BỘ KHOA HỌC VÀ CÔNG NGHỆ
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÁO CÁO
ĐỒ ÁN MÔN HỌC

ĐỀ TÀI: PHÂN TÍCH VÀ TẤN CÔNG THUẬT TOÁN MÃ HÓA DES BẰNG KỸ THUẬT BRUTE-FORCE, TỪ ĐIỂN

Môn học: Mật mã học cơ sở

Giảng viên hướng dẫn: Th.S Nguyễn Hoàng Thành

Thực hiện bởi nhóm sinh viên, bao gồm:

1.	Hồ Văn Tâm	N22DCAT046	<Trưởng nhóm>
2.	Trần Thị Ánh Nguyệt	N22DCAT040	<Thành viên>
3.	Trương Quỳnh Như	N22DCAT041	<Thành viên>
4.	Nguyễn Xuân Quang	N22DCAT043	<Thành viên>
5.	Trần Phúc Tiến	N22DCAT057	<Thành viên>

TP.HCM, tháng 6/2025

MỤC LỤC

DANH SÁCH HÌNH, BẢNG

TÓM TẮT

Thuật toán mã hóa DES (Data Encryption Standard) là một trong những chuẩn mã hóa đối xứng được sử dụng phổ biến trong nhiều thập kỷ trước khi bị thay thế bởi các thuật toán mạnh hơn như AES. Mặc dù có vai trò lịch sử quan trọng, DES hiện nay được xem là không còn đủ an toàn do độ dài khóa ngắn (56 bit), dễ bị tấn công bởi các phương pháp dò tìm khóa.

Đề tài này tập trung phân tích chi tiết cấu trúc và nguyên lý hoạt động của thuật toán DES, đồng thời triển khai và đánh giá hai kỹ thuật tấn công phổ biến nhằm phá vỡ tính bảo mật của nó: tấn công brute-force (thử toàn bộ không gian khóa) và tấn công từ điển (dựa trên tập hợp các khóa hoặc mật khẩu phổ biến). Bằng cách xây dựng mô hình tấn công thực nghiệm, đề tài này giúp làm rõ tính hiệu quả và giới hạn của từng phương pháp, đồng thời chỉ ra mức độ dễ bị tổn thương của DES trong thực tế.

Kết quả nghiên cứu cho thấy rằng với sự phát triển của phần cứng hiện đại, DES hoàn toàn có thể bị bẻ khóa trong một khoảng thời gian ngắn, đặc biệt khi kết hợp với các kỹ thuật tối ưu hóa hoặc sử dụng GPU. Từ đó, đề tài nhấn mạnh sự cần thiết trong việc chuyển đổi sang các kỹ thuật mã hóa mạnh hơn như AES, đồng thời nâng cao nhận thức về tầm quan trọng của việc lựa chọn thuật toán và quản lý khóa trong bảo mật thông tin.

CHƯƠNG I. TỔNG QUAN

1. Giới thiệu đề tài.

Trong bối cảnh bùng nổ thông tin và sự phụ thuộc ngày càng tăng của xã hội vào các hệ thống kỹ thuật số, an toàn thông tin trở thành một trong những mối quan tâm hàng đầu. Mật mã học, với vai trò là xương sống của an ninh mạng, đã và đang phát triển không ngừng nhằm bảo vệ tính bảo mật, toàn vẹn và xác thực dữ liệu. Trong lịch sử phát triển của mật mã hiện đại, thuật toán mã hóa DES (Data Encryption Standard) chiếm một vị trí quan trọng. DES từng là một trong những thuật toán mã hóa đối xứng được sử dụng rộng rãi nhất trên toàn cầu, trở thành tiêu chuẩn cho nhiều ứng dụng bảo mật từ các giao dịch tài chính đến thông tin liên lạc.

Mặc dù ngày nay DES không còn được xem là an toàn trước các phương pháp tấn công hiện đại do sự phát triển vượt bậc của công nghệ tính toán và các kỹ thuật phá mã tinh vi, việc phân tích các lỗ hổng của nó vẫn cung cấp những bài học quý giá về thiết kế hệ thống mật mã và đánh giá rủi ro bảo mật. Đặc biệt, việc tìm hiểu sâu về các kỹ thuật tấn công truyền thống như Brute-force và tấn công từ điển giúp sinh viên và các nhà nghiên cứu có cái nhìn thực tế về cách thức các hệ thống mật mã bị khai thác. Brute-force là phương pháp tấn công thử toàn bộ không gian khóa, trong khi tấn công từ điển dựa trên việc thử các khóa từ một danh sách các khoá có khả năng cao hoặc phổ biến. Nắm vững các kỹ thuật này không chỉ củng cố kiến thức lý thuyết mà còn trang bị kỹ năng thực hành cần thiết để xây dựng và bảo vệ các hệ thống an toàn hơn trong tương lai.

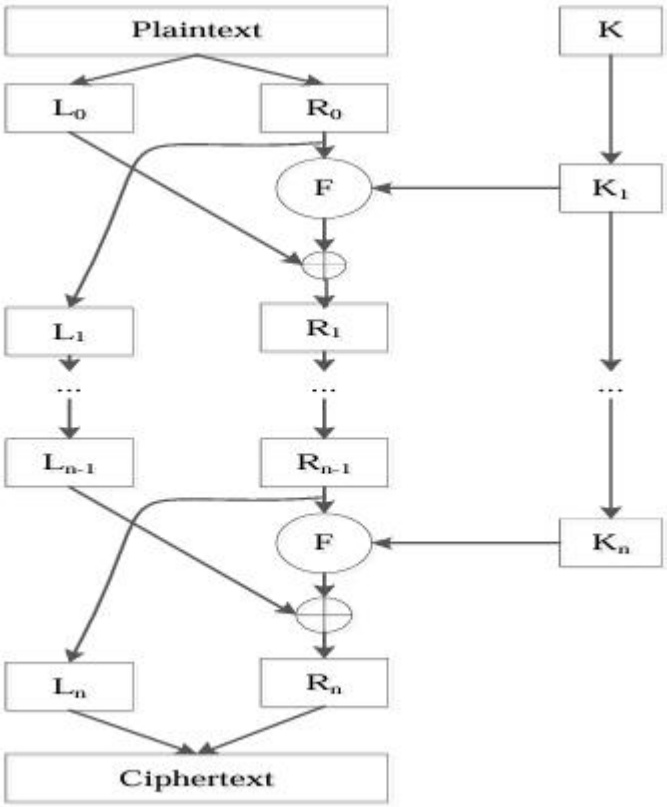
2. Cơ sở lý thuyết.

2.1. Thuật toán DES (Data Encryption Standard)

2.1.1. Giới thiệu mô hình Feistel:

Mô hình Feistel là một dạng tiếp cận khác so với mạng SP. Mô hình so Host Feistel đề xuất, cũng là sự kết hợp các phép thay thế và hoán vị. Đầu vào của thuật toán mã hóa là khối văn bản gốc có chiều dài 2w bit và một khóa K. Khối thông điệp gốc được chia thành 2 nửa L_0 và R_0. Hai nửa này đi qua n vòng xử lý rồi tổ hợp để tạo ra khối bản mã hóa. Mỗi vòng i có đầu vào L_{i-1} và R_{i-1} lấy từ vòng trước đó, giống như

1 khóa con K_i, lấy từ tổng thể K. Nói chung, những khóa con K_i khác với khóa K và các khóa khác.



Hình: Mô hình Feistel

Tất cả các vòng có cấu trúc tương tự nhau. Một thay thế được thực hiện trên một nửa trái của dữ liệu. Điều này được thực hiện bằng cách áp dụng một hàm vòng F cho nửa bên phải của dữ liệu và sau đó lấy exclusive-OR của các đầu ra của hàm đó và một nửa còn lại của dữ liệu. Tất cả các vòng có cùng một cấu trúc chung cho mỗi vòng, nhưng là tham số của mỗi vòng là khóa con K_i. Sau sự thay thế, một hoán vị được thực hiện mà bao gồm việc trao đổi hai nửa của dữ liệu. Việc thực hiện chính xác của một mạng Feistel phụ thuộc vào sự lựa chọn của các tham số và đặc điểm thiết kế sau:

Kích thước khối: khối kích thước lớn hơn có nghĩa là bảo mật cao hơn (tất cả những thứ khác bằng nhau), nhưng giảm đi tốc độ mã hóa/ giải mã của một thuật toan cho trước. Theo truyền thống, một kích thước khối 64 bit được coi là một sự cân bằng hợp lý và được phổ cập trong thiết kế gần mã hóa khối.

Kích thước khóa: kích thước lớn hơn có nghĩa là an toàn hơn, nhưng có thể làm giảm tốc độ mã hóa/giải mã. Việc bảo mật cao hơn đạt được bằng cách tấn công brute-force tốt hơn. Kích thước khóa của 64 bit hoặc ít hơn bây giờ nhiều người xem là không đủ an toàn, và 128 bit đã trở thành một kích thước khóa thông thường.

Số vòng: Bản chất của mã hóa Feistel là một vòng duy nhất cung cấp bảo mật không đầy đủ nhưng nhiều vòng cung cấp sẽ tăng cường bảo mật hơn. Một kích thước điển hình là 16 vòng.

Thuật toán sinh khóa phụ: phức tạp hơn trong thuật toán này nên dẫn đến khó khăn lớn hơn của phân tích mật mã.

Hàm vòng: Một lần nữa, phức tạp hơn thường có ý nghĩa chống phân tích mật mã tốt hơn. Có hai quan tâm khác trong thiết kế của một mã hóa Feistel.

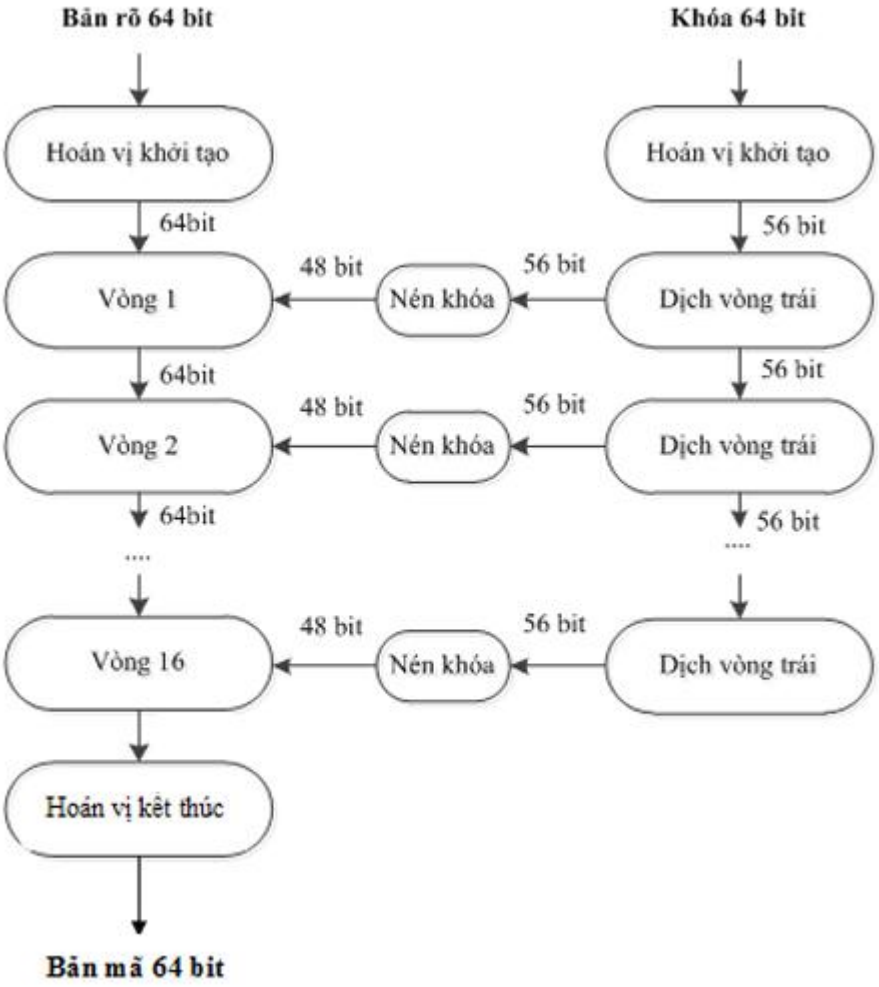
Tốc độ phần mềm mã hóa/giải mã: Trong nhiều trường hợp, mã hóa được nhúng trong các ứng dụng hoặc các chức năng tiện ích. Vì vậy tốc độ thực hiện của thuật toán sẽ trở thành một mối quan tâm.

Dễ dàng trong việc phân tích: Mặc dù muốn làm cho thuật toán khó khăn nhất có thể để chống lại các phân tích mã phát triển ở một mức độ cao hơn.

2.1.2. Giới thiệu chung về DES:

Vào năm 1973, khi lĩnh vực máy tính ngày càng phát triển, nhu cầu ứng dụng bảo mật vào các mục đích dân sự được đặt ra. Lúc này Cục tiêu chuẩn quốc gia Hoa Kỳ kêu gọi các công ty Mỹ thiết lập một chuẩn mã hóa quốc gia. Mã hóa Lucifer của công ty IBM được chọn và sau một vài sửa đổi của cơ quan an ninh Hoa Kỳ, mã hóa Lucifer đã trở thành mã hóa tiêu chuẩn DES (Data Encryption Standard). Qua quá trình sử dụng mã DES đã chứng tỏ độ an toàn cao được ứng dụng rộng rãi trong những năm 1970 và 1980.

DES là dạng mã hóa khối với dữ liệu vào kích thước 64 bit và khóa 64 bit, trong đó thực sử dụng 56 bit (còn gọi là kích thước hiệu dụng của khóa) và 8 bit dùng cho kiểm tra chẵn lẻ. Một ưu điểm của DES là sử dụng chung một giải thuật cho cả khâu mã hóa và khâu giải mã. DES là mã thuộc hệ mã Feistel gồm 16 vòng, ngoài ra DES có thêm một hoán vị khởi tạo trước khi vòng 1 và một hoán vị kết thúc sau 16 vòng. Mỗi vòng của DES dùng khóa con có kích thước 48 bit được trích ra từ khóa chính.



Hình 1: Sơ đồ tổng quát quá trình mã hóa DES

Sơ đồ mã hóa DES gồm 3 phần, phần thứ nhất là các hoán vị khởi tạo và hoán vị kết thúc. Phần thứ hai là các vòng Feistel. Phần thứ ba là thuật toán sinh khóa con.

2.1.3. Các thành phần trong DES:

2.1.3.1. Hoán vị khởi tạo và hoán vị kết thúc:

Giá trị đầu vào là khối M có kích thước 64 bit *m1,m2,m3,..m64.* Hoán vị khởi tạo sẽ thực hiện biến đổi các giá trị các bit theo nguyên tắc sau: ($m_1,m_2,m_3...,m_{64}$ -> $m_{58},m_{50},m_{42},..,m_7$).

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Hình 2: Bảng hoán vị khởi tạo IP

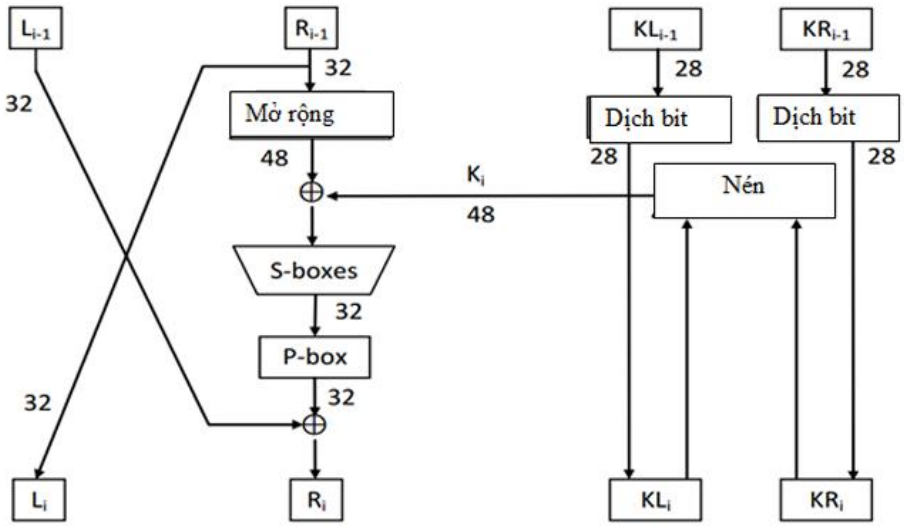
Hoán vị kết thúc hoán đổi các bit theo nguyên tắc sau: ($c_1,c_2,c_3,..,c_{64}$ -> $c_{40},c_8,c_{48},..,c_{25}$)

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Hình 3: Bảng hoán vị kết thúc IP^{-1}

Hoán vị kết thúc chính là hoán vị nghịch đảo của hoán vị khởi tạo. Đối với known plaintext hay chosen-plaintext attack, hoán vị khởi tạo và hoán vị kết thúc không có ý nghĩa bảo mật, sự tồn tại của hai hoán vị trên được nhận định do yếu tố lịch sử.

2.1.3.2. Các vòng của DES:

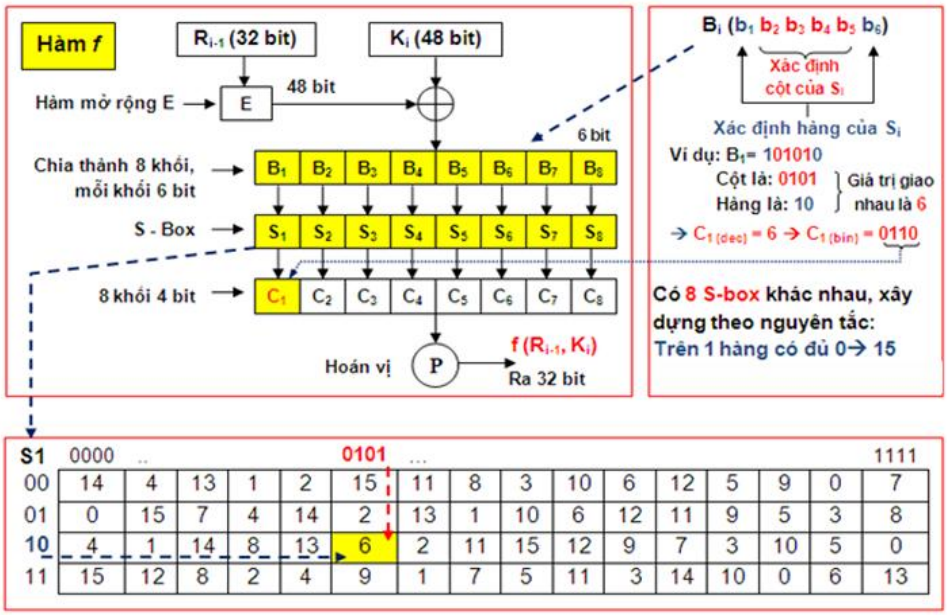


Hình 4: Mô hình vòng lặp DES

Trong DES biểu thức hàm F là:

$$F(R_{i-1}, K_i) = P-box(s-boxes(Expand(R_{i-1}) \oplus K_i))$$

Trong đó hàm Expand (Mở rộng) vừa mở rộng vừa mở rộng R_{i-1} từ 32 bit lên 48 bit. Hàm S-boxes lại nén 48 bit xuống cong 32 bit. Hàm P-box là một hoán vị 32 bit.



Hình 5: Mô tả hàm F trong DES

Hàm Expand: hàm thực hiện hoán vị và mở rộng từ 332bit lên 48 bit theo nguyên tắc sau: $r_1,r_2,..,r_{32}$ -> $r_{32}r_1,r_2,..,r_1$

E					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

48 bit

Hình 6: Bảng Expand DES

Hàm S-boxes của DES biến đổi một số 48 bit thành một số 32 bit. Hàm S-boxes được chia thành 8 hàm S-box con, mỗi hàm biến đổi số 6 bit thành số 4 bit.

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	1110 0100 1101 0001 0010 1111 1011 1000 0011 1010 0110 1100 0101 1001 0000 0111
	01	0000 1111 0111 0100 1110 0010 1101 0001 1010 0110 1100 1011 1001 0101 0011 1000
	10	0100 0001 1110 1000 1101 0110 0010 1011 1111 1100 1001 0111 0011 1010 0101 0000
	11	1111 1100 1000 0010 0100 1001 0001 0111 0101 1011 0011 1110 1010 0000 0110 1101

Hình 7: Bảng S-box 1

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	1111 0001 1000 1110 0110 1011 0011 0100 1001 0111 0010 1101 1100 0000 0101 1010
	01	0011 1101 0100 0111 1111 0010 1000 1110 1100 0000 0001 1010 0110 1001 1011 0101
	10	0000 1110 0111 1011 1010 0100 1101 0001 0101 1000 1100 0110 1001 0011 0010 1111
	11	1101 0 1010 0001 0011 1111 0100 0010 1011 0110 0111 1100 0000 0101 1110 1001

Hình 8: Bảng S-box 2

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	1010 0000 1001 1110 0110 0011 1111 0101 0001 1101 1100 0111 1011 0100 0010 1000
	01	1101 0111 0000 1001 0011 0100 0110 1010 0010 1000 0101 1110 1100 1011 1111 0001
	10	1101 0110 0100 1001 1000 1111 0011 0000 1011 0001 0010 1100 0101 1010 1110 0111
	11	0001 1010 1101 0000 0110 1001 1000 0111 0100 1111 1110 0011 1011 0101 0010 1100

Hình 9: Bảng S-box 3

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	0111 1101 1110 0011 0000 0110 1001 1010 0001 0010 1000 0101 1011 1100 0100 1111
	01	1101 1000 1011 0101 0110 1111 0000 0011 0100 0111 0010 1100 0001 1010 1110 1001
	10	1010 0110 1001 0000 1100 1011 0111 1101 1111 0001 0011 1110 0101 0010 1000 0100
	11	0011 1111 0000 0110 1010 0001 1101 1000 1001 0100 0101 1011 1100 0111 0010 1110

Hình 10: Bảng S-box 4

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	0010 1100 0100 0001 0111 1010 1011 0110 1000 0101 0011 1111 1101 0000 1110 1001
	01	1110 1011 0010 1100 0100 0111 1101 0001 0101 0000 1111 1010 0011 1001 1000 0110
	10	0100 0010 0001 1011 1010 1101 0111 1000 1111 1001 1100 0101 0110 0011 0000 1110
	11	1011 1000 1100 0111 0001 1110 0010 1101 0110 1111 0000 1001 1010 0100 0101 0011

Hình 11: Bảng S-box 5

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	1100 0001 1010 1111 1001 0010 0110 1000 0000 1101 0011 0100 1110 0111 0101 1011
	01	1010 1111 0100 0010 0111 1100 1001 0101 0110 0001 1101 1110 0000 1011 0011 1000
	10	1001 1110 1111 0101 0010 1000 1100 0011 0111 0000 0100 1010 0001 1101 1011 0110
	11	0100 0011 0010 1100 1001 0101 1111 1010 1011 1110 0001 0111 0110 0000 1000 1101

Hình 12: Bảng S-box 6

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	0100 1011 0010 1110 1111 0000 1000 1101 0011 1100 1001 0111 0101 1010 0110 0001
	01	1101 0000 1011 0111 0100 1001 0001 1010 1110 0011 0101 1100 0010 1111 1000 0110
	10	0001 0100 1011 1101 1100 0011 0111 1110 1010 1111 0110 1000 0000 0101 1001 0010
	11	0110 1011 1101 1000 0001 0100 1010 0111 1001 0101 0000 1111 1110 0010 0011 1100

Hình 13: Bảng S-box 7

		$b_1b_2b_3b_4$
		0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b_0b_5	00	1101 0010 1000 0100 0110 1111 1011 0001 1010 1001 0011 1110 0101 0000 1100 0111
	01	0001 1111 1101 1000 1010 0011 0111 0100 1100 0101 0110 1011 0000 1110 1001 0010
	10	0111 1011 0100 0001 1001 1100 1110 0010 0000 0110 1010 1101 1111 0011 0101 1000
	11	0010 0001 1110 0111 0100 1010 1000 1101 1111 1100 1001 0000 0011 0101 0110 1011

Hình 14: Bảng S-box 8

Có thể thấy, mỗi hàm S-box con là một phép thay thế Substitution. Các hàm S-box con không khả nghịch, do đó hàm S-boxes cũng không không khả nghịch. Sự phức tạp này của S-boxes là yếu tố chính làm cho DES có độ an toàn cao

P-box cũng thực hiện hoán vị 32 bit đầu vào theo nguyên tắc:

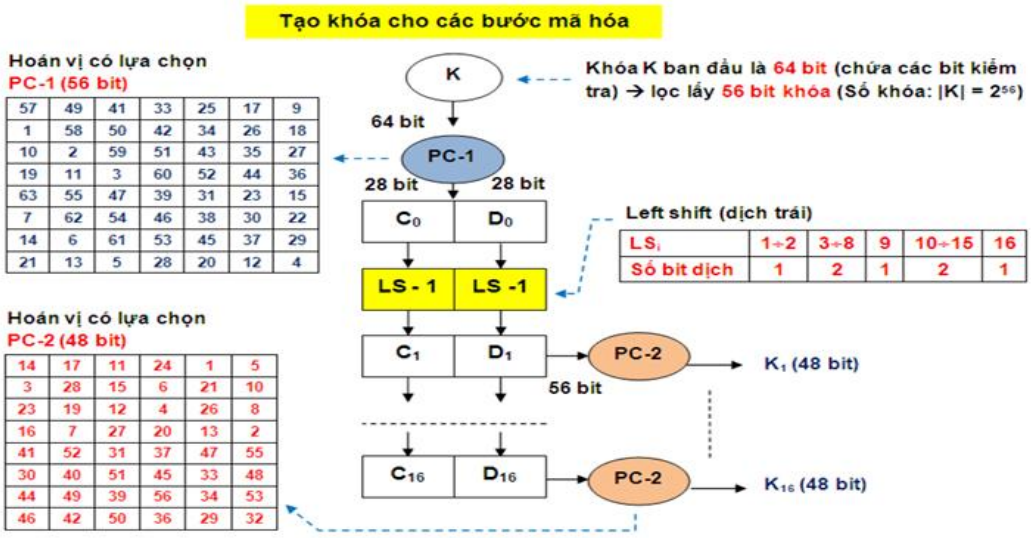
P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Hình 15: Bảng P-box

2.1.3.3. Thuật toán sinh khóa con:

Thuật toán sinh khóa con cho từng vòng mã hóa của DES được thể hiện trong hình dưới. Khóa K 64 bit ban đầu được rút trích và hoán vị thành một khóa 56 bit (tức chỉ siwr dụng 56 bit) theo quy tắc nén khóa. Khóa 56 bit này được chia thành 2 nửa trái và phải KL_0 và KR_0, mỗi nửa có kích thước 28 bit. Tại vòng thứ i (i = 1, 2, ,9, 16) KL_{i-1} và KR_{i-1} được dịch trái 1 bit, còn lại dịch trái 2 bit.

Cuối cùng khóa Ki của mỗi vòng được tạo ra bằng cách hoán vị và nén 56 bit của KL_i và KR_i thành 48 bit



Hình 16: Mô hình các bước tạo khóa của DES

Lưu ý: DES có khóa 64 bit nhưng trên thực tế chỉ 56 bit thực sự dùng để mã hóa và số lượng khóa có thể là khoảng 72 triệu tỷ khóa (2^{56}). Và với không gian khóa như vậy, một cuộc tấn công brute-force là khả thi với các máy tính mạnh mẽ và hệ thống phân tán hiện nay. Vào năm 1998, tổ chức EF xây dựng một máy chuyên dụng gọi là "DES Cracker" với chi phí 250.000 USD. Máy này có thể phá khóa DES trong vài ngày bằng cách thử tất cả các khóa có thể. Với công nghệ hiện đại (như GPU, FPGA hoặc các hệ thống tính toán đám mây), việc thử 2^{56} khóa có thể được thực hiện ngắn hơn thường chỉ vài giờ. Do đó DES không còn được xem là an toàn và đã được thay thế bằng 3DES, AES.

2.1.4. Quy trình mã hóa/ giải mã:

Tạo 16 khóa con

Bước 1: IP - hoán vị ban đầu: Dữ liệu 64 bit được sắp xếp lại thứ tự bit theo bảng IP

Bước 2: Chia dữ liệu: L_0 và R_0 mỗi nửa 32 bit

Bước 3: 16 vòng lặp Feistel: mỗi vòng

$L_i = R_{i-1}$

$R_i = L_i \oplus F(R_{i-1}, K_i)$

Với F là hàm phức hợp bao gồm: Mở rộng (32 -> 48 bit), XOR với khóa con K_i. S-box (48 -> 32 bit), hoán vị P-box

Bước 4: Kết hợp và hoán vị cuối

Lưu ý: Quy trình giải mã cũng dùng lại chính quy trình mã hóa, chỉ khác là sử dụng các khóa con theo thứ tự ngược lại: K_{16} -> K_{15} -> … -> K_1 . Vì DES sử dụng cấu trúc feistel, cho phép đảo ngược quy trình, bất kể hàm thay thế S-boxes hay hoán vị P-box có phức tạp thế nào. Điều này có nghĩa là, nếu ta áp dụng các vòng Feistel với thứ tự khóa ngược lại, ta sẽ hoàn nguyên các biến đổi đã thực hiện trong quá trình mã hóa. Giải mã tương đương với mã hóa ngược, khi giải mã DES áp dụng cùng một thuật

toán mã hóa nhưng sử dụng khóa con theo thứ tự ngược lại. Điều này đảm bảo rằng mỗi vòng giải mã sẽ đảo ngược phép biến đổi của vòng mã hóa tương ứng.

2.1.5. Khóa yếu

Khóa yếu trong thuật toán mã hóa DES là những khóa mà khi sử dụng sẽ làm giảm độ an toàn của quá trình mã hóa. Cụ thể, với một số khóa nhất định, thuật toán DES trở nên dễ bị tấn công hoặc có những tính chất không mong muốn, chẳng hạn:

Mã hóa 2 lần cho cùng một bản rõ sẽ cho ra chính bản rõ đó. Điều này làm vô hiệu hóa hiệu ứng bảo mật khi mã hóa nhiều lần.

Có 3 nhóm khóa yếu làm suy yếu hệ thống:

Khóa yếu: số lượng 4: Khi mã hóa 2 lần cùng với khóa, kết quả là văn bản gốc

Khóa bán yếu: số lượng 6 cặp: Mã hóa bằng khóa này cho kết quả có thể giải mã bằng một khóa khác.

Khóa gần yếu: số lượng 48: Có tính chất giống khóa bán yếu, nhưng ít phổ biến hơn.

VD: 0x0101010101010101,0xFEFEFEFEFEFEFEFE, 0x1F1F1F1F0E0E0E0E , 0xE0E0E0E0F1F1F1F1

=> Các khóa này có cấu trúc đối xứng hoặc lặp lại, khiến quá trình sản sinh khóa con trong DES kém hiệu quả, làm giảm tính ngẫu nhiên cần thiết cho mã hóa an toàn.

2.1.6. Khóa mạnh

Khóa yếu trong thuật toán mã hóa DES là những khóa không có tính chất làm suy yếu quá trình mã hóa, tức là:

Không thuộc nhóm khóa yếu, khóa bán yếu hay khóa gần yếu

Khi sử dụng để mã hóa, chúng sinh ra các khóa con (subkeys) khác nhau cho mỗi vòng, đảm bảo tính ngẫu nhiên cao.

Không có tính chất đối xứng hay lặp lại trong cấu trúc bit. Đảm bảo độ phân tán và nhiễu hóa tốt (diffusion & confusion), đúng theo nguyên lý mã hóa của Shannon.

VD: 0xA1B2C3D4E5F60718 => Đây là một khóa 64-bit (trong đó chỉ có 56 bit có ý nghĩa, cong 8 bit dùng làm parity), không có cấu trúc lặp hoặc đối xứng. Khi đưa vào thuật toán DES, sẽ sinh ra 16 khóa con khác nhau - đủ tính chất của một khóa mạnh.

2.2. Kỹ thuật tấn công Brute-force:

2.2.1. Khái niệm tấn công Brute-force

Tấn công brute force là phương pháp đột nhập vào tài khoản hoặc hệ thống được bảo vệ bằng mật khẩu bằng cách thử mọi tổ hợp ký tự có thể. Tấn công brute force là loại tấn công mạng mà kẻ tấn công sử dụng hệ thống tự động để đoán tổ hợp tên người dùng và mật khẩu chính xác để truy cập vào hệ thống hoặc trang web. Loại tấn công này thường được sử dụng để truy cập vào trang web, tài khoản hoặc các hệ thống an toàn khác. Bằng cách thử nhiều tổ hợp tên người dùng và mật khẩu khác nhau, kẻ tấn công cuối cùng có thể đoán được tổ hợp đúng, cho phép chúng truy cập vào hệ thống. Điều quan trọng đối với các doanh nghiệp là phải cập nhật các biện pháp bảo mật của mình để bảo vệ chống lại các cuộc tấn công brute force.

2.2.2. Cách thức hoạt động

Trong một cuộc tấn công brute force, tin tặc sẽ sử dụng phần mềm tinh vi để thử một cách có hệ thống hàng nghìn hoặc thậm chí hàng triệu tổ hợp ký tự cho đến khi tìm ra mật khẩu chính xác. Nếu tin tặc thực hiện thành công một cuộc tấn công brute force, chúng có thể truy cập vào hệ thống và bất kỳ dữ liệu nào có trong đó.

Sau đây là cách thức hoạt động của một cuộc tấn công brute force:

1. Đầu tiên, tin tặc xác định hệ thống mục tiêu. Đây có thể là trang web, mạng, tài khoản quản trị, tài khoản người dùng hoặc hệ thống khác được bảo vệ bằng mật khẩu.
2. Để bắt đầu cuộc tấn công, tin tặc sử dụng một bot nhập mật khẩu dự đoán vào một trường biểu mẫu trên hệ thống mục tiêu.
3. Bot sẽ đợi phản hồi từ trang web sau mỗi lần đoán để xác định xem dự đoán đó có đúng hay không.
4. Nếu bot đoán được mật khẩu thành công, tin tặc có thể truy cập vào hệ thống và mọi dữ liệu có trong đó.
5. Nếu bot không đoán được mật khẩu, tin tặc có thể sử dụng các phương pháp mạnh hơn như tấn công từ điển, bảng cầu vồng, tấn công kết hợp, v.v.

2.2.3. Hiệu quả

Kích thước không gian khóa càn lớn, brute force càng lâu, độ phức tạp là O(2^n).
Tốc độ xử lý phần cứng: GPU/CPU mạnh hơn sẽ thử được nhiều khóa mỗi giây
Số lượng thiết bị song song: tăng hiệu suất đáng kể nếu dùng phân tán hoặc hệ thống song song.
Tính hợp lệ bản rõ: Nếu có thể xác định bản rõ đúng nhanh (dựa vào định dạng), quá trình kết thúc sớm hơn.

2.3. Kỹ thuật tấn công từ điển:

2.3.1. Khái niệm tấn công từ điển

Tấn công từ điển là một loại tấn công mạng trong đó kẻ tấn công độc hại sử dụng danh sách các từ và cụm từ để truy cập vào hệ thống. Tấn công từ điển sử dụng một chuỗi từ hoặc cụm từ có mục tiêu để cố gắng truy cập vào hệ thống an toàn. Nó được sử dụng để truy cập trái phép vào tài khoản người dùng hoặc giải mã dữ liệu nhạy cảm. Cuộc tấn công này hoạt động bằng cách khai thác thực tế là nhiều người sử dụng các từ hoặc cụm từ phổ biến làm mật khẩu hoặc sử dụng các biến thể của cùng một mật khẩu. Tấn công từ điển thường được sử dụng kết hợp với các loại tấn công khác, chẳng hạn như tấn công bằng cách dùng vũ lực hoặc tấn công bảng cầu vồng, để làm cho chúng thành công hơn.

2.3.2. Cách thức hoạt động

Trong một cuộc tấn công từ điển, tin tặc sẽ sử dụng một tệp từ điển để thử một cách có hệ thống hàng nghìn hoặc thậm chí hàng triệu mật khẩu thường dùng được liệt kê trong tệp đó cho đến khi tìm ra mật khẩu chính xác. Nếu tin tặc thực hiện thành công một cuộc tấn công từ điển, chúng có thể truy cập vào hệ thống và bất kỳ dữ liệu nào có trong đó.

Sau đây là cách thức hoạt động của một cuộc tấn công từ điển:

1. Kẻ tấn công tạo hoặc tìm thấy danh sách các mật khẩu hoặc từ phổ biến và đưa chúng vào tệp từ điển. Các tệp như vậy có thể dễ dàng tìm thấy trực tuyến.

2. Để bắt đầu cuộc tấn công, tin tặc sử dụng một bot nhập mật khẩu dự đoán vào một trường biểu mẫu trên hệ thống mục tiêu.
3. Bot sẽ đợi phản hồi từ trang web sau mỗi lần đoán để xác định xem dự đoán đó có đúng hay không.
4. Nếu mật khẩu dự đoán không chính xác, bot sẽ chuyển sang mật khẩu tiếp theo trong tệp từ điển.
5. Bot sẽ tiếp tục quá trình này cho đến khi tìm được mật khẩu đúng.
6. Khi bot tìm thấy tên người dùng và mật khẩu chính xác, tin tặc có thể truy cập vào trang web và dữ liệu của trang web.

2.3.3. Hiệu quả

Tốc độ: nhanh hơn brute-force do chỉ thử các khóa có khả năng có xác suất cao
Hiệu suất: Cao nếu người dùng nhập mật khẩu yếu hoặc phổ biến
Chi phí tài nguyên: Tương đối thấp (tùy vào độ dài từ điển và tốc độ xử lý)
Xác suất thành công: Cao nếu danh sách từ điển tốt, thấp nếu mật khẩu ngẫu nhiên phức tạp.

2.4. So sánh kỹ thuật tấn công Brute-force và tấn công từ điển

Tấn công brute force thử tất cả các tổ hợp ký tự có thể cho đến khi một tổ hợp có hiệu quả, trong khi tấn công từ điển thu hẹp các tổ hợp thành danh sách các mật khẩu phổ biến hoặc đã biết. Danh sách này có thể bao gồm các mật khẩu phổ biến được nhiều người sử dụng cũng như thông tin đăng nhập bị rò rỉ. Các mật khẩu thường được sắp xếp theo mức độ phổ biến, nghĩa là những mật khẩu phổ biến nhất sẽ được kiểm tra trước. Do đó, tấn công từ điển thường tốn ít thời gian hơn so với tấn công brute force hoàn toàn, nhưng cũng kém hiệu quả hơn đối với các mật khẩu duy nhất, chưa bị rò rỉ.

Tấn công Brute Force	**Tấn công từ điển**

Cố gắng đoán mật khẩu bằng cách thử một cách có hệ thống mọi tổ hợp ký tự có thể có	Cố gắng đoán mật khẩu bằng cách thử một cách có hệ thống mọi từ có thể có trong từ điển
Chậm và tốn nhiều tính toán	Nhanh nhưng bị giới hạn bởi các từ trong từ điển
Có thể đoán mật khẩu có độ dài bất kỳ	Thường giới hạn ở mật khẩu có độ dài hợp lý

CHƯƠNG II. THIẾT KẾ HỆ THỐNG

1. Xây dựng thuật toán mã hóa/giải mã DES

1.1. Thuật toán mã hóa DES

1.1.1. Khởi tạo:

Sử dụng hàm def __init__(self) để khởi tạo các bảng cần cho quá trình mã hóa như IP, IP^{-1}, PC-1, PC-2, S-boxes, P-box, E.

```
def __init__(self):
    self.IP = [58, 50, 42, 34, 26, 18, 10, 2,
               60, 52, 44, 36, 28, 20, 12, 4,
               62, 54, 46, 38, 30, 22, 14, 6,
               64, 56, 48, 40, 32, 24, 16, 8,
               57, 49, 41, 33, 25, 17, 9, 1,
               59, 51, 43, 35, 27, 19, 11, 3,
               61, 53, 45, 37, 29, 21, 13, 5,
               63, 55, 47, 39, 31, 23, 15, 7]

    self.IP_1 = [40, 8, 48, 16, 56, 24, 64, 32,
                 39, 7, 47, 15, 55, 23, 63, 31,
                 38, 6, 46, 14, 54, 22, 62, 30,
                 37, 5, 45, 13, 53, 21, 61, 29,
                 36, 4, 44, 12, 52, 20, 60, 28,
                 35, 3, 43, 11, 51, 19, 59, 27,
                 34, 2, 42, 10, 50, 18, 58, 26,
                 33, 1, 41, 9, 49, 17, 57, 25]

    self.E = [32, 1, 2, 3, 4, 5,
              4, 5, 6, 7, 8, 9,
              8, 9, 10, 11, 12, 13,
              12, 13, 14, 15, 16, 17,
              16, 17, 18, 19, 20, 21,
              20, 21, 22, 23, 24, 25,
              24, 25, 26, 27, 28, 29,
              28, 29, 30, 31, 32, 1]
```

Hình 17: Hàm khởi tạo __init__(self)

```
self.PC_1 = [57, 49, 41, 33, 25, 17, 9,
             1, 58, 50, 42, 34, 26, 18,
             10, 2, 59, 51, 43, 35, 27,
             19, 11, 3, 60, 52, 44, 36,
             63, 55, 47, 39, 31, 23, 15,
             7, 62, 54, 46, 38, 30, 22,
             14, 6, 61, 53, 45, 37, 29,
             21, 13, 5, 28, 20, 12, 4]

self.PC_2 = [14, 17, 11, 24, 1, 5,
             3, 28, 15, 6, 21, 10,
             23, 19, 12, 4, 26, 8,
             16, 7, 27, 20, 13, 2,
             41, 52, 31, 37, 47, 55,
             30, 40, 51, 45, 33, 48,
             44, 49, 39, 56, 34, 53,
             46, 42, 50, 36, 29, 32]

self.shifts = [1, 1, 2, 2, 2, 2, 2, 2,
               1, 2, 2, 2, 2, 2, 2, 1]
```

1.1.2. Hoán vị ban đầu:

Dữ liệu plaintext được chuyển thành ma trận nhị phân 8x8, sau đó hoán vị theo bảng IP

```
def permute_matrix(self, matrix, ip_table):
    permuted = []
    for pos in ip_table:
        pos -= 1
        row = pos // 8
        col = pos % 8
        permuted.append(matrix[row][col])
    result_matrix = []
    for i in range(0, 64, 8):
        result_matrix.append(permuted[i:i + 8])
    return result_matrix
```

Hình 18: Hàm chuyển plaintext thành ma trận 8x8

```
matrix = self.bin_string_to_bit_matrix(plaintext_bin)
for row in matrix:
    textProcess.insert(END, str(row) + "\n")
# Sau khi qua bảng IP
matrix_after_IP = self.permute_matrix(matrix, self.IP)
textProcess.insert(END, "Sau IP (ma trận 8x8):\n")
for row in matrix_after_IP:
    textProcess.insert(END, str(row) + "\n")
```

Hình 19: Hoán vị theo bảng IP

1.1.3. Chia thành 2 nửa:

Ma trận 8x8 được chia làm 2 nửa 4x8: L_0 và R_0

```
def split_matrix_L0_R0(self, matrix):
    L0 = matrix[:4]
    R0 = matrix[4:]
    return L0, R0
```

Hình 20: Hàm chia ma trận thành L0 và R0

1.1.4. Sinh khóa con:

Áp dụng hoán vị PC-1 để rút gọn khóa từ 64 bit còn 56 bit

```
# Qua bảng PC-1
key_pc1 = self.permute_matrix_pc1(key_matrix, self.PC_1)
```

```
def permute_matrix_pc1(self, matrix, table):
    permuted = []
    for pos in table:
        pos -= 1
        row = pos // 8
        col = pos % 8
        permuted.append(matrix[row][col])
    return permuted
```

Hình 21: Hàm áp dụng PC-1

Tách thành 2 phần C0 và D0 (mỗi phần 28 bit), sau đó thực hiện phép dịch trái. Nếu đang ở vòng thứ 1, 2, 9, 16 thì thực hiện dịch trái 1 bit, còn nếu ở các vòng còn lại thì thực hiện dịch trái 2 bit.

```
def split_CD(self, bitlist56):
    C_bits = bitlist56[:28]
    D_bits = bitlist56[28:]
    C0 = [C_bits[i:i + 7] for i in range(0, 28, 7)]
    D0 = [D_bits[i:i + 7] for i in range(0, 28, 7)]
    return C0, D0
```

Hình 22: Hàm chia khóa thành 2 nửa C0 và D0

```
def left_shift(self, matrix, n):
    flat_bits = ""
    for row in matrix:
        for bit in row:
            flat_bits += str(bit)
    shifted_bits = flat_bits[n:] + flat_bits[:n]
    shifted_matrix = []
    for i in range(4):
        row = []
        for j in range(7):
            row.append(int(shifted_bits[i * 7 + j]))
        shifted_matrix.append(row)
    return shifted_matrix
```

Hình 23: Hàm dịch bit

Kết hợp C và D ta được 56 bit, sau đó áp dụng bảng PC-2, ta được key 48 bit để thực hiện XOR với R mở rộng.

```
def permute_matrix_pc2(self, C_matrix, D_matrix, table):
    combined = C_matrix + D_matrix
    permuted = []
    for pos in table:
        pos -= 1
        row = pos // 7
        col = pos % 7
        permuted.append(combined[row][col])
    result_matrix = []
    for i in range(0, 48, 6):
        result_matrix.append(permuted[i:i + 6])
    return result_matrix
```

Hình 24:Hàm áp dụng PC-2

1.1.5. Vòng lặp Feistel:

Mở rộng R từ 32 bit thành 48 bit (sử dụng bảng E), sau đó thực hiện XOR với khóa k 48 bit đã có trước đó sau khi áp dụng PC-2

```
# Sau khi mở rộng R
expanded_R = self.expand(R)
textProcess.insert(END, "Expansion R (ma trận 8x6):\n")
for row in expanded_R:
    textProcess.insert(END, str(row) + "\n")
# XOR E(R) với Subkey
xor_result = self.xor_matrices(expanded_R, subkey_matrix)
textProcess.insert(END, "Kết quả sau XOR (ma trận 8x6):\n")
for row in xor_result:
    textProcess.insert(END, str(row) + "\n")
```

Hình 25: Thực hiện mở rộng R và XOR với khóa k

Áp dụng S-box để giảm từ 48 bit xuống còn 32 bit. Hàm S-box có nhiệm vụ chia 48 bit thành 8 hộp, mỗi hộp có 6 bit. Sau khi đi qua hàm này, đầu ra của nó sẽ chỉ còn 8 hộp, mỗi hộp 4 bit (tức là từ 48 bit xuống còn 32 bit)

```
def sbox_substitution(self, matrix8x6):
    output_bits = ""
    for sbox_index in range(8):
        block = matrix8x6[sbox_index]
        row = (block[0] << 1) + block[5]
        column = (block[1] << 3) + (block[2] << 2) + (block[3] << 1) + block[4]
        sbox_value = self.S[sbox_index][row][column]
        sbox_bits = bin(sbox_value)[2:].zfill(4)
        output_bits += sbox_bits
    output_matrix = []
    for i in range(0, 32, 4):
        row = [int(bit) for bit in output_bits[i:i + 4]]
        output_matrix.append(row)
    return output_matrix
```

Hình 26: Hàm xử lý khi qua S-boxes

Sau khi qua S-boxes, tiếp tục cho qua bảng hoán vị P. Lấy kết quả đó, đem đi XOR với L trước đó ta thu được R_{i+1}. Còn $L_{i+1} = R_i$ trước đó.

1.1.5. Hoán vị ngược IP-1:

Đây là bước cuối cùng khi thực hiện mã hóa DES. Tức là sau khi hoàn tất 16 vòng, ghép R và L và áp dụng bảng IP^{-1} để tạo ciphertext.

```
pre_output = self.merge_LR(R, L)
textProcess.insert(END, "Hoán vị cuối cùng (R16|L16) trước IP-1:\n")
for row in pre_output:
    textProcess.insert(END, str(row) + "\n")
cipher_matrix = self.permute_matrix(pre_output, self.IP_1)
textProcess.insert(END, "Ciphertext (sau IP-1, ma trận 8x8):\n")
for row in cipher_matrix:
    textProcess.insert(END, str(row) + "\n")
```

Hình 27: Biến đổi bước cuối mã hóa DES

1.2. Thuật toán giải mã DES

Thuật toán giải mã thực hiện gần như tương tự thuật toán mã hóa, nhưng các khóa con được sử dụng theo thứ tự ngược lại (từ round 16 đến 1). Toàn bộ quá trình giải mã được thực hiện bằng cách đảo ngược các bước trong mã hóa cùng khóa.

```
# Giải mã 16 vòng: dùng subkey đảo ngược
for round_num, i in enumerate(reversed(range(16))):
    textProcess.insert(END, f"Decrypt Round {round_num + 1}:\n")
    expanded_R = self.expand(R)
    textProcess.insert(END, "Expansion R (ma trận 8x6):\n")
    for row in expanded_R:
        textProcess.insert(END, str(row) + "\n")
    xor_result = self.xor_matrices(expanded_R, subkeys[i])
    textProcess.insert(END, "Kết quả sau XOR (ma trận 8x6):\n")
    for row in xor_result:
        textProcess.insert(END, str(row) + "\n")
    sbox_output = self.sbox_substitution(xor_result)
    textProcess.insert(END, "Sau S-box (ma trận 8x4):\n")
    for row in sbox_output:
        textProcess.insert(END, str(row) + "\n")
    p_output = self.permute_P(sbox_output)
    textProcess.insert(END, "Sau Permutation P (ma trận 8x4):\n")
    for row in p_output:
        textProcess.insert(END, str(row) + "\n")
    f_output = self.xor_L_and_f(L, p_output)
    textProcess.insert(END, "Sau khi XOR với L (ma trận 4x8):\n")
    for row in f_output:
        textProcess.insert(END, str(row) + "\n")
    L, R = R, f_output
    merged_LR = self.merge_LR(L, R)
    textProcess.insert(END, "Hợp L và R (ma trận 8x8):\n")
    for row in merged_LR:
        textProcess.insert(END, str(row) + "\n")
```

Hình 28: Giải mã với Subkey đảo ngược

2. Thuật toán Brute-force

Thuật toán Brute-force thực hiện thử tất cả các khóa có thể có trong không gian khóa 2^{64} (DES 64 bit, trong đó 56 bit hiệu dụng), theo thứ tự tăng dần.

Các bước:

1. Nhập ciphertext và plaintext kỳ vọng

```
if len(ciphertext_bytes) % 8 != 0:
    error_msg = "Lỗi: Độ dài ciphertext không chia hết cho 8 byte\n"
    textProcess.insert(END, error_msg)
    textProcess.see(END)
    textProcess.update()
    return "Không tìm được", None, 0
plaintext_bytes = plaintext_expect.encode('utf-8')
pad_len = (8 - (len(plaintext_bytes) % 8)) % 8
if pad_len != 0:
    plaintext_bytes += b' ' * pad_len
if len(plaintext_bytes) != len(ciphertext_bytes):
    error_msg = "Lỗi: Độ dài plaintext và ciphertext không khớp\n"
    textProcess.insert(END, error_msg)
    textProcess.see(END)
    textProcess.update()
    return "Không tìm được", None, 0
```

2. Duyệt từng giá trị khóa từ 0 đến 2^{64} - 1

```
for key_int in range(total_keys):
```

3. Với mỗi khóa:

Giải mã Ciphertext

```
key_matrix = self.bin_string_to_bit_matrix(key_bin)
key_pc1 = self.permute_matrix_pc1(key_matrix, self.PC_1)
C0, D0 = self.split_CD(key_pc1)
C_matrices, D_matrices = self.generate_CD(C0, D0)
subkeys = []
for i in range(16):
    subkey_matrix = self.permute_matrix_pc2(C_matrices[i], D_matrices[i], self.PC_2)
    subkeys.append(subkey_matrix)
all_plaintexts = []
success = True
for c_block, p_block in zip(ciphertext_blocks, plaintext_blocks):
    ciphertext_block_hex = c_block.hex()
    ciphertext_bin = self.hex_to_bin(ciphertext_block_hex)
    matrix = self.bin_string_to_bit_matrix(ciphertext_bin)
    matrix_after_IP = self.permute_matrix(matrix, self.IP)
    L, R = self.split_matrix_L0_R0(matrix_after_IP)
    for i in reversed(range(16)):
        expanded_R = self.expand(R)
        xor_result = self.xor_matrices(expanded_R, subkeys[i])
        sbox_output = self.sbox_substitution(xor_result)
        p_output = self.permute_P(sbox_output)
        f_output = self.xor_L_and_f(L, p_output)
        L, R = R, f_output
    pre_output = self.merge_LR(R, L)
    plaintext_matrix = self.permute_matrix(pre_output, self.IP_1)
    flat_bits = ''.join(str(bit) for row in plaintext_matrix for bit in row)
    plaintext_hex = self.bin_to_hex(flat_bits)
    plaintext_bytes_res = bytes.fromhex(plaintext_hex)
    all_plaintexts.append((plaintext_hex, plaintext_bytes_res))
    if plaintext_bytes_res != p_block:
        success = False
        break
```

So sánh kết quả với plaintext kỳ vọng. Nếu có trùng khớp thì kết thúc và trả về khóa

```
for idx, (hex_val, plain_bytes) in enumerate(all_plaintexts):
    try:
        plain_str = plain_bytes.decode('utf-8').rstrip()
    except:
        plain_str = '<Không thể decode>'
    msg_block = f"  Block {idx + 1}: Plaintext (hex): {hex_val} -> {plain_str}\n"
    textProcess.insert(END, msg_block)
    textProcess.see(END)
    textProcess.update()
if success:
    end_time = time.time()
    msg_found = (
        "========== ĐÃ TÌM THẤY KHÓA ==========\n"
        f"Key (bin): {key_bin}\n"
        f"Key (hex): {key_hex}\n"
```

4. Ghi lại thời gian bắt đầu, kết thúc và số lần thử khóa.

```
f"Thời gian bắt đầu: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(start_time))}\n"
f"Thời gian kết thúc: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(end_time))}\n"
f"Tổng thời gian chạy: {end_time - start_time:.2f} giây\n"
f"Tổng số key đã thử: {key_counter}\n"
```

3. Thuật toán Dictionary attack:

Thuật toán brute-force từ điển sử dụng một danh sách khóa được lưu sẵn trong file dictionary.txt, thường là những chuỗi mật khẩu phổ biến, để tiết kiệm thời gian so với brute-force toàn phần.

Các bước:

1. Nhập ciphertext và plaintext kỳ vọng

```
if len(ciphertext_bytes) % 8 != 0:
    msg = "Lỗi: Độ dài ciphertext không chia hết cho 8 byte\n"
    textProcess.insert(END, msg)
    textProcess.see(END)
    textProcess.update()
    return "Không tìm thấy", None, 0, None, None
plaintext_bytes = plaintext_expect.encode('utf-8')
pad_len = (8 - (len(plaintext_bytes) % 8)) % 8
if pad_len != 0:
    plaintext_bytes += b' ' * pad_len
if len(plaintext_bytes) != len(ciphertext_bytes):
    msg = "Lỗi: Độ dài plaintext và ciphertext không khớp\n"
    textProcess.insert(END, msg)
    textProcess.see(END)
    textProcess.update()
    return "Không tìm thấy", None, 0, None, None
```

2. Đọc file từ điển, lấy danh sách các chuỗi khóa dạng string

```
try:
    with open(dictionary_file, 'r', encoding='utf-8') as f:
        keys = [line.strip() for line in f if line.strip()]
except FileNotFoundError:
    msg = f"Không tìm thấy file: {dictionary_file}\n"
    textProcess.insert(END, msg)
    textProcess.see(END)
    textProcess.update()
    return "Không tìm thấy", None, 0, None, None
```

3. Với mỗi khóa:

Chuyển thành dạng hex đủ 16 kí tự

```
key_bytes = key_str.encode('utf-8')
key_hex = key_bytes.hex().zfill(16)
key_bin = self.hex_to_bin(key_hex).zfill(64)
```

Giải mã ciphertext

```
key_matrix = self.bin_string_to_bit_matrix(key_bin)
key_pc1 = self.permute_matrix_pc1(key_matrix, self.PC_1)
C0, D0 = self.split_CD(key_pc1)
C_matrices, D_matrices = self.generate_CD(C0, D0)
subkeys = []
for i in range(16):
    subkey_matrix = self.permute_matrix_pc2(C_matrices[i], D_matrices[i], self.PC_2)
    subkeys.append(subkey_matrix)
all_plaintexts = []
success = True
for c_block, p_block in zip(ciphertext_blocks, plaintext_blocks):
    ciphertext_block_hex = c_block.hex()
    ciphertext_bin = self.hex_to_bin(ciphertext_block_hex)
    matrix = self.bin_string_to_bit_matrix(ciphertext_bin)
    matrix_after_IP = self.permute_matrix(matrix, self.IP)
    L, R = self.split_matrix_L0_R0(matrix_after_IP)
    for i in reversed(range(16)):
        expanded_R = self.expand(R)
        xor_result = self.xor_matrices(expanded_R, subkeys[i])
        sbox_output = self.sbox_substitution(xor_result)
        p_output = self.permute_P(sbox_output)
        f_output = self.xor_L_and_f(L, p_output)
        L, R = R, f_output
    pre_output = self.merge_LR(R, L)
    plaintext_matrix = self.permute_matrix(pre_output, self.IP_1)
    flat_bits = ''.join(str(bit) for row in plaintext_matrix for bit in row)
    plaintext_hex = self.bin_to_hex(flat_bits)
    plaintext_bytes_res = bytes.fromhex(plaintext_hex)
    all_plaintexts.append((plaintext_hex, plaintext_bytes_res))
    if plaintext_bytes_res != p_block:
        success = False
        break
msg_try = f"Thử key: {key_str} (hex: {key_hex})\n"
```

4. So sánh với plaintext kỳ vọng. Nếu trùng khớp thì kết thúc và trả về khóa.

```
if success:
    end_time = time.time()
    msg_found = (
        "========== ĐÃ TÌM THẤY KHÓA ==========\n"
        f"Key tìm được: {key_str}\n"
        f"Key (hex): {key_hex}\n"
        f"Thời gian bắt đầu: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(start_time))}\n"
        f"Thời gian kết thúc: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(end_time))}\n"
        f"Tổng thời gian chạy: {end_time - start_time:.2f} giây\n"
        f"Tổng số key đã thử: {key_counter}\n"
    )
```

CHƯƠNG III. TRIỂN KHAI HỆ THỐNG

1. Khái quát kịch bản

Ở đồ án này, phương pháp thám mã chính được sử dụng là kiểu vét cạn (brute-force), và được triển khai theo 2 dạng: dạng vét cạn toàn bộ tổ hợp có thể có của khóa và dò theo từ điển. Có hai điều cần phải làm rõ ở đây chính là: một, chúng ta xem khóa của thuật toán DES như là một mật khẩu trong hệ ASCII thay vì bit - điều kiện tiên quyết để có thể thành công dò được khóa bằng phương pháp brute force từ điển; và hai, theo phân loại mật mã học thì đồ án sẽ tập trung vào trình bày cuộc tấn công ciphertext-only attack.

Vì bản chất đây là tấn công brute force nên chúng ta sẽ thừa nhận rằng khóa yếu chính là những khóa thuộc những tổ hợp sẽ được thử trước nhất chứ không bàn về những tính chất toán học của một khóa khiến nó yếu, và khóa mạnh có thể là một khóa bất kỳ, miễn là thuật toán brute force phải dò đến bit thứ 64 của khóa đó, bởi vì độ phức tạp thời gian khi dò tất cả những tổ hợp từ bit thứ i chính là một hàm số mũ 2^i (ở đây độ dài của plain text sẽ không được đưa vào phân tích).

Kịch bản tấn công sẽ được chia làm 2 phần chính: duyệt khóa mạnh và duyệt khóa yếu theo từng phương pháp brute force: tuần tự (vét cạn toàn bộ tổ hợp) và dò theo từ điển.

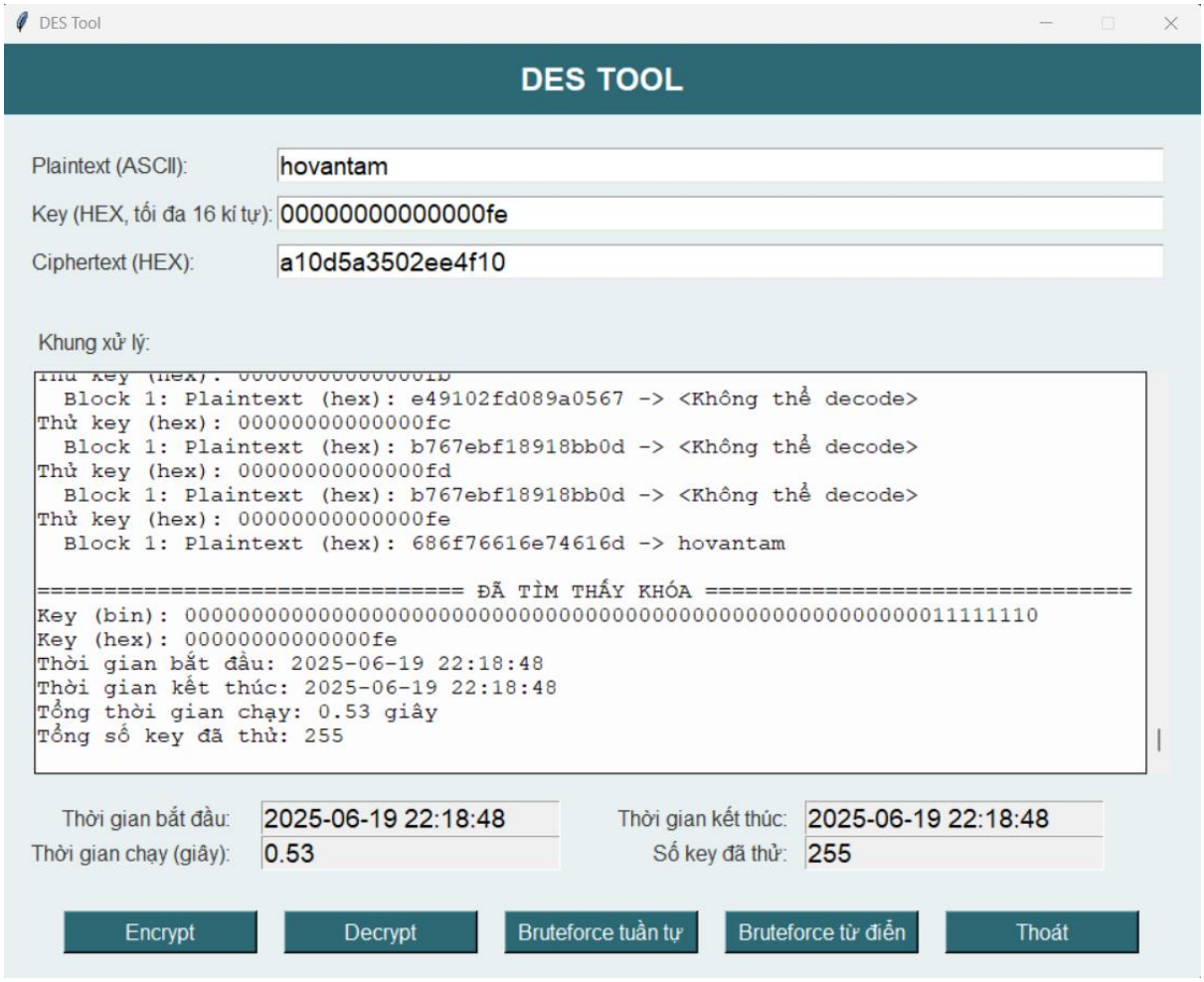
Plain text xuyên suốt kịch bản sẽ là “hovantam”, với độ dài quy ước bằng đúng 64 bit - số bit mà 1 khối DES xử lý.

2. Khóa yếu

2.1. Brute force tuần tự

2.1.1. Test 1

- p = hovantam, c = $a10d5a3502ee4f10_H$, k = $00000000000000ff_H$



Hình 29: Brute-force (khóa yếu): Kết quả test1

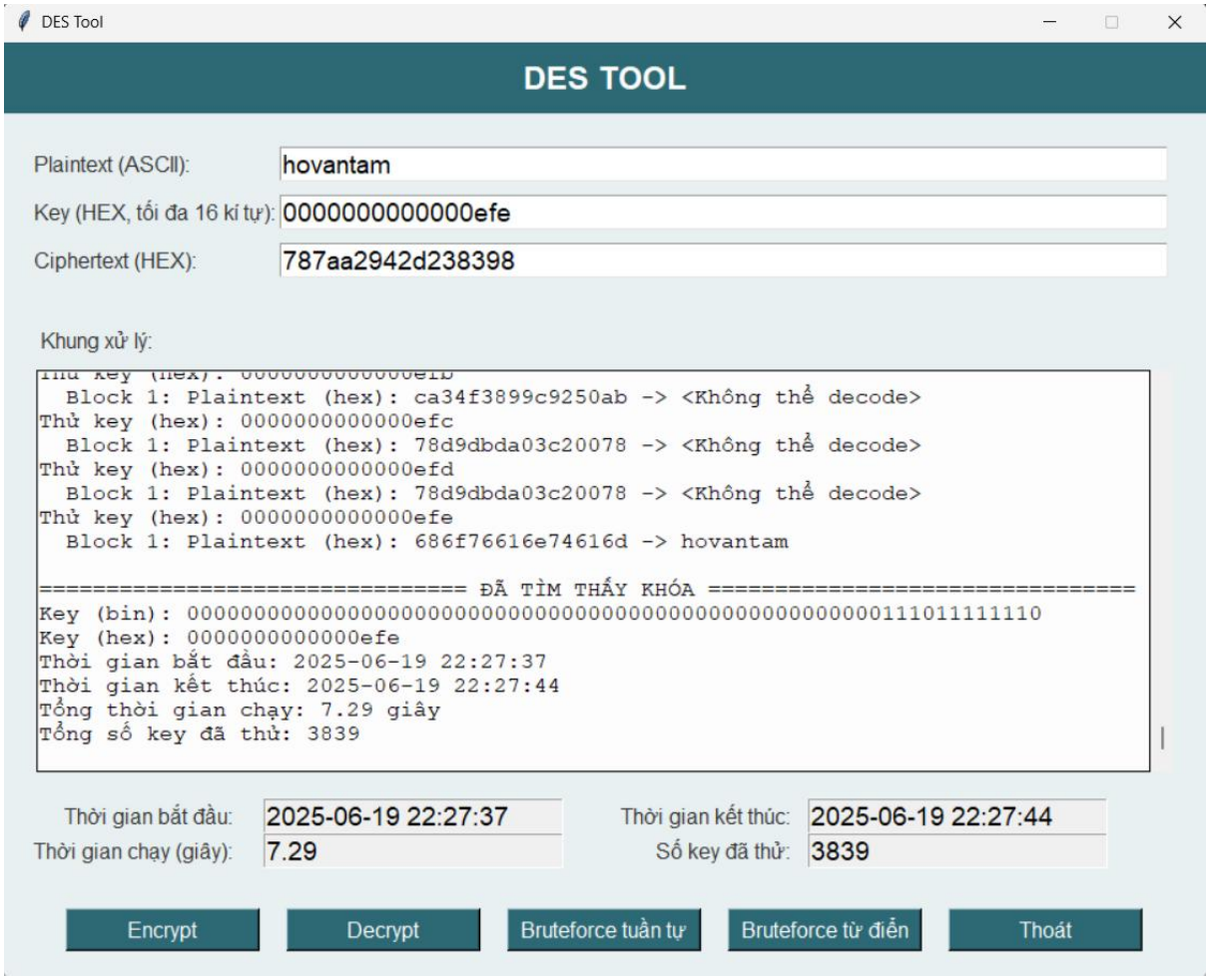
Thời gian duyệt: 0,53 giây

Thời gian duyệt trung bình cho 1 key: 481 giây

Khóa tìm thấy: $00000000000000fe_H$

2.1.2. Test 2

- p = hovantam, c = $787aa2942d238398_H$, k = $0000000000000fff_H$



Hình 30: Brute-force (khóa yếu): Kết quả test2

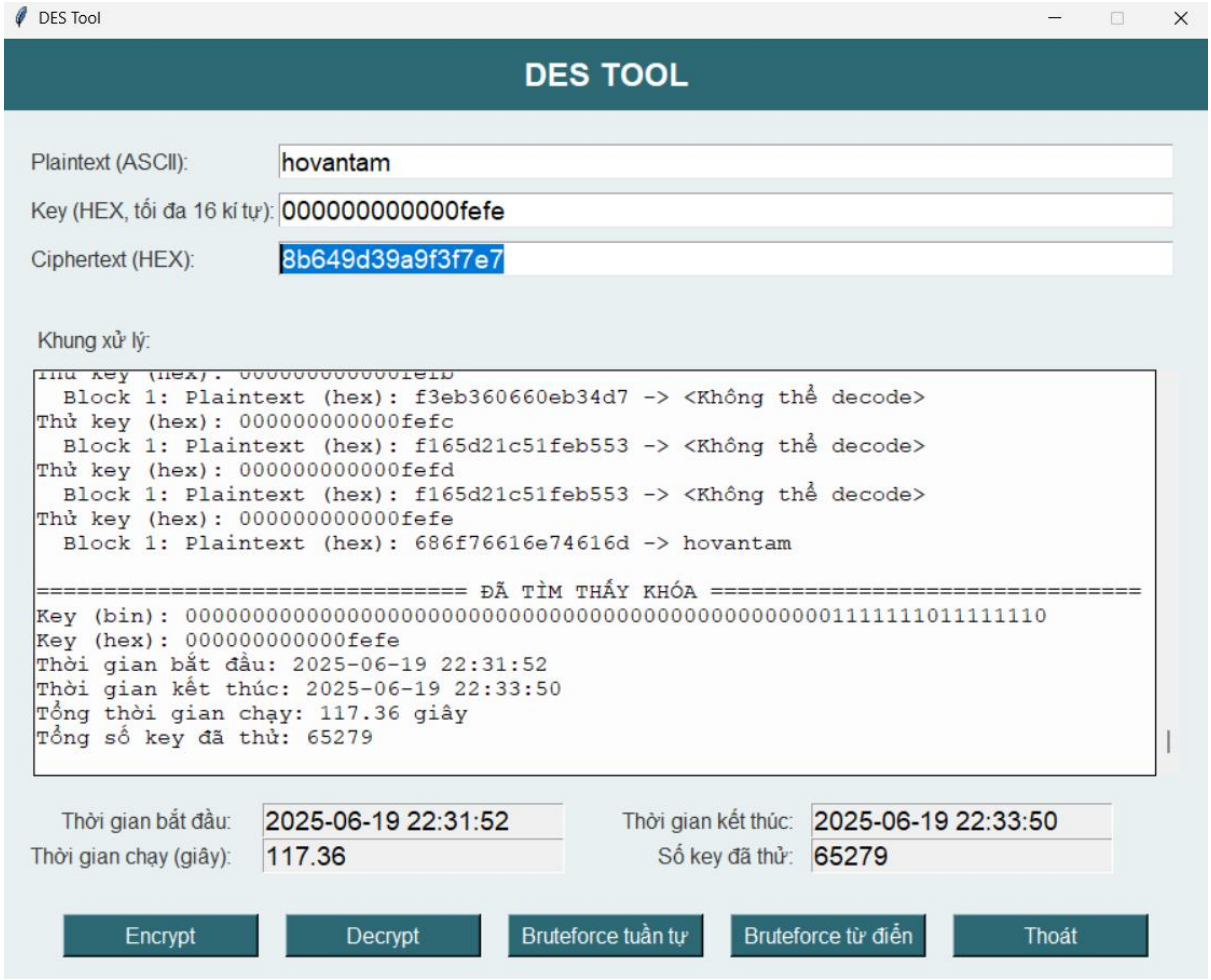
Thời gian duyệt: 7,29 giây

Thời gian duyệt trung bình cho 1 key: 526 giây

Khóa tìm thấy: $0000000000000efe_H$

2.1.3. Test 3

- p = hovantam, c = $8b649d39a9f3f7e7_H$, k = $000000000000ffff_H$



Hình 31: Brute-force (khóa yếu): Kết quả test3

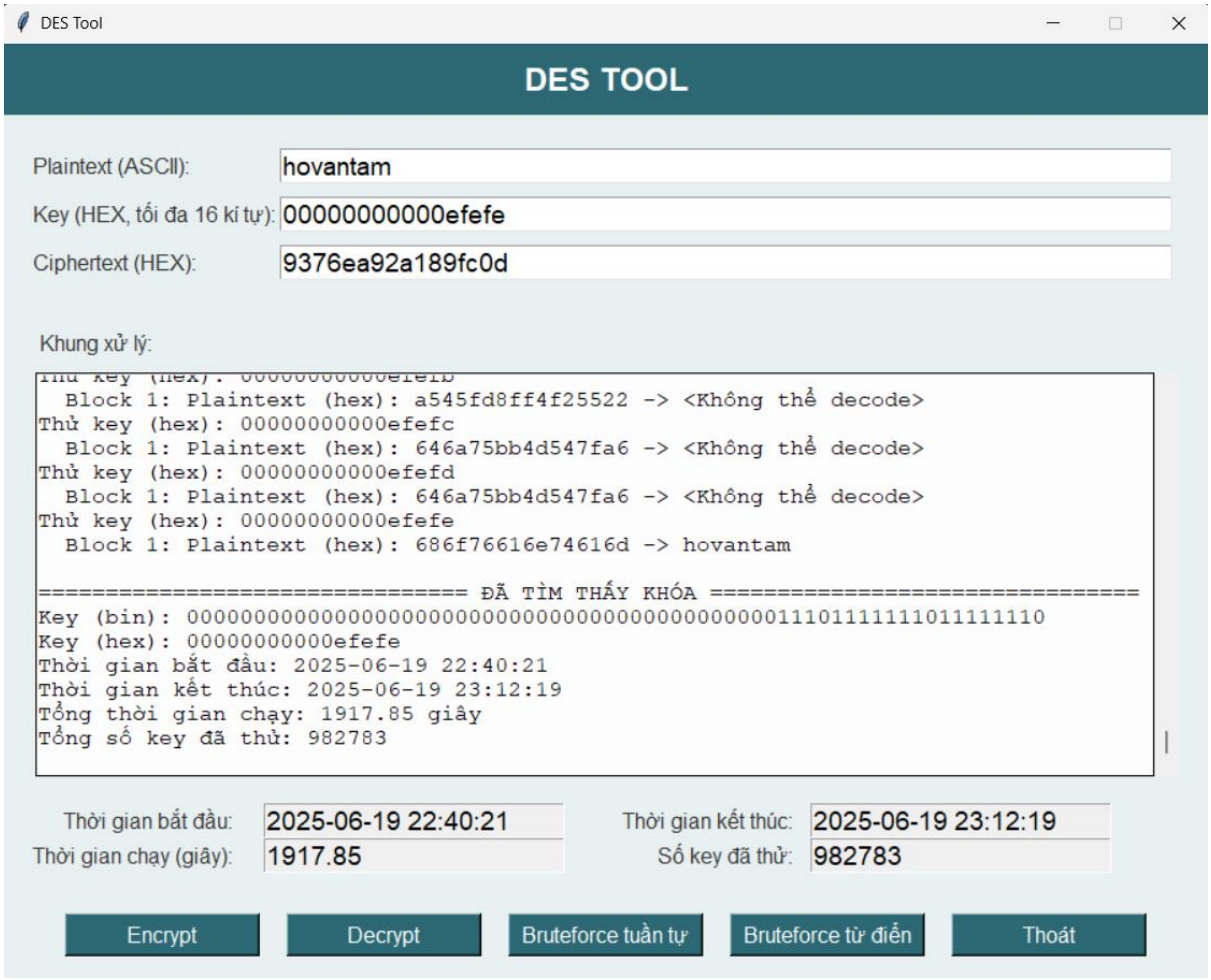
Thời gian duyệt: 117,36 giây

Thời gian duyệt trung bình cho 1 key: 526 giây

Khóa tìm thấy: $000000000000fefe_H$

2.1.4. Test 4

- p = hovantam, c = $9376ea92a189fc0d_H$, k = $00000000000fffff_H$



Hình 32: Brute-force (khóa yếu): Kết quả test4

Thời gian duyệt: 1917,85 giây

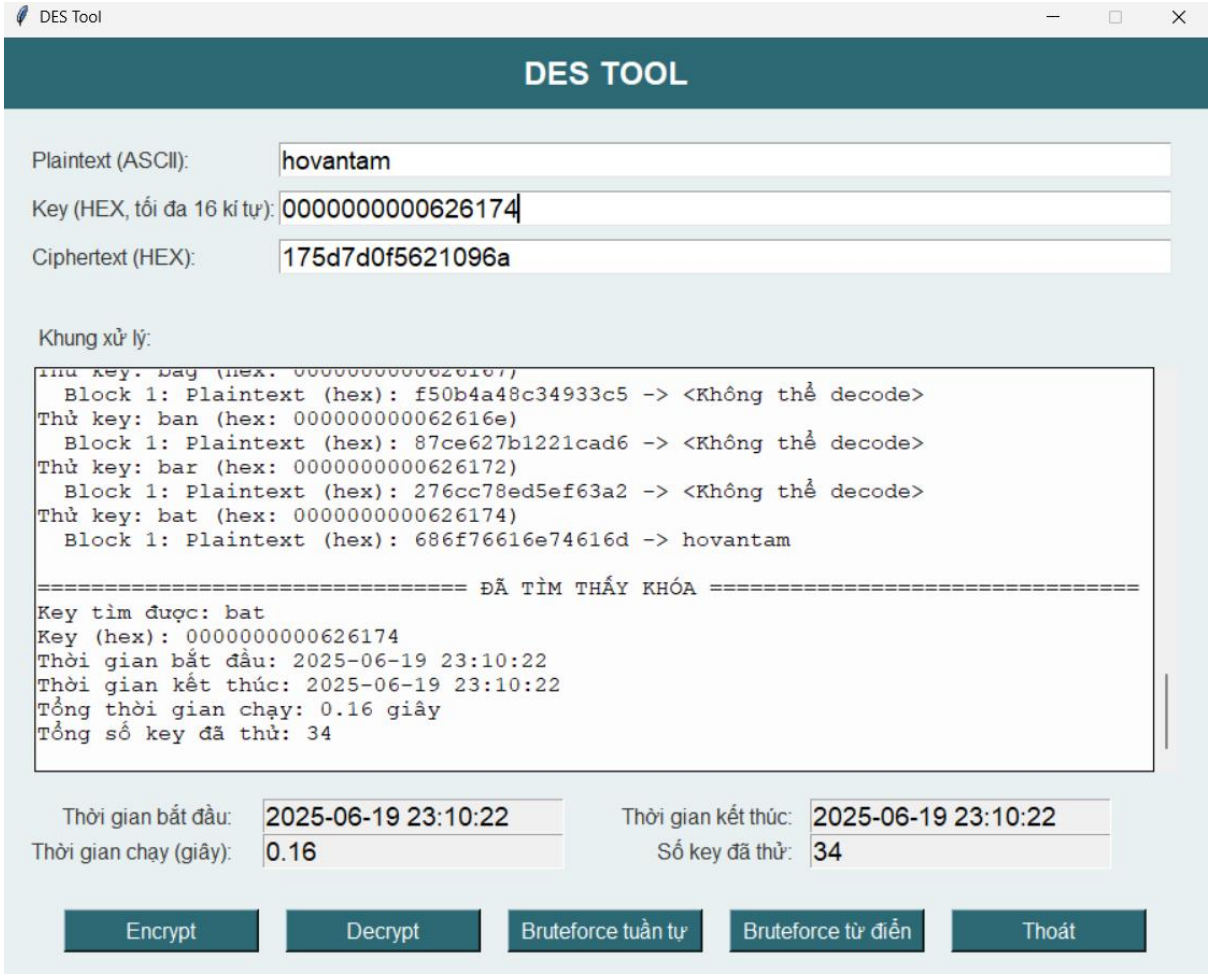
Thời gian duyệt trung bình cho 1 key: 526 giây

Khóa tìm thấy: $00000000000efefe_H$

2.2. Brute force từ điển

2.2.1. Test 1

- p = hovantam, c = $175d7d0f5621096a_H$, k = “cat” = 0000000000636174_H



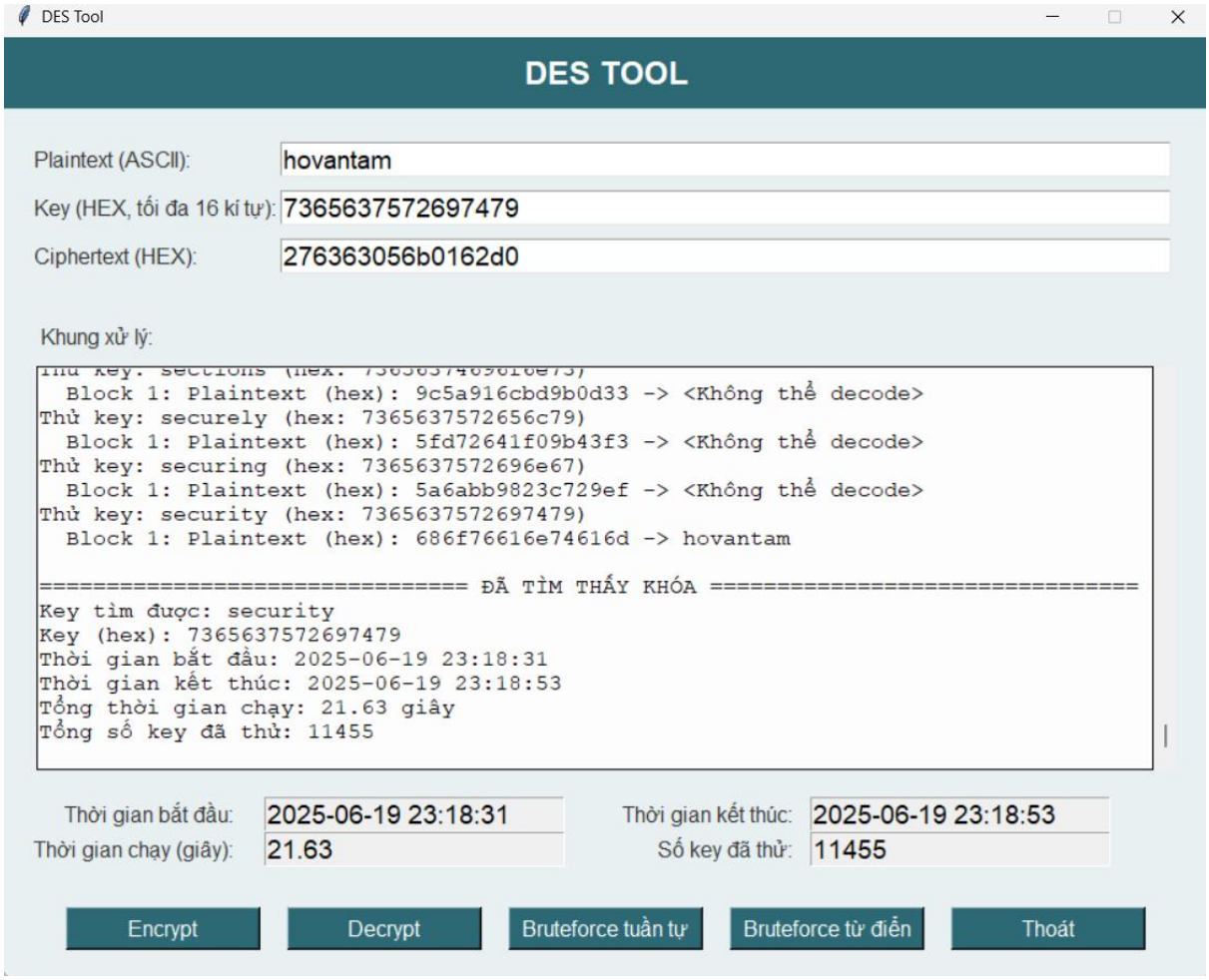
Hình 33: Dictionary (khóa yếu): Kết quả test1

Thời gian duyệt: 0,16 giây

Khóa tìm thấy: 0000000000626174_H

2.2.2. Test 2

- p = hovantam, c = $276363056b0162d0_H$, k = “security” = 7365637572697479_H



Hình 34: Dictionary (khóa yếu): Kết quả test2

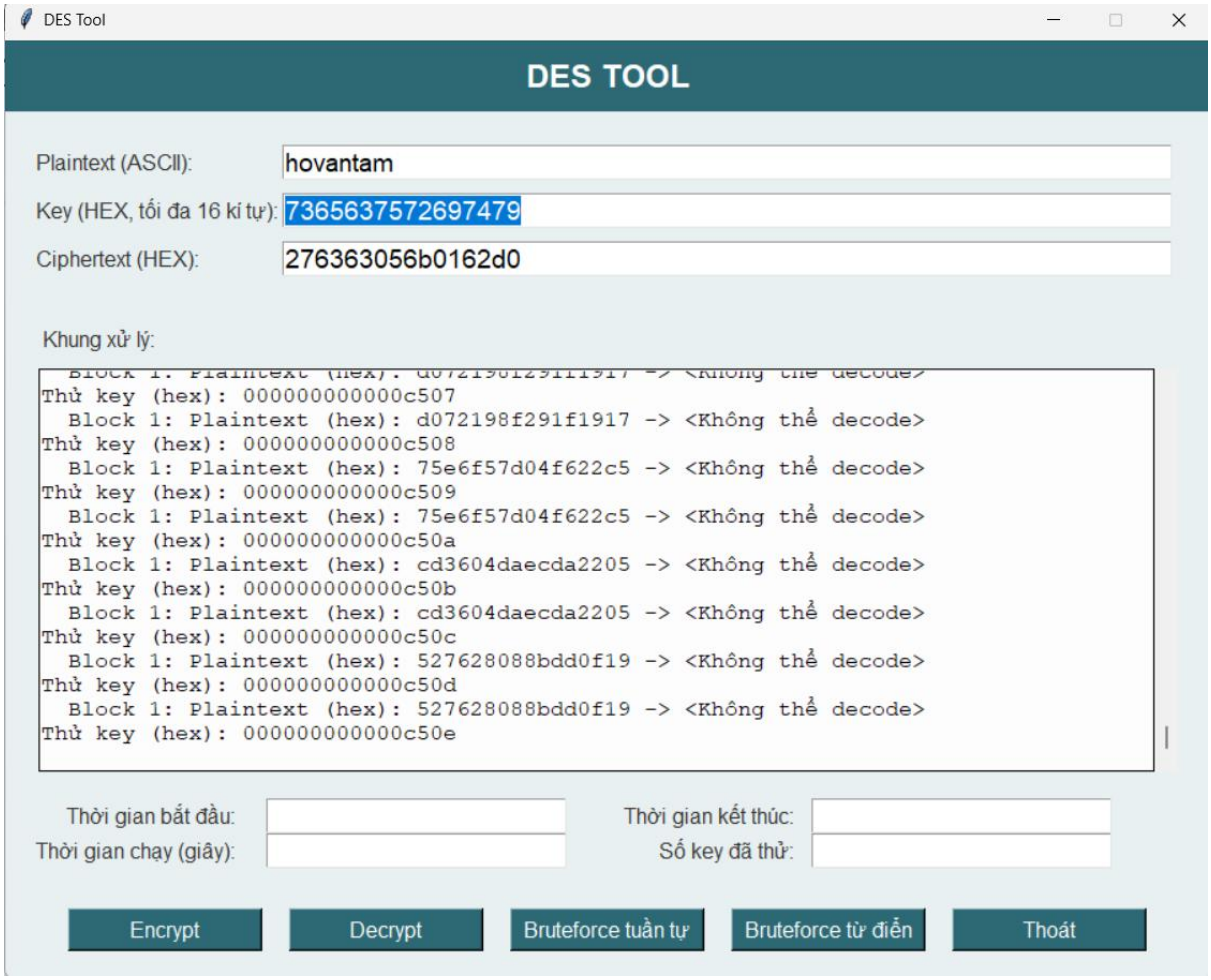
Thời gian duyệt: 21,63 giây

Khóa tìm thấy: 7365637572697479_H

3. Khóa mạnh

3.1. Brute force tuần tự

- p = hovantam, c = $276363056b0162d0_H$, k = “security” = 7365637572697479_H



Hình 35: Brute-force (khóa mạnh): Kết quả

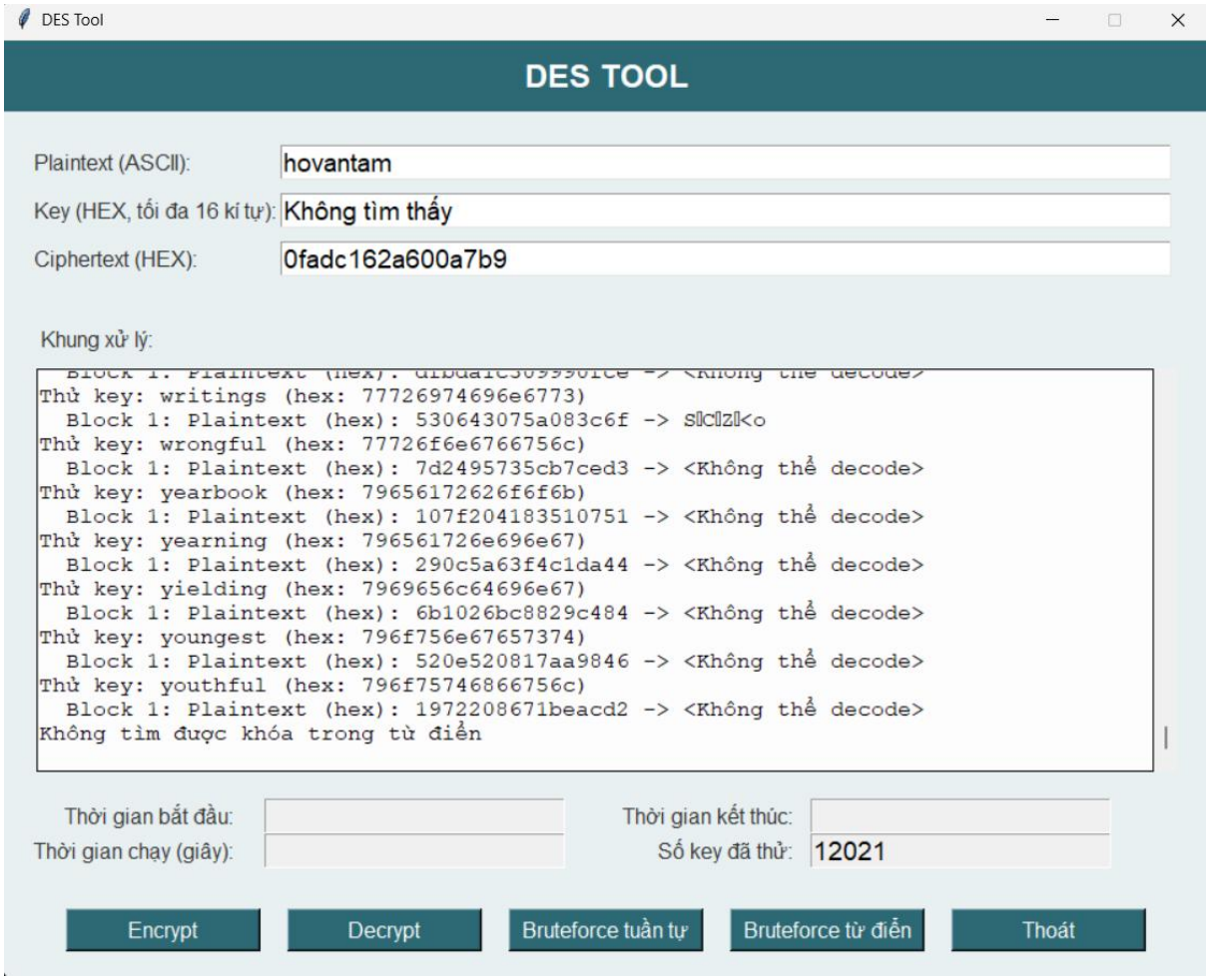
k=01110011 01100101 01100011 01110101 01110010 01101001 01110100 01111001_B

Vậy thuật toán brute force phải dò đến vị trí bit thứ 63, với tổng số tổ hợp là 2^{63} khóa.

Thời gian duyệt ước tính: (số key/giây)×2^{63} giây.

3.2. Brute force từ điển

- p = hovantam, c = $0fadc162a600a7b9_H$, k = “xinloiem” = $78696e6c6f69656d_H$



Hình 36: Dictionary (khóa mạnh): Kết quả

Không tìm thấy khóa để cho ra bản rõ có ý nghĩa.

CHƯƠNG IV. KẾT LUẬN

1. So sánh 2 phương pháp brute force

	Tuần tự	Từ điển
Số khả năng	2^{64}	12021
Tính khả thi	100% chắc chắn	Không chắc chắn
Độ phức tạp thời gian	$O(2^n)$	O(n)
Số key/giây	526	529
Tổng thời gian dò	526×2^{64} giây	23 giây

Bảng

2. Đánh giá tính an toàn của thuật toán mã hóa DES

2.1 Điểm yếu chính về độ dài khóa

Nguyên nhân chủ yếu khiến DES không còn an toàn là độ dài khóa chỉ 56 bit quá nhỏ. Với không gian khóa 2^{56}, DES đã trở thành mục tiêu dễ dàng cho các cuộc tấn công brute force.

2.2 Khóa yếu và nửa yếu

DES có 4 khóa yếu mà tất cả 16 khóa con đều giống nhau, nghĩa là ($Z_1 = Z_2 = ... = Z_{16}$), khiến cho quá trình mã hóa và giải mã trở thành giống hệt nhau.

2.3 Nghi ngờ về cửa sau

Từ khi ra đời, DES đã gây tranh cãi do các thành phần thiết kế bí mật và nghi ngờ về việc NSA có thể đã cài đặt cửa sau để có thể phá khóa dễ dàng hơn. Các đặc tính của S-box vẫn được NSA che giấu, tạo ra nghi ngờ về khả năng tồn tại các trapdoor.

3. Hướng cải tiến

3.1. Tối ưu thuật toán brute force

Thuật toán brute force tuần tự trong đồ án có độ phức tạp thời gian là 2^{64}, vì nó duyệt tuần tự các tổ hợp 64 bit gồm cả các parity bit một cách tăng dần, nên có thể thấy khóa dò ra ứng với bản rõ có nhiều hơn 1, cụ thể là có nhiều nhất 2^8 khóa như vậy.

Vì vậy, có thể giảm độ phức tạp về thời gian bằng cách giảm xuống số vòng lặp từ 2^{64} xuống còn 2^{56} bằng cách liệt kê toàn bộ các tổ hợp của C_0, D_0 thay vì khóa k.

3.2. Áp dụng kỹ thuật khuếch tán để trộn lẫn các từ ngữ thông thường với ký tự đặc biệt, tạo ra nhiều tổ hợp mới

Thuật toán brute force từ điển trong đồ án chỉ sử dụng một tập các từ vựng phổ biến từ 3 đến tối đa 8 ký tự tương ứng với 64 bit khóa. Tập dữ liệu này chỉ sử dụng các chữ cái viết thường chứ không sử dụng chữ số, chữ cái viết hoa hay các ký tự đặc biệt nên chỉ có hơn 12000 từ, thấp hơn rất nhiều so với tổng số tổ hợp 95^8 trong bảng ASCII có thể có của khóa 64 bit.

Vì vậy, chúng ta có thể sử dụng các kỹ thuật khuếch tán để trộn lẫn các từ ngữ thông thường với các ký tự đặc biệt hay viết hoa các chữ cái đầu để tạo ra một tập dữ liệu lớn hơn, tăng tỉ lệ brute force từ điển thành công.

4. Đánh giá 2 phương pháp brute force

Qua các kịch bản trên, ta thấy có một sự đánh đổi giữa tính khả thi và thời gian thực hiện giữa 2 phương pháp. Chúng đều tồn tại những ưu điểm lẫn hạn chế, nên người tấn công cần hiểu rõ sự đánh đổi ấy để lựa chọn phương pháp tấn công phù hợp với hoàn cảnh hay mục tiêu của bản thân.

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Chợ, Bài giảng Mật mã học cơ sở, Học viện công nghệ bưu chính viễn thông, 2021

[2] TS. Thái Thanh Tùng, Giáo trình Mật mã học và Hệ thống an toàn thông tin, NXB Thông tin và Truyền thông

[3] William Stallings, "Cryptography and Network Security: Principles and Practice", 2023

[4] Bruce Schneier, "Applied Cryptography", Chapter 1 v& Chapter 6

[5] Electronic Frontier Foundation (EFF), “Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design” , 1998

[6] Mark Stamp , "Information Security: Principles and Practice", Chapter 4 & Chapter 5

[7] Nguyễn Khanh Văn, Cơ sở an toàn thông tin, Đại học Bách khoa Hà Nội, 2014, 230 trang.

[8] Mitsuru Matsui, Linear Cryptanalysis of DES-Cipher.

[9] “A Survey of Password Cracking Techniques and Tools” , 2020

[10] “A Chosen-Plaintext Attack on DES” , Eli Biham , 1991