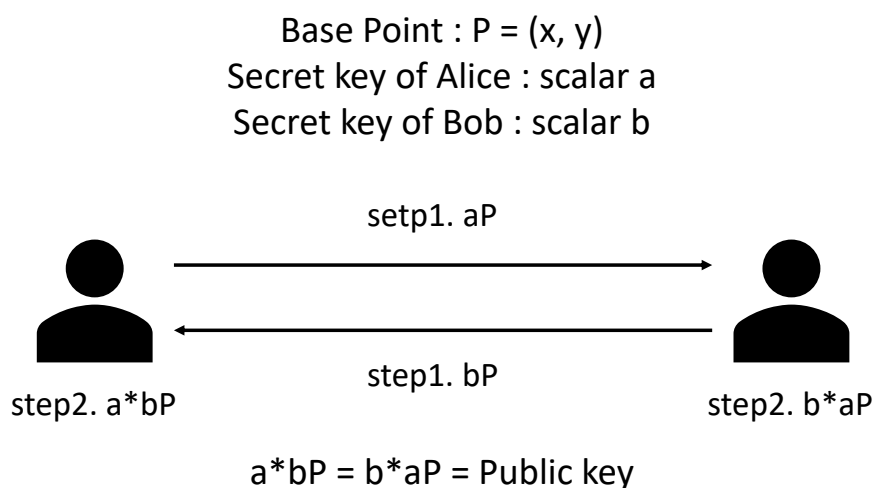


Figure 6. ECDH Process.

3.5. Providing Side Channel Resistance

Atomic Block-Based ECDH Implementation

The scalar multiplication operation of elliptic curve cryptography is vulnerable to simple power analysis (SPA). This is because the scalar multiplication operation operates $ECDBL$ and $ECADD$ when the 1-bit of the scalar integer is 1, and calculates only $ECDBL$ when the 1-bit of the scalar integer is 0, resulting in different power consumption. In addition, $ECADD$ is only performed when the scalar multiplication operation is 1 thus, the use of branch statements is vulnerable to timing attacks. Countermeasures for side-channel analysis against scalar multiplication of elliptic curve cryptography have been proposed [12–14].

In [14], an atomic block, an algorithm for countering SPA, which is a side-channel attack method of RSA and elliptic curve cryptography, was proposed. In the Scalar multiplication operation, an atomic block is applied to $ECDBL$ and $ECADD$ to be safe against SPA, which is a side-channel attack, and the existing atomic block operation repeats in the order of multiplication, addition, subtraction, and addition operations to perform a Scalar multiplication operation. $ECADD$ and $ECDBL$ to which the atomic block is applied are shown in Table 3. In order to safely perform $ECDBL$ and $ECADD$ through an atomic block, a fake operation must be added. For the existing atomic block, 17 fake operations were added for $ECDBL$ and 32 for $ECADD$. If $ECDBL$ and $ECADD$ are configured through the calculation process shown in Table 3, the same power waveform is repeated when

an attacker measures the power consumption for scalar multiplication, so it is safe for SPA. In addition, it is safe for TA because branch statements are not required when implementing atomic blocks. When exchanging keys between web environment and another environment, using a branch statement in scalar multiplication inside ECDH is vulnerable to TA. Therefore, we present a secure key exchange protocol to users when using crypto libraries by applying an atomic block which is a security method for TA and SPA to scalar multiplication.

Table 3. Existing atomic block method.

<i>ECDBL</i>		<i>ECADD</i>	
$T_0 \leftarrow a,$		$T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$	
$T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$		$T_7 \leftarrow X_2, T_8 \leftarrow Y_2, T_9 \leftarrow Z_2$	
1.	6.	1.	9.
$T_4 \leftarrow T_1 \cdot T_1$	$T_2 \leftarrow T_2 \cdot T_2$	$T_4 \leftarrow T_9 \cdot T_9$	$T_3 \leftarrow T_3 \cdot T_9$
$T_5 \leftarrow T_4 + T_4$	$T_2 \leftarrow T_2 + T_2$	*	*
*	*	*	*
$T_5 \leftarrow T_4 + T_4$	*	*	*
2.	7.	2.	10.
$T_5 \leftarrow T_3 \cdot T_3$	$T_5 \leftarrow T_3 \cdot T_3$	$T_1 \leftarrow T_1 \cdot T_4$	$T_3 \leftarrow T_3 \cdot T_5$
$T_1 \leftarrow T_1 + T_1$	*	*	*
*	$T_5 \leftarrow -T_5$	*	*
*	*	*	*
3.	8.	3.	11.
$T_5 \leftarrow T_5 \cdot T_5$	$T_5 \leftarrow T_5 \cdot T_5$	$T_4 \leftarrow T_4 \cdot T_9$	$T_6 \leftarrow T_5 \cdot T_5$
*	$T_1 \leftarrow T_1 + T_5$	*	*
*	*	*	*
*	$T_1 \leftarrow T_1 + T_5$	*	*
4.	9.	4.	12.
$T_5 \leftarrow T_0 \cdot T_5$	$T_2 \leftarrow T_2 \cdot T_2$	$T_2 \leftarrow T_2 \cdot T_4$	$T_1 \leftarrow T_1 \cdot T_6$
$T_4 \leftarrow T_4 + T_5$	$T_2 \leftarrow T_2 + T_2$	*	*
*	*	*	$T_4 \leftarrow -T_4$
$T_5 \leftarrow T_2 + T_2$	$T_5 \leftarrow T_1 + T_5$	*	*
5.	10.	5.	13.
$T_3 \leftarrow T_3 \cdot T_5$	$T_4 \leftarrow T_4 \cdot T_5$	$T_4 \leftarrow T_3 \cdot T_3$	$T_5 \leftarrow T_5 \cdot T_6$
*	$T_2 \leftarrow T_2 + T_4$	*	$T_6 \leftarrow T_1 + T_2$
*	$T_2 \leftarrow -T_2$	*	$T_2 \leftarrow -T_2$
*	*	*	$T_6 \leftarrow T_2 + T_6$
		6.	14.
		$T_5 \leftarrow T_4 \cdot T_7$	$T_1 \leftarrow T_4 \cdot T_4$
		*	$T_1 \leftarrow T_1 + T_5$
		$T_5 \leftarrow -T_5$	$T_6 \leftarrow -T_6$
		$T_5 \leftarrow T_1 + T_5$	$T_1 \leftarrow T_1 + T_6$
		7.	15.
		$T_4 \leftarrow T_4 \cdot T_8$	$T_2 \leftarrow T_2 \cdot T_5$
		*	$T_1 \leftarrow T_1 + T_6$
		$T_4 \leftarrow -T_4$	*
		$T_4 \leftarrow T_2 + T_4$	$T_6 \leftarrow T_1 + T_6$
		8.	16.
		$T_4 \leftarrow T_4 \cdot T_8$	$T_4 \leftarrow T_4 \cdot T_6$
		*	$T_2 \leftarrow T_2 + T_4$
		$T_4 \leftarrow -T_4$	*
		$T_4 \leftarrow T_2 + T_4$	*

4. Previous Crypto Implementations in Web Environment

4.1. CHAM Algorithm in JavaScript and Web Assembly

The original CHAM algorithm (Figure 4) is divided into odd and even rounds, and swaps the position of the word at the end of each round. Use $2 \times k/w$ round keys repeatedly. The words of

the original CHAM algorithm return to their original positions every four rounds. Thus, as shown in Figure 7, it is possible to maintain the position of each word by calculating the necessary values for each round without performing a swap. This method is faster because the swap process in the original CHAM algorithm is not used [9]. The CHAM and AES algorithms are implemented with Web Assembly and demonstrate faster performance than JavaScript implementation [16].

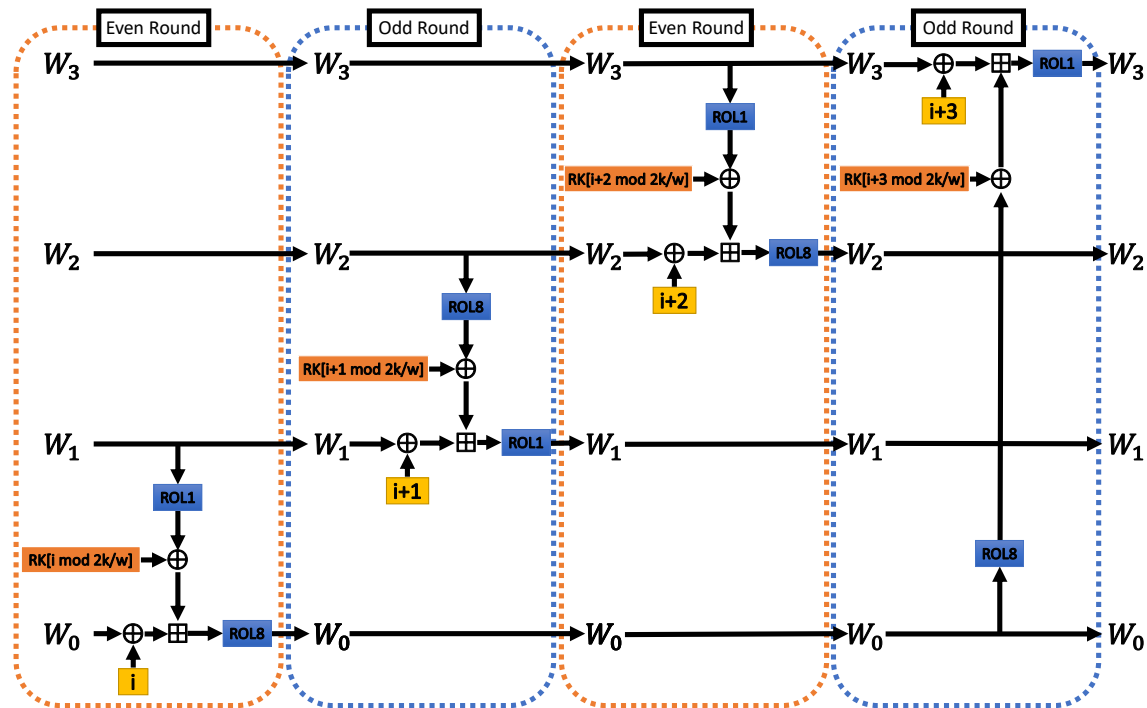


Figure 7. CHAM 4-round combining process.

4.2. Crypto Implementations on Web Assembly Environment

In [17], *HACL** [18], *libsodium* [19], and the proposed *WHACL** [17] libraries are converted to Web Assembly to compare performance. *HACL** is a verified library of cryptographic primitives that is implemented in *Low** and compiled to C via KreMLin [20]. *Libsodium* is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing, and more. *WHACL** is the library proposed in [17]. In Table 4, (A) is a *HACL** library compiled with C using KreMLin and then compiled as Web Assembly through Emscripten, (B) is *libsodium* compiled with Web Assembly through Emscripten, and (C) is *WHACL** compiled with KreMLin. Looking at Table 4, *HACL** is slower than *libsodium* in Curve25519 and Ed25519. *HACL** depends on 128-bit arithmetic in C compilers such as gcc and clang. *Libsodium* converts to 32-bit implementation and operates. Web Assembly also encodes 128-bit integers into 64-bit integer pairs. Due to these characteristics, there is a difference in performance when converting *HACL** and *libsodium* libraries to Web Assembly. As a result, when using a cryptographic algorithm by converting the code implemented in a web-based application into a Web Assembly, implementing a cryptographic algorithm in consideration of the characteristics of such Web Assembly helps to improve performance.

In [21], the official implementation of Picnic [22], which was NIST’s second round candidate for the standardization of quantum tolerance encryption, was converted into Web Assembly, and its performance was measured in Chrome, Firefox, and Microsoft Edge. Comparing Tables 5 and 6, as a result, Web Assembly shows a result that is about 2 ~ 3 times slower than that of C.

Table 4. Performance evaluation of *HACL**. (A) is *HACL*/C*, (B) is *libsodium*, and (C) is *WHACL**. (1k : 1000, B : Byte) [17].

Algorithm (Blocksize, #Rounds)	(A)	(B)	(C)
Curve25519 (1 k)	0.83 s	0.15 s	4.05 s
Chacha20 (4 kB, 100 k)	1.86 s	1.74 s	6.62 s
Salsa21 (4 kB, 100 k)	1.55 s	2.24 s	5.52 s
Ed25519 sign (16 kB, 1 k)	3.01 s	0.27 s	15.6 s
Ed25519 verify (16 kB, 1 k)	3.07 s	0.24 s	15.6 s
Poly1305_32 (16 kB, 10 k)	0.27 s	0.19 s	-
Poly1305_64 (16 kB, 10 k)	1.93 s	0.19 s	11.5 s
SHA2_256 (16 kB, 10 k)	1.64 s	1.84 s	3.5 s
SHA2_512 (16 kB, 10 k)	1.16 s	1.21 s	3.2 s

Table 5. Performance of *picnic* (C implementation) [22].

Parameters	Sign	Verify
Picnic-L1-FS	2.82 ms	2.34 ms
Picnic-L1-UR	3.49 ms	2.87 ms
Picnic2-L1-FS	106.91 ms	42.64 ms
Picnic-L3-FS	6.74 ms	5.66 ms
Picnic-L3-UR	8.64 ms	7.12 ms
Picnic2-L3-FS	328.68 ms	99.27 ms
Picnic-L5-FS	12.37 ms	10.59 ms
Picnic-L5-UR	15.02 ms	12.64 ms
Picnic2-L5-FS	708.82 ms	178.63 ms

Table 6. Performance of *picnic* (Web Assembly implementation) [21].

Parameters	Firefox		Edge		Chrome	
	Sign	Verify	Sign	Verify	Sign	Verify
Picnic-L1-FS	6.67 ms	4.97 ms	8.22 ms	6.56 ms	6.62 ms	6.86 ms
Picnic-L1-UR	8.36 ms	6.36 ms	9.64 ms	7.70 ms	9.61 ms	7.82 ms
Picnic-L3-FS	15.57 ms	12.98 ms	18.54 ms	15.78 ms	18.38 ms	15.56 ms
Picnic-L3-UR	20.11 ms	16.47 ms	22.86 ms	19.08 ms	22.58 ms	19.10 ms
Picnic-L5-FS	27.25 ms	23.01 ms	32.93 ms	29.45 ms	32.62 ms	28.34 ms
Picnic-L5-UR	33.92 ms	28.70 ms	39.91 ms	34.72 ms	38.84 ms	33.12 ms
Picnic-L1-full	5.64 ms	3.82 ms	5.05 ms	3.35 ms	5.01 ms	3.26 ms
Picnic-L3-full	10.06 ms	7.32 ms	8.94 ms	6.66 ms	8.75 ms	6.38 ms
Picnic-L5-full	16.49 ms	13.00 ms	16.12 ms	12.61 ms	16.02 ms	12.26 ms
Picnic3-L1	21.90 ms	17.58 ms	19.63 ms	16.02 ms	19.54 ms	15.46 ms
Picnic3-L3	48.57 ms	38.26 ms	43.74 ms	35.32 ms	43.80 ms	34.58 ms
Picnic3-L5	80.54 ms	59.59 ms	75.38 ms	55.75 ms	73.57 ms	54.30 ms

5. Proposed Web Assembly-Based Crypto Library Implementation

5.1. Proposed Implementation of Revised CHAM

The revised CHAM algorithm is an ARX-based lightweight cipher, and is an algorithm that is safer for differential attacks than the original CHAM. The revised CHAM algorithm is safe for differential attacks because it increases the number of rounds of the original CHAM algorithms, CHAM-64/128, CHAM-128/128, and CHAM-128/256. With this method, we implement the revised CHAM algorithm to be safe for differential attacks by implementing it using Web Assembly. The number of words in the plaintext entering the input value from the original CHAM algorithm is 4. The original CHAM algorithm swaps the place of four words that make up the plaintext at the end of one round.

Rather than swapping four words for each round [9], as shown in Figure 7, it uses a feature that returns to the original words every four rounds to improve performance. At the end of each

round, the round algorithm is calculated using the necessary values while maintaining the position of each word without swapping by removing the word swapping process from the existing algorithm to induce a faster round operation. In CHAM algorithms, the plaintext and 1-word of the key are 16-bit in CHAM-64/128. In Figures 3 and 4, 16-bit word is used as the input to *ROL8*, *ROL1*, and Keyschedule. Algorithms 3 and 4 present a method to pre-compute the input values of *ROL8*, *ROL1*, and Keyschedule, through which the resultant values are 16-bit and are calculated in advance from 0×0 to $0 \times ffff$, the number of all 16-bit inputs. Whenever the *ROL1*, *ROL8*, and Keyschedule functions were required, they used a method of taking and using the result values based on the input computed in the pre-built table rather than the operation.

Algorithm 3 Generation of Rotation Left Shift Table

Output: *ROL1-Table*[$0 \times ffff$], *ROR8-Table*[$0 \times ffff$], *ROR1-Table*[$0 \times ffff$]
 1: **for** $i = 0 \times 0$ to $0 \times ffff$ **do**
 2: *ROL1-Table*[i] \leftarrow *ROL1*(i)
 3: *ROL8-Table*[i] \leftarrow *ROL8*(i)
 4: **end for**

Algorithm 4 Generation of Keyschedule Table

Output: *Key1-Table*[$0 \times ffff$], *Key2-Table*[$0 \times ffff$]
 1: **for** $i = 0 \times 0$ to $0 \times ffff$ **do**
 2: *Key1-Table*[i] $\leftarrow i \oplus$ *ROL1*(i) \oplus *ROL8*(i)
 3: *Key2-Table*[$(i + k/w) \oplus 1$] $\leftarrow i \oplus$ *ROL8*(i) \oplus *ROL11*(i)
 4: **end for**

First, Algorithm 3 is used to create the precomputation table for *ROL1* and *ROL8*. Then, the Keyschedule table is created using Algorithm 4. For the *ROL1* and *ROL8* operations in Algorithm 4, the Keyschedule table can be created faster by using the table created in Algorithm 3.

5.2. Proposed Implementation of ECDH with Side Channel Resistance

In the literature [11], the *NAF* algorithm used a negative representation to reduce the number of 1s for scalar k . As the number of *ECADD* decreases as much as the number of 1, scalar multiplication is possibly faster than before. The *wNAF* algorithm processes *ECADD* for w -bit at once thus, the *wNAF* algorithm realizes a faster scalar multiplication than the binary left to right scalar multiplication algorithm. To process w -bit, pre-computation is required for odd values in the range $[-2^w, 2^{w-1} - 1]$. It can be used at variable points due to the relatively low cost of pre-computation.

To use *wNAF*, conversion from scalar k to *NAF_w*(k) is required, which is realized in the same manner as Algorithm 5. The *NAF_w*(k) can be up to 1 bit longer than the existing k , and the maximum nonzero density will be $\frac{1}{w+1}$. Multiplication for the overall scalar k is performed in the same manner as Algorithm 6. In the pre-calculation, one *ECDBL* and $2^w - 2$ *ECADD* operations are required, and, in the scalar multiplication process, 1 *ECDBL* and $\frac{1}{w+1}$ *ECADD* operations are required. The *wNAF* algorithm is safe for SPA because it uses the number of holes in the range $[-2^w, 2^{w-1} - 1]$. Depending on the bit size of Scalar k , the number of pre-computed *ECADDs* varies. Therefore, it is vulnerable to TA, and it is implemented to be safe for TA using atomic blocks.

Algorithm 5 Computing the width- w NAF of a positive integer

Input : Window width w , positive integer k .
Output : $NAF_w(k)$

- 1: $i \leftarrow 0$.
- 2: **while** $k \geq 1$ **do**
- 3: **if** k is odd **then**
- 4: $k_i \leftarrow k \bmod 2^w, k \leftarrow k - k_i$
- 5: **else**
- 6: $k_i \leftarrow 0$
- 7: **end if**
- 8: $k \leftarrow k/2, i \leftarrow i + 1$
- 9: **end while**
- 10: **return** $(k_{i-1}, k_{i-2}, \dots, k_1, k_0)$

Algorithm 6 Window NAF method for point multiplication

Input : Window width w , positive integer $k, P \in E(\mathbb{F}_q)$
Output : kP

- 1: Use Algorithm 5 to compute $NAF_w(k) = \sum_{i=0}^{l-1} k_i 2^i$
- 2: Compute $P_i = iP$ for $i \in \{1, 3, 5, \dots, 2^{w-1} - 1\}$
- 3: $Q \leftarrow \infty$
- 4: **for** i from $l - 1$ **downto** 0 **do**
- 5: $Q \leftarrow 2Q$
- 6: **if** $k_i \neq 0$ **then**
- 7: **if** $k_i > 0$ **then**
- 8: $Q \leftarrow Q + P_{k_i}$
- 9: **else**
- 10: $Q \leftarrow Q - P_{-k_i}$
- 11: **end if**
- 12: **end if**
- 13: **end for**
- 14: **return** (Q)

Atomic block is safe for SCA by repeating the same process regardless of 0 or 1 in scalar multiplication operation. Atomic block provides safety for SCA by making it difficult to distinguish between *ECDBL* and *ECADD* by adding fake operations to make calculations in a regular order. In this paper, we present the operation process of a new atomic block by reducing the fake operation in the previous atomic block [14].

As seen in Table 3, the existing atomic blocks in the literature [14] consist of $*$, $+$, $-$, and $+$. The method we propose is an improved method, assuming that only $P = (X, Y, Z)$ is used. We propose a method to reduce the number of fake operations by changing the block configuration of the existing atomic block to the configuration of $*$, $+$, $-$. The proposed atomic block composes *ECDBL* and *ECADD* into 10 and 16 blocks by removing one addition in one block process, respectively.

Therefore, 10 and 16 addition operations in *ECDBL* and *ECADD* are reduced compared to the existing atomic block. As for the existing atomic block, *ECDBL* has nine fake additions and eight fake subtractions, and *ECADD* has 22 fake additions and 10 fake subtractions. In the proposed atomic block, *ECDBL* has six fake subtractions, *ECADD* has nine fake additions and nine fake subtractions. Finally, the proposed atomic block reduced nine fake additions and two fake subtractions in *ECDBL* and 13 fake additions and one fake subtraction in *ECADD* compared to the existing atomic block. The proposed atomic block operation process is shown in Table 7.

Table 8 lists the number of additions, subtractions, and multiplications of the original w NAF, the existing atomic block, and the proposed atomic block.

Table 7. Proposed atomic block method.

ECDBL		ECADD	
$T_0 \leftarrow a,$ $T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$		$T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$ $T_{10} \leftarrow X_2, T_{11} \leftarrow Y_2, T_{12} \leftarrow Z_2$	
1.	6.	1.	9.
$T_4 \leftarrow T_1 \cdot T_1$	$T_7 \leftarrow T_0 \cdot T_7$	$T_4 \leftarrow T_{12} \cdot T_{12}$	$T_5 \leftarrow T_7 \cdot T_7$
$T_5 \leftarrow T_4 + T_4$	$T_4 \leftarrow T_4 + T_7$	*	*
*	*	*	*
2.	7.	2.	10.
$T_7 \leftarrow T_3 \cdot T_3$	$T_5 \leftarrow T_3 \cdot T_3$	$T_1 \leftarrow T_4 \cdot T_1$	$T_6 \leftarrow T_5 \cdot T_1$
$T_4 \leftarrow T_5 + T_4$	$T_5 \leftarrow T_7 + T_7$	*	$T_1 \leftarrow T_6 + T_6$
*	$T_5 \leftarrow -T_5$	$T_1 \leftarrow -T_1$	$T_1 \leftarrow -T_1$
3.	8.	3.	11.
$T_5 \leftarrow T_2 \cdot T_2$	$T_6 \leftarrow T_4 \cdot T_4$	$T_5 \leftarrow T_3 \cdot T_3$	$T_4 \leftarrow T_8 \cdot T_8$
$T_8 \leftarrow T_5 + T_5$	$T_1 \leftarrow T_6 + T_5$	*	*
*	$T_1 \leftarrow -T_6$	*	*
4.	9.	4.	12.
$T_9 \leftarrow T_2 \cdot T_3$	$T_8 \leftarrow T_8 \cdot T_9$	$T_7 \leftarrow T_5 \cdot T_{10}$	$T_5 \leftarrow T_7 \cdot T_5$
$T_3 \leftarrow T_9 + T_9$	$T_7 \leftarrow T_7 + T_6$	$T_7 \leftarrow T_7 + T_{10}$	$T_1 \leftarrow T_4 + T_1$
*	$T_8 \leftarrow -T_8$	*	$T_5 \leftarrow -T_5$
5.	10.	5.	13.
$T_7 \leftarrow T_7 \cdot T_7$	$T_4 \leftarrow T_7 \cdot T_4$	$T_4 \leftarrow T_4 \cdot T_{12}$	$T_3 \leftarrow T_3 \cdot T_{12}$
$T_9 \leftarrow T_8 + T_8$	$T_2 \leftarrow T_4 + T_8$	*	$T_1 \leftarrow T_1 + T_5$
*	*	$T_1 \leftarrow -T_1$	$T_1 \leftarrow -T_4$
		6.	14.
		$T_2 \leftarrow T_2 \cdot T_4$	$T_2 \leftarrow T_5 \cdot T_2$
		*	$T_6 \leftarrow T_6 + T_4$
		*	*
		7.	15.
		$T_5 \leftarrow T_5 \cdot T_3$	$T_8 \leftarrow T_8 \cdot T_6$
		*	$T_2 \leftarrow T_8 + T_2$
		$T_2 \leftarrow -T_2$	*
		8.	16.
		$T_8 \leftarrow T_5 \cdot T_{11}$	$T_3 \leftarrow T_3 \cdot T_7$
		$T_8 \leftarrow T_8 + T_2$	*
		$T_2 \leftarrow -T_2$	*

Table 8. Operation count comparison (M: Field multiplication, A: Field addition, and S: Field subtraction).

	ECDBL			ECADD		
	M	A	S	M	A	S
$wNAF$	8	5	4	16	1	6
Existing Atomic Block $wNAF$	10	20	10	16	32	16
Proposed Atomic Block $wNAF$	10	10	10	16	16	16

5.3. Proposed Implementation of HMAC

When a web-based application communicates with other environments, it encrypts the data using various cryptographic algorithms, and then sends the encrypted data. For the sent encrypted data, it is necessary to determine whether it was sent without damage. Encrypted data can be confirmed whether it has been transmitted normally using HMAC, which is MAC made using SHA-256. Implementing HMAC as Web Assembly allows web-based applications to authenticate faster than JavaScript [6,15].

6. Performance Analysis

In the environment of Table 9, the proposed crypto library was implemented as Web Assembly and JavaScript, was compared in Web browsers Chrome, Firefox, and Microsoft Edge to evaluate the performance. Tables 10–18 show the results of the implementation of existing algorithms and the proposed methods, i.e., the revised CHAM algorithm, $wNAF$, SHA-256, and HMAC.

Table 9. Running environment.

Operating System	Window 10 Education
CPU	Intel i5-8250U 1.60 GHz
RAM	8.00 GB
SW	(1) Chrome 85.0.4183.83 (2) Firefox 79.0 (3) Microsoft Edge 84.0.522.63
Languages	(1) JavaScript (2) Web Assembly
$wNAF$ Window width w	4

Table 10. Revised CHAM algorithm performance in Chrome (CPB : Cycle Per Byte).

Algorithm	Language	Optimization Techniques	Average Timing	CPB
revised CHAM-64/128	JavaScript	4-round combining	0.0000013 s	260
revised CHAM-128/128	JavaScript	4-round combining	0.0000018 s	180
revised CHAM-128/256	JavaScript	4-round combining	0.0000021 s	210
This work CHAM-64/128	Web Assembly	4-round combining	0.0000006 s	120 (2.1 times)
This work CHAM-128/128	Web Assembly	4-round combining	0.0000006 s	60 (3 times)
This work CHAM-128/256	Web Assembly	4-round combining	0.0000007 s	70 (3 times)
This work CHAM-64/128	Web Assembly	4-round combining precomputation table	0.0000005 s	100 (1.2 times)

Table 11. Revised CHAM algorithm performance in Firfox (CPB : Cycle Per Byte).

Algorithm	Language	Optimization Techniques	Average Timing	CPB
revised CHAM-64/128	JavaScript	4-round combining	0.0000013 s	260
revised CHAM-128/128	JavaScript	4-round combining	0.0000010 s	100
revised CHAM-128/256	JavaScript	4-round combining	0.0000015 s	150
This work CHAM-64/128	Web Assembly	4-round combining	0.0000006 s	120 (2.1 times)
This work CHAM-128/128	Web Assembly	4-round combining	0.0000006 s	60 (1.6 times)
This work CHAM-128/256	Web Assembly	4-round combining	0.0000007 s	70 (2.1 times)
This work CHAM-64/128	Web Assembly	4-round combining precomputation table	0.0000005 s	100 (1.2 times)

Table 12. Revised CHAM algorithm performance in Microsoft Edge (CPB : Cycle Per Byte).

Algorithm	Language	Optimization Techniques	Average Timing	CPB
revised CHAM-64/128	JavaScript	4-round combining	0.0000012 s	240
revised CHAM-128/128	JavaScript	4-round combining	0.0000013 s	130
revised CHAM-128/256	JavaScript	4-round combining	0.0000020 s	200
This work CHAM-64/128	Web Assembly	4-round combining	0.0000006 s	120 (2 times)
This work CHAM-128/128	Web Assembly	4-round combining	0.0000007 s	70 (1.8 times)
This work CHAM-128/256	Web Assembly	4-round combining	0.0000007 s	70 (2.8 times)
This work CHAM-64/128	Web Assembly	4-round combining precomputation table	0.0000005 s	100 (1.2 times)

Tables 10–12 are the results of measuring the implemented CHAM family algorithm in Chrome, Firefox, and Microsoft Edge. The revised CHAM family algorithm has a 4-round combination method,

and additionally, CHAM-64/128 is implemented with JavaScript and Web Assembly by applying a pre-computation method. As a result, the revised CHAM algorithm with the applied 4-round combining method showed an improved performance, in Chrome, Firefox, and MicrosoftEdge, by 2.1, 2.1, and 2 times for CHAM-64/128, 3, 1.6, and 1.8 times for CHAM-128/128, and 3, 2.1, and 2.8 times for CHAM-128/256. Pre-computation applied to CHAM-64/128 shows a 1.2 times performance improvement than existing revised CHAM-64/128 in three web browsers.

Table 13. *wNAF* algorithm performance in Chrome (CPB: Cycle Per Byte), ($w = 4$).

Algorithm	Language	Average Timing	CPB	Performance Overhead
original <i>wNAF</i>	JavaScript	0.000012 s	300	-
Existing Atomic Block <i>wNAF</i>	JavaScript	0.0000179 s	447	49%
Proposed Atomic Block <i>wNAF</i>	JavaScript	0.0000146 s	365	21%
original <i>wNAF</i>	Web Assembly	0.0000011 s	27 (11 times)	-
Existing Atomic Block <i>wNAF</i>	Web Assembly	0.0000017 s	42 (10 times)	55%
Proposed Atomic Block <i>wNAF</i>	Web Assembly	0.0000013 s	32 (11 times)	18%

Table 14. *wNAF* algorithm performance in Firefox (CPB: Cycle Per Byte), ($w = 4$).

Algorithm	Language	Average Timing	CPB	Performance Overhead
original <i>wNAF</i>	JavaScript	0.0000146 s	365	-
Existing Atomic Block <i>wNAF</i>	JavaScript	0.0000162 s	405	10%
Proposed Atomic Block <i>wNAF</i>	JavaScript	0.0000155 s	387	6%
original <i>wNAF</i>	Web Assembly	0.0000012 s	30 (12 times)	-
Existing Atomic Block <i>wNAF</i>	Web Assembly	0.0000015 s	37 (10 times)	23%
Proposed Atomic Block <i>wNAF</i>	Web Assembly	0.0000013 s	32 (12 times)	6%

Table 15. *wNAF* algorithm performance in Microsoft Edge (CPB: Cycle Per Byte), ($w = 4$).

Algorithm	Language	Average Timing	CPB	Performance Overhead
original <i>wNAF</i>	JavaScript	0.0000129 s	322	-
Existing Atomic Block <i>wNAF</i>	JavaScript	0.0000209 s	522	62%
Proposed Atomic Block <i>wNAF</i>	JavaScript	0.0000175 s	437	35%
original <i>wNAF</i>	Web Assembly	0.0000011 s	27 (11 times)	-
Existing Atomic Block <i>wNAF</i>	Web Assembly	0.0000015 s	37 (14 times)	37%
Proposed Atomic Block <i>wNAF</i>	Web Assembly	0.0000012 s	30 (14 times)	11%

Tables 13–15 are the result tables measured in Chrome, Firefox, and Microsoft Edge after implementing the original *wNAF*, the existing atomic block *wNAF*, and the proposed atomic block *wNAF* with JavaScript and Web Assembly. As a result of measurement in Chrome, Firefox, and Microsoft Edge, Web Assembly improved more than JavaScript, for the original *wNAF* by 11, 12, and 11 times, the existing atomic block *wNAF* by 10, 10, and 14 times, and the proposed *wNAF* by 11, 12, and 14 times. As shown in Table 8, the atomic block increases the number of operations compared to the existing *ECDBL* and *ECADD*, resulting in performance overhead. Therefore, in the case of the existing atomic block *wNAF*, performance overhead of 55, 23, and 37% occurs. However, in the case of the atomic block *wNAF* proposed in $P = (X, Y, Z)$, the number of operations is reduced, resulting in a performance overhead of 18%, 6%, and 11%, and scalar multiplication is possible faster than the conventional atomic block.

Table 16. HMAC algorithm performance in Chrome (CPB: Cycle Per Byte).

Algorithm	Language	Average Timing	CPB
SHA-256	JavaScript	0.0000163 s	203
HMAC	JavaScript	0.0000558 s	697
This work SHA-256	Web Assembly	0.0000022 s	27 (7.5 times)
This work HMAC	Web Assembly	0.0000074 s	92 (7.5 times)

Table 17. HMAC algorithm performance in Firefox (CPB: Cycle Per Byte).

Algorithm	Language	Average Timing	CPB
SHA-256	JavaScript	0.0000173 s	216
HMAC	JavaScript	0.0001852 s	2315
This work SHA-256	Web Assembly	0.0000016 s	20 (10.8 times)
This work HMAC	Web Assembly	0.0000075 s	93 (24.8 times)

Table 18. HMAC algorithm performance in Microsoft Edge (CPB: Cycle Per Byte).

Algorithm	Language	Average Timing	CPB
SHA-256	JavaScript	0.0000177 s	221
HMAC	JavaScript	0.0000555 s	693
This work SHA-256	Web Assembly	0.0000016 s	20 (11 times)
This work HMAC	Web Assembly	0.0000078 s	97 (7.1 times)

Tables 16–18 are the results of measuring HMAC, a MAC made using SHA-256 and SHA-256 implemented with JavaScript and Web Assembly in Chrome, Firefox, and Microsoft Edge. As a result, Web Assembly showed a higher performance by 7.5, 10.8, and 11 times for SHA-256, and 7.5, 24.8, and 7.1 times for HMAC, over JavaScript in Chrome, Firefox, and Microsoft Edge.

7. Conclusions

In this paper, we proposed a crypto library by implementing a cryptographic algorithm using Web Assembly to improve the performance of cryptographic algorithms in web-based applications. The block cipher, key exchange algorithm, and MAC algorithm were implemented directly in JavaScript and Web Assembly and were compared. As the block cipher, we employed a lightweight cipher (i.e., the CHAM algorithm), applied the four-round combining method, and applied revised CHAM algorithm method, which is secure against differential attacks. Algorithms implemented in Web Assembly and JavaScript were measured in Chrome, Firefox, and Microsoft Edge. In case of block cipher, 2.1, 2.1, and 2 times for CHAM-64/128, 3, 1.6, and 1.8 times for CHAM-128/128, and 3, 2.1, and 2.8 times for CHAM-128/256 showed improvement in performance. CHAM-64/128 to which the pre-computation method was applied showed a performance improvement of 1.2 times in three web browsers than when the algorithm was not applied. For the key exchange algorithm, *wNAF* was applied to P-256. The atomic block method, which is an algorithm corresponding against SPA and TA, was also applied. When applying the existing atomic block and proposed atomic block to *wNAF*, we checked how much the performance overhead appeared in comparison to the original *wNAF* due to the increased number of operations, and how much the proposed atomic block improved over the existing atomic block. For this purpose, each algorithm implemented in Web Assembly and JavaScript was measured in Chrome, Firefox, and Microsoft Edge. As a result, Web Assembly improved over JavaScript, for the original *wNAF* by 11, 12, and 11 times, the existing atomic block *wNAF* by 10, 10, 14 times, and the proposed *wNAF* by 11, 12, and 14 times. Existing atomic block *wNAF* shows a performance overhead of 55%, 23%, and 37% compared to the original *wNAF*. However, the atomic block *wNAF* was proposed to be used at $P = (X, Y, Z)$ showing performance overheads of 18%, 6% and 11%. The message authentication code was HMAC, which uses SHA-256 to create a MAC. As a result of the measurement, Web Assembly showed higher performance over JavaScript by 7.5, 10.8, and 11 times for SHA-256, and 7.5, 24.8, and 7.1 times for HMAC.

Web Assembly will continue to evolve through several web browser companies. Web Assembly is intended to be used together, not as a replacement for JavaScript. Therefore, with the development of Web Assembly in future, the function call time between Web Assembly and JavaScript will gradually decrease. Thus, from a cryptographic algorithm perspective in future, Web Assembly will be an appropriate language to use. Cryptographic algorithms with a lot of mathematical operations use Web Assembly, and additionally, it will be more efficient from a Web-based application perspective if

it is configured using a JavaScript library of various functions. Web Assembly works in a SIMD way, and therefore, there is a disadvantage that Web Assembly is slower when processing the same amount of data than the cryptographic algorithm using SIMD which is currently being studied. However, Web Assembly is being developed to support the SIMD method, supporting quite a few intrinsic functions, and is continuously evolving. In addition, an API called WebGPU is being created that can use the functions of a graphic card in a web environment. WebGPU enables the SIMD operation using a graphic card in a web environment. In addition, WebGPU is evolving to support use with Web Assembly. Eventually, we will be able to encrypt and decrypt large amounts of data at high speed when we can use high-performance functions in the web environment such as Web Assembly and WebGPU in future. Currently, there are various attack methods for cryptographic algorithms, but our proposed crypto library only applied a differential attack for block ciphers and SPA and TA for key exchange. We plan to investigate possible attack methods for cryptographic algorithms in the web environment in future and study to improve response algorithms suitable for attack methods. In addition, we will study further because it will be possible to optimize cryptographic algorithms in web-based applications through the support of Web Assembly's SIMD and WebGPU. As such, research on cryptographic libraries used in web-based applications through the development of Web Assembly and support for various functions in the future will be of valuable study.

Author Contributions: Writing—original draft, B.P. and J.S.; Writing—review and editing, S.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF), grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494).

Conflicts of Interest: The authors declare no conflict of interest.

Reference

1. Telecommunications Technology Association. Security Management Guidelines for Web Environment Establishment & Operation. 2006. Available online: http://www.tta.or.kr/data/ttas_view.jsp?rn=1&by=asc&order=publish_date&totalSu=16253&pk_num=TTAS.KO-10.0090/R1&nowSu=5594 (accessed on 2 November 2020).
2. Zakas, N.C. *Professional Javascript for Web Developers*; John Wiley & Sons: Hoboken, NJ, USA, 2009.
3. Rossberg, A.; Titzer, B.L.; Haas, A.; Schuff, D.L.; Gohman, D.; Wagner, L.; Zakai, A.; Bastien, J.F.; Holman, M. Bringing the web up to speed with WebAssembly. *Commun. ACM* **2018**, *61*, 107–115. [CrossRef]
4. Rossberg, A. WebAssembly Specification Release 1.1. 2020. Available online: <https://webassembly.github.io/spec/core/> (accessed on 30 October 2020).
5. Roh, D.; Koo, B.; Jung, Y.; Jeong, I.; Lee, D.; Kwon, D.; Kim, W.H. Revised Version of Block Cipher CHAM. In Proceedings of the Information Security and Cryptology—ICISC 2019—22nd International Conference, Seoul, Korea, 4–6 December 2019; Revised Selected Papers; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11975, pp. 1–19.
6. Federal Information Processing Standards Publications 198-1(FIPS PUBS). In *The Keyed-Hash Message Authentication Code (HMAC)*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2008.
7. Barker, E.; Chen, L.; Roginsky, A.; Vassilev, A.; Davis, R. *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*; Technical Report; National Institute of Standards and Technology(NIST): Gaithersburg, MD, USA, 2018.
8. Federal Information Processing Standards Publications 186-4(FIPS PUBS). In *Digital Signature Standard (DSS)*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2013.
9. Park, C.; Park, T.; Seo, H.; Kim, H. Optimization of CHAM Encryption Algorithm Based on Javascript. In Proceedings of the Tenth International Conference on Ubiquitous and Future Networks, ICUFN 2018, Prague, Czech Republic, 3–6 July 2018; pp. 774–778.

10. Koo, B.; Roh, D.; Kim, H.; Jung, Y.; Lee, D.; Kwon, D. CHAM: A Family of Lightweight Block Ciphers for Resource-Constrained Devices. In Proceedings of the Information Security and Cryptology—ICISC 2017—20th International Conference, Seoul, Korea, 29 November–1 December 2017; Revised Selected Papers; Kim, H., Kim, D.C., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10779, pp. 3–25.
11. Hankerson, D.; Menezes, A.J.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2006.
12. Möller, B. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks. In Proceedings of the Information Security, 4th International Conference, ISC 2001, Malaga, Spain, 1–3 October 2001; Davida, G.I., Frankel, Y., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2200, pp. 324–334.
13. Izu, T.; Möller, B.; Takagi, T. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks. In Proceedings of the Progress in Cryptology—INDOCRYPT 2002, Third International Conference on Cryptology in India, Hyderabad, India, 16–18 December 2002; Menezes, A., Sarkar, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2551, pp. 296–313.
14. Chevallier-Mames, B.; Ciet, M.; Joye, M. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Trans. Comput.* **2004**, *53*, 760–768. [[CrossRef](#)]
15. Federal Information Processing Standards Publications 180-4(FIPS PUBS). In *Secure Hash Standard (SHS)*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2015.
16. An, K.; Kwon, H.; Kim, H.; Seo, H. Implementation of Ultra-Light Block Cipher CHAM Optimization Using Web Assembly. *J. Korea Inst. Inf. Secur.* 2019. Available online: https://github.com/solowal/PUBLICATION/blob/master/2019/%EC%9B%B9%20%EC%96%B4%EC%85%88%EB%B8%94%EB%A6%AC%EB%A5%BC%20%ED%99%9C%EC%9A%A9%ED%95%9C%20%EC%B4%88%EA%B2%BD%EB%9F%89%20%EB%B8%94%EB%A1%9D%EC%95%94%ED%98%B8%20CHAM%20%EC%B5%9C%EC%A0%81%ED%99%94%20%EA%B5%AC%ED%98%84_%EB%85%BC%EB%AC%B8.pdf (accessed on 2 November 2020).
17. Protzenko, J.; Beurdouche, B.; Merigoux, D.; Bhargavan, K. Formally Verified Cryptographic Web Applications in WebAssembly. In Proceedings of the 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, 19–23 May 2019; pp. 1256–1274.
18. Zinzindohoué, J.K.; Bhargavan, K.; Protzenko, J.; Beurdouche, B. HACL*: A Verified Modern Cryptographic Library. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–3 November 2017; pp. 1789–1806.
19. Bernstein, D.J.; Denis, F. Libsodium—A Modern, Portable, Easy to Use Crypto Library. 2019. Available online: <https://github.com/lemons/libodium> (accessed on 30 October 2020).
20. Protzenko, J.; Zinzindohoué, J.K.; Rastogi, A.; Ramananandro, T.; Wang, P.; Béguelin, S.Z.; Delignat-Lavaud, A.; Hritcu, C.; Bhargavan, K.; Fournet, C.; et al. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* **2017**, *1*, 17:1–17:29. [[CrossRef](#)]
21. Rösch, J. Efficient implementation of Picnic. Available online: <https://is.muni.cz/th/pbn05/> (accessed on 30 October 2020).
22. Chase, M.; Derler, D.; Goldfeder, S.; Katz, J.; Kolesnikov, V.; Orlandi, C.; Ramacher, S.; Rechberger, C.; Slamanig, D.; Wang, X.; et al. The Picnic Signature Scheme Design Document. 2020. Available online: <https://github.com/microsoft/Picnic/blob/master/spec/design-v2.2.pdf> (accessed on 2 November 2020).

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).