CUSTOMIZATION SOLUTION FOR SCRYPT KEY DERIVATION TO IMPROVE THE SECURITY OF STORED DATA

Nguyen Van Nghi*, Do Quang Trung, Vu Ba Linh
Academy of Cryptography Techniques, 141 Chien Thang, Tan Trieu, Thanh Tri, Ha Noi

ARTICLE INFO

Received: 31/10/2022
Revised: 26/12/2022
Published: 26/12/2022

KEYWORDS

Scrypt
ChaCha20/8
Salsa20/8
SHA-3
AES-CBC

ABSTRACT

Scrypt is a password-based key derivation function used in many storage security applications. Scrypt is based on a memory-hard structure that is resistant to brute force attacks using specialized hardware. The security of the scrypt function depends mainly on cryptographic algorithms that are part of the memory-hard structure Salsa20/8 and SHA256. This article shows how to customize the scrypt function with new cryptographic algorithms such as ChaCha20 and SHA-3 to improve the security of the custom scrypt function, thereby combining with the AES algorithm to apply for encryption and decrypt stored data with reasonable time. The goal of the solution is to create a more secure commented key derivation function in near-equivalent performance, in acceptable time, to secure the stored data. The results are proven based on the comparison of theoretical and experimental basis in C/C++ programming language. The results of analysis and experiment show that the solution is good to apply data storage security in practice.

GIẢI PHÁP TÙY BIẾN HÀM DẪN XUẤT KHÓA SCRYPT NÂNG CAO ĐỘ AN TOÀN CHO BẢO MẬT DỮ LIỆU LƯU TRỮ

Nguyễn Văn Nghị*, Đỗ Quang Trung, Vũ Bá Linh
Học viện Kỹ thuật Mật mã, 141 Chiến Thắng, Tân Triều, Thanh Trì, Hà Nội

THÔNG TIN BÀI BÁO

Ngày nhận bài: 31/10/2022
Ngày hoàn thiện: 26/12/2022
Ngày đăng: 26/12/2022

TỪ KHÓA

Scrypt
ChaCha20/8
Salsa20/8
SHA-3
AES-CBC

TÓM TẮT

Scrypt là hàm dẫn xuất khóa dựa trên mật khẩu được sử dụng trong nhiều ứng dụng bảo mật dữ liệu lưu trữ. Scrypt dựa trên cấu trúc memory-hard, có khả năng chống lại tấn công vét cạn sử dụng phần cứng chuyên dụng. Sự an toàn của hàm scrypt phụ thuộc chủ yếu vào các thuật toán mật mã thành phần bên trong cấu trúc memory-hard là Salsa20/8 và SHA256. Bài viết này trình bày cách thức tùy biến hàm scrypt với các thuật toán mật mã mới như ChaCha20/8 và SHA-3 nhằm nâng cao độ an toàn cho hàm scrypt tùy biến, từ đó kết hợp với thuật toán AES để ứng dụng trong mã hóa, giải mã dữ liệu lưu trữ với thời gian hợp lý. Mục tiêu của giải pháp là tạo ra hàm dẫn xuất khóa được nhận xét về độ an toàn cao hơn trong hiệu suất gần tương đương, thời gian chấp nhận được để bảo mật dữ liệu lưu trữ. Các kết quả được chứng minh dựa trên việc so sánh cơ sở lý thuyết và thực nghiệm bằng ngôn ngữ lập trình C/C++. Các kết quả phân tích và thực nghiệm cho thấy giải pháp là tốt để áp dụng bảo mật lưu trữ dữ liệu trên thực tế.

DOI: https://doi.org/10.34238/tnu-jst.6836

* Corresponding author. *Email: nghinv@actvn.edu.vn.com*

1. Giới thiệu

Bảo vệ dữ liệu lưu trữ là một những vấn đề quan trọng trong đảm bảo an toàn thông tin và giải pháp hiệu quả để bảo vệ dữ liệu trong trường hợp này chính là sử dụng mật mã để mã hóa dữ liệu lưu trữ. Để thuận tiện cho người dùng và đảm bảo sự an toàn, các hàm dẫn xuất khóa được sử dụng để dẫn xuất khóa mật mã, sử dụng để mã hóa, giải mã, từ các giá trị mật khẩu có entropy thấp, do người dùng tự đặt. Yêu cầu đặt ra đối với các hàm dẫn xuất khóa là phải mạnh để chống lại các tấn công [1], [2] để tìm ra khóa mật mã được sử dụng để mã hóa, giải mã. Một số hàm dẫn xuất khóa phổ biến hiện nay có thể kể đến là scrypt [1] – [3], PBKDF2 [4], Bcrypt...

Scrypt được giới thiệu bởi Colin Percivak vào năm 2009 [2] và chuẩn hóa năm 2016 [1]. Đây là hàm dẫn xuất khóa dựa trên mật khẩu mới nhất và được đánh giá có độ an toàn cao hơn so với các hàm dẫn xuất khóa công bố trước đó như PBKDF2 [4] hoặc Bcrypt. Scrypt được sử dụng phổ biến để sinh khóa mật mã trong các ứng dụng bảo mật hiện nay. Tuy nhiên, các thành phần mật mã được sử dụng trong cấu trúc của scrypt là Salsa20/8, SHA256 [1] – [3] đều đã có những phiên bản mới là ChaCha20 [5] – [10] và SHA-3 [11], [12] được chứng minh là an toàn hơn. Do đó, tùy biến bằng cách thay thế các thành phần mật mã bên trong scrypt, cụ thể là thay thế Salsa20 và SHA256 thành ChaCha20 và SHA-3 là khả thi và có thể nâng cao độ an toàn cho hàm scrypt, khi đó, hàm scrypt tùy biến được sử dụng trong bảo mật dữ liệu lưu trữ với thuật toán mã hóa khóa đối xứng AES (Advanced Encryption Standard) sẽ làm tăng độ an toàn cho dữ liệu được lưu trữ.

Xuất phát từ ý tưởng trên, bài viết này chúng tôi đề xuất cách thức tùy biến hàm scrypt và ứng dụng hàm scrypt tùy biến trong bảo mật dữ liệu lưu trữ với mã khối AES chế độ xích khối mã (Cipher Block Chaining - CBC). Mục tiêu của giải pháp này nhằm làm tăng độ an toàn cho dữ liệu lưu trữ trong khi vẫn đạt được hiệu suất tốt, khả thi với các tham số scrypt được khuyến cáo vào năm 2017 [13]. Các mục tiêu được thực hiện dựa trên việc so sánh, phân tích các kết quả đã được công bố và thực nghiệm bằng ngôn ngữ lập trình C/C++.

Bài viết được bố cục thành mục chính: sau phần 1 giới thiệu, phần 2 mô tả kiến trúc hàm scrypt, phần 3 trình bày cách thức tùy biến nâng cao độ an toàn hàm scrypt, phần 4 đề xuất thực nghiệmứng dụng hàm scrypt tùy biến trong bảo mật dữ liệu lưu trữ, cuối cùng là phần kết luận và tài liệu tham khảo được tham chiếu trong bài viết.

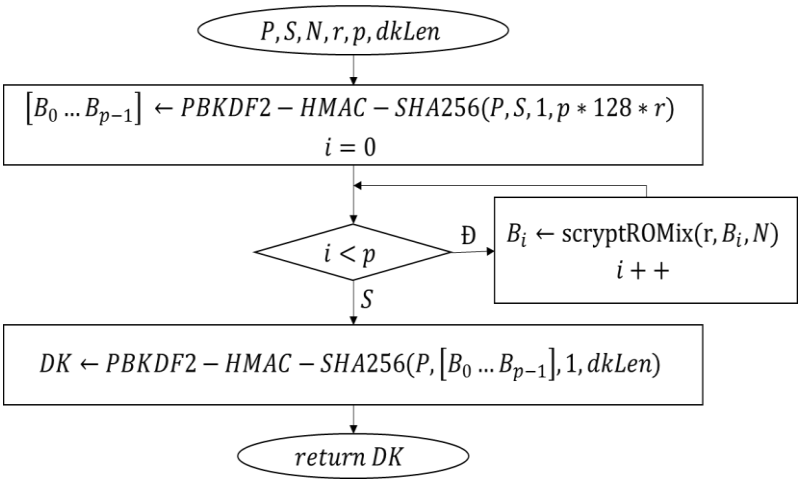
2. Kiến trúc hàm scrypt

Hàm scrypt sử dụng các tham số sau [1] – [3]:

- P: một mật khẩu được người dùng lựa chọn.
- $Salt$: một chuỗi ký tự, thường được tạo duy nhất và giả ngẫu nhiên, giúp chống lại tấn công bảng cầu vồng lên hàm băm.
- r: tham số kích thước khối, có tác dụng tinh chỉnh kích thước và hiệu suất đọc bộ nhớ tuần tự, thường sử dụng là 8.
- N: tham số chi phí của CPU/bộ nhớ, là lũy thừa của 2, lớn hơn 1 và nhỏ hơn $2\text{^}(128 * r/8)$.
- p: tham số song song, là một số nguyên dương thỏa mãn $p \leq ((2\text{^}32 - 1) * hLen)/MFLen$.
- $dkLen$: chiều dài đầu ra dự kiến bằng byte của khóa dẫn xuất, là một số nguyên dương thỏa mãn $dkLen \leq (2\text{^}32 - 1) * hLen$.
- $hLen$: độ dài byte của hàm băm trong hàm PBKDF2 (là 32 đối với SHA256)
- $MFLen$: độ dài byte của đầu ra hàm BlockMix, được xác định là $r * 128$ trong RFC7914.

Hàm scrypt được thiết kế với mô hình hàm memory-hard tuần tự nhằm chống lại tấn công vét cạn sử dụng phần cứng song song [2]. Trong hình 1, hàm scryptROMix đóng vai trò là hàm memory-hard tuần tự và sẽ được mô tả chi tiết ở phần 2.1 của mục này. Hàm scrypt là sự kết hợp của hàm memory-hard tuần tự với hàm giả ngẫu nhiên được lựa chọn là PBKDF2 – HMAC – SHA256, sẽ được trình bày phần 2.3 của phần này. Thuật toán scrypt có đầu vào [1], [2] là các tham số: $P, S, N, r, p, dkLen$; đầu ra của thuật toán là khóa dẫn xuất, có độ dài $dkLen$ byte.

Hình 1 mô tả sơ đồ khối kiến trúc hàm dẫn xuất khóa dựa trên mật khẩu scrypt.



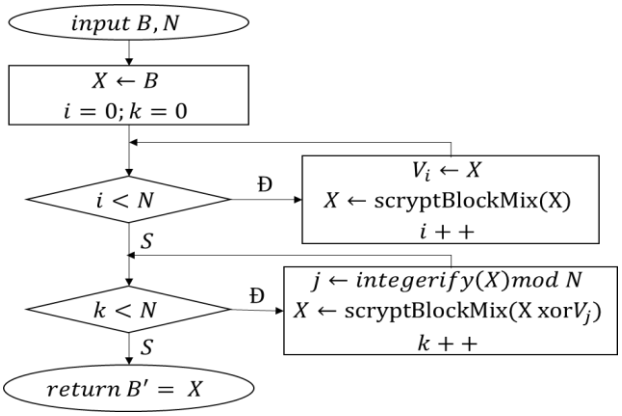
Hình 1. *Sơ đồ kiến trúc hàm dẫn xuất khóa scrypt*

Thuật toán PBKDF2 được sử dụng để tạo ra p khối có độ dài $MFLen$ từ P, S được cung cấp và số lần lặp là 1. Các khối này được trộn một cách độc lập bằng cách sử dụng hàm memory-hard tuần tự scryptROMix. Đầu ra cuối cùng được tạo ra bằng cách áp dụng PBKDF2 và sử dụng các khối đầu ra của hàm scryptROMix làm salt và số lần lặp là 1.

Các tham số N, r và p có thể điều chỉnh theo dung lượng bộ nhớ và năng lực tính toán của bộ vi xử lý của máy tính người dùng. Theo khuyến nghị với giá trị $r = 8$ và $p = 1$ sẽ đem lại khả năng xử lý tốt, nhưng với sự tăng dung lượng bộ nhớ và khả năng tính toán song song hóa của CPU các máy tính ngày nay thì việc tăng các giá trị r và p sẽ cho hiệu suất tốt hơn [2].

2.1. scryptROMix

Hàm memory-hard tuần tự trong scrypt được sử dụng gọi là scryptROMix. Hàm này tính toán một lượng lớn các giá trị ngẫu nhiên và sau đó chúng được lưu trữ một cách ngẫu nhiên trong RAM. Điều này giúp chống lại các tấn công vét cạn bằng các mạch phần cứng song song.



Hình 2. *Sơ đồ hàm memory-hard tuần tự scryptROMix*

Thuật toán scryptROMix trong scrypt được mô tả như sau:

Đầu vào:

- B – Byte vector đầu vào có độ dài $128 * r$ byte.
- N – Tham số chi phí CPU/Memory, là lũy thừa của 2, lớn hơn 1 và nhỏ hơn $2\text{^}(128 * r/8)$.

Đầu ra: B' – Byte vector đầu ra có độ dài $128 * r$ byte.

Quá trình thực thi được mô tả bằng sơ đồ thuật toán hình 2.

Thuật toán scryptBlockMix được sử dụng trong thuật toán scryptROMix đóng vai trò như một hàm băm mà có các thuộc tính đáp ứng yêu cầu để hàm ROMix trở thành hàm memory-hard tuần tự. Thuật toán scryptBlockMix được miêu tả ở phần 2.2 của mục này.

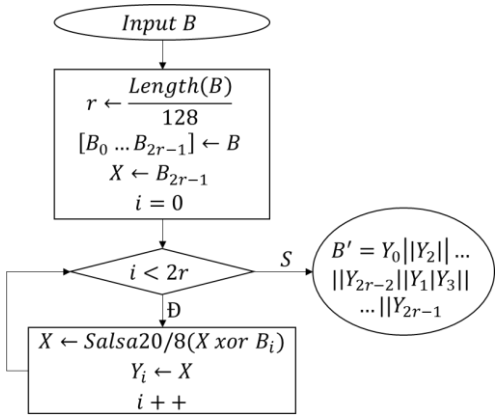
2.2. scryptBlockMix

Thuật toán scryptBlockMix của scrypt là thuật toán BlockMix của hàm Memory-Hard tuần tự với hàm băm H là hàm Salsa20/8 Core [1].

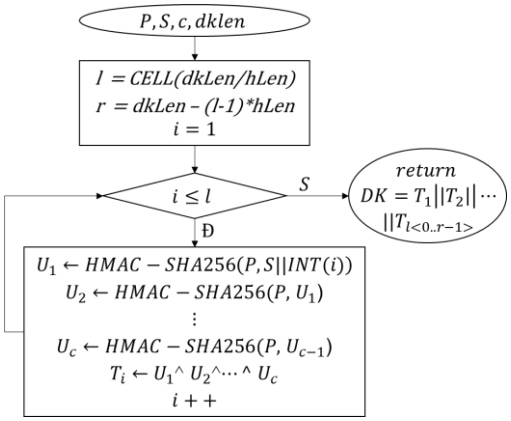
Đầu vào: $B_0 \| B_1 \| \ldots \| B_{2r-1}$– chuỗi byte đầu vào (kích thước $128 * r$ byte), được coi là $2 * r$ khối 64-byte. Với mỗi thành phần của B là một khối 64-byte.

Đầu ra: $B'_0 \| B'_1 \| \ldots \| B'_{2r-1}$.

Chuỗi đầu vào B được chia là $2r$ khối 64 byte. Sau đó, thực hiện vòng lặp $2r$ lần sử dụng hàm Salsa20/8 Core, đóng vai trò như một hàm băm. Kết quả đầu ra được sắp xếp theo thứ tự các khối chẵn đến các khối lẻ. Quá trình thực thi BlockMix được miêu tả ở hình 3.



Hình 3. *Sơ đồ thuật toán hàm scryptBlockMix*



Hình 4. *Sơ đồ thuật toán PBKDF2-HMAC-SHA256*

2.3. PBKDF2-HMAC-SHA256

Trong scrypt, hàm PBKDF2 được sử dụng với hàm HMAC-SHA256 [1]-[4]. Khi đó thuật toán của hàm PBKDF2 được mô tả như sau [4]:

Đầu vào:

- P – Mật khẩu, một chuỗi byte.
- S – Salt, một chuỗi byte.
- c – Số lần lặp, một số nguyên.
- $dkLen$ – độ dài byte của khóa dẫn xuất, một số nguyên lớn nhất là $(2\text{^}32 - 1) * hLen$.

Đầu ra:

DK – khóa dẫn xuất, một chuỗi $dkLen$ – byte.

Thuật toán sẽ tính toán l số khối có độ dài $hLen$ byte và r là số byte trong khối cuối cùng (chú ý $CELL(x)$ là hàm "ceiling" kết quả trả về là giá trị nhỏ nhất lớn hơn hoặc bằng x). Sau đó, phép lặp l vòng được thực hiện với hàm giả ngẫu nhiên là HMAC – SHA256. Cuối cùng, ghép các khối và khóa dẫn xuất là $dkLen$ byte đầu tiên. Quá trình thực thi thuật toán được thể hiện trong hình 4.

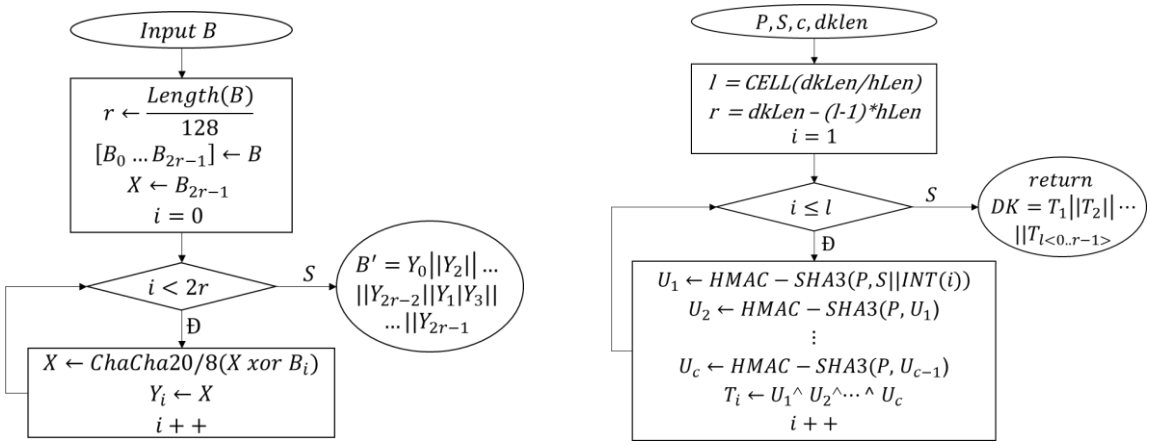
3. Tùy biến hàm scrypt nâng cao độ an toàn

3.1. Cách thức tùy biến hàm scrypt

Theo kiến trúc hàm scrypt đã được trình bày trong phần 2, các thuật toán mật mã sử dụng là mã dòng Salsa20/8 Core, trong hàm ROMix, và HMAC-SHA256 trong hàm PBKDF2. Do hàm

scrypt được thiết kế với mô hình hàm memory-hard tuần tự để chống lại tấn công vét cạn sử dụng phần cứng song song nên hai thuật toán mật mã này đóng vai trò quan trọng quyết định sự an toàn của scrypt trước các tấn công phân tích mã phổ biến như tấn công tuyến tính, lượng sai và tuyến tính kết hợp lượng sai [1]-[4]. Hiện nay, phiên bản mới của thuật toán mã dòng Salsa và hàm băm SHA-256 tương ứng là ChaCha [5] và SHA-3 [11] đã được công bố với độ an toàn tốt hơn rất nhiều (chúng tôi sẽ so sánh độ an toàn giữa Salsa và ChaCha, giữa SHA-2 và SHA-3 ở phần đánh giá độ an toàn của hàm scrypt tùy biến so với phiên bản gốc).

Để nâng cao sự an toàn cho scrypt, chúng tôi đã tiến hành tùy biến lại hàm scrypt bằng cách thay thế mã dòng Salsa bằng mã dòng ChaCha và thay thế hàm băm SHA-256 bằng SHA-3.



(a) *Sơ đồ thuật toán BlockMix với ChaCha20/8* **(b)** *Sơ đồ thuật toán PBKDF2 với hàm SHA3*

Hình 5. *Cách tùy biến hàm BlockMix và FBKDF2 của scrypt*

ChaCha là mã dòng được phát triển từ Salsa bởi cùng nhà mật mã học Daniel Bernstein. ChaCha20 hoạt động với 32-bit đầu vào, 256-bit khóa và bộ đếm 32-bit và tạo ra một chuỗi các khối khóa 512-bit. Điểm khác nhau giữa biến thể ChaCha20 và Salsa20 là ở việc thiết kế sắp xếp bảng trạng thái và 4 phép Quarter-Round [5], [6], [9], [10]. Điều này làm tăng độ khuếch tán trên mỗi vòng của ChaCha trong khi vẫn đạt được hiệu suất tương đương Salsa [5], [6], [9], [10].

Tương tự Salsa20/8, ChaCha20/8 là phiên bản thuật toán ChaCha20 với việc giảm số vòng lặp từ 20 vòng xuống 8 vòng. Việc thay thế Salsa20/8 bằng Chacha20/8 trong khối BlockMix được thực hiện dễ dàng theo hình 5a bởi vì hai thuật toán có chung cấu trúc và kích thước đầu vào và đầu ra.

Hàm SHA-256 được sử dụng trong hàm HMAC-SHA256 để làm hàm giả ngẫu nhiên trong hàm dẫn xuất khóa PBKDF2. Tiến hành thay thế SHA-256 thành SHA-3 trong hàm dẫn xuất khóa PBKDF2 theo hình 5b.

3.2. Đánh giá hàm scrypt tùy biến

3.2.1. Phân tích sự an toàn hàm scrypt tùy biến

Độ an toàn của hàm scrypt được chứng minh an toàn dưới góc nhìn của mô hình memory-hard tuần tự, phụ thuộc chính vào hàm Salsa20/8 trong BlockMix, và SHA256 trong hàm BPKDF2-HMAC-SHA256. Do đó, quá trình đánh giá sự an toàn hàm scrypt đã tùy biến với phiên bản ban đầu chính là đi phân tích sự an toàn giữa hai phiên bản mã dòng Salsa20/Chacha20 và hai phiên bản hàm băm SHA-2/SHA-3.

- So sánh sự an toàn giữa ChaCha20 và Salsa20

Bảng 1 so sánh sự an toàn giữa Salsa20 và ChaCha20 theo các tấn công lượng sai [5] – [9]. Từ bảng 1, độ phức tạp về thời gian và lượng dữ liệu dùng để tấn công lượng sai lên ChaCha20 là cao hơn so với Salsa20. Điều đó có thể thấy được ChaCha20 là an toàn hơn so với Salsa20.

Bảng 1. *Kết quả tấn công lượng sai lên Salsa20 và ChaCha20* [9]

Thuật toán	Loại tấn công	Vòng/ Độ dài khóa	Độ phức tạp	
			Thời gian	Dữ liệu
Salsa20	Tấn công lượng sai của Aumasson	7/256	2^151	2^26
		8/256	2^251	2^31
		7/128	2^111	2^21
	Tấn công mở rộng tấn công lượng sai của Aumasson	7/256	2^148	2^24
		8/256	2^250	2^27
		7/128	2^109	2^19
ChaCha20	Tấn công lượng sai của Aumasson	6/256	2^139	2^30
		7/256	2^248	2^27
		6/128	2^107	2^30
	Tấn công mở rộng tấn công lượng sai của Aumasson	6/256	2^136	2^28
		7/256	2^246.5	2^27
		6/128	2^105	2^28

Ngoài ra, đối với một số tấn công phổ biến khác đối với Chacha20 như bảng 2 thì hầu hết là không khả thi hoặc không tìm thấy [9]. Đối với Salsa20 thì chưa có công bố nào liên quan các tấn công này. Các phân tích này khẳng định rằng ChaCha20 là an toàn hơn so với Salsa20.

Bảng 2. *Đánh giá các tấn công lên mã dòng ChaCha20* [9]

Thuật toán	Tấn công	Đánh giá
ChaCha20	Thám lượng sai	Không tìm được
	Thám tuyến tính	Không tìm được
	Tấn công phân biệt	Không tìm thấy
	Phân tích ước lượng và định đoạt	Không tìm thấy
	Tấn công đánh đổi thời gian bộ nhớ dữ liệu	Không khả thi
	Tấn công đại số	Không tìm thấy
	Tấn công quá trình khởi tạo	Không tìm thấy
	Phân tích năng lượng vi sai	Không khả thi
	Phân tích năng lượng đơn	Không khả thi
	Tấn công Cache Timing	Không tìm thấy
	Phân tích tiêm lỗi	Không khả thi

- So sánh sự an toàn của SHA-3 và SHA-2

Hàm băm SHA-2 sử dụng kiến trúc Merkle-Damgard trong thiết kế, trong khi đó SHA-3 sử dụng kiến trúc Sponge với phép hoán vị Keccak. Kiến trúc Sponge của SHA-3 đem lại độ an toàn cao hơn so với SHA-2 [11], [12], cụ thể mô tả trong bảng 3.

Bảng 3. *Các tham số an toàn cho SHA-2 và SHA-3*

Thuật toán		Kích thước đầu ra	An toàn kháng tấn công va chạm (bit)	An toàn kháng tấn công mở rộng độ dài (bit)
SHA-2	SHA-224	224	112	32
	SHA-256	256	128	0
	SHA-384	384	192	128 ($\leq$ 384)
	SHA-512	512	256	0
	SHA-512/224	224	112	288
	SHA-512/256	256	128	256
SHA-3	SHA3-224	224	112	448
	SHA3-256	256	128	512
	SHA3-384	384	192	768
	SHA3-512	512	256	1024

SHA-3 và SHA-2 có cùng thông số để chống lại các tấn công va chạm nhưng SHA-3 lại có thông số cao hơn khi xét đến tham số chống lại tấn công mở rộng độ dài.

Hàm dẫn xuất khóa scrypt được chứng minh là an toàn, chống lại tấn công vét cạn, theo mô hình memory-hard tuần tự, mô hình này được chứng minh an toàn theo mô hình tiên chi ngẫu nhiên cho hàm băm. Theo tài liệu RFC [1], độ toàn hàm scrypt được giả định rằng hàm BlockMix với Salsa20/8 Core không có bất kỳ "lối tắt" nào để cho phép thực hiện lặp lại một cách dễ dàng mô hình tiên tri ngẫu nhiên (điều này sẽ làm cho scrypt không bị bất kỳ một tấn công nào mạnh hơn các thuật toán chung để tính ROMix). Từ các kết luận ở trên, ChaCha an toàn hơn Salsa và SHA-3 an toàn hơn SHA-2 đây là luận điểm đúng đắn để khẳng định rằng hàm scrypt tùy biến là an toàn hơn so với phiên bản gốc.

3.2.2. Đánh giá về hiệu suất hàm scrypt tùy biến

Để đánh giá hiệu suất tùy biến, chúng tôi sẽ lần lượt thực hiện gọi các hàm scrypt tùy biến (trên cùng thiết bị phần cứng tại cùng thời điểm) với các tham số được lựa chọn lần lượt như sau:

- Test-1: $N = 4096, r = 8, p = 1$ (bộ nhớ 4Mb)
- Test-2: $N = 16384, r = 8, p = 1$ (bộ nhớ 16Mb)
- Test-3: $N = 1048576, r = 8, p = 1$ (bộ nhớ 1Gb)

Kết quả được thể hiện ở Bảng 4.

Bảng 4. *Bảng so sánh hiệu suất thực thi giữa hàm scrypt gốc và hàm scrypt tùy biến*

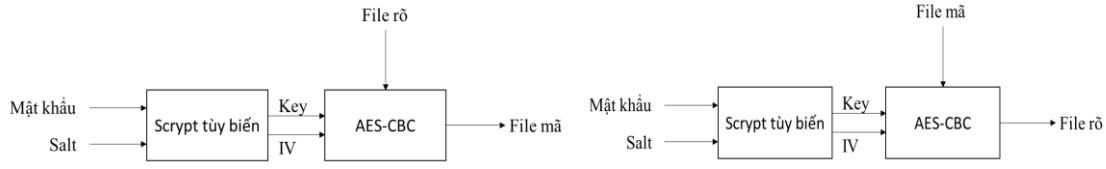
Thuật toán	Test-1 (ticks)	Test-2 (ticks)	Test-3 (ticks)
scrypt[SHA-2-256,Salsa20/8]	10202	44483	3240613
scrypt[SHA-2-256,ChaCha20/8]	13162	65796	3706145
scrypt[SHA-3-256,Salsa20/8]	14261	52375	3519960
scrypt[SHA-3-256,ChaCha20/8]	14085	54166	3656830

Trong đó: ticks là đơn vị đo thời gian bên trong hệ thống của máy tính, tương ứng với tốc độ xung nhịp của mỗi loại CPU. Mỗi hệ điều hành có cách đếm các ticks riêng để cập nhật ngày, giờ.

Từ bảng thống kê, thấy rằng hàm scrypt tùy biến có tốc độ xấp xỉ tốc độ của hàm scrypt gốc.

4. Ứng dụng hàm scrypt tùy biến trong bảo mật dữ liệu lưu trữ

Để ứng dụng hàm dẫn xuất khóa scrypt tùy biến và trong bảo mật dữ liệu lưu trữ, chúng tôi đề xuất giải pháp sử dụng hàm scrypt tùy biến để tạo một khóa dẫn xuất có độ dài 384 bit, khóa dẫn xuất này được tách thành khóa mã hóa Key 256 bit và 128 bit vector khởi tạo IV của thuật toán mã hóa AES-CBC để tiến mã hóa file dữ liệu rõ. Sơ đồ giải pháp được thể hiện như hình 6a.



(a) **(b)**

Hình 6. *Sơ đồ giải pháp ứng dụng scrypt trong bảo mật dữ liệu lưu trữ*

Quá trình giải mã là tương tự, việc sinh khóa giải mã (256 bit) và IV (128 bit) từ hàm dẫn xuất khóa dựa trên mật khẩu bằng hàm scrypt tùy biến với các cách thức tương tự như ở quá trình mã hóa. Quá trình giải mã được thể hiện như hình 6b.

Quá trình cài đặt giải pháp được thực hiện bằng ngôn ngữ C, C++ với bộ công cụ lập trình Qt, trên hệ điều hành Windows. Thuật toán scrypt tùy biến được sử dụng là các thuật toán mật mã tùy biến là SHA-3-256 và ChaCha20/8. Thuật toán AES-CBC được sử dụng bằng cách gọi đến thư viện OpenSSL. Quá trình cài đặt chương trình mã hóa, giải mã với scypt tùy biến và AES-

CBC thành công, các tham số được lựa chọn cho hàm dẫn xuất khóa scrypt tùy biến là: $N = 1048576, r = 8, p = 1$ [13]. Quá trình mã hóa, giải mã các file dữ liệu lưu trữ trên hệ điều hành Windows 10 với nền tảng phần cứng là chip Intel® Core i3-4160 3.6GHz, Ram 16GB. Hiệu năng quá trình mã hóa/giải mã được đưa ra trong bảng 5 (dựa trên số liệu trung bình cộng 10 lần thực nghiệm mã hóa/giải mã tập tin) như sau:

Bảng 5. *Thời gian thực hiện mã hóa và giải mã dữ liệu lưu trữ bằng giải pháp đã được đề xuất*

Dung lượng tập tin	Mã hóa (giây)	Giải mã (giây)
1kB	2,05	2,05
1m	2,05	2,05
5m	2,1	2,1
10m	2,2	2,21
100m	3,61	3,68

Các kết quả thực nghiệm trong bảng 5 cho thấy, giải pháp đề xuất là phù hợp để mã hóa, giải mã các dữ liệu lưu trữ, đặc biệt là các tài liệu, văn bản điện tử lưu trữ.

5. Kết luận

Bài viết này đã đề xuất cách thức tùy biến hàm scrypt bằng cách thay thế Salsa20/8 thành ChaCha20/8, SHA256 thành SHA-3. Có nhiều bằng chứng để tin rằng, cách tùy biến này làm cho hàm scrypt tùy biến an toàn hơn so với hàm scrypt gốc trước các tấn công lên các thành phần mật mã. Chúng tôi cũng đã đề xuất một giải pháp ứng dụng hàm scrypt tùy biến để lưu trữ dữ liệu, với giải pháp đề xuất được cài đặt trên ngôn ngữ C++, cho thấy rằng, hiệu xuất thực thi mã hóa, giải mã là nhanh, phù hợp để có thể sử dụng trong thực tế. Trong bài viết này, chúng tôi chỉ tập trung vào quá trình tùy biến và phân tích sự an toàn dựa trên các tấn công lên các thành phần mật mã mà chưa phân tích các tham số được sử dụng, điều này chưa đánh giá đầy đủ được sự an toàn của hàm scrypt tùy biến khi thay đổi các tham số và chưa tối ưu được thời gian thực thi. Đây sẽ là hướng mà chúng tôi sẽ nghiên cứu trong thời gian tới để có thể tối ưu hàm dẫn xuất khóa scrypt tùy biến cả về mặt thời gian và độ an toàn cần thiết.

TÀI LIỆU THAM KHẢO/ REFERENCES

[1] C. Percival and S. Josefsson, *The scrypt Password-Based Key Derivation Function*, RFC 7914, August 2016.

[2] C. Percival, *Stronger key derivation via sequential memory-hard functions*, University of California, Santa Barbara, 2009.

[3] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, "Scrypt Is Maximally Memory-Hard," *EUROCRYPT 2017, Part III, LNCS 10212*, 2017, pp. 33–62.

[4] B. Kaliski, *PKCS #5: Password-Based Cryptography Specification Version 2.0*, RFC 2898, September 2000.

[5] D. J. Bernstein, *ChaCha, a variant of Salsa20*, The University of Illinois at Chicago, Chicago-USA, 2008.

[6] P. Yadav, I. Gupta, and S. K. Murthy, "Study and analysis of eSTREAM cipher Salsa and ChaCha," *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, 2016, pp. 90-94, doi: 10.1109/ICETECH.2016.7569218.

[7] S. Miyashita, R. Ito, and A. Miyaji, "PNB-based Differential Cryptanalysis of ChaCha Stream Cipher," *Cryptology ePrint Archive*, 2021, Art. no. 1537.

[8] M. Coutinho and T. C. S. Neto, "Improved Linear Approximations to ARX ciphers and attacks against ChaCha." *Cryptology ePrint Archive*, 2021, Art. no.224.

[9] KDDI Research Inc., *Security Analysis of ChaCha20-Poly1305*, CRYPTREC-EX-2601-2016, 2017.

[10] P. McLaren, W. J. Buchanan, G. Russell, and Z. Tan, "Deriving ChaCha20 key streams from targeted memory analysis," *Journal of Information Security and Applications*, vol. 48, 2019, Art. no. 102372.

[11] National Institute of Standards and Technology (NIST), *FIPS 202: Sha-3 Standard: Permutation-Based Hash And Extendable Output Functions*, August 2015.

[12] N. Bagheri, N. Ghaedi, and S. K. Sanadhya, "Differential Fault Analysis of SHA-3," in *Progress in Cryptology -- INDOCRYPT 2015, Lecture Notes in Computer Science*, A. Biryukov and V Goyal (eds), vol. 9462, pp. 253–269, 2015, doi: 10.1007/978-3-319-26617-6_14.

[13] S. Nakov, *Practical Cryptography for Developers*, SoftUni, 2018.