Politechnika Białostocka

Sprawozdanie z pracowni specjalistycznej

Bezpieczeństwo Sieci Komputerowych

Ćwiczenie numer: 4

Temat: GENERATORY LICZB PSEUDOLOSOWYCH. SZYFRY STRUMIENIOWE

Link do repozytorium: https://github.com/pb-students/BSK-02-lfsr

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:10.04.2022

1. Teoria

Generatory liczb pseudolosowych- program, który na podstawie niewielkiej ilości informacji generuje deterministycznie ciąg bitów, który pod pewnymi względami jest nieodróżnialny od ciągu uzyskanego z prawdziwie losowego źródła.

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

2.1 Zaimplementuj generator liczb pseudolosowych bazujący na metodzie LFSR. Wielomian powinien być podawany przez użytkownika jako parametr. Program powinien generować bity „w nieskończoność” aż do momentu zatrzymania algorytmu przez użytkownika.

2.1.1

Wersja bazowa, która została wykorzystana do stworzenia finalnej wersji 2.1.2

a) kod źródłowy

```
function decTobin(text){
    var numberArray = [...text].map(x => parseInt(x))
    .filter(x => !isNaN(x));
    const maxNumber = Math.max(...numberArray)
    let mapped = numberArray.reduce((prev, current) => prev + current, 0)
    .toString(2)
    .split("")
    .reverse()
    .join("");
    for(let i=0;i<maxNumber - mapped.length;i++){
        mapped += "0"
    }
    return mapped
}
```

```
function main(text){
   let wynik = []
   var binarna = decTobin(text)
   const powtorzenia = checkPowtorzenia(binarna)
   // znalezienie indeksow dla xor
   const index = [] // indexy dla rezultatu xor
    for(let i=0;i<binarna.length;i++){
        if(binarna[i] == "1")
        index.push(i)
    }
    for(let i=0;i<powtorzenia-1;i++){
        let temp =""
        let temp2 =""
        temp = (binarna[index[0]] ^ binarna[index[1]])
        for(let i = 2;i<binarna.length - 2;i++){
            temp = (temp ^ binarna[index[i]])
        }
        let b = temp
        let a = ""
        a = a+b
        a = a+binarna
        a = a.slice(0,binarna.length)
        wynik.push(a)
        binarna = a
    }
    return wynik
```

Omówienie:

-Funkcja decTobin- funkcja na początku parsuje stringa na inta, w wyniku czego każdy znak zostaje na cyfrę, jeżeli taka zmiana nie jest możliwa, dzięki filtrowi pozbywamy się tych stringów, które nie mogą zostać zamienione(NaN). Kolejnym krokiem jest dodanie do siebie liczb, zamiana na postać binarną jako string, a na końcu odwrócenie wyniku(żeby najstarszy bit był po prawej stronie)

-Funkcja main- do zmiennej var przypisujemy wynik działania funkcji decTobin opisanej wyżej, sprawdzamy za pomocą funkcji checkPowtorzenia(prosta funkcja, która zwraca liczbę powtórzeń w pętli w funkcji main) ilość powtórzeń. Kolejnym krokiem jest wyznaczenie miejsc, które program będzie porównywał(XoR) w głównej pętli. W głównej pętli zaczynamy od ustalenia wyniku XoRa dla aktualnego ciągu, następnie tworzymy nowy ciąg, który rozpoczyna się od tego wyniku(temp). Do takiego ciągu dodajemy nasz obecnie sprawdzany ciąg i obcinamy ostatnią cyfrę funkcją slice. Ostateczny wynik wrzucamy do tablicy nazwanej wynik

b) Przykładowe wyniki

```
console.log(main("x^3+x^6+x^5"))  [ '10111', '01011', '00101', '10010', '11001', '11100', '01110' ]
```

```
console.log(main("x^1+x^5"))  [ '0011', '1001', '0100' ]
```

```
console.log(main("x^2+x^3+x^7+x^8"))
```

```
[ '0001010',
  '0000101',
  '1000010',
  '0100001',
  '0010000',
  '1001000',
  '0100100',
  '1010010',
  '1101001',
  '0110100',
  '0011010',
  '1001101',
  '1100110',
  '1110011',
  '1111001' ]
```

2.1.2

a) kod źródłowy- omówienie

BSK-02-lsfr/src/composables/compilePolynomial.ts

```ts
const EXPRESSION_REGEX = /^x\s*\*\*\s*(\d+)$/      Kasper Seweryn, 55 minut temu • feat: add useLFSR composable …

export default (polynomialString: string) => {
  const exponents = new Set()
  let hasConstantOne = false
  for (const expression of polynomialString.split('+').map(e => e.trim())) {
    switch (expression) {
      case '1':
        hasConstantOne = true
        break;

      case 'x':
        exponents.add(0)
        break;

      default:
        if (!EXPRESSION_REGEX.test(expression)) throw new Error(`Invalid expression: ${expression}`)

        const exponent = +(expression.match(EXPRESSION_REGEX)?.[1] ?? 0) - 1
        if (exponent < 0) throw new Error(`Invalid expression: Exponent cannot be less or equal than 0: ${expression}`)
        if (exponents.has(exponent)) throw new Error(`Invalid expression: Exponent already exists: ${expression}`)
        exponents.add(exponent)
    }
  }
```

```
if (!hasConstantOne) {
  throw new Error(`Invalid polynomial: Polynomial has to include a constant '1'`)
}

if (exponents.size < 2) {
  throw new Error(`Invalid polynomial: Polynomial has to include at least 2 expressions with x`)
}

return exponents
```

Omówienie: Zaczynamy od ustawienia naszego wyrażenia regularnego, aby format wykładnika przy x był poprawny. Kolejno wykładniki zapisujemy do kolekcji Set. Wykładniki dodajemy do kolekcji w pętli, która iteruje po wprowadzonym Stringu(w pętli od razu oddzielamy znak + i przycinamy dany wielomian). Następnie instrukcje if mają za zadnie poinformować nas o złym formacie wprowadzonego wielomianu. W tworzeniu programu założyliśmy, że w wielomianie muszą wystąpić przynajmniej 2 wyrażenia z x.

BSK-02-lsfr/src/composables/ useLFSR.ts

```
import compilePolynomial from "./compilePolynomial"

function* lfsr (taps: Set<number>, startingState: number) {
  const max = Math.max(...taps)
  const tapsArray = [...taps]
  // NOTE: Limit startingState to only max + 1 bits
  let lfsr = startingState & ((1 << (max + 1)) - 1)      Kasper Seweryn, 59 minut te
  while (true) {
    yield lfsr & 1

    // NOTE: Add missing zeros to the beginning of the binary string
    const binaryString = lfsr.toString(2).padStart(max + 1, '0')

    let bit = +binaryString[max - tapsArray[1]] ^ +binaryString[max - tapsArray[0]]
    for (const tap of tapsArray.slice(2)) {
      bit ^= +binaryString[max - tap]
    }
    lfsr >>= 1
    lfsr |= bit << max
  }
}
export default (polynomial: string, startingState: number = 0xdeadbeef) => {
  const taps = compilePolynomial(polynomial)
  return lfsr(taps, startingState)
}
```

Omówienie: Zaczynamy od ustalenia długości naszego ciągu(Math.max w kolekcji taps) oraz zmienienia Set na zwykłą tablicę. Kolejno do zmiennej lsfr przypisujemy startingState ograniczony do długości maksymalnego indeksu bitu zwiększonego o 1. W nieskończonej pętli tworzymy ciąg binarny, który tworzymy poprzez toString oraz uzupełniając go zerami do wartości max+1. Zmienna bit jest wynikiem operacji XOR dla dwóch pierwszych bitów. Zaraz po tym w pętli for wykonujemy operację XOR dla wcześniejszego bitu oraz następnego, aż otrzymamy ostateczny wynik. Na koniec nasz ciąg binarny

przesuwamy w prawo o jedną pozycję. Ostatnim krokiem jest dodanie z przodu bitu, który mamy pod zmienną bit.

BSK-02-lsfr/src/composables/ useByteLFSR.ts

```
const ARRAY_8 = [...Array(8)]

export default (polynomial: string, startingState: number) => {
  const lfsr = useLFSR(polynomial, startingState)

  const generator = function* () {
    while (true) {
      yield parseInt(ARRAY_8.map(() => lfsr.next().value).join(''), 2)
    }
  }

  return generator()
}
```

Omówienie: Pobieramy 8-bitów z LFSR-a, wstawiamy je do stringa, aby potem użyć parseInt(). Dzięki takiej operacji otrzymujemy 1 bajt danych. W trakcie czytania danych z pliku dane otrzymujemy jako Uint8Array, czyli jako array 1 bajtowych danych. Na końcu XORujemy 1 bajt danych z 1 bajtem z LFSRa

b) testy

BSK-02-lfsr/test/lfsr.test.ts

```
test('with starting state', async () => {
  const lfsr = useLFSR('1 + x + x ** 4', 0b0110)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([0, 1, 1, 0, 0, 1, 0, 0])
})

test('2bit-initial', async () => {
  const lfsr = useLFSR('1 + x + x ** 2', 0b10)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([0,1,1,0,1,1,0,1])
})

test('3bit-initial', async () => {
  const lfsr = useLFSR('1 + x + x ** 2', 0b101)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([1,0,1,1,0,1,1,0])
})

test('4bit-initial with 4 bits to XOR', async () => {
  const lfsr = useLFSR('1 + x + x ** 2 + x ** 3 + x ** 4', 0b1010)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([0,1,0,1,0,0,1,0])
})
```

```
test('3bit-initial', async () => {
  const lfsr = useLFSR('1 + x + x ** 2', 0b101)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([1,0,1,1,0,1,1,0])
})

test('4bit-initial with 4 bits to XOR', async () => {
  const lfsr = useLFSR('1 + x + x ** 2 + x ** 3 + x ** 4', 0b1010)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([0,1,0,1,0,0,1,0])
})

test('4bit-initial with 3 bits to XOR', async () => {
  const lfsr = useLFSR('1 + x + x ** 2 + x ** 3 ', 0b1010)

  const byte = [...Array(8)].map(() => lfsr.next().value)
  expect(byte).toEqual([0,1,0,1,0,1,0,1])
})
```

2.2 Zaimplementuj kryptosystem bazujący na schemacie Synchronous Stream Cipher dla podanego wielomianu. Do generowania klucza wykorzystaj metodę LFSR. Na potrzeby działania algorytmu informację jawną przekształć do postaci binarnej. Program powinien mieć możliwość obsługi plików różnego typu (szyfrowanie zarówno plików tekstowych, jak i co najmniej jednego innego formatu plików).

Szyfrowanie

```
const encrypt = async (file: MaybeRef<File>) => {
  const lfsr = useByteLFSR(
    get(polynomial),
    get(startingState)
  )

  const byteArray = new Uint8Array(await file.arrayBuffer())

  // NOTE: We're attaching a MIME type header
  const header = Uint8Array.from(Array.from(get(file).type + ';').map(letter => letter.charCodeAt(0)))

  return new File([
    header.map(byte => byte ^ lfsr.next().value),
    byteArray.map(byte => byte ^ lfsr.next().value),
  ], 'encrypted.bin', { type: 'application/octet-stream' })
}
```

Funkcja przyjmuje jako parametr dowolny plik. Inicjujemy generator lfsr za pomocą wcześniej stworzonej funkcji, do której przekazujemy wielomian oraz stan początkowy. Następnie zamieniamy plik w kolekcję bajtów i tworzymy nagłówek do zakodowanej postaci poprzez wyciągnięcie go z pliku. W tym nagłówku dodajemy na końcu ‘;’ jako wiadomość o końcu nagłówka, która jest potrzebna do dekodowania. Funkcja zwraca zakodowany plik, gdzie najpierw zakodowywany jest nagłówek pliku, w którym jest informacja o typie a następnie całość jest xor’owana. Zastosowany typ octet-stream dodajemy, by skojarzyć plik jako binarny. Nagłówek dodajemy po to, by nie trzeba było domyślać się rozszerzenia pliku.

Deszyfrowanie

```
const decrypt = async (file: MaybeRef<File>) => {
  const lfsr = useByteLFSR(
    get(polynomial),
    get(startingState)
  )

  const byteArray = new Uint8Array(await file.arrayBuffer())
  const decrypted = byteArray.map(byte => byte ^ lfsr.next().value)
  const separatorIndex = decrypted.indexOf(';'.charCodeAt(0))

  const type = [...decrypted.slice(0, separatorIndex)].map(charCode => String.fromCharCode(charCode)).join('')
  return new File(
    [decrypted.slice(1 + separatorIndex)],
    `decrypted.${mimedb[type]?.extensions[0] ?? 'bin' }`,
    { type: type in mimedb ? type : 'application/octet-stream' }
  )
}
```

Funkcja przyjmuje jako parametr zaszyfrowany plik. Inicjujemy generator LFSR z wielomianem i stanem początkowym. Operacją XOR odszyfrowujemy kolekcję bajtów, a następnie przypisujemy do zmiennej ‘separatorIndex’ zapisujemy pozycję końca nagłówka pliku mówiącego o typie pliku. Jest to możliwe dzięki oddzieleniu nagłówka pliku i pozostałości znakiem ‘;’. Odczytujemy typ pliku i zwracamy odkodowany plik. Rozszerzenie określamy za pomocą mimedb, a w przypadku problemów zwracamy go domyślnie w rozszerzeniu bin.

3. Wnioski

Zaprojektowany generator LFSR oraz szyfr strumieniowy działa poprawnie. Rejestr inicjujemy losowym ciągiem bitów oraz wielomianem i może on generować nieskończenie długi ciąg, jednak zaczynają się one potem powtarzać w zależności od postaci wielomianu. Generator stworzony został na potrzeby zaprojektowania szyfru strumieniowego. W szyfrze tekst jawny przetwarzamy bit po bicie z wygenerowanym kluczem co pozwala na otrzymanie postaci zaszyfrowanej. Stworzony projekt pozwala na zaszyfrowanie dowolnego typu plików i odszyfrowanie go ze znanym rozszerzeniem.