



# Kryptografia stosowana

---

Atak Coppersmitha

Adam Mielewski  
Bartłomiej Roszczyk  
Paweł Sołowiej  
Adrian Zalewski  
Wiktor Zawadzki

## Spis treści

1. Problem faktoryzacji .....	3
2. RSA .....	3
2.1. Definicje .....	3
2.2. Opis działania .....	3
2.2.1. Generowanie kluczy .....	3
2.2.2. Szyfrowanie .....	4
2.2.3. Deszyfrowanie .....	4
2.3. Problem RSA .....	4
2.4. Faktoryzacja RSA .....	5
2.5. Podsumowanie .....	5
3. Kryptografia oparta na kratkach .....	6
3.1. Definicja kraty .....	6
3.2. Redukcja LLL .....	7
3.2.1. Właściwości redukcji LLL .....	8
3.2.2. Pseudoalgorytm redukcji LLL .....	8
4. Atak Coppersmitha .....	9
4.1. Opis teoretyczny i specyfika ataku .....	9
4.2. Warunki ataku .....	10
4.3. Praktyczne zastosowania i przykład działania .....	10
4.3.1. Stereotyped Messages .....	10
4.3.2. Factoring $N$ with high bits known .....	11
4.3.3. Podatność ROCA .....	11
4.4. Implementacja ataku .....	13
4.4.1. Faktoryzacja $n$ z częściową znajomością jednej z liczb pierwszych .....	13
4.4.2. Stereotyped messages .....	14
Bibliografia .....	15

# 1. Problem faktoryzacji

Faktoryzacja liczby  $N$  polega na znalezieniu jej dwóch czynników  $p$  i  $q$ , które są liczbami pierwszymi.

Problem staje się coraz trudniejszy im wybrano większe liczby  $p$  i  $q$ . Liczba możliwych kombinacji rośnie wykładniczo wraz ze wzrostem liczby  $N$ . Jeśli  $N$  jest bardzo dużą liczbą, zadanie to staje się ekstremalnie trudne i czasochłonne dla klasycznych superkomputerów.

Faktoryzacja jest trudna z powodu braku znanych algorytmów, które mogłyby rozwiązać problem w czasie wielomianowym. Najszybszy znany algorytm faktoryzacji - **GNFS** (General Number Field Sieve) posiada złożoność  $O\left(e^{c(\log(N))^{\frac{1}{3}}(\log(\log(N)))^{\frac{2}{3}}}\right)$ . Jest to zredukowana, lecz nadal wykładnicza złożoność.

## 2. RSA

Algorytm RSA to najpopularniejszy algorytm kryptografii asymetrycznej. Jest wykorzystywany w szyfrowaniu, podpisach cyfrowych i uwierzytelnianiu. Jego bezpieczeństwo opiera się na trudności problemu faktoryzacji dużej liczby  $N = pq$ , gdzie  $p$  i  $q$  to duże liczby pierwsze.

### 2.1. Definicje

Symbol	Opis
$s$	parametr bezpieczeństwa - długość klucza w bitach
$\text{Gen}(1^s)$	bezpieczny algorytm generowania klucza, argumentem jest parametr bezpieczeństwa
$\varphi(n)$	funkcja Eulera (tocjent)
$\mathcal{A}$	adwersarz

**Problem trudny** - problem obliczeniowy z parametrem bezpieczeństwa  $s$  jest łatwy, jeśli może zostać rozwiązany przez algorytm probabilistyczny o złożoności  $s^{O(1)}$ , gdy  $s \rightarrow \infty$ , tzn. algorytm probabilistyczny o wielomianowej złożoności czasowej (PPT)  $\rightarrow$  łatwy = PPT (w przeciwnym razie jest trudny).

### 2.2. Opis działania

Opis skupia się na elemencie szyfrowania i deszyfrowania algorytmem RSA.

#### 2.2.1. Generowanie kluczy

W celu rozpoczęcia pracy z algorytmem należy wygenerować dwa klucze - prywatny oraz publiczny. Klucze są ze sobą powiązane własnościami matematycznymi.

- **klucz publiczny** - klucz udostępniany wszystkim, dzięki niemu druga strona może zaszyfrować wiadomość, która może być odszyfrowana wyłącznie powiązanym kluczem prywatnym
- **klucz prywatny** - nie należy udostępniać go nikomu, poznanie klucza prywatnego umożliwia złamanie totalne algorytmu RSA i odszyfrowanie wszystkich wiadomości

Algorytm  $\text{Gen}(1^s) \rightarrow ((p, q), e, d)$  generuje:

#### 1. Wartości $p$ i $q$ , które są:

- liczbami pierwszymi o długości bitowej  $s$  (obecnie min. 2048 bitów),
- podobnej długości bitowej (ok.  $\frac{n}{2}$  bitów każda),
- odpowiednio daleko od siebie,
- $(p - 1)$  oraz  $(q - 1)$  nie posiadają małych dzielników.

#### 2. Liczbę $e$ - **wykładnik publiczny**, który jest względnie pierwszy z $\varphi(N)$ . Obecnie zaleca się stosowanie $e = 65537$ . Jest to największa znana liczba pierwsza Fermata postaci $2^{2^n} + 1$ . Wybór tego wykładnika posiada wiele zalet:

3. **Liczbę  $d$  - wykładnik prywatny**, który jest odwrotnością modularną wykładnika  $e$  modulo  $\varphi(N)$ .

Problemem trudnym dla stron nieznających rozkładu  $N = p \cdot q$  jest obliczenie wartości funkcji Eulera  $\varphi(N)$ :

którą można uprościć do:

Ostatecznie otrzymujemy parę kluczy wygenerowanych przez algorytm  $Gen(1^s)$ :

- Klucze te są matematycznie powiązane i stanowią nierozdzielalną parę do szyfrowania oraz deszyfrowania.

Korzystając z wygenerowanych kluczy możemy zaszyfrować wiadomość używając klucza publicznego. Taką wiadomość odszyfrować może wyłącznie osoba znająca klucz publiczny.

- $m$  - wiadomość (plaintext)
- $c$  - zaszyfrowana wiadomość (ciphertext)

$$c = m^e \bmod N \tag{2.4}$$

Posiadając **klucz prywatny (N,d)** powiązany z kluczem publicznym, którym zaszyfrowana została wiadomość, deszyfracji dokonujemy w następujący sposób:

$$m = c^d \bmod N \tag{2.5}$$

Posiadając  $(N, e)$  wygenerowane przez algorytm *Gen* oraz losową resztę  $y \in Z_N$  należy obliczyć  $x$ , takie że  $y = x^e \bmod N$ .

1.  $\text{Gen}(1^s) \rightarrow (N, e)$
2. pick  $x \in Z_N$
3.  $y = x^e \bmod N$
4.  $\mathcal{A}(N, e, y) \rightarrow z$
5. **return**  $1_{x=z}$

Powyższy problem uważany jest jako trudny. RSA jest bezpieczne pod względem deszyfracji pod założeniem CCA (Chosen Ciphertext Attack) - decryption under CCA.

## 2.4. Faktoryzacja RSA

Problem polega na faktoryzacji liczby  $N$  wygenerowanej przez algorytm  $Gen$ .

**Gra**

1.  $Gen(1^s) \rightarrow N$
2.  $\mathcal{A}(N) \rightarrow (p, q)$
3. **return**  $1_{1 < p, q < N, N=pq}$

Powyższy problem uważany jest jako trudny. RSA jest bezpieczne pod względem odzyskania klucza pod założeniem (Chosen Plaintext/Ciphertext Attack) - key recovery under CPCA.

Domyślny algorytm RSA niestety nie jest bezpieczny przeciwko odróżnianiu (IND-CCA security), ponieważ jest algorytmem deterministycznym. Rozszerzone wersje RSA np. RSA-OAEP jest krypto-systemem bezpiecznym przeciwko IND-CCA.

## 2.5. Podsumowanie

Algorytm RSA jest uważany jako bezpieczny, ponieważ opiera swoje działanie na **Problemie RSA** oraz **Problemie Faktoryzacji**, które są obecnie uważane jako trudne.

Wyzwanie stanowią komputery kwantowe, które dzięki algorytmowi Shora są w stanie rozwiązać **Problem faktoryzacji** w czasie wielomianowym, co detronizuje go jako problem trudny.

### 3. Kryptografia oparta na kratkach

Kryptografia oparta na kratkach (*lattice-based cryptography*) opiera swoje działanie na problemach związanych z kratami, takich jak SVP, CVP, LWE, które są odporne na znane ataki kwantowe.

#### 3.1. Definicja kraty

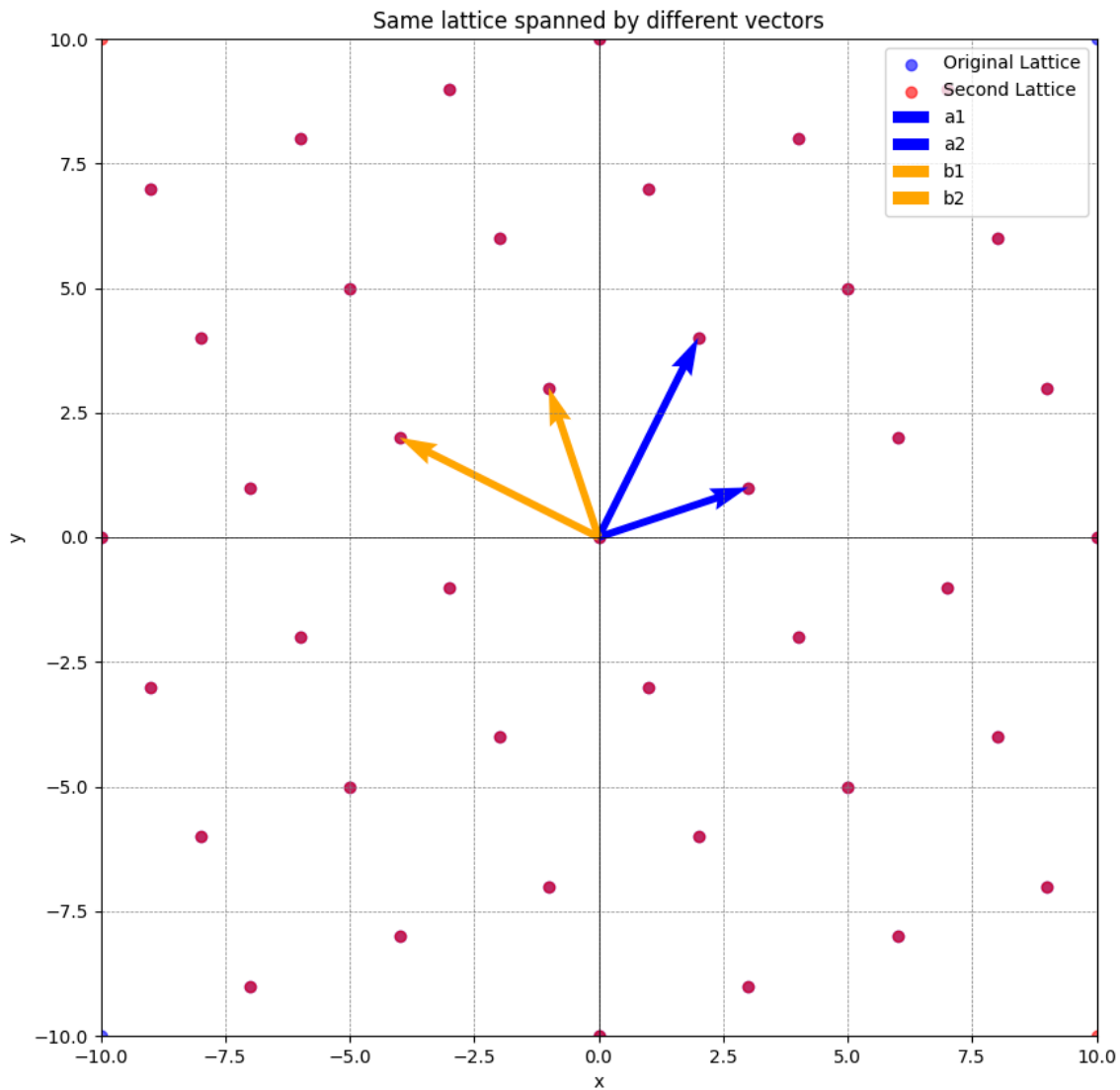
Kratę  $\mathcal{L}$  definiuje się jako dyskretną podgrupę  $\mathbb{R}^n$ , która rozpiną  $\mathbb{R}^n$  współczynnikami rzeczywistymi. Natomiast ten dokument ogranicza się do krat takich że  $\mathcal{L} \subseteq \mathbb{Z}^n$ . [1]

Bazę  $\mathcal{L}$  definiujemy jako wektor  $\mathcal{B} = (b_1, b_2, \dots, b_n)$  taki, że:

$$\mathcal{L} = \mathcal{L}(\mathcal{B}) = \mathcal{B} \cdot \mathbb{Z}^n = \left\{ \sum_{i=1}^n c_i b_i : c_i \in \mathbb{Z} \right\} \quad (3.1)$$

Przy czym wektory  $b_i (1 \leq i \leq n)$  są liniowo niezależne.

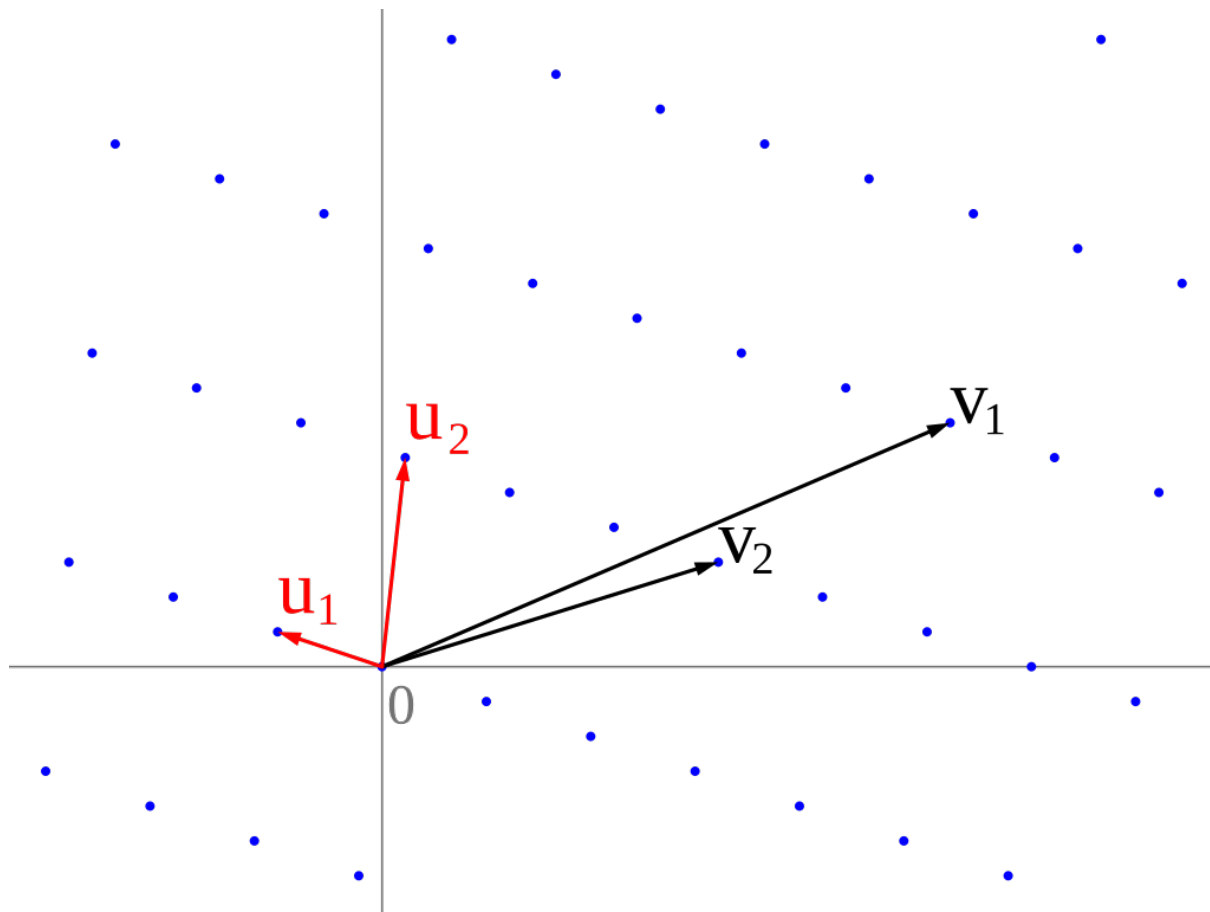
Dla przykładu mamy dwa wektory takie, że  $a_1 = [3, 1]$ ,  $a_2 = [2, 4]$ , które rozpinają kratę  $\mathcal{L}$ . Ponadto tę samą kratę  $\mathcal{L}$  rozpinają zupełnie inne wektory  $b_1 = (-1, 3)$ ,  $b_2 = (-4, 2)$ .



Rysunek 1: Przykład tej samej kraty  $\mathcal{L}$  rozpinanej przez różne wektory

### 3.2. Redukcja LLL

Celem redukcji bazy kraty  $\mathcal{B}_1$  jest znalezienie takiej bazy  $\mathcal{B}_2$  tej samej kraty, z krótszymi, niemal ortogonalnymi wektorami. Algorytmy wykorzystywane w tym celu mają zazwyczaj złożoność czasową  $O(2^n)$ , gdzie  $n$  oznacza wymiar kraty. Przytoczonym przykładem będzie tu algorytm LLL (Lenstra–Lenstra–Lovász).[2]



Rysunek 2: Redukcja kraty w dwóch wymiarach: czarne wektory stanowią pierwotną bazę, a czerwone są zredukowaną bazą.

Redukcję LLL można zdefiniować w następujący sposób: mając bazę  $\mathcal{B} = \{b_1, \dots, b_n\}$ , definiujemy bazę ortogonalną powstałą w wyniku przeprowadzenia ortogonalizacji Grama-Schmidta  $\mathcal{B}^* = \{b_1^*, \dots, b_n^*\}$  oraz współczynniki Grama-Schmidta  $\mu_{i,j}$  dla każdego  $1 \leq j \leq i \leq n$ . Wtedy baza  $\mathcal{B}$  jest LLL-zredukowana jeśli istnieje parametr  $\delta$ , taki że  $\delta \in (\frac{1}{4}, 1]$  oraz:

1. Dla każdego  $1 \leq j \leq i \leq n$  :  $|\mu_{i,j}| \leq \frac{1}{2}$  (gwarantuje to redukcję długości)
2. Dla każdego  $k \in \{2, n\}$  :  $\delta \|b_{k-1}^*\|^2 \leq \|b_k^*\|^2 + \mu_{k,k-1}^2 \|b_{k-1}^*\|^2$  (warunek Lovásza)

Estymując wartość  $\delta$  można określić jak dobrze zredukowana jest baza. Większe wartości  $\delta$  prowadzą do silniejszej redukcji. Przy czym warto zaznaczyć że złożoność wielomianowa jest możliwa tylko dla  $\delta \in (\frac{1}{4}, 1]$ .

### 3.2.1. Właściwości redukcji LLL

Niech  $\mathcal{B} = \{b_1, \dots, b_n\}$  będzie LLL-zredukowaną bazą kraty  $\mathcal{L}$ . Wówczas dla redukcji LLL można określić poniższe właściwości

1. Pierwszy wektor bazy nie może być znacznie większy niż najkrótszy niezerowy wektor, tzn.:

$$\|b_1\| \leq \left( \frac{2}{\sqrt{4\delta - 1}} \right)^{n-1} \cdot \lambda_1(\mathcal{L}) \quad (3.2)$$

2. Pierwszy wektor jest też ograniczony przez wyznacznik kraty:

$$\|b_1\| \leq \left( \frac{2}{\sqrt{4\delta - 1}} \right)^{\frac{n-1}{2}} \cdot \det(\mathcal{L})^{\frac{1}{n}} \quad (3.3)$$

3. Iloczyn norm wektorów bazy nie może być większy niż wyznacznik kraty:

$$\prod_{i=1}^n \|b_i\| \leq 2^{\frac{n(n-1)}{4}} \cdot \det(\mathcal{L}) \quad (3.4)$$

### 3.2.2. Pseudoalgorytm redukcji LLL

Niech  $B = \{v_1, \dots, v_n\}$  będzie bazą kraty  $\mathcal{L}$  takiej że  $\mathcal{L} \subseteq \mathbb{Z}^n$ . Algorytm opisany poniżej wykonuje się w skończonej liczbie kroków i zwraca LLL-zredukowaną bazę. Dla  $B = \max(\|v_i\|)$  algorytm wykonuje główną pętlę  $k$  nie więcej niż  $O(n^2 \log(n) + n^2 \log(B))$  razy.

```
1  Input a basis {v1,..., vn} for a lattice L
2  Set k = 2
3  Set v1 = v1 *
4  Loop while k ≤ n
5      Loop Down j = k - 1, k - 2, ..., 2, 1
6          Set v[k] = v[k] - [μ-(k,j)]v[j]
7      End j Loop
8      If ||vk||2 ≥ (3/4 - μ) ||v-(k-1)||2
9          Set k = k + 1
10     Else
11         Swap vk-1 and vk
12         Set k = max(k - 1, 2)
13     End If
14 End k Loop
15 Return LLL reduced basis {v1,..., vn}
```

Program 1: Pseudokod algorytmu redukcyjnego LLL [3]



## 4. Atak Coppersmitha

### 4.1. Opis teoretyczny i specyfika ataku

Metoda Coppersmitha, zaproponowana przez Dona Coppersmitha, jest metodą znajdowania miejsc zerowych wielomianów modulo dana liczba całkowita. Metoda ta wykorzystuje algorytm redukcji Lenstra–Lenstra–Lovász (LLL) do znalezienia wielomianu, który ma te same miejsca zerowe co docelowy wielomian, ale mniejsze współczynniki.

Podejście Coppersmitha polega na sprowadzeniu rozwiązywania modularnych równań wielomianowych do rozwiązywania wielomianów nad liczbami całkowitymi.

$$\text{niech } f(x) = x^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0 \quad (4.1)$$

$$f(x_0) \equiv 0 \pmod{N} \text{ dla } |x_0| < N^{\frac{1}{n}} \quad (4.2)$$

Znajdowanie pierwiastków wielomianów w ciele liczb wymiernych  $Q$  jest względnie szybkie, na przykład przy użyciu metody Newtona, jednak taki algorytm nie działa w przypadku wielomianów modulo  $N$ . Idea stojąca za metodą Coppersmitha polega na znalezieniu innego wielomianu  $g$  związanego z  $f$ , który ma ten sam pierwiastek modulo  $N$ , ale posiada tylko małe współczynniki. Jeśli współczynniki oraz  $x_0$  są na tyle małe, że  $|g(x_0)| < N$  nad liczbami całkowitymi, wtedy mamy  $g(x_0) = 0$  co oznacza, że  $x_0$  jest pierwiastkiem funkcji  $g$  w  $Q$  i może być łatwo znaleziony.

Algorytm Coppersmitha wykorzystuje algorytm redukcji bazy sieci Lenstra–Lenstra–Lovász (LLL) do skonstruowania wielomianu  $g$  o małych współczynnikach.

**Główne kroki algorytmu:**

**1. Definicja kraty:**

- Wielomian  $f(x)$  jest przekształcany w wektor współczynników. Zbiorczo rozważa się kilka wielomianów powstałych przez pomnożenie  $f(x)$  przez potęgi  $x$  i  $M$ . Tworzona jest macierz bazowa kraty na podstawie tych wielomianów.

**2. Redukcja bazy krat:**

- Na macierzy bazowej stosuje się algorytm LLL, który znajduje krótkie wektory w kratce. Każdy wektor odpowiada wielomianowi z mniejszymi współczynnikami.

**3. Znajdowanie pierwiastków:**

- Pierwszy wektor zredukowanej bazy odpowiada wielomianowi  $g(x)$ , który zachowuje pierwiastki  $x_0$  oryginalnego  $f(x)$ . Dzięki temu pierwiastki  $x_0$  mogą zostać odnalezione za pomocą standardowych metod, takich jak interpolacja lub metoda Newtona.

**Konstrukcja wielomianu  $g(x)$**

1. Rozważamy wielomian  $f(x)$  o stopniu  $d$ , taki że:

$$f(x_0) \equiv 0 \pmod{N} \quad (4.3)$$

gdzie  $x_0$  to mały pierwiastek, czyli  $|x_0| < X$

2. Tworzymy nowy wielomian  $g(x)$  jako kombinację liniową innych wielomianów:

$$h_{i,j} = x^j N^i f^{m-i}(x) \quad (4.4)$$

gdzie  $m$  to wybrany parametr

Można zauważyć, że każdy pierwiastek  $x_0$  wielomianu  $f(x)$  spełnia również  $h_{i,j}(x_0) = 0 \pmod{N^m}$ . Łącząc te wielomiany w odpowiedni sposób, możemy uzyskać wielomian  $g(x)$ .

**Kraty i redukcja bazy** - Wielomian  $g(x)$  jest reprezentowany przez współczynniki jako wektor w kracie. Kombinacja liniowa wielomianów  $h_{i,j}$  tworzy bazę kraty  $L$ , a algorytm redukcji baz (LLL) pozwala znaleźć wektor z małymi współczynnikami.

**Warunek na  $g(x)$**  - Jeśli współczynniki  $g(x)$  są wystarczająco małe, to dla wszystkich  $|x_0| < X$ :

$$g(x_0) = 0 \pmod{N^m} \quad \text{ i } \quad |g(x_0)| < N^m \quad (4.5)$$

To implikuje, że  $g(x_0) = 0$  w dziedzinie liczb całkowitych.

## 4.2. Warunki ataku

Metoda Coppersmitha jest głównie wykorzystywana w atakach na RSA, gdy:

- Znana jest część wiadomości
- Znane jest przybliżenie jednej z liczb pierwszych z których składa się  $N$
- Publiczny wykładnik jest zbyt mały.

Niech  $N$  będzie liczbą całkowitą o nie znanych czynnikach która ma dzielnik  $b \geq N^\beta, 0 < \beta \leq 1$ .

Niech  $f(x)$  będzie wielomianem o stopniu  $\delta$  i niech  $c \geq 1$ .

Wtedy w czasie  $O(c\delta^5 \log^9(N))$  można znaleźć rozwiązania  $x_0$  równania:

$$f(x) = 0 \pmod{b} \text{ dla } |x_0| \leq cN^{\frac{\beta^2}{\delta}} \quad (4.6)$$

## 4.3. Praktyczne zastosowania i przykład działania

Uwzględniając opisane warunki, konieczne aby atak Coppersmitha był skuteczny, potencjalne praktyczne zastosowania to:

- ataki na ograniczone lub zawierające błędy implementacje kryptosystemu RSA,
- połączenie z innymi atakami np. side-channel lub fault-injection, które ujawniają dodatkowe informacje,

Potencjalnymi celami są więc urządzenia optymalizowane pod kątem ograniczonych zasobów sprzętowych, takie jak Moduły TPM (Trusted Platform Module), smartcardy i karty SIM, bezpieczne moduły sprzętowe (HSM),

### 4.3.1. Stereotyped Messages

Stereotypowa wiadomość to taka, która ma przewidywalny wzorec lub strukturę, np. „YOUR PASSWORD IS: {XXXXXX}”. Kiedy takie wiadomości są szyfrowane za pomocą RSA, szczególnie w najprostszej formie  $C = M^e \pmod{n}$  atakujący może wykorzystać przewidywalność wiadomości, aby odzyskać tekst jawny  $M$ . W tym celu wykorzystuje się metodę Howgrave-Grahama, która sama wykorzystuje metodę Coppersmitha oraz redukcję LLL. Niech  $M_0$  oznacza znany fragment tekstu jawnego  $M$ , a  $x$  część nieznana:

$$M = M_0 + x \quad (4.7)$$

Wówczas operację szyfrowania można zapisać jako:

$$C = (M_0 + x)^e \pmod{N} \quad (4.8)$$

Powyższe równanie można przekształcać w postać wielomian względem  $x$ :

$$f(x) = (M_0 + x)^e - C \quad (4.9)$$

Jeśli  $x_0$  jest odpowiednio małe, takie że  $x_0 < N^{\frac{1}{e}}$  to można je szybko obliczyć. Niech  $t = \deg(f(x))$  oraz  $x < X$  dla pewnego znanego  $X$ , wówczas chcemy znaleźć takie  $g(x)$  i pierwiastek  $x_0$ , że  $g(x_0) \in$

$n\mathbb{Z}$ . Z racji tego, że  $f(x)$  mamy już zdefiniowane oraz wiemy że  $f(x_0) \in n\mathbb{Z}$  dla szukanego  $x_0$ . Zatem  $g(x)$  można określić tak jak poniżej:

$$g(x) \in n\mathbb{Z} + nx\mathbb{Z} + nx^2\mathbb{Z} + nx^{t-1}\mathbb{Z} + f(x)\mathbb{Z} \quad (4.10)$$

W celu odnalezienia  $g(x)$  wykorzystany zostanie algorytm LLL, założmy że  $e = 3$  wówczas  $f(x) = (M_0 + x)^3 - C$ . Wówczas macierz współczynników takiego wielomianu można rozwinąć w następujący sposób:

$$M = \begin{pmatrix} X^3 & 3X^2M_0 & 3XM_0^2 & M_0^3 - C \\ 0 & nX^2 & 0 & 0 \\ 0 & 0 & nX & 0 \\ 0 & 0 & 0 & n \end{pmatrix} \quad (4.11)$$

Na tak powstałej macierzy przeprowadzamy redukcję LLL. Na podstawie nowej macierzy i jej najkrótszego wektora tworzymy wielomian  $g(x)$ . Na końcu należy znaleźć pierwiastki takiego wielomianu, znaleziony pierwiastek  $x_0$  jest brakującą częścią szyfrogramu:  $M = M_0 + x_0$

#### 4.3.2. Factoring N with high bits known

Atak opiera się na założeniu że atakujący zna  $m$  najstarszych bitów jednej z liczb pierwszych (których iloczyn wynosi  $n$ ). Czyli znamy np. aproksymację  $p'$  (jako wzorec bitów) liczby pierwszej  $p$ . Założmy że naszą liczbę pierwszą  $p$  możemy wyrazić następująco:

$$p = p' + r \quad (4.12)$$

Gdzie  $r$  jest nieznaną częścią reszty bitów. Na tej podstawie możemy określić następujący wielomian:

$$f(x) = p' + x \quad (4.13)$$

Celem jest znalezienie pierwiastka  $x_0$  powyższego wielomianu. Na początku należy zdefiniować granicę  $X$  taką, że  $r \leq X$ . Kolejnym krokiem jest zdefiniowanie macierzy odpowiadającej współczynnikom następujących wielomianów:  $Xf(x)$ ,  $f(X)$ ,  $n$ .

$$M = \begin{pmatrix} X^2 & Xp' & 0 \\ 0 & X & p' \\ 0 & 0 & n \end{pmatrix} \quad (4.14)$$

Następnie należy dokonać redukcji LLL na macierzy  $M$ . Na podstawie nowej macierzy i jej najkrótszego wektora tworzymy wielomian  $g(x)$ . Pozostało jedynie obliczyć pierwiastki  $x_i$  wielomianu  $g(x)$  i sprawdzić czy  $p' + x_i$  dzieli  $n$ .

#### 4.3.3. Podatność ROCA

ROCA (Return of Coppersmith's Attack) [4], [5] to podatność odkryta w 2017 roku, wynikająca z podejścia do generowania kluczy RSA stosowanego w podatnych wersjach biblioteki oprogramowania RSALib dostarczanej przez Infineon Technologies i wbudowanej w wiele kart inteligentnych, modułów Trusted Platform Modules (TPM) i modułów zabezpieczeń sprzętowych (HSM), w tym tokenów YubiKey.

W bibliotece, do wygenerowania modułu  $N$  używane były liczby pierwsze  $p$ ,  $q$  postaci:

$$p = k \cdot M + (65537^a \bmod M) \quad (4.15)$$

gdzie parametry  $k$  i  $a$  są nieznanne.  $M$  jest liczbą pierwotną (iloczynem pierwszych  $n$  kolejnych liczb pierwszych,  $n$  zależne od długości klucza).

Klucze publiczne wygenerowane w ten sposób można szybko rozpoznać, próbując obliczyć dyskretny logarytm klucza publicznego mod  $M$  przy podstawie 65537 używając algorytmu Pohliga–Hellmana, ponieważ  $M$  jest liczbą gładką.

Klucze wygenerowane w ten sposób mają mniejszą entropię i klucz prywatny może być odzyskany za pomocą wariantu metody Coppersmitha.

## 4.4. Implementacja ataku

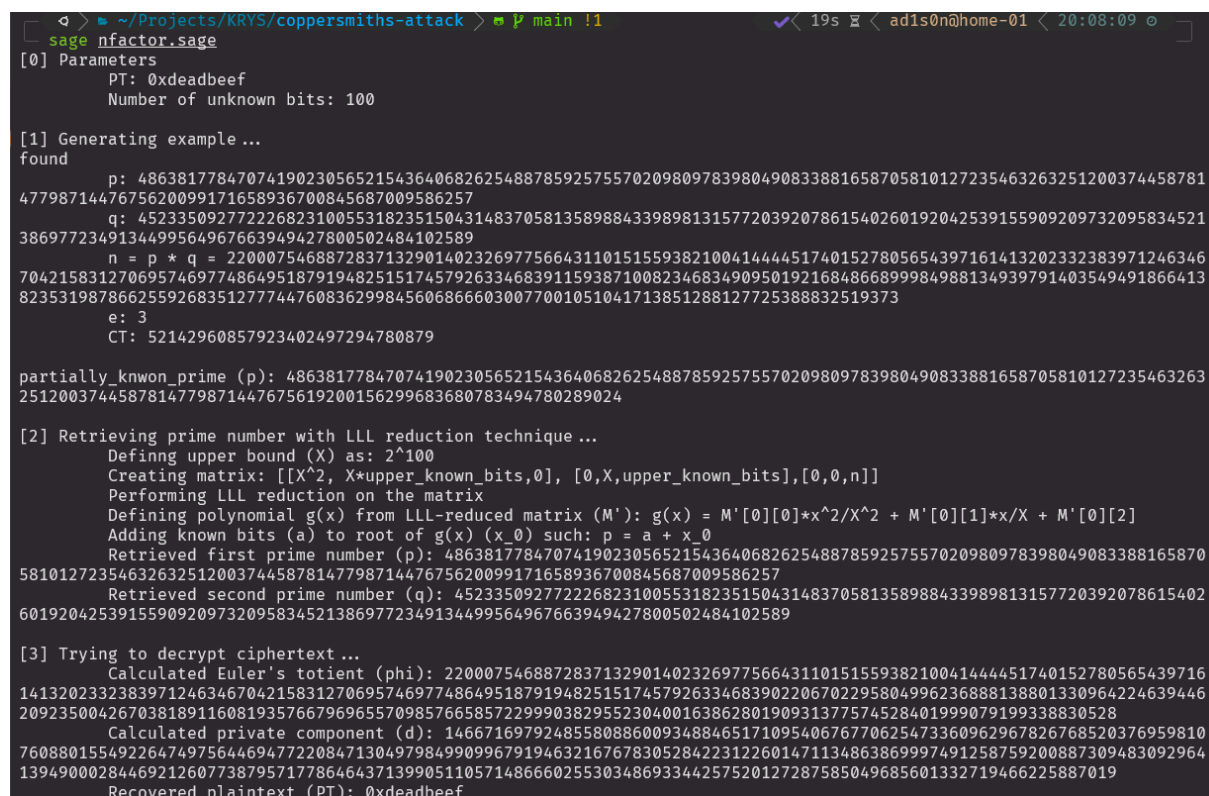
Pełną implementację można znaleźć w repozytorium GitHub pod tym linkiem: <https://github.com/adi7312/coppersmiths-attack/tree/main>

### 4.4.1. Faktoryzacja $n$ z częściową znajomością jednej z liczb pierwszych

Zakładamy że jako atakujący znamy  $m$  pierwszych bitów jednej z liczb pierwszych. Pełny opis teoretyczny tego ataku znajduje się w sekcji 4.4.2. Przykładowy atak został zaimplementowany przy pomocy Sage. Poniżej znajduje się funkcja odpowiedzialna za odzyskanie liczby pierwszej  $p$ . Pełny kod załączony jest na końcu dokumentu.

```
1 def retrieve_prime(n, upper_known_bits, number_of_bits):
2     """
3     Based on Coppersmith Method - utilizes LLL reduction
4     number_of_bits - this is number of UNKNOWN bits!
5     """
6
7     X = 2^(number_of_bits)
8     M = matrix( [[X^2, X*upper_known_bits, 0], [0, X, upper_known_bits], [0, 0, n]] )
9     M_LLL = M.LLL()
10    Q = M_LLL[0][0]*x^2/X^2+M_LLL[0][1]*x/X + M_LLL[0][2]
11
12    p = upper_known_bits + Q.roots(ring=ZZ)[0][0]
13    q = n / p
14
15    return p,q
```

Program 2: Funkcja odpowiedzialna za odzyskanie liczb pierwszych



```
< ~/Projects/KRYS/coppersmiths-attack > P main !1
sage nfactor.sage
[0] Parameters
    PT: 0xdeadbeef
    Number of unknown bits: 100

[1] Generating example...
found
    p: 486381778470741902305652154364068262548878592575570209809783980490833881658705810127235463263251200374458781
4779871447675620099171658936700845687009586257
    q: 45233509277226823100553182351504314837058135898843398981315772039207861540260192042539155909209732095834521
3869772349134499564967663949427800502484102589
    n = p * q = 220007546887283713290140232697756643110151559382100414444517401527805654397161413202332383971246346
704215831270695746977486495187919482515174579263346839115938710082346834909501921684866899984988134939791403549491866413
82353198786625592683512777447608362998456068666030077001051041713851288127725388832519373
    e: 3
    CT: 52142960857923402497294780879

partially_kwon_prime (p): 486381778470741902305652154364068262548878592575570209809783980490833881658705810127235463263
2512003744587814779871447675619200156299683680783494780289024

[2] Retrieving prime number with LLL reduction technique...
Defining upper bound (X) as: 2^100
Creating matrix: [[X^2, X*upper_known_bits, 0], [0, X, upper_known_bits], [0, 0, n]]
Performing LLL reduction on the matrix
Defining polynomial g(x) from LLL-reduced matrix (M'): g(x) = M'[0][0]*x^2/X^2 + M'[0][1]*x/X + M'[0][2]
Adding known bits (a) to root of g(x) (x_0) such: p = a + x_0
Retrieved first prime number (p): 48638177847074190230565215436406826254887859257557020980978398049083388165870
58101272354632632512003744587814779871447675620099171658936700845687009586257
Retrieved second prime number (q): 452335092772268231005531823515043148370581358988433989813157720392078615402
601920425391559092097320958345213869772349134499564967663949427800502484102589

[3] Trying to decrypt ciphertext...
Calculated Euler's totient (phi): 22000754688728371329014023269775664311015155938210041444451740152780565439716
141320233238397124634670421583127069574697748649518791948251517457926334683902206702295804996236888138801330964224639446
209235004267038189116081935766796965570985766585722999038295523040016386280190931377574528401999079199338830528
Calculated private component (d): 14667169792485580886009348846517109540676770625473360962967826768520376959810
76088015549226474975646947722084713049798499099679194632167678305284223122601471134863869997491258759200887309483092964
139490002844692126077387957177864643713990511057148666025530348693344257520127287585049685601332719466225887019
Recovered plaintext (PT): 0xdeadbeef
```

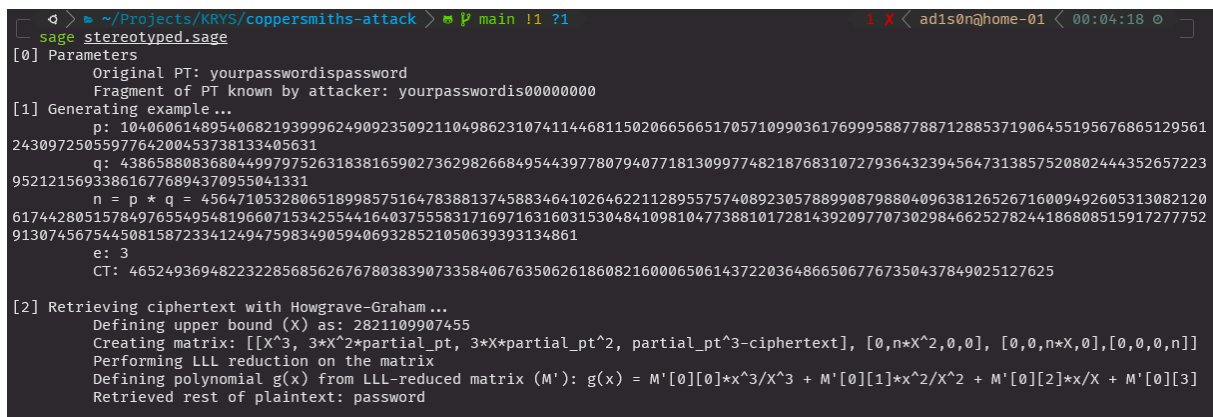
Rysunek 3: Wynik uruchomienia programu

#### 4.4.2. Stereotyped messages

Zakładamy że atakujący ma częściową znajomość dotyczącą tekstu jawnego w przypadku poniższym znana jest część „yourpasswordis00000000”, gdzie zera oznaczają część nieznaną atakującemu. Poniższy przykład działa dla przypadku gdy  $e = 3$ .

```
1 def retrieve_plaintext(partial_pt, n, upper_bound, ciphertext):
2     X = upper_bound
3     M = matrix(
4         [[X^3, 3*X^2*partial_pt, 3*X*partial_pt^2, partial_pt^3-ciphertext],
5          [0,n*X^2,0,0],
6          [0,0,n*X,0],
7          [0,0,0,n]]
8     )
9     M_LLL = M.LLL()
10    Q = M_LLL[0][0]*x^3/X^3+M_LLL[0][1]*x^2/X^2+M_LLL[0][2]*x/X + M_LLL[0][3]
11    retrieved_pt = Q.roots(ring=ZZ)[0][0]
12    return retrieved_pt
```

Program 3: Funkcja odpowiedzialna za odzyskanie liczb pierwszych



```
> ~/Projects/KRYS/coppersmiths-attack > P main !1 ?1
sage stereotyped.sage
[0] Parameters
    Original PT: yourpasswordispassword
    Fragment of PT known by attacker: yourpasswordis00000000
[1] Generating example...
    p: 1040606148954068219399962490923509211049862310741144681150206656651705710990361769995887788712885371906455195676865129561
    2430972505597764200453738133405631
    q: 4386588083680449979752631838165902736298266849544397780794077181309977482187683107279364323945647313857520802444352657223
    952121569338616776894370955041331
    n = p * q = 4564710532806518998575164783881374588346410264622112895575740892305788990879880409638126526716009492605313082120
    6174428051578497655495481966071534255441640375558317169716316031530484109810477388101728143920977073029846625278244186808515917277752
    913074567544508158723341249475983490594069328521050639393134861
    e: 3
    CT: 4652493694822322856856267678038390733584067635062618608216000650614372203648665067767350437849025127625
[2] Retrieving ciphertext with Howgrave-Graham...
    Defining upper bound (X) as: 2821109907455
    Creating matrix: [[X^3, 3*X^2*partial_pt, 3*X*partial_pt^2, partial_pt^3-ciphertext], [0,n*X^2,0,0], [0,0,n*X,0],[0,0,0,n]]
    Performing LLL reduction on the matrix
    Defining polynomial g(x) from LLL-reduced matrix (M'): g(x) = M'[0][0]*x^3/X^3 + M'[0][1]*x^2/X^2 + M'[0][2]*x/X + M'[0][3]
    Retrieved rest of plaintext: password
```

Rysunek 4: Wynik uruchomienia programu

## Bibliografia

- [1] J. S. K. Dong Pyo Chi Jeong Woon Choi i T. Kim, „Lattice Based Cryptography for Beginners”. [Online]. Dostępne na: <https://eprint.iacr.org/2015/938.pdf>
- [2] P. Q. Nguyen, „Hermite's Constant and Lattice Algorithms”, w *The LLL Algorithm: Survey and Applications*, P. Q. Nguyen i B. Vallée, Red., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 19–69. doi: [10.1007/978-3-642-02295-1\\_2](https://doi.org/10.1007/978-3-642-02295-1_2).
- [3] J. Hoffstein, J. Pipher, i J. H. Silverman, *An Introduction to Mathematical Cryptography*, 2. wyd. w Undergraduate Texts in Mathematics. Springer New York, 2014, s. XVII, 538. doi: <https://doi.org/10.1007/978-1-4939-1711-2>.
- [4] M. Nemec, M. Sys, P. Svenda, D. Klinec, i V. Matyas, „The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli”, w *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, w CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, s. 1631–1648. doi: [10.1145/3133956.3133969](https://doi.org/10.1145/3133956.3133969).
- [5] F. Picca, „Analysis of the ROCA vulnerability”. [Online]. Dostępne na: <https://bitsdeep.com/posts/analysis-of-the-roca-vulnerability>