**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**

**Trường Đại Học Khoa Học Tự Nhiên**

**Khoa Công Nghệ Thông Tin**

|  |
| --- |
| **NGUYỄN THỊ NGÂN HÀ - 19C11004** |
| **ĐẶNG MINH TUẤN - 19C11010** |

ĐỒ ÁN MÔN HỌC

**MÃ HOÁ**

**ĐỀ TÀI : MÃ HÓA MẬT KHẨU NGƯỜI DÙNG VÀ THÔNG TIN TRUYỀN TẢI**

**GVHD: TS. NGUYỄN ĐÌNH THÚC**

**TP.HCM-12/2020**

MỤC LỤC

[LỜI MỞ ĐẦU 4](#_Toc61188962)

[PHẦN 1. TÌM HIỂU VỀ RSA 5](#_Toc61188963)

[1.1. Giới thiệu khái niệm Mã hóa 5](#_Toc61188964)

[1.2. Các loại mã hóa? 5](#_Toc61188965)

[1.3. Quy trình mã hóa của mã hóa bất đối xứng. 6](#_Toc61188966)

[1.3.1. Tạo khóa - Key generation 6](#_Toc61188967)

[1.3.2. Phân phối khóa - Key distribution 6](#_Toc61188968)

[1.3.3. Encryption 7](#_Toc61188969)

[1.3.4. Decryption 7](#_Toc61188970)

[1.4. Ví dụ thuật toán RSA 7](#_Toc61188971)

[1.4.1. Tạo khóa - Key generation 7](#_Toc61188972)

[1.4.2. Phân phối khóa - Key distribution 7](#_Toc61188973)

[1.4.3. Encryption 7](#_Toc61188974)

[1.4.4. Decryption 7](#_Toc61188975)

[PHẦN 2. CÀI ĐẶT – THỰC NGHIỆM MÔ HÌNH MÃ HOÁ VỚI RSA 9](#_Toc61188976)

[2.1. Cài Đặt Thuật Toán RSA Trên C++ 9](#_Toc61188977)

[2.2. Thử nghiệm RSA trên C++ 11](#_Toc61188978)

[2.3. Cài Đặt Thuật Toán RSA Trên JAVASCIPT 12](#_Toc61188979)

[2.4. Chạy Thuật Toán RSA Trên JAVASCIPT 12](#_Toc61188980)

[2.4.1. Encrypt 12](#_Toc61188981)

[2.4.2. Decrypt 12](#_Toc61188982)

[2.5. Nhận xét và đánh giá 13](#_Toc61188983)

[PHẦN 3. HÀM BĂM 13](#_Toc61188984)

[3.1. Khái niệm hàm băm 14](#_Toc61188985)

[3.2. Cấu trúc của hàm băm 15](#_Toc61188986)

[3.3. Một số tính chất của hàm băm 15](#_Toc61188987)

[PHẦN 4. TÌM HIỂU VỀ BCRYPT 17](#_Toc61188988)

[4.1. Thuật toán Blowfish 17](#_Toc61188989)

[4.2. Hàm băm Bcrypt 19](#_Toc61188990)

[PHẦN 5. CÀI ĐẶT VÀ THỰC NGHIỆM MÔ HÌNH LƯU MẬT KHẨU NGƯỜI DÙNG VỚI BCRYPT, MÃ HÓA THÔNG TIN TRUYỀN TẢI VỚI RSA 21](#_Toc61188991)

[5.1. Thực nghiệm mô hình lưu mật khẩu người dùng với bcrypt 21](#_Toc61188992)

[5.1.1. Giao diện log in 21](#_Toc61188993)

[5.1.2. Mật khẩu sau khi đã được lưu trữ trong csdl: 21](#_Toc61188994)

[5.1.3. Kiểm tra mật khẩu đăng nhập. 22](#_Toc61188995)

[5.2. Thực nghiệm mô hình mã hóa thông tin truyền tải với RSA. 23](#_Toc61188996)

[5.2.1. Khởi tạo RSA. 23](#_Toc61188997)

[5.2.2. Lấy thông tin public key từ server. 23](#_Toc61188998)

[5.2.3. Lấy thông tin private key. 24](#_Toc61188999)

[5.2.4. Gửi thông điệp từ client tới server. 24](#_Toc61189000)

[5.2.5. Gửi thông điệp từ server tới client. 24](#_Toc61189001)

[KẾT LUẬN 26](#_Toc61189002)

[TÀI LIỆU THAM KHẢO 26](#_Toc61189003)

# LỜI MỞ ĐẦU

Từ xa xưa, con người đã sử dụng mã hóa trong nhiều ứng dụng cuộc sống như dùng để gửi tin nhắn với hy vọng rằng tin đó không bị đọc bởi ai ngoài người nhận tin, hoặc các trò chơi đơn giản như truy tìm kho báu cũng đòi hỏi người chơi buộc phải cố gắng giải mã để tìm ra được thông điệp về đường đi. Ngày nay, chúng ta có máy tính với khả năng lưu trữ và trao đổi một lượng lớn thông tin thì yêu cầu về mã hoá lại càng là một nhu cầu cấp thiết, mã hoá đã không còn chỉ bảo mật tin nhắn mà mở rộng ra cho các mục đích khác chẳng hạn có thể sử dụng cho những mục đích tinh vi hơn như để xác nhận người gửi tin nhắn để đảm bảo tính an toàn.

Ngoài ra, mã hoá còn được ứng dụng trong lĩnh vực lưu trữ thông tin, đảm bảo tính riêng tư và quyền lợi cho người dùng. Làm thế nào để lưu trữ thông tin mật khẩu người dùng một cách hiệu quả (mật khẩu ngắn nhưng vẫn đảm bảo được tính bảo mật cao)?

Trong đề tài này, nhóm tập trung vào tìm hiểu kỹ thuật mã hoá dựa trên hệ mã công khai để mã hoá một thông điệp nào đó là RSA và tìm hiểu về cơ chế băm của BCRYPT từ các bước thực thi cũng như cài đặt nó trên một ứng dụng website đơn giản và đưa ra một số kết luận mà nhóm đã học tập được thông qua đề tài này.

# PHẦN 1. TÌM HIỂU VỀ RSA

## 1.1. Giới thiệu khái niệm Mã hóa

Mã hóa là cách xáo trộn dữ liệu chỉ để hai bên trao đổi thông tin có thể hiểu được. Về mặt kỹ thuật, đó là quá trình chuyển đổi văn bản gốc sang bản mã. Nói một cách đơn giản hơn, mã hóa lấy dữ liệu có thể đọc được và thay đổi nó để dữ liệu này không giống như ban đầu. Mã hóa yêu cầu sử dụng khóa mã hóa: một tập hợp các giá trị toán học mà cả người gửi và người nhận tin nhắn được mã hóa đều biết.

Ví dụ.



## 1.2. Các loại mã hóa?

Hiện nay có 2 loại mã hóa:

* **Mã hóa đối xứng:** chỉ có một khóa và tất cả các bên giao tiếp sử dụng cùng một khóa để mã hóa và giải mã. Khóa ở đây có thể là 1 thuật toán (mã hóa và giải mã) nào đó, hoặc là 1 key nào đó trong 1 thuật toán đã phổ biến. Có thể chia mã hóa đối xứng thành 3 loại:
  + Mã hóa cổ điển: chỉ dùng thuật toán, không cần key.
  + Mã hóa 1 chiều (hash)
  + Mã hóa đối xứng: dùng thuật toán, cần key cho cả 2 bên.

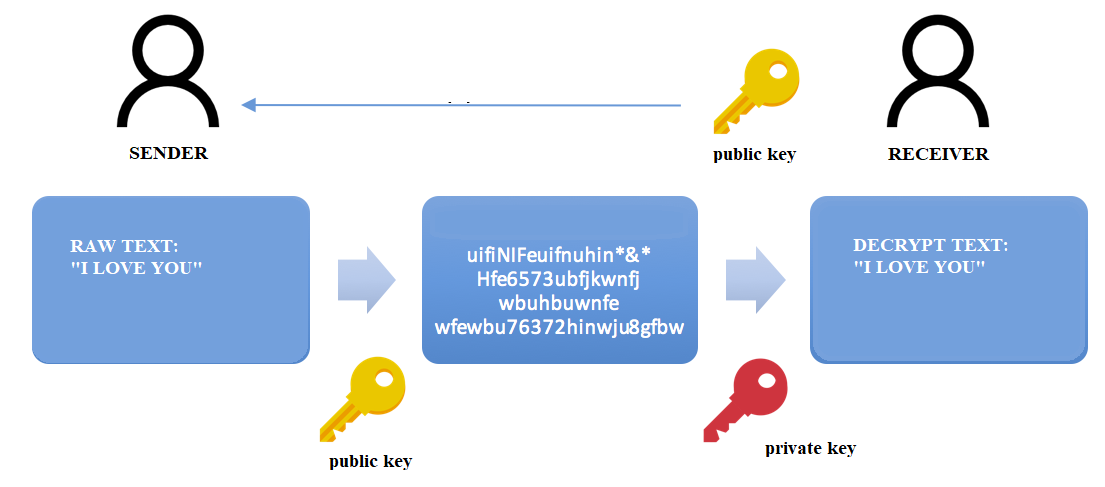
*Mã hóa đối xứng có điểm yếu là 2 bên phải trao đổi key để encrypt và decrypt, do đó nếu bị tấn công theo dạng* ***man-in-the-middle attack*** *thì vẫn có thể bị lộ thông tin.*

* **Mã hóa bất đối xứng** (khóa công khai): có hai khóa: một khóa được sử dụng để mã hóa và một khóa khác được sử dụng để giải mã. Một trong hai khóa có thể được sử dụng cho một trong hai hành động, nhưng dữ liệu được mã hóa bằng khóa đầu tiên chỉ có thể được giải mã bằng khóa thứ hai và ngược lại. Một khóa được giữ riêng tư, trong khi một khóa được chia sẻ công khai, cho bất kỳ ai sử dụng – do đó tên “khóa chung”.

*Điểm yếu của mã hoá đối xứng được khắc chế trong mã hoá bất đối xứng. Ý tưởng là thay vì gửi chìa khoá cho phía client, thì server sẽ gửi ổ khoá, để client khoá thông điệp bí mật trong một chiếc hộp, và chỉ có server có thể giải mã được. Cho nên các client sẽ không đọc được thông điệp của nhau, và chỉ có sever với private key mới mở khoá được những chiếc hộp này.*

Phần tìm hiểu của nhóm sẽ dựa trên mã hóa bất đối xứng (mã hóa công khai).

## 1.3. Quy trình mã hóa của mã hóa bất đối xứng.



**Quá trình mã hóa gồm 4 bước:**

### 1.3.1. Tạo khóa - Key generation

Tạo 2 số nguyên tố lớn P, Q

Tính N = P \* Q

Tính Phi = (P - 1)(Q - 1)

Chọn E sao cho GCD(E, phi) = 1

Chọn D sao cho E \* D = 1 (mod Phi)

*🡪 Ta có public key: E, N*

*🡪 Ta có private key: D, N*

### 1.3.2. Phân phối khóa - Key distribution

Với mã hóa đối xứng, việc phân phối khóa không đòi hỏi quá nhiều sự phức tạp cũng như bảo mật. Còn mã hóa bất đối xứng, khóa công khai được public cho tất cả mọi người.

### 1.3.3. Encryption

Việc mã hóa được thực hiện từ phía người gửi, dựa vào khóa công khai (E, N).

Văn bản cần mã hóa: M

Phía người gửi cần tính toán giá trị C = ME mod N

Sau đó gửi C cho người nhận.

### 1.3.4. Decryption

Việc giải mã được thực hiện từ phía người nhận, dựa vào khóa bí mật (D, N).

Văn bản giải mã M = CD mod N

## 1.4. Ví dụ thuật toán RSA

### 1.4.1. Tạo khóa - Key generation

|  |  |
| --- | --- |
| p = 61 | — số nguyên tố thứ nhất (giữ bí mật hoặc hủy sau khi tạo khóa) |
| q = 53 | — số nguyên tố thứ hai (giữ bí mật hoặc hủy sau khi tạo khóa) |
| n = pq = 3233 | — môđun (công bố công khai) |
| e = 17 | — số mũ công khai |
| d = 2753 | — số mũ bí mật |

### 1.4.2. Phân phối khóa - Key distribution

Khóa công khai là cặp (e, n) = (17,3233) . Khóa bí mật là (d,n) = (2753,3233).

### 1.4.3. Encryption

Hàm mã hóa là:

encrypt(m) = me mod n = m17 mod 3233. Với m là văn bản rõ.

Để mã hóa văn bản có giá trị 123, ta thực hiện phép tính:

encrypt(123) = 12317 mod 3233 = 855

### 1.4.4. Decryption

Hàm giải mã là:

decrypt(c) = cd mod n = c2753 mod 3233 với c là văn bản mã.

Để giải mã văn bản có giá trị 855, ta thực hiện phép tính:

decrypt(855) = 8552753 mod 3233 = 123

Như vậy qua Phần 1, chúng ta đã hiểu rõ về quy trình mã hoá RSA, tiếp đến ở phần 2 chúng ta sẽ cùng tìm hiểu về quá trình cài đặt và thực nghiệm mô hình trên ngôn ngữ C++.

# PHẦN 2. CÀI ĐẶT – THỰC NGHIỆM MÔ HÌNH MÃ HOÁ VỚI RSA

## 2.1. Cài Đặt Thuật Toán RSA Trên C++

Thuật toán RSA yêu cầu lập trình và debug nhiều, do đó nhóm chọn IDE là C++ để dễ dàng cài đặt.

RSA được cài đặt gồm 2 lớp:

* Lớp BigInt dùng để lưu trữ và xử lý số nguyên lớn.
* Các biến dữ liệu:

|  |
| --- |
| int \_value = 0; #giá trị dùng cho debug với số int nhỏ  bool neg = false; #dấu (âm/dương) của số nguyên  bool bnumber[1024\*8] = { false }; #mảng bit lưu trữ giá trị của số nguyên lớn  int head = 0; #con trỏ đầu  int tail = 0; #con trỏ đuôi  int digits = 0; #số lượng bit có nghĩa trong số nguyên |

|  |
| --- |
| #Các hàm khởi tạo, kiểm tra số chẵn, nhân và chia cho 2  void SetValue(long long int number);  bool isEven();  void Devision\_to\_2();  void Multiply\_to\_2();  bool isPositive();  void RandomPrimeCandidate(); #hàm tạo ngẫu nhiên số nguyên tố lớn  void Random(int minnumbers, int maxnumbers);#hàm tạo ngẫu nhiên số nguyên lớn với lượng bit cho trước  #Hàm khởi tạo từ chuỗi HEX, xuất ra chuỗi HEX  std::string ToString();  bool FromString(std::string input);  #Các hàm so sánh số nguyên lớn  friend bool operator == (BigInt first, BigInt second);  friend bool operator != (BigInt first, BigInt second);  friend bool operator != (BigInt first, int second);  friend bool operator < (BigInt first, BigInt second);  friend bool operator <= (BigInt first, BigInt second);  friend bool operator > (BigInt first, BigInt second);  friend bool operator > (BigInt first, int second);  friend bool operator >= (BigInt first, BigInt second);  void operator = (long long int number);  #Các hàm swap và abs  friend void swap(BigInt &first, BigInt &second);  friend BigInt BigIntAbs(BigInt bigint);    #Các hàm cộng trừ nhân chia, mod  friend BigInt Plus(BigInt first, BigInt second);  friend BigInt Minus(BigInt first, BigInt second);  friend BigInt Multiply (BigInt first, BigInt second);  friend BigInt Divide (BigInt first, BigInt second);  friend BigInt Modulo(BigInt first, BigInt second);  friend BigInt Modulo2(BigInt first, BigInt second);  friend BigInt PowerMod(BigInt first, BigInt second, BigInt mod);  friend BigInt MulMod(BigInt first, BigInt second, BigInt mod); |

* Lớp CRSA dùng để cài đặt RSA

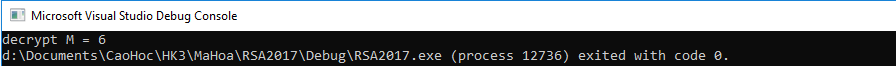
|  |
| --- |
| BigInt N;  BigInt E;  BigInt D; |
| #Hàm khởi tạo RSA  void Init();  BigInt gcd(BigInt a, BigInt b);  #Hàm test giả nguyên tố  bool fermat\_testing(BigInt n, int a);  bool fermat\_testing(BigInt x) ;  #Hàm khởi tạo số nguyên tố P, Q  BigInt PrimeGen();  BigInt rel\_prime(BigInt phi); #Hàm tính E  BigInt encrypt(BigInt m); #Hàm mã hóa  BigInt calculate\_d(BigInt phi, BigInt e); #Hàm tính D  BigInt decrypt(BigInt c); #Hàm giải mã |

## 2.2. Thử nghiệm RSA trên C++

Hàm chạy thử RSA như sau:

|  |
| --- |
| int main()  {  CRSA rsa;  rsa.Init();  BigInt M = 6;  BigInt c = rsa.encrypt(M);  BigInt M1 = rsa.decrypt(c);  std::cout << "decrypt M = " << M1.\_value;  } |

Kết quả chạy thử RSA như sau:



## 2.3. Cài Đặt Thuật Toán RSA Trên JAVASCIPT

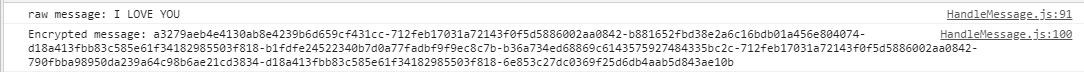
Do thuật toán RSA đã được cài đặt chi tiết trên C++, trên Java nhóm chỉ sử dụng lớp BigInt từ Opensouce để tính toán mã hóa và giải mã.

|  |
| --- |
| #Hàm mã hóa RSA    function EncryptMessage(txtMessage){      var result = "";      for (var i = 0; i < txtMessage.length; i++) {        var m = bigInt(txtMessage.charCodeAt(i));        var c = m.modPow (BigInt\_E,BigInt\_N);        result = result + c.toString(16);        if(i < txtMessage.length - 1)        result = result + "-";      }      return result;    } |
| #Hàm giải mã RSA    function DecryptMessage(txtMessageEncrypted){        strMessages = txtMessageEncrypted.split('-');      var result = "";      for (var i = 0; i < strMessages.length; i++) {       var c = new bigInt(strMessages[i], 16);        var m = c.modPow (BigInt\_private\_D,BigInt\_private\_N);        var number\_M = Number(m);        var charM = String.fromCharCode(number\_M);        result = result + charM;      }      return result;    } |

## 2.4. Chạy Thuật Toán RSA Trên JAVASCIPT

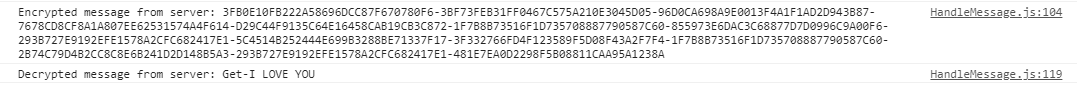
### 2.4.1. Encrypt

Thực hiện mã hóa với message “I LOVE YOU”:



### 2.4.2. Decrypt

Thực hiện giải mã với message “GET-I LOVE YOU” từ server:



## 2.5. Nhận xét và đánh giá

Quá trình cài đặt RSA trên C++ tương đối phức tạp, một số hạn chế của RSA như sau:

* Với n không đủ lớn thì sẽ không đảm bảo với những cuộc tấn công brute-force để tìm ra được p và q.
* Quá trình gửi khóa cần chống lại được [tấn công đứng giữa](https://vi.wikipedia.org/w/index.php?title=T%E1%BA%A5n_c%C3%B4ng_%C4%91%E1%BB%A9ng_gi%E1%BB%AFa_(m%E1%BA%ADt_m%C3%A3)&action=edit&redlink=1) (*man-in-the-middle attack*).
* Một điểm nữa cần nhấn mạnh là khóa bí mật *d* phải đủ lớn. Nếu thì chắc chắn tìm ra được p,q.

Điều đặc biệt là như đã nói ở trên mã hóa (Encryption) là hành động xáo trộn thông tin theo cách mà chỉ người nào đó có khóa tương ứng mới có thể xắp xếp lại và đọc nó. Mã hóa là một hàm hai chiều. Chúng ta mã hóa một cái gì đó, thì sau này chúng ta giải mã nó sau này. Điều quan trọng là mã hóa có thể đảo ngược. Nhưng Hashing thì không.

Vì vậy với một số hạn chế của RSA, nhóm tiến hành tìm hiểu hàm băm BCRYPT và cài đặt nó cho một ứng dụng website.

# PHẦN 3. HÀM BĂM

Ngày nay, các thông điệp sử dụng chữ ký điện tử có độ dài bất kỳ, thậm chí lên đến vài Megabyte. Trong khi đó, thuật toán chữ ký điện tử được lại áp dụng trên các thông điệp có độ dài cố định và thường tương đối ngắn, chẳng hạn như phương pháp DSS (*Digital Signature Standard*) sử dụng chữ ký 320 bit trên thông điệp 160 bit. Để giải quyết vấn đề này, chúng ta có thể chia nhỏ thông điệp cần ký thành các đoạn nhỏ có độ dài thích hợp và ký trên từng mảnh thông điệp này. Tuy nhiên, giải pháp này lại có nhiều khuyết điểm và không thích hợp áp dụng trong thực tế:

Chẳng hạn như, khi văn bản cần được ký quá dài thì số lượng chữ ký được tạo ra sẽ rất nhiều và kết quả nhận được là một thông điệp có kích thước rất lớn.

Các phương pháp chữ ký điện tử có độ an toàn cao đều đòi hỏi chi phí tính toán cao và do đó, tốc độ xử lý rất chậm. Việc áp dụng thuật toán tạo chữ ký điện tử nhiều lần trên một văn bản sẽ thực hiện rất lâu.

Từng đoạn văn bản sau khi được ký có thể dễ dàng bị thay đổi thứ tự hay bỏ bớt đi mà không làm mất đi tính hợp lệ của văn bản. Việc chia nhỏ văn bản sẽ không thể bảo đảm được tính toàn vẹn của thông tin ban đầu cần được ký.

Để giải quyết những khuyết điểm đã nêu trên người ta xây dựng hàm băm, nghĩa là trước tiên sẽ xây dựng một bảng băm (nơi lưu trữ dữ liệu sao cho có thể tìm lại một cách nhanh nhất) và sau đó sử dụng một hàm ánh xạ (hàm băm) giá trị của chuỗi thành chỉ số trên bảng băm mà lưu trữ dữ liệu này.

## 3.1. Khái niệm hàm băm

Hàm băm mật mã là hàm toán học chuyển đổi một thông điệp có độ dài bất kỳ thành một dãy bit có độ dài cố định (tùy thuộc vào thuật toán băm). Dãy bit này được gọi là thông điệp rút gọn (message digest) hay giá trị băm (hash value), đại diện cho thông điệp ban đầu. Với *h* được biểu thị cho kí hiệu hàm băm.

*h* không phải là một song ánh. Do đó, với thông điệp *x* bất kỳ, tồn tại thông điệp *x*’*x* sao cho *h*(*x*) = *h*(*x*’). Lúc này, ta nói rằng “có sự đụng độ xảy ra”. Một hàm băm *h* được gọi là an toàn (hay “ít bị đụng độ”) khi không thể xác định được (bằng cách tính toán) cặp thông điệp *x* và *x*’ thỏa mãn *x* *x*’ và *h*(*x*) = *h*(*x*’). Trên thực tế, các thuật toán băm là hàm một chiều, do đó, rất khó để xây dựng lại thông điệp ban đầu từ thông điệp rút gọn.

Hàm băm giúp xác định được tính toàn vẹn dữ liệu của thông tin: mọi thay đổi, dù là rất nhỏ, trên thông điệp cho trước, ví dụ như đổi giá trị 1 bit, đều làm thay đổi thông điệp rút gọn tương ứng. Tính chất này hữu ích trong việc phát sinh, kiểm tra chữ ký điện tử, các đoạn mã chứng nhận thông điệp, phát sinh số ngẫu nhiên, tạo ra khóa cho quá trình mã hóa...

Hàm băm là nền tảng cho nhiều ứng dụng mã hóa. Có nhiều thuật toán để thực hiện hàm băm, trong số đó, phương pháp SHA-1 và MD5 thường được sử dụng khá phổ biến ở những thập niên 1990.

## 3.2. Cấu trúc của hàm băm

Hầu hết các hàm băm mật mã đều có cấu trúc giải thuật như sau:

Cho trước một thông điệp *M* có độ dài bất kỳ. Tùy theo thuật toán được sử dụng, chúng ta có thể cần bổ sung một số bit vào thông điệp này để nhận được thông điệp có độ dài là bội số của một hằng số cho trước. Chia nhỏ thông điệp thành từng khối có kích thước bằng nhau: *M*1, *M*2, ...*Ms*

Gọi *H* là trạng thái có kích thước *n* bit, *f* là “hàm nén” thực hiện thao tác trộn khối dữ liệu với trạng thái hiện hành. Khởi gán *H*0 bằng một vector khởi tạo nào đó  
*Hi* = *f*(*Hi*−1,*Mi*) với *i*=1,2,3,...,*s*. *Hs* chính là thông điệp rút gọn của thông điệp *M* ban đầu.

## 3.3. Một số tính chất của hàm băm

* ***Tính một chiều***

Hàm băm được xem là hàm một chiều khi cho trước giá trị băm, không thể tái tạo lại thông điệp ban đầu, hay còn gọi là “tiền ảnh” (“pre-image”). Như vậy, trong trường hợp lý tưởng, cần phải thực hiện hàm băm cho khoảng 2nthông điệp để tìm ra được “tiền ảnh” tương ứng với một giá trị băm.

Nếu tìm ra được một phương pháp tấn công cho phép xác định được “tiền ảnh” tương ứng với một giá trị băm cho trước thì thuật toán băm sẽ không còn an toàn nữa.

Cách tấn công nhằm tạo ra một thông điệp khác với thông điệp ban đầu nhưng có cùng giá trị băm gọi là tấn công “tiền ảnh thứ hai” (second pre-image attack).

* ***Tính an toàn của hàm băm đối với hiện tượng đụng độ.***

Hàm băm được xem là an toàn đối với hiện tượng đụng độ khi rất khó tìm được hai thông điệp có cùng giá trị băm.

Hàm băm không chỉ dùng trong mã hoá chữ kí số, ngày nay rất nhiều giải thuật ra đời như Bcrypt, Scrypt,... dựa trên ý tưởng chung của hàm băm nhằm đáp ứng các nhu cầu bảo mật khác của người dùng như bảo mật và lưu trữ mật khẩu. Trong phạm vi đồ án này nhóm tập trung tìm hiểu về giải thuật Bcrypt.

# PHẦN 4. TÌM HIỂU VỀ BCRYPT

Bcrypt là một hàm băm mật khẩu được thiết kế bởi Niels Provos và David Mazières, dựa trên mật mã Blowfish (Bcrypt sử dụng 128 bit salt và 192 bit encrypt) và được trình bày tại USENIX năm 1999. Bên cạnh việc kết hợp salt để bảo vệ khỏi các cuộc tấn công phá mã, bcrypt là một hàm tự thích ứng theo thời gian số lần lặp có thể được tăng lên để làm cho nó chậm hơn vì vậy nó vẫn chống lại các cuộc tấn công tìm kiếm brute-force ngay cả khi sức mạnh tính toán ngày càng tăng.

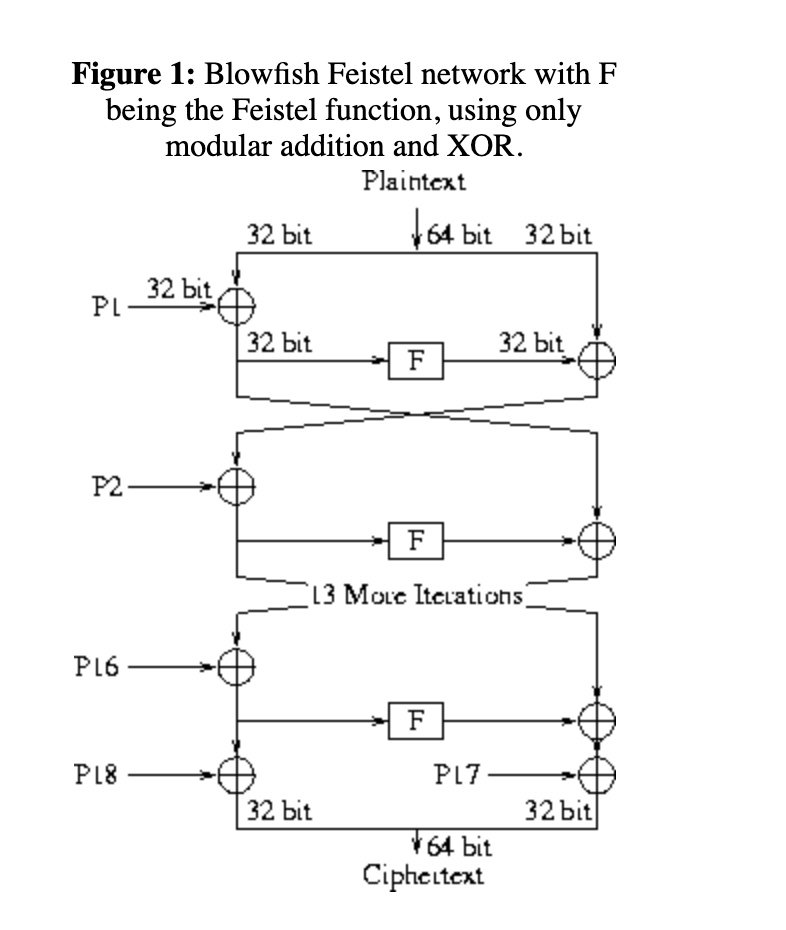
## 4.1. Thuật toán Blowfish

Blowfish là một mật mã khối 64 bit, có cấu trúc như một mạng Feistel 16 vòng. Nó sử dụng 18 khóa con 32-bit, từ P1 đến P4 mà nó lấy từ khóa mã hóa. Các khóa con được gọi chung là P-Array.

Blowfish mã hóa bằng cách chia khối đầu vào (plaintext) 64 bit thành hai nửa 32 bit, L0 và R0. Nửa quan trọng nhất, L0 được XOR với khóa con P0, và được sử dụng làm đầu vào cho một hàm F. Kết quả của hàm đó là XOR với nửa bên phải R0. Hai nửa sau đó được đổi chỗ cho nhau và toàn bộ quá trình lặp lại 15 lần nữa với tổng cộng 16 lần lặp.

Sau 16 vòng, hai nửa lại được hoán đổi (hoàn tác hiệu ứng của lần hoán đổi thứ 16) và mỗi nửa được XOR với một khóa con 32 bit khác.

Quá trình này được minh họa bằng đồ thị trong hình bên dưới:



Với 1 16, kí hiệu biểu thị XOR.

Hàm F trong Blowfish sử dụng bốn mảng, S1...S4 bắt nguồn từ khóa mã hóa. Mỗi mảng chứa 256 từ 32 bit. Các mảng hoạt động như hộp thay thế hoặc hộp S, thay thế đầu vào 8 bit bằng đầu ra 32 bit. F chia đầu vào 32 bit của nó thành bốn byte 8 bit, a, b, c và d, với một byte quan trọng nhất. Nó thay thế từng byte bằng nội dung của hộp S và kết hợp các kết quả như sau:

với biểu thị phép cộng modulo 232:

Dựa trên cơ chế hoạt động của Blowfish thuật toán Eksflowfish được trình bày lại để phù hợp với giải thuật hàm băm bcrypt. Bây giờ chúng ta sẽ xem xét qua mô tả cho quá trình sinh key dựa trên khối salt, tham số cost và khoá encrypt với tên gọi là eksblowfish. Mục tiêu chính của Eksblowfish được thiết kế để lấy mật khẩu do người dùng chọn làm khóa và chống lại các cuộc tấn công vào các khóa đó.

Mã giả của EksBlowfishSetup:

|  |
| --- |
| EksBlowfishSetup(cost, salt, key)  State 🡨 InitState()  State 🡨 ExpandKey (state, salt, key)  Repeat (2cost)  State 🡨ExpandKey(state, 0, salt)  State 🡨 ExpandKey(state, 0, key)  Return state |

EksBlowfishSetup có ba tham số đầu vào: cost, salt và key endcrypt. Nó trả về một tập hợp các khóa con và S-box, còn được gọi là *key schedule*.

Tham số cost kiểm soát mức độ tốn kém của *schedule* chính để tính toán. Salt là một giá trị 128-bit dùng để cập nhật *key schedule* để cùng một khóa không phải lúc nào cũng tạo ra kết quả giống nhau. Cuối cùng, đối số key là khóa mã hóa bí mật, có thể do người dùng chọn mật khẩu lên đến 56 byte (bao gồm byte 0 kết thúc khi khóa là chuỗi ASCII).

EksBlowfishSetup bắt đầu bằng cách gọi InitState, một hàm sao chép các chữ số của số trước tiên vào các khóa con, sau đó vào S-box.

ExpandKey (state, salt, key) sửa đổi P-Array và S-box dựa trên giá trị của salt 128 bit và khóa có độ dài thay đổi. Đầu tiên nó XOR tất cả các khóa con trong mảng P bằng khóa mã hóa. 32 bit đầu tiên của khóa được XOR với P1, 32 bit tiếp theo với P2, v.v. Khóa được sử dụng theo chu kỳ nghĩa là khi tới những bit kết thúc mà vẫn chưa XOR hết các P thì nó bắt đầu sử dụng lại các bit từ đầu để XOR với khóa con.

Sau đó, ExpandKey blowfish mã hóa 64 bit đầu tiên của đối số salt của nó bằng cách sử dụng khoá của state hiện tại. Bản mã kết quả thay thế các khóa con P1 và P2. Bản mã tương tự cũng được XOR với salt 64 bit thứ hai và kết quả được mã hóa với state mới của lịch khóa. Đầu ra của mã hóa thứ hai thay thế các khóa con P3 và P4. Nó cũng được XOR với 64-bit salt đầu tiên và được mã hóa để thay thế P5 và P6. Quá trình tiếp tục, xen kẽ giữa salt 64 bit thứ nhất và thứ hai. Khi ExpandKey hoàn thành việc thay thế các mục nhập trong P-Array, nó sẽ tiếp tục thay thế hai mục nhập S-box cùng một lúc. Sau khi thay thế hai mục nhập cuối cùng của S-box cuối cùng S4 [254] và S4 [255] ExpandKey trả về schedule key.

Trong tính toán ExpandKey (state, 0, khóa), một khối 128 bit 0 được sử dụng thay vì salt. Điều này tương đương với một lần lặp lại schedule tiêu chuẩn. Lệnh gọi tới ExpandKey (state, 0, salt) chỉ đơn giản coi salt như một khóa 16 byte.

Sau khi gọi InitState để khởi tạo một khóa mới với các chữ số của, EksBlowfishSetup gọi ExpandKey bằng salt và khóa mã hoá. Điều này đảm bảo rằng tất cả state tiếp theo phụ thuộc vào cả hai và không có phần nào của thuật toán có thể được tính toán trước mà không có cả salt và khóa. Sau đó, ExpandKey được gọi luân phiên với salt và sau đó là khóa với 2cost lần lặp. Đối với tất cả ngoại trừ lệnh gọi đầu tiên của ExpandKey, tham số thứ hai là một khối 128 bit 0. Điều này gần giống với key schedule blowfish ban đầu và cũng cho phép EksBlowfishSetup được triển khai hiệu quả hơn trên các kiến ​​trúc CPU có ít thanh ghi.

Với nội dung thay đổi và không thể đoán trước của P-array và S-Boxes sẽ làm giảm khả năng áp dụng của các tối ưu hóa chưa được biết đến. Ngoài ra, eksblowfish S-Boxes yêu cầu 4 KB bộ nhớ được truy cập và sửa đổi liên tục. Do đó, các S-Box không thể được chia sẻ trong suốt quá trình và các S-Box riêng biệt phải tồn tại cho mọi lần thực thi đồng thời. Điều này hạn chế đáng kể tính hữu ích của bất kỳ nỗ lực nào để chuyển mạng Feistel trong phần cứng.

## 4.2. Hàm băm Bcrypt

Bcrypt sử dụng salt 128 bit và mã hóa giá trị hàm băm 192 bit. Nó tận dụng lợi thế của thiết lập khóa đắt đỏ trong eksblowfish.

Thuật toán bcrypt chạy trong hai giai đoạn, được phác thảo trong mã giả bên dưới. Trong giai đoạn đầu, EksBlowfishSetup được gọi với chi phí, salt và mật khẩu, để khởi tạo state của eksblowfish. Hầu hết thời gian của bcrypt được dành cho schedule đắt tiền. Theo đó, giá trị 192-bit “OrpheanBeholderScryDoubt” được mã hóa 64 lần bằng cách sử dụng eksblowfish ở chế độ ECB với state từ giai đoạn trước. Đầu ra là giá và salt 128 bit được nối với kết quả của vòng mã hóa.

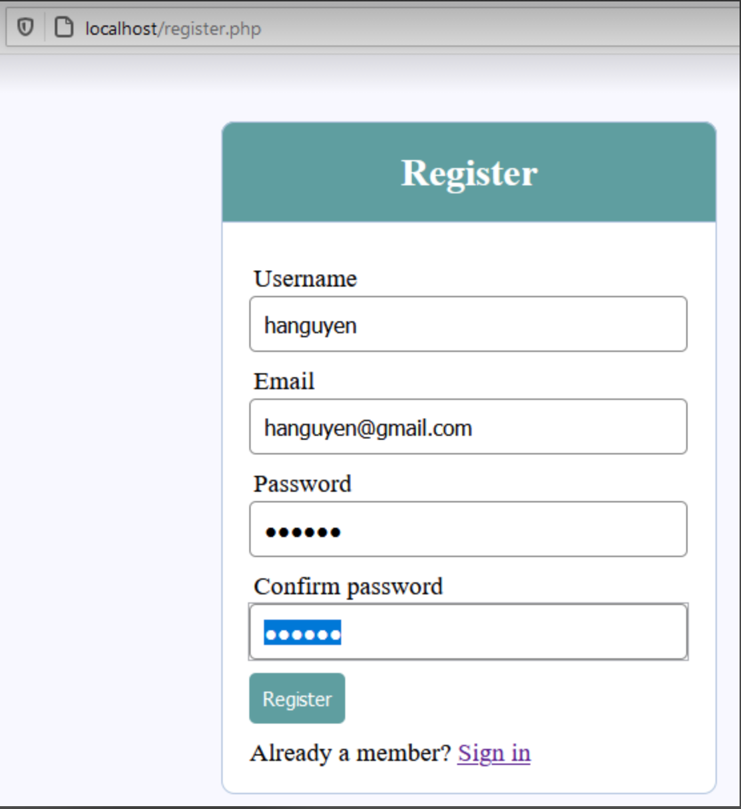
|  |
| --- |
| Bcrypt (cost, salt, pwd)  State 🡨 EksBlowfishSetup (cost, salt, key)  Ctext 🡨 “OrpheanBeholderScryDoubt”  Repeat (64)  Ctext 🡨 EncryptECB (state, ctext)  Return Concatenate (cost, salt, ctext) |

Như đã nói ở trên, một hàm băm mật khẩu an toàn phải đáp ứng một số tiêu chí quan trọng: khả năng chống tiền ảnh thứ hai (second pre-image attack), không gian salt đủ lớn để đánh bại các cuộc tấn công tính toán trước và chi phí phù hợp nhất có thể. Bcrypt đạt được cả ba thuộc tính và nó có các giá trị hữu ích trong những năm gần đây.

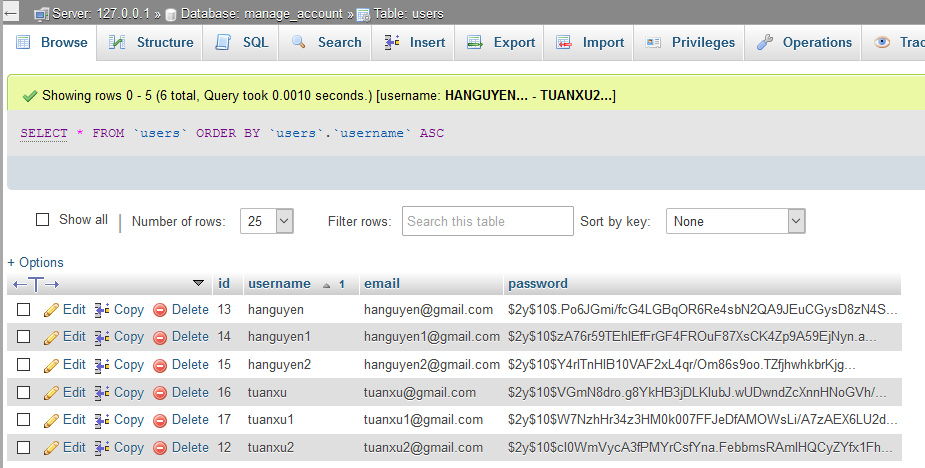
# PHẦN 5. CÀI ĐẶT VÀ THỰC NGHIỆM MÔ HÌNH LƯU MẬT KHẨU NGƯỜI DÙNG VỚI BCRYPT, MÃ HÓA THÔNG TIN TRUYỀN TẢI VỚI RSA

## 5.1. Thực nghiệm mô hình lưu mật khẩu người dùng với bcrypt

### 5.1.1. Giao diện log in



### 5.1.2. Mật khẩu sau khi đã được lưu trữ trong csdl:

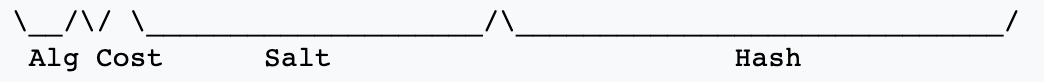


***Nhận xét:*** Phần password của người dùng đã được băm theo đúng cú pháp



*Ví dụ:*





*Cụ thể:*

$2a$: Định danh thuật toán băm (bcrypt).

10: Yếu tố chi phí cost (210 = 1,024 vòng).

SMu25HBP7jEQJcMkmF8ED: 16 byte (128-bit) Salt, được mã hóa base64 thành 22 ký tự.

9A1UzlNWrjhe1em9Jl8v7T84DdfT3tG: 24 byte (192 bit) Hash, base64 được mã hóa thành 31 ký tự.

### 5.1.3. Kiểm tra mật khẩu đăng nhập.

Do mật khẩu đã được lưu dưới dạng mã hóa, ta không thể so sánh theo cách thông thường mà phải sử dụng hàm verify riêng:

***password\_verify($password, $password\_hash\_stored)***

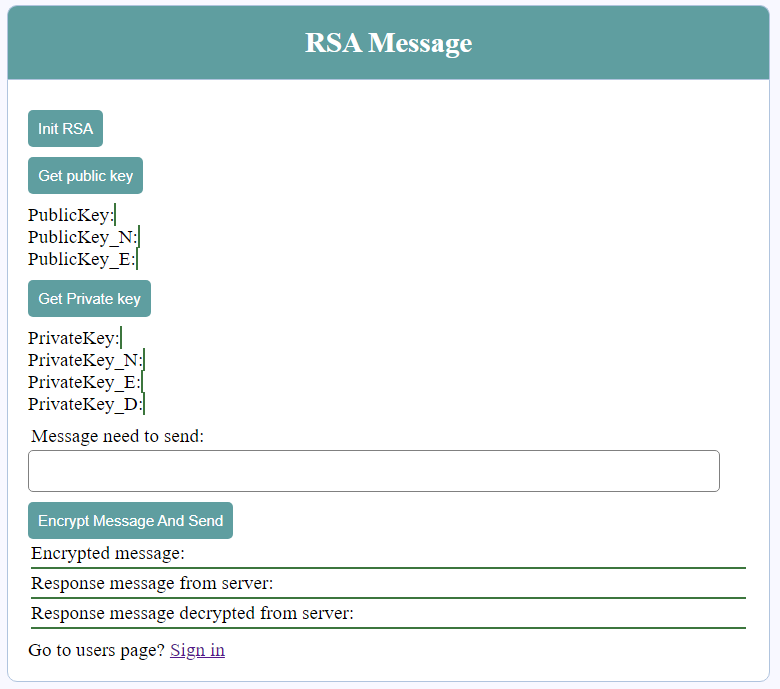
*Trong đó:*

* ***$password :*** *abc*
* ***$password\_hash\_stored :*** *$2y$10$bjOdOBQ7Y/7B4cfg9fMOg.KKh7Un9uJAxdVvNKG9s9P1EYI5taPdy*

Hàm này sử dụng lại salt để băm ra một chuỗi mới. Sau đó so sánh 2 chuỗi sau khi băm, nếu nó trùng nhau thì là đúng password.

## 5.2. Thực nghiệm mô hình mã hóa thông tin truyền tải với RSA.

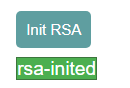
Giao diện thực nghiệm:



### 5.2.1. Khởi tạo RSA.

Nút Init RSA có tính năng khởi tạo RSA phía server. Tạo ra P,Q là 2 số nguyên tố lớn, tính N, E và D.

Khởi tạo thành công thì có thông báo:



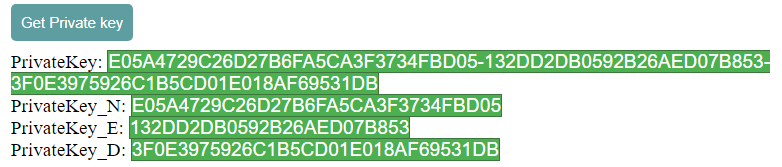
### 5.2.2. Lấy thông tin public key từ server.

Như đã nói ở phần trên N, E là public key của server và client connect tới server để lấy thông tin.



### 5.2.3. Lấy thông tin private key.

Client có thể tự tạo private key, hoặc nhập private key từ file hoặc text box. Ở đây nhóm lấy thông tin private key từ server.



### 5.2.4. Gửi thông điệp từ client tới server.

Client lấy message từ textbox, encrypt message theo public key đã lấy được (N,E) và send thông tin mã hóa cho server.

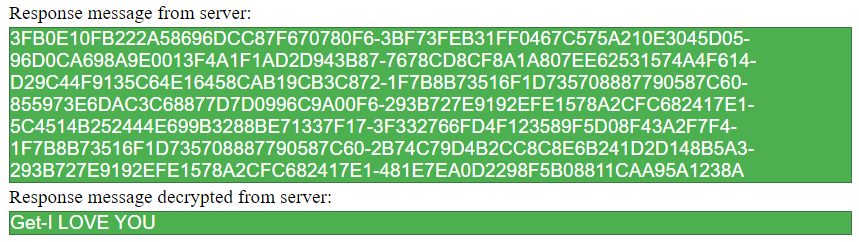


### 5.2.5. Gửi thông điệp từ server tới client.

Server nhận thông điệp đã mã hóa từ client (N,E), sau đó giải mã ra dựa vào private key của server (N,D).

Server tạo ra message mới : “GET” + message từ client (ví dụ: GET-I LOVE YOU), mã hóa theo public key của client (Nc, Ec) và gửi trả về cho client.

Client nhận thông điệp từ server, giải mã theo pivate key của client, sau đó hiển thị ra cho người dùng client.



Phần thực nghiệm dựa trên source code: *https://codewithawa.com/posts/complete-user-registration-system-using-php-and-mysql-database.*

# KẾT LUẬN

Qua đề tài này, nhóm đã học hỏi và hệ thống được rất nhiều kiến thức bổ ích từ môn học, hiểu được vai trò của mã hoá nói chung và bản chất của một hệ mã công khai RSA cũng như hàm băm BCRYPT.

Về mặt lý thuyết cũng đã hiểu được về cơ chế hoạt động của RSA, quy trình mã hoá một thông điệp M (plaint text) và giải mã một bản mã C (cipher) thông qua quá trình cài đặt trên C++.

Đồng thời cũng đã tìm hiểu về vai trò và cơ chế hoạt động, quá trình sinh key dựa trên Blowfish của hàm băm BCRYPT, cài đặt được chúng trên một ứng dụng cụ thể.

Hiểu và phân biệt được nét khác biệt rõ nhất giữa mật mã (hàm song ánh) và hàm băm (hàm đơn ánh) và ứng dụng nó vào bài toán thực tế.

Tuy nhiên, về kỹ thuật lập trình bản thân nhóm cũng chỉ dừng lại ở mức cài đặt thư viện Bcrypt sẵn có thay vì xây dựng một hàm băm theo từng bước cụ thể.

# TÀI LIỆU THAM KHẢO

***Tiếng Việt:***

[1] Bùi Doãn Khanh, Nguyễn Đình Thúc, Giáo trình “*Mã hoá thông tin- Lý thuyết và Ứng dụng*”,NXB Lao động xã hội, 2004.

[2] Dương Anh Đức, Trần Minh Triết, Giáo trình “*Mã hoá và Ứng dụng*”, 2005.

[3] Nguyễn Đình Thúc, Bùi Doãn Khanh, “*Mã hoá thông tin với Java – Mã hoá – Mật mã*”, NXB Lao động xã hội, 2006.

[4] website: https://ssl.vn

[5] Link Source tham khảo: *https://codewithawa.com/posts/complete-user-registration-system-using-php-and-mysql-database*

***Tiếng Anh:***

[6] Niels Provos and David Mazierer, *A future-Adaptable Password Scheme*, The OpenBSD Project, 1976.