

# Projekt 1 - prúdová šifra

Michal Ormoš  
Kryptografia 2018/2019

10. března 2019

## Abstrakt

Táto správa popisuje riešenie prvého projektu z Kryptografie a to prelomenie neznámej prúdovej šifry. V úvode predstavuje zadanie a vysvetľuje princíp prúdovej šifry. V ďalšej časti popisuje ručné a SAT riešenie. V riešení sa odkazuje na zdrojové súbory v jazyku Python, ktorým bolo riešenie dosiahnuté. Popisovať budeme len zaujímavé a podstatne pasáže riešenia.

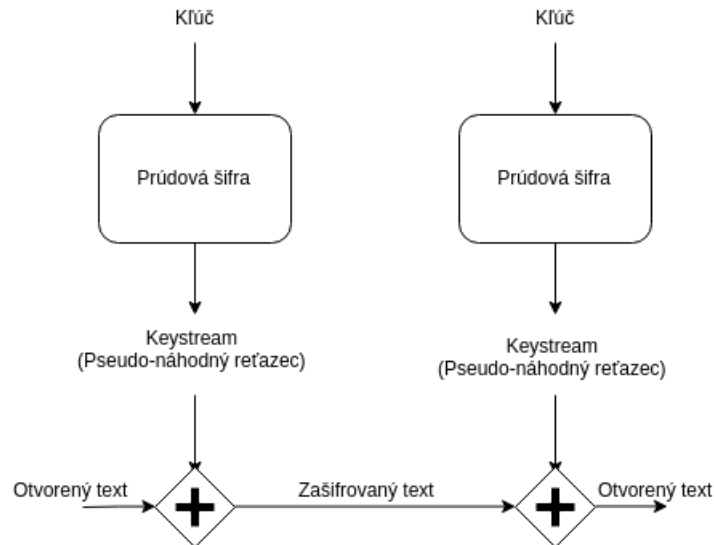
## 1 Úvod

Cielom tohto projektu je prelomiť neznámu synchronnú prúdovú šifru, ktorou boli zašifrované súbory. Bolo nám prezradené, že súbory boli zašifrované neznámou prúdovou šifrou a kľúčom o dĺžke 29 znakov vo formáte `KRY{24 znakov ASCII textu}`.

## 2 Analýza

Predložený archív obsahoval štyri súbory, bez dodatočných informácií. Ich význam som odhadol a dedukoval nasledne:

- bis.txt - nešifrovaný plaintext [509B]
- bis.txt.enc - zašifrovaný plaintext textu bis.txt [512B]
- hint.gif.enc - zašifrovaný obrázok vo formáte gif [20970B]
- super\_cipher.py.enc - zašifrovaný skript [714B]



Obrázek 1: Prúdová šifra

Prúdová šifra je symetrický šifrovací algoritmus, ktorý používa kľúč pre zašifrovanie jednotlivých bitov textu, obrázok 1. Narozdiel od one-time-pad šifier, nevyužíva XOR len na zašifrovanie správy na náhodný string, ale XORuje správu na pseudonáhodnú sekvenciu bitov, ktorá sa nazýva keystream. Keystream je generovaný z malej sekvencie naozaj náhodných bitov. Dešifrovanie prebieha rovnakým spôsobom a to XORom zašifrovaného textu s keystreamom generovaným z rovnakej malej sekvencie náhodných bitov.<sup>1</sup>

### 3 Ručné riešenie

#### 3.1 Keystream

Ako prvý krok som zvolil pokus o odhalenie keystreamu, použitého pre šifrovanie súboru `bis.txt`. Keďže som k súboru `bis.txt` mal odpovedajúci zašifrovaný text `bis.txt.enc`, použil som na nich operáciu `xor`. Podľa vzorca:

$$\text{bis.txt} \oplus \text{bis.txt.enc} = \text{keystream}$$

Vo výsledku som získal keystream, ktorý som následne aplikoval na súbor `super_cypher.py.enc`

$$\text{keystream} \oplus \text{super\_cypher.py.end} = \text{super\_cypher.py}$$

Tým som odhalil odšifrovaný plaint text súboru `super_cypher.py` Keystream bol kratší než obsah tohto zašifrovaného skriptu, a tak sa mi po darilo odkryť len časť tohto súboru. Našťastie táto časť obsahovala dôležité konštanty a algoritmus, ktorým bol tento keystream generovaný.

SUB = [0, 1, 1, 0, 1, 0, 1, 0]

N\_B = 32

N = 8 \* N\_B

<sup>1</sup><https://www.globalspec.com/reference/81191/203279/2-6-stream-ciphers>

```

# Next keystream
def step(x):
    x = (x & 1) << N+1 | x << 1 | x >> N-1
    y = 0
    for i in range(N):
        y |= SUB[(x >> i) & 7] << i
    return y

# Keystream init
keystream = int.from_bytes(args.key.encode(), 'little')
print(keystream)
for i in range(N//2):
    keystream = step(keystream)

```

Postupu získania tejto časti tajmostva zodpovedá v zdrojových kódach funkcia `xorKnownTexts` v súbore `solution.py`

### 3.2 Odhalenie celého obsahu `super_cipher`

V dešifrovanom súbore som objavil dôležité konšanty `SUB`, `N_B` a algoritmus, ktorým bol keystream generovaný. Rovnakým spôsobom ako v predošlom kroku som tento keystream aplikoval aj na súbor `hint.gif.enc`. Ten mi odhalil druhú časť skriptu `super_cypher.py` a to riadky ktoré spracujú vstup.

```

plaintext = sys.stdin.buffer.read(N_B)
while plaintext:
    sys.stdout.buffer.write((
        int.from_bytes(plaintext, 'little') ^ keystream
    ).to_bytes(N_B, 'little'))
    keystream = step(keystream)
    plaintext = sys.stdin.buffer.read(N_B)

```

Súbor `hint.gif.enc` nepriniesol žiadne ďalšie informácie, ktoré by som mohol použiť a ďalej som sa ním už nezaoberal. Postupu získania tejto časti tajmostva zodpovedá v zdrojových kódach funkcia `xorKnownTexts` v súbore `solution.py`

Súbor `bis.txt` má zašifrovaný väčšiu veľkosť ako dešifrovaný, ale to vo výsledku nehraje rolu, keďže vidíme, podľa konštanty `N_B`, že keystream sa po 32B vždy opakuje

### 3.3 Reverzácia funkcie `step`

Po dôkladnom naštudovaní skriptu, kde som predpokladal, že žiadne ďalšie riadky neobsahuje, mi bolo jasné, že chýbajúci parameter `key` môžem získať len reverzáciou funkcie `step`. Funkcia `step` funguje nasledujúcim spôsobom.

Rotácia:

- vezme vstup a vypočíta z neho výstup, vždy o veľkosti 32B.
- vezme najpravejší bit vstupu a posunie sa o  $N+1$  bitov doľava
- hodnota vstupu sa posunie o 1 bit doľava

- hodnota vstupu sa posunie o N-1 bitov doprava
- prevedie sa logický OR troch predchádzajúcich krokov

Substitúcia:

- vektor x sa posunie o i bitov doprava
- vezmu sa tri najpravejšie hodnoty posunutého vektoru x
- vezme sa hodnota vektoru SUB, pozícia ktorú určujú bity z predchádzajúceho kroku
- prevedie sa posun získanej hodnoty o i bitov doľava
- prevedie sa logický OR získanej hodnoty s hodnotou z predchádzajúceho kroku i-1

Získaný keystream je teda výsledok v premennej y, my sa budeme snažiť získať reverzným algoritmom hodnotu x.

Najťažším krokom bolo reverzovať substitúciu, ktorá sa vykonáva pre každý bit. Vstupom substitúcie je aktuálny bit a dva predchádzajúce bity. Spolu teda tri bity, indexujúce čísla od 0 do 7. Tento index sa následne používa ako index prístupu do poľa SUB. Indexovaným prvkom je nahradený bit aktuálnej pozície. Najväčší problém spočíval v tom, že najvyššie dva bity nie sú známe a je teda potrebné ich uhádnuť. Ako najlepšia možnosť sa javí skúšať všetky možnosti. Ako podmienka nám poslúži fakt, že spodné dva bity sa musia rovnať vrchným dvom bitom.

```
if (x & 3) == (x >> 256)
```

Postupným aplikovaním reverzného kroku step som narazil na problém, že posledná spomenutá podmienka sa nerovnála, alebo vo výsledku som dostal kľúč, ktorý sa nedal spatne preložiť do kódu ASCII. Keďže som vedel fakt, že kľúč bude vo formáte `KRY{24 znakov ASCII textu}`, avšak nemohol som si byť istý, že algoritmus získaný zo súboru `super_cypher.py` je správny a mohol byť rovnako podvrhnutý.

Najväčší problém nastával práve pri poli SUB, z ktorého spätnou substitúciou vychádzali hodnoty, ktoré neodpovedali mojim dedukciám a podmienke. Podozrenie spolu s diskusiou so spolužiakmi dospeli k bodu, že veriť samotnému dešifrovanému skriptu stačiť nebude a preto som sa rozhodol rovnako vyskúšať kombinácie poľa SUB, teda nepoužiť hodnoty, ktoré som objavil, ale vyskúšať permutáciou všetkých 256 možností.

Pri každej možnosti SUB, ktorá nesplnila moju podmienku, alebo obsahovali na konci nezmyselné znaky som zahodil a pokračoval ďalšou permutáciou SUB.

### 3.4 Záver ručného riešenia

Výsledný získaný jedinečný kľúč bol:

**KRY{xormos00-3aa5d40d35f624b}**

V tomto riešení som získal, pochopil a aplikoval reverzáciu funkcie step na získanie kľúča. Narazil som na problém falošného poľa SUB, ktorý mi zabral v riešení najviac času.

Výsledok som si overil tak, že som pomocou skriptu `super_cypher.py` s pozmeneným SUB (úpravou `[0, 1, 1, 0, 0, 1, 0, 1]` na `[0, 1, 1, 0, 1, 0, 1, 0]`) a mnou získaným kľúčom aplikoval na pôvodný súbor `bis.txt.enc` a získal rovnaké znenie súboru `bis.txt`.

Riešeniu zodpovedajú funkcie `brakeCipherKey` a `stepReversed` v súbore `solution.py`. Časová náročnosť bola testovaná na školskom serveri merlin a v priemere prelomenie zabralo 6 sekúnd.

## 4 SAT riešenie

V tomto riešení som už využil poznatky z ručného riešenia a to presne, znenie algoritmu pomocou ktorého bolo šifrovanie vykonané a fakt, že pole SUB je vo výsledku iné ako pôvodne algoritmus preznetoval.

Riešenie s pomocou SAT solvera je podobné ručnému riešenie, okrem substitúcie, ktorú tu budeme riešiť pomocou SAT solvera. Ten pozostáva z nasledujúcich krokov:

1. Prevod aktuálneho kľúča na formulu v boolovej algebre. Každému jednotlivému bitu keystreamu je vytvorená jedna premenná. Z toho je následne vytvorená formula v konjunktívnej normálnej forme, kde jej každý člen je podformula a popisuje jednotlivé bity nasledovne, obdobne ako pri ručnom riešení s opraveným poľom SUB.
  - bit je nulový, existujú štyri možnosti ako môžu vyzerat dva predchádzajúce bity
  - bit je jednotkový, existujú rovnako štyri možnosti ako môžu vyzerat dva predchádzajúce bity

Jedná sa o štyri podformule spojené logickým OR.

$$(-v \ \& \ -v1 \ \& \ -v2) \mid (v \ \& \ (v1 \mid v2)) \mid (v \ \& \ -v1 \ \& \ -v2) \mid (-v \ \& \ (v1 \mid v2))$$

2. Ohodnotenie premených, v spolupráci s knižnicou `satisfy`. Vstupom je formula vo formáte boolovej algebry a výstupom je binárna hodnota toho či je formula riešiteľná (True), alebo nie je riešiteľná (False)
3. Konverzia ohodnoteným premenných späť na kľúč. Prechádzame bit po bite.
  - bit je jednotkový ak je jeho i-tá premenná ohodnotená ako riešiteľná
  - bit je nulový, ak je i-tá premenná ohodnotená inak

## 5 Záver

V projekte sme si vyskúšali prelomenie šifry pomocou ručného riešenia a SAT solveru. Riešenie pomocou SAT solveru som považoval, že zaujímave vďaka nutnosti využiť formuly boolovej algebry. Výsledok SAT riešenia:

**KRY{xormos00-3aa5d40d35f624b}**

## A Použitie programu

Skript `solution.py` nepotrebuje žiadne ďalšie dodatočne inštalácie. Spúšťa sa s jediným argumentom, ktorým je cesta k zdrojovým súborom popísaným v kapitole Úvod.

Príklad spustenia:

```
python3 solution.py in/
```

V prípade nezadania vstupného argumentu očakáva súbory v zložke `in/`.

Skript `solution_sat.py` potrebuje doinštalovať rozšírenia pomocou skriptu `install.sh`. Jeho spustenie je obdobné ako pri skripte `solution.py` a to pomocou jediného vstupného argumentu, cesta k vstupným súborom.

Príklad spustenia:

```
python3 solution_sat.py in/
```

V prípade nezadania vstupného argumentu očakáva súbory v zložke in/.