Politechnika Białostocka

Sprawozdanie z pracowni specjalistycznej

Bezpieczeństwo Sieci Komputerowych

Ćwiczenie numer: 6

Temat: Kryptosystemy symetryczne- algorytmy DES, RSA

Link do repozytorium: https://github.com/pb-students/BSK-03-des

Link do sprawozdania: https://github.com/pb-students/BSK-raport-05-des

Link do filmu prezentującego działanie: https://streamable.com/07z4do

Wykonujący ćwiczenie:

Paweł Orzel

Łukasz Hossa

Kacper Seweryn

Studia dzienne

Kierunek: Informatyka

Semestr: VI

Grupa zajęciowa: Grupa PS 10

Prowadzący ćwiczenie: mgr inż. Katarzyna Borowska

Data wykonania ćwiczenia:08.05.2022

1. Teoria

DES – symetryczny szyfr blokowy zaprojektowany w 1975 roku przez IBM na zlecenie ówczesnego Narodowego Biura Standardów USA. Od 1976 do 2001 roku stanowił standard federalny USA, a od roku 1981 standard ANSI dla sektora prywatnego (znany jako *Data Encryption Algorithm*).

Szyfr strumieniowy- algorytm symetryczny, który szyfruje oddzielnie każdy bit wiadomości. Algorytm ten składa się z generatora strumienia bitowego, będącego kluczem szyfrującym oraz elementu dodającego

2. Realizacja zadania

Wykonaj program realizujący szyfrowanie oraz deszyfrowanie z wykorzystaniem algorytmu DES.

Wszystkie potrzebne permutacje znajdują się w **BSK-03-des/src/constants-des.ts**

a) kod źródłowy- omówienie

BSK-03-des/src/composables/use64BitBlock.ts

```ts
export default async (file: File) => {
  const arrayBuffer = await file.arrayBuffer()
  const fillBlocks = (8 - arrayBuffer.byteLength % 8) || 8

  const buffer = new Uint8Array(arrayBuffer.byteLength + fillBlocks)
  buffer.set(new Uint8Array(arrayBuffer), 0)
  buffer.set(new Uint8Array(
    [...Array(fillBlocks)].map((_, i) => i === fillBlocks - 1 ? fillBlocks : 0)),
    arrayBuffer.byteLength
  )

  return new BigUint64Array(buffer.buffer)
}
```

Omówienie: hook odpowiada za sprawdzenie czy blok danych jest 64-bitowy oraz dopełnieniu bloku do 64-bitów w razie stwierdzenia niekompletności bloku

BSK-03-des/src/composables/useDES.ts

```ts
export default (key = 108755180518133317n) => {
  const encrypt = async (file: MaybeRef<File>) => {
    const blocks = await use64BitBlocks(get(file))
    const blockPairs32 = [...blocks].map(block => {
      const bitBlock = permutate(new BitArray(block), IP)
      return cast64BitBlock(Uint32Array, BigInt(`0b${bitBlock.join('')}`))
    })

    const permutedChoice = permutate(cast64BitBlock(BitArray, key), PC)
    let keyC = parseInt(permutedChoice.slice(0, 28).join(''), 2)
    let keyD = parseInt(permutedChoice.slice(28).join(''), 2)

    const keys = []
    for (let i = 0; i < SHIFT.length; i++) {
      keyC <<= SHIFT[i]
      keyC &= 0xfffffff
      keyD <<= SHIFT[i]
      keyD &= 0xfffffff

      keys.push(permutate([...new BitArray(keyC, 28), ...new BitArray(keyD, 28)], PC2))
    }
```

Omówienie: do zmiennej blocks przypisujemy nasz blok 64-bitowy(wcześniej dopełniony jeśli była taka potrzeba) i permutujemy zgodnie z permutacją początkową(IP), nasz klucz musimy zredukować (permutowanym wyborem PC) następnie dzielimy blok na dwa bloki 28-bitowe(keyC oraz keyD) . W każdej z 28-bitowych części klucza robimy przesunięcie w lewo o X bitów. Operację wykonujemy SHIFT-razy(SHIFT to stała zadeklarowana w constants-des.ts).

Na koniec tworzymy klucz poprzez połączenie keyC oraz keyD,a następnie odpowiednio permutując połączenie ciągiem PC2.

```
const result = []
for (const [left, right] of blockPairs32) {
  const newBlocks: [BitArray, BitArray][] = []

  const rightBitArray = new BitArray(right)
  const leftBitArray = new BitArray(left)

  for (let i = 0; i < SHIFT.length; i++) {
    const key = keys[i]

    const rightExtended = permutate(rightBitArray, E)
      .map((bit, i) => bit ^ key[i])

    const bits32 = [...Array(8).keys()].map(i => {
      return rightExtended.slice(0 + 6 * i, 6 + 6 * i)
    }).map(([y1, x1, x2, x3, x4, y2], i) => {
      const x = parseInt([x1, x2, x3, x4].join(''), 2)
      const y = parseInt([y1, y2].join(''), 2)
      return new BitArray(S[i][y][x], 4)
    }).reduce((acc: number[], a) => [...acc, ...a], [])

    newBlocks.push([
      rightBitArray,
      permutate(bits32, P).map((bit, i) => bit ^ leftBitArray[i])
    ])
  }
```

Omówienie: Na początku tworzymy tablice (rightBitArray-Rn, leftBitArray-Ln, key). Do zmiennej key przepisujemy kolejne elementy wcześniej uzyskanego klucza. Aby przekształcić 32-bitowy blok przypisujemy do nowej zmiennej rightExtended wynik permutacji(E) na Rn, a następnie XOR'ujemy te tablice, uzyskując ciąg 48 bitów. Kolejnym krokiem jest podzielenie tablicy rightExtended na 8 ciągów, gdzie każdy ma 6 bitów. Po podzieleniu tworzymy dwie zmienne x oraz y. Zmienna x odpowiada za przechowywanie indeksu wiersza, a zmienna y za indeks kolumny. Łączące uzyskane indeksy wiemy z jakiej tablicy S[i] będziemy czytać dane(gdzie i odpowiada kolejnemu blokowi). Na koniec wywołujemy funkcję reduce względem wartości przyrostowej z każdego wywołania i kolejnego elementu tablicy w celu sprowadzenia tej tablicy do pojedynczej wartości.

Do zmiennej newBlocks wrzucamy R_{16}(rightBitArray) oraz blok 32-bitowy(bits32), który po wcześniejszych przekształceniach XOR'ujemy z leftBitArray.

```
    const [leftBlock, rightBlock] = newBlocks[SHIFT.length - 1]
    result.push(BigInt(`0b${permutate([...rightBlock, ...leftBlock], IIP).join('')}`))
  }

  return new File([
    new BigUint64Array(result).buffer
  ], 'encrypted.bin', { type: 'application/octet-stream' })
}
```

Omówienie: Łączymy otrzymane bloki(zmieniamy bloki miejscami), a następnie poddajemy permutacji(IIP). Na koniec zwracamy tablicę do pliku.

```
const decrypt = async (file: MaybeRef<File>) => {

  const blocks = await use64BitBlocks(get(file))
  const blockPairs32 = [...blocks].map(block => {
    const bitBlock = permutate(new BitArray(block), IP)
    return cast64BitBlock(Uint32Array, BigInt(`0b${bitBlock.join('')}`))
  })

  const permutedChoice = permutate(cast64BitBlock(BitArray, key), PC)
  let keyC = parseInt(permutedChoice.slice(0, 28).join(''), 2)
  let keyD = parseInt(permutedChoice.slice(28).join(''), 2)

  const keys = []
  for (let i = 0; i < SHIFT.length; i++) {
    keyC <<= SHIFT[i]
    keyC &= 0xfffffff
    keyD <<= SHIFT[i]
    keyD &= 0xfffffff

    keys.push(permutate([...new BitArray(keyC, 28), ...new BitArray(keyD, 28)], PC2))
  }
  const reversedKeys = keys.reverse() // reversing keys for decrypt
```

```
const reversedKeys = keys.reverse() // reversing keys for decrypt

const result = []
for (const [left, right] of blockPairs32) {
  const newBlocks: [BitArray, BitArray][] = []

  const rightBitArray = new BitArray(right)
  const leftBitArray = new BitArray(left)

  for (let i = 0; i < SHIFT.length; i++) {
    const key = reversedKeys[i]

    const rightExtended = permutate(rightBitArray, E)
      .map((bit, i) => bit ^ key[i])

    const bits32 = [...Array(8).keys()].map(i => {
      return rightExtended.slice(0 + 6 * i, 6 + 6 * i)
    }).map(([y1, x1, x2, x3, x4, y2], i) => {
      const x = parseInt([x1, x2, x3, x4].join(''), 2)
      const y = parseInt([y1, y2].join(''), 2)
      return new BitArray(S[i][y][x], 4)
    }).reduce((acc: number[], a) => [...acc, ...a], [])

    newBlocks.push([
      rightBitArray,
      permutate(bits32, P).map((bit, i) => bit ^ leftBitArray[i])
    ])
  }

  const [leftBlock, rightBlock] = newBlocks[SHIFT.length - 1]
  result.push(BigInt(`0b${permutate([...rightBlock, ...leftBlock], IIP).join('')}`))
}

return new File([
  new BigUint64Array(result).buffer
], 'decrypted.bin', { type: 'application/octet-stream' })
```

Omówienie: Operacja deszyfrowania działa tak samo jak szyfrowanie, jedyną różnicą jest odwrócenia klucza

3. Wnioski

Algorytm DES od pierwszych lat dwudziestego pierwszego wieku uznawany jest za szyfr, który nie jest w stanie zapewnić odpowiedniego zabezpieczenia, głównie ze względu na relatywnie

krótki sekretny klucz, podatny na ataki siłowe (ang. *brute force attacks*). W 2001 roku został zastąpiony przez AES, który otrzymał miano standardowego szyfru dla sektora prywatnego. DES jest wciąż jednym z najlepiej zbadanych i najpopularniejszych algorytmów szyfrujących.