Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 12(4): 252-256, Dec. 2014 Regular paper

Low-Power Encryption Algorithm Block Cipher in JavaScript

Hwajeong Seo and Howon Kim*, *Member, KIICE*

Department of Computer Engineering, Pusan National University, Pusan 609-735, Korea

Abstract

Traditional block cipher Advanced Encryption Standard (AES) is widely used in the field of network security, but it has high overhead on each operation. In the 15th international workshop on information security applications, a novel lightweight and low-power encryption algorithm named low-power encryption algorithm (LEA) was released. This algorithm has certain useful features for hardware and software implementations, that is, simple addition, rotation, exclusive-or (ARX) operations, non-Substitute-BOX architecture, and 32-bit word size. In this study, we further improve the LEA encryptions for cloud computing. The Web-based implementations include JavaScript and assembly codes. Unlike normal implementation, JavaScript does not support unsigned integer and rotation operations; therefore, we present several techniques for resolving this issue. Furthermore, the proposed method yields a speed-optimized result and shows high performance enhancements. Each implementation is tested using various Web browsers, such as Google Chrome, Internet Explorer, and Mozilla Firefox, and on various devices including personal computers and mobile devices. These results extend the use of LEA encryption to any circumstance.

Index Terms: Assembly, Block cipher, JavaScript, Low-power encryption, Web application

I. INTRODUCTION

With the development of cloud computing environments, several JavaScript encryption libraries have been suggested for encrypting data in a Web browser [1-5]. JavaScript cryptography is used in mashups to provide secure cross-origin messaging by using fragment identifiers [6]. Furthermore, Mozilla Firefox extensions are written in JavaScript. Further, Adobe Air and Mozilla Prism are full-fledged environments for developing desktop applications in variants of JavaScript. The most popular encryption method is AES [7]. The block cipher is designed for an 8-bit processor. Even though thousands of works provide high performance on processors with a large word size, many high-performance block ciphers that are a considerably more favorable choice for cloud computing environments that need thousands of encryptions within a second have been developed.

In 2013, the low-power encryption algorithm (LEA) was proposed by the Attached Institute of ETRI [8]. This algorithm has software-friendly architecture, and efficient implementation results on a wide range of computational devices from high-end machines, such as personal computers, to low-end microprocessors have been reported in previous papers [8, 9]. In this paper, we present LEA implementations on cloud computing, including JavaScript and ASM JavaScript. This result can contribute to widening the usage of the LEA block cipher, particularly in the field of cloud computing.

The rest of this paper is organized as follows: in Section II, we briefly discuss the basic specifications of LEA and target platforms. In Section III, we present the novel implementation techniques. In Section IV, we evaluate the

Received 15 August 2014, **Revised** 20 October 2014, **Accepted** 14 November 2014
***Corresponding Author** Howon Kim (E-mail: howonkim@pusan.ac.kr. Tel: +82-51-510-3927)
Department of Computer Engineering, Pusan National University, 2 Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 609-735, Korea.

Open Access **http://dx.doi.org/10.6109/jicce.2014.12.4.252** print ISSN: 2234-8255 online ISSN: 2234-8883

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/li-censes/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Copyright ⓒ The Korea Institute of Information and Communication Engineering

performance of the proposed methods in terms of clock cycles and compare the results with those reported in related works. Finally, Section V concludes this paper.

II. SPECIFICATIONS OF LEA

LEA is a block cipher with a 128-bit block. The key size is 128, 192, and 256 bits. The number of rounds is 24, 28, and 32 for 128-, 192- and 256-bit keys, respectively. The process consists of key scheduling, encryption, and decryption. The following subsection presents the notations of the key scheduling and encryption operations.

Table 1. Key schedule for LEA

Input: master key K, constants _. **Output: round key RK.**
1. T[0] = K[0], T[1] = K[1], T[2] = K[2], T[3] = K[3].
2. for i = 0 to 23
3. T[0] = ROL1(T[0] + ROLi(delta [i mod 4])
4. T[1] = ROL3(T[1] + ROLi + 1(delta [i mod 4])
5. T[2] = ROL6(T[2] + ROLi + 2(delta [i mod 4])
6. T[3] = ROL11(T[3] + ROLi + 3(delta[i mod 4])
7. RKi = (T[0], T[1], T[2], T[1], T[3], T[1])
8. end for
9. return RK

LEA: low-power encryption algorithm.

Table 2. Encryption for LEA

Input: plaintext P, round key RK. **Output: ciphertext C**
1. X0[0] = P[0], X0[1] = P[1], X0[2] = P[2], X0[3] = P[3].
2. for i = 0 to 23
3. Xi + 1[0] = ROL9(Xi[0] ⊕ RKi[0]) + (Xi[1] ⊕ RKi[1])
4. Xi + 1[1] = ROR5(Xi[1] ⊕ RKi[2]) + (Xi[2] ⊕ RKi[3])
5. Xi + 1[2] = ROR3(Xi[2] ⊕ RKi[4]) + (Xi[3] ⊕ RKi[5])
6. Xi + 1[3] = Xi[0]
7. end for
8. C[0] = X24[0], C[1] = X24[1], C[2] = X24[2], C[3] = X24[3].
9. return C

LEA: low-power encryption algorithm.

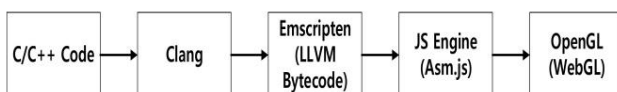

Fig. 1. Computation process of ASM.javascript.

A. Key Schedule

Key scheduling generates a sequence of round keys RKi as follows: it uses several constants for generating round keys, which are defined as delta[8] = {0xc3efe9db, 0x44626b02, 0x79e27c8a, 0x78df30ec, 0x715ea49e, 0xc785da0a, 0xe04ef22a, 0xe5c40957}. The detailed key scheduling process for 128-bit keys is presented in Table 1.

B. Encryption/Decryption

The LEA encryption procedure consists of 24 rounds for 128-bit keys, 28 rounds for 192-bit keys, and 32 rounds for 256-bit keys. For 24, 28, and 32 rounds, it encrypts a 128-bit plaintext P = (P[0], P[1], P[2], P[3]) generating a 128-bit ciphertext C = (C[0], C[1], C[2], C[3]) with 128-, 192-, and 256-bit keys. We omit the description of the decryption procedure because it is simply considered the inverse of the encryption procedure. The detailed encryption process for 128-bit keys is described in Table 2.

III. PROPOSED METHOD

A. Target Platform: Cloud Platforms

In some applications, client-side encryption is needed before data are uploaded to a server cloud. Since the web browser is becoming the universal tool for interacting with remote servers, it is natural to ask whether existing browsers can perform encryption without installing additional client-side software. In [10], a small, fast AES implementation in JavaScript for a browser has been studied. These researchers used various lookup table approaches and native x86 code for the implementation. Furthermore, they presented several comparison reports on different browsers and ciphers. However, there are no LEA results on cloud computing environments; therefore, in this paper, we provide detailed results for various environments. Programs written in JavaScript and asm.js are compared. Between them, native C-based JavaScript shows higher performance under all circumstances. By using asm.js, we can compile all former C/C++ applications into JavaScript with the support of Mozilla's Emscripten project. Emscripten takes in C/C++ code, passes it through LLVM, and converts the LLVM-generated bytecode into JavaScript, also known as asm.js, which is a subset of JavaScript. If the compiled asm.js code is doing some rendering, then it is most likely being handled by WebGL. Thus, the entire pipeline is technically making use of JavaScript. The detailed process is depicted in Fig. 1. This type of JavaScript highly optimizes the source code and shows higher performance than ordinary JavaScript.

B. Implementation of LEA Block Cipher

Block cipher consists of key scheduling and encryption operations. Key scheduling generates round keys that can be precomputed offline. After round key computation, the keys are used for the encryption. There are two main approaches to establish block cipher computations. First, an on-the-fly method generates a round key on the spot and then, directly encrypts plaintext with these round key pairs. The main benefits of the method are that storage for the round key is not needed and the source code size is reduced by rolling the encryption and key scheduling within the short length of the program. The algorithm for the on-the-fly method is presented in Table 3.

Second, a separated computation mode literally executes the key scheduling and encryption processes separately. The round keys are extracted and then, stored into temporal buffers of a certain size. Then, these values are simply loaded and used during the encryption or decryption processes. The advantage of this method is that by eliminating the key generation process, a fast computation time is obtained. The algorithm for the separated mode is presented in Table 4. In this paper, we focused on fast implementation, because cloud computing environments need high-speed performance and the platforms have sufficient capacity to retain round keys and source codes.

From the viewpoint of programming, JavaScript is an abstract form of a high-level language that enables users to write programs easily and reduce logical errors. For the LEA implementation, we established basic ARX operations in the Java language because these are basic components of LEA. The variables are explicitly defined in a 32-bit format.

Table 3. On-the-fly method

Input: master key K, constants delta, plaintext P. **Output: ciphertext C.**
1. T[0] = K[0], T[1] = K[1], T[2] = K[2], T[3] = K[3].
2. X0[0] = P[0], X0[1] = P[1], X0[2] = P[2], X0[3] = P[3].
2. for i = 0 to 23
4. T[0] = ROL1(T[0] + ROLi(delta[i mod 4))
5. T[1] = ROL3(T[1] + ROLi + 1(delta[i mod 4))
6. T[2] = ROL6(T[2] + ROLi + 2(delta[i mod 4))
7. T[3] = ROL11(T[3] + ROLi + 3(delta[i mod 4))
8. Xi + 1[0] = ROL9(Xi[0] ⊕ T[0]) + (Xi[1] ⊕ T[1])
9. Xi + 1[1] = ROR5(Xi[1] ⊕ T[2]) + (Xi[2] ⊕ T[1])
10. Xi + 1[2] = ROR3(Xi[2] ⊕ T[3]) + (Xi[3] ⊕ T[1])
11. Xi + 1[3] = Xi[0]
12. end for
13. C[0] = X24[0], C[1] = X24[1], C[2] = X24[2], C[3] = X24[3].
14. return C

manually implemented the unsigned int data type. In Table 5, However, in JavaScript, there is no unsigned int type, so we unsigned int rotation is realized in JavaScript. JavaScript provides unsigned int right shift (>>>) but not left shift (<<). To convert the signed int into the unsigned int type, we executed an unsigned int right shift by zero. The detailed descriptions are given in Table 6. For the addition and bitwise exclusive-or, basic arithmetic and logical operations, such as + and ^ were exploited.

We tried to unroll the LEA encryption process. The looped process usually presents a small program size but slow performance due to the computation of loop handling. However, we found strange results, which were contradictory to our general knowledge. A detailed discussion of these results is presented in the evaluation section. The program was also executed in a separated form; therefore, first, we conducted key scheduling and constructed whole round key pairs, and then, we conducted encryptions with the round keys. For a higher optimal implementation, we compiled the C/C++ implementation using Emscripten to output JavaScript. This method generates JavaScript from the native C language; therefore, compared with JavaScript, the program is highly optimized. However, the original JavaScript implementation works on various servers and platforms without code modifications, unlike C-based coding.

Table 4. Separated computation method

Input: master key K, constants delta, plaintext P. **Intermediate: round key RK.** **Output: ciphertext C.**
1. T[0] = K[0], T[1] = K[1], T[2] = K[2], T[3] = K[3].
2. for i = 0 to 23
3. T[0] = ROL1(T[0] + ROLi(delta[i mod 4))
4. T[1] = ROL3(T[1] + ROLi + 1(delta[i mod 4))
5. T[2] = ROL6(T[2] + ROLi + 2(delta[i mod 4))
6. T[3] = ROL11(T[3] + ROLi + 3(delta[i mod 4))
7. RKi = (T[0]; T[1]; T[2]; T[1]; T[3]; T[1])
8. end for
9. X0[0] = P[0], X0[1] = P[1], X0[2] = P[2], X0[3] = P[3].
10. for i = 0 to 23
11. Xi + 1[0] = ROL9(Xi[0] ⊕ RKi[0]) + (Xi[1] ⊕ RKi[1])
12. Xi + 1[1] = ROR5(Xi[1] ⊕ RKi[2]) + (Xi[2] ⊕ RKi[3])
13. Xi + 1[2] = ROR3(Xi[2] ⊕ RKi[4]) + (Xi[3] ⊕ RKi[5])
14. Xi + 1[3] = Xi[0]
15. end for
16. C[0] = X24[0], C[1] = X24[1], C[2] = X24[2], C[3] = X24[3].
17. return C

IV. EVALUATION

We evaluated the performance of the proposed method on Intel Core i7-3770 powered by 3.4 GHz with 8-GB RAM. We used 64-bit Windows 7 as the operating system, and JavaScript 1.5 and Emscripten 1.12.0 as the development tool and compiler, respectively. The target browsers were Chrome 33.0.1750.154m, Internet Explorer 10, and Firefox 27.0.1. We analyzed performance by measuring the time taken in milliseconds with getUTCMilliseconds() in terms of the browser, language, and loop/unroll modes, and have presented the results in Table 6. For browsers, significant differences were observed. Internet Explorer showed the slowest performance among browsers. The performance of Chrome was faster than that of Firefox with JavaScript but was slower than that with asm.js. Between JavaScript and asm.js, JavaScript was slower than asm.js by about 41%–90.5%. This implies that irrespective of the web browser, asm.js had better performance but required a considerably large amount of source code and an Emscipten environment.

For achieving further improvements, we implemented unrolled and looped versions for LEA and tested both versions in both languages. Strangely, JavaScript always exhibited poorer performance in the case of the unrolled version. We assumed that JavaScript was operated over a virtual machine for wide computability. When the virtual machine launched the program, a long program would generate additional overhead. This would lead to performance degradation from our perspective. Compared with the previous AES implementation, LEA exhibited an improved performance of 94.3% and 69.8% in the case of JavaScript with Chrome and Firefox, respectively, that of 96.9%, 33.3%, and 97.1% in the case of asm.js with Chrome, Internet Explorer, and Firefox, respectively. Only JavaScript with Internet Explorer exhibited 14.8% degradation in performance.

In order to show the compatibility of JavaScript, we measured performance even on a cellphone. The target device was Samsung Galaxy S3 supporting Cortex-A9 and 1.4-GHz quad core with 1-GB RAM. The operating system was Android 4.3. We executed our programs over Mobile Chrome 34.0.1847, Webkit 4.3, and Firefox 28.0.1. The detailed results are presented in Table 7. Among the three browsers, Chrome showed the best performance. We think that Google has highly optimized Chrome to operate over the Android platform. Further, the basic browser of Android, Webkit, showed the second best performance. Finally, Firefox ranked last.

Table 5. Rotation method on JavaScript

```
function ROL(input, offset){
input = ((input<<offset)>>>0) | (input>>>(32-offset));
return input
  }//Left rotation
```

```
function ROR(input, offset){
input = ((input<<32-offset)>>>0) | (input>>>(offset));
return input
  }//Right rotation
```

Table 6. Comparison results of JavaScript on desktop

Method	Key	Enc	Code size
Proposed			
128-bit J, C [L]	114.8	65.9	2309
128-bit J, I.E. [L]	998.8	1232.5	2309
128-bit J, FF [L]	252.9	246.5	2309
128-bit J, C [U]	114.8	240.1	5510
128-bit J, I.E. [U]	998.8	1997.5	5510
128-bit J, FF [U]	252.9	276.3	5510
128-bit A, C [L]	63.8	36.1	224642
128-bit A, I.E. [L]	688.5	716.1	224642
128-bit A, FF [L]	44.6	23.4	224642
128-bit A, C [U]	63.8	42.5	228614
128-bit A, I.E. [U]	688.5	637.5	228614
128-bit A, FF [U]	44.6	25.5	228614
AES			
128-bit J, C [3]	n/a	1176	6057
128-bit J, I.E. [3]	n/a	1073.7	6057
128-bit J, FF [3]	n/a	818.1	6057

Key and Enc are measured in cycles/byte and code size in bytes.
LEA: low-power encryption algorithm, AES: Advanced Encryption Standard, J: JavaScript, A: asm.js, C: Chrome, I.E.: Internet Explorer, FF: Firefox, [L]: looped, [U]: unrolled.

Table 7. Comparison results of JavaScript on mobile platform

Method	Key	Enc	Code size
Proposed LEA			
128-bit J, C [L]	1032	350	2309
128-bit J, W [L]	1466	934	2309
128-bit J, FF [L]	2345	1102	2309
128-bit J, C [U]	1032	1697	5510
128-bit J, W [U]	1466	2614	5510
128-bit J, FF [U]	2345	3132	5510
AES			
128-bit J, C [3]	n/a	3524	6057
128-bit J, W [3]	n/a	4581	6057
128-bit J, FF [3]	n/a	4012	6057

Key and Enc are measured in cycles/byte and code size in bytes.
LEA: low-power encryption algorithm, AES: Advanced Encryption Standard, J: JavaScript, C: Chrome, W: Webkit, FF: Firefox, [L]: looped, [U]: unrolled.

V. CONCLUSION

In this paper, we implemented the LEA algorithm in JavaScript languages. In order to provide efficient implementations, we tried to use JavaScript and ASM JavaScript.

Furthermore, various browsers including Firefox, Internet Explorer, and Chrome were considered. In this study, we also explored the trade-off between code size and speed in cloud computing environments. In the future, we intend to focus on authenticated encryption with the proposed method in various cloud computing environments.

ACKNOWLEDGMENTS

This research was supported by the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the Information Technology Research Center support program (NIPA-2014-H0301-14-1048) supervised by the National IT Industry Promotion Agency (NIPA).

REFERENCES

[1] Marco and G. Cesare, Clipperz online password manager [Internet], Available: http://www.clipperz.com.

[2] Google Browser Sync [Internet], Available: http://www.google.com/tools/firefox/browsersync/.

[3] E. Styere, Javascript AES Example [Internet], Available: http://people.eku.edu/styere/Encrypt/JS-AES.html.

[4] J. Walker, JavaScrypt: browser-based cryptography tools [Internet], Available: http://www.fourmilab.ch/javascrypt.

[5] Javascript implementation of AES in counter mode [Internet], Available: http://www.movable-type.co.uk/scripts/aes.html.

[6] C. Jackson, A. Barth, and J. Mitchell, "Securing frame communication in browsers," in *Proceedings of 17th USENIX Security Symposium*, San Jose, CA, 2008.

[7] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - the Advanced Encryption Standard*. Heidelberg: Springer, 2002.

[8] D. Hong, J. K. Lee, D. C. Kim, D. Kwon, K. H. Ryu, and D. G. Lee, "LEA: a 128-bit block cipher for fast encryption on common processors," in *Information Security Applications.* Heidelberg: Springer, pp. 3-27, 2014.

[9] D. Lee, D. C. Kim, D. Kwon, and H. Kim, "Efficient hard-ware implementation of the lightweight block encryption algorithm LEA," *Sensors*, vol. 14, no. 1, pp. 975-994, 2014.

[10] E. Stark, M. Hamburg, and D. Boneh, "Symmetric cryptography in JavaScript," in *Proceedings of the Computer Security Applications Conference (ACSAC2009)*, Honolulu, HI, pp. 373-381, 2009.

Hwajeong Seo
received his B.S.E.E. from Pusan National University, Pusan, Republic of Korea, in 2010. He also received his M.S. and Ph.D. in Computer Engineering from the same university. His research interests include sensor networks, information security, elliptic curve cryptography, and RFID security.

Howon Kim
received his B.S.E.E. from Kyungpook National University, Daegu, Republic of Korea, in 1993, and his M.S. and Ph.D. in Electronic and Electrical Engineering from Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea, in 1995 and 1999, respectively. From July 2003 to June 2004, he studied with the COSY group at the Ruhr-University of Bochum, Germany. He was a senior member of the technical staff at the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea. He is currently working as an associate professor with the Department of Computer Engineering, School of Computer Science and Engineering, Pusan National University, Busan, Republic of Korea. His research interests include RFID technology, sensor networks, information security, and computer architecture. Currently, his main research focus is on mobile RFID technology and sensor networks, public key cryptosystems, and their security issues. He is a member of the IEEE and the International Association for Cryptologic Research (IACR).