Téma 2 – Principy kryptografie

Hashovací funkce

V prvním čísle jsme se zabývali blokovými šiframi. V tomto se podíváme na další podstatný koncept, bez kterého by se kryptografie neobešla, a tím jsou hashovací neboli jednosměrné funkce.

Již nyní se můžete těšit na třetí číslo, ve kterém blokové šifry a hashovací funkce využijeme dohromady. Najdete v něm také řešení úloh z prvního i druhého čísla. Pokud jste nám zatím nezaslali řešení úloh z prvního čísla, můžete je tedy posílat i nadále.

Koncept

V kryptografii nazýváme hashovací funkcí H takovou funkci, která pro libovolný vstup vrátí výstup pevně stanovené délky a splňuje následující podmínky:

1. Pro daný hash c je obtížné najít x takové, že $H(x) = c$. Tedy pro zadaný hash je obtížné najít vzor, který je pomocí hashovací funkce na tento hash převeden.

2. Pro daný vstup x je obtížné najít y tak, aby $H(x) = H(y)$. Neboli je obtížné najít druhou vstupní hodnotu se stejným hashem.

3. Je obtížné najít vstupy x a y, aby $H(x) = H(y)$. Neboli je obtížné najít libovolné dvě vstupní hodnoty se stejným hashem.

Označením, že je něco obtížné, v tomto případě chceme vyjádřit, že toho nelze dosáhnout výrazně snáze než pomocí zkoušení variant hrubou silou. Pokud budeme mít hashovací funkci s výstupem o velikosti 64 bitů a zkusíme zahashovat $2^{64} + 1$ hodnot, určitě najdeme dvě se stejným hashem. Dokonce s velkou pravděpodobností najdeme dvě hodnoty se stejným hashem mnohem dříve. Neměla by ale existovat žádná metoda, která je statisticky úspěšnější než zkoušet hrubou silou hashovat náhodné hodnoty.

Úloha 1 [1b]: *V kryptografii je obecně za úspěšný považován libovolný útok, který má alespoň 50% šanci na úspěch. Představme si ideální hashovací funkci, která má výstup o velikosti 64 bitů. Kolik výpočtů hashovací funkce budeme muset provést, abychom s pravděpodobností alespoň 50 % našli dvě hodnoty se stejným hashem?*

Zároveň ještě od hashovací funkce z praktických důvodů typicky chceme, aby drobná změna vstupního textu vedla k velké změně výsledného hashe a abychom z hashe nedokázali získat ani část informace o vstupu.

Využití

Hashovací funkce mají celou řadu uplatnění. Jedním z nejpodstatnějších je ověření, že nějaká data nebyla změněna. Stačí si z dat spočítat hash a ten nějakým důvěryhodným způsobem uložit. Mohu si ho třeba i elektronicky podepsat (i k tomu se v rámci tohoto tématu ještě dostaneme). Pokud chceme později ověřit, že v datech nedošlo k žádné úpravě, stačí z nich znovu spočítat hash a hashe porovnat.

Zcela jiné využití hashovacích funkcí najdeme například při ukládání hesel. Představme si webovou stránku, kam se uživatelé přihlašují pomocí jména a hesla. Aby hesla nemusela být na serveru uložena jako prostý text, který si může kdokoli přečíst, bývá zvykem na serveru ukládat pouze hashe hesel. Ve chvíli, kdy se uživatel chce přihlásit, pošle na server heslo. Na serveru je z tohoto hesla spočítán hash a porovnán s uloženým záznamem. Pokud jsou stejné, uživatel zadal nejspíš správné heslo a je přihlášen.

Problém 2 [2b+]: *Hashování hesel je určitě vhodným postupem. Pokud ale jednoduše spočítáme hash z hesla, stále můžeme při odcizení hashů čelit určitým rizikům. Napadá vás, kde jsou nedostatky takového postupu? Dokážete najít a popsat, jak ukládat hesla lépe? Napovíme, že možností na vylepšení existuje celá řada.*

Praktická realizace

Už víme, jaké vlastnosti by hashovací funkce měla mít. Zbývá vyřešit otázku, jak takovou funkci sestrojit.

V současné době můžeme za nejrozšířenější hashovací funkce považovat MD5, SHA1 nebo rodinu funkcí SHA2 (SHA256, SHA512). Ty všechny využívají pouze klasické binární operace and, or, xor, modulární sčítání a rotace, případně shifty. Více informací o těchto operacích najdete v prvním čísle. Zmíněno tam není akorát modulární sčítání. Na této operaci ale není nic složitého – jedná se o klasické sčítání s čísly pevné maximální délky, kde u nejvyššího řádu nezohledňujeme přenos výš. Vedle uživatelského vstupu (libovolné délky) funkce vždy využívají i řadu pevných konstant.

Funkce RIKSHA

Abychom si mohli s konstrukcí hashovacích funkcí trochu pohrát, tak pro vás máme naši vlastní funkci jménem RIKSHA [čti *rikša*]. Princip, jak funkce pracuje, je podobný jako u běžných hashovacích funkcí.

Nabízíme vám ji ve formě funkčního zdrojového kódu pro Python, o kterém doufáme, že je dostatečně názorný. Pokud by vám nebylo jasné, co některá část kódu dělá, nebojte se poslat dotaz do tématkové e-mailové konference.

```
#!/usr/bin/env python3
import sys
```

```
# konstanty
c0 = 0x56
c1 = 0x37
c2 = 0xa5
c3 = 0x9d
c4 = 0xbf

# inicializace casti vysledneho hashe
h0, h1, h2, h3, h4, h5, h6, h7 = c0, c1, c0, c1, c0, c1, c0, c1

# rotace vlevo o b bitu 8-bitoveho cisla n
def rotl(n, b):
    return ((n << b) | (n >> (8 - b))) & 0xff

# pridej "binarni 1", dopln 0x00 na nasobek 8 bytu
def pad(text):
    text += '80'
    text += '0' * (16 - len(text) % 16)
    return text

# uprav hash podle casti vstupu
def update(chunk):
    global h0, h1, h2, h3, h4, h5, h6, h7
    # rozdel vstup na (decimalni) cisla
    d = []
    for i in range(8):
        d.append(int(chunk[2*i:2*i+2], 16))
    # spocitej mezihodnoty
    a0 = ((d[2] << 3) ^ (d[4] << 3) ^ (d[6] << 3) ^ (c2 << 5)
         ) & 0xff
    a1 = ((d[1] << 4) ^ (d[3] << 4) ^ (d[5] << 4) ^ c3) & 0xff
    a2 = rotl(d[0], 3) ^ rotl(d[7], 5) ^ c2
    a3 = ((d[0] << 7) & (d[3] >> 2) ^ (c3 << 2)) & 0xff
    a4 = (rotl(d[0], 3) ^ rotl(d[7], 5) ^ (d[4] << 7) ^ c4
         ) & 0xff
    # uprav hodnotu hashe
    h0_puvodni = h0
    h0 = (h0 + h1 + (a0 << 5)) & 0xff
    h1 = (h1 + h2 + (a1 << 3)) & 0xff
    h2 = (h2 + h3 + (a2 << 2)) & 0xff
    h3 = (h3 + h4 + a3) & 0xff
    h4 = (h4 + h5 + a2 + a4) & 0xff
    h5 = (h5 + h6 + a3) & 0xff
```

```
    h6 = (h6 + h7 + a2 + a4) & 0xff
    h7 = (h7 + h0_puvodni + (a1 << 4)) & 0xff

# spocitej RIKSHA (vstup v hex ASCII bez mezer)
text = pad(sys.argv[1])
while len(text) > 0:
    update(text[0:16])
    text = text[16:]
print('%.2X'*8 % (h0, h1, h2, h3, h4, h5, h6, h7))
```

Zdrojový kód v elektronické podobě si můžete stáhnout z našeho webu. Na vás je prozkoumat, jestli je RIKSHA dobrou hashovací funkcí.

Úloha 3 [1b]: *Všimněte si funkce* `pad`*, která připojuje za zprávu před doplnění nulami vždy stejnou konstantu. Tento postup jsme si vypůjčili od běžně používaných hashovacích funkcí. Napadá vás, k čemu je tato konstrukce dobrá?*

Úloha 4 [2b]: *Dokážete najít dvě vstupní hodnoty, pro které je výsledný hash stejný?*

Úloha 5 [3b]: *Víte, že pomocí funkce RIKSHA byl zahashován text, který není delší než 8 bytů. Výsledný hash je* `02365E1E061E62BA`*. Dokážete najít nějaký (ne nutně stejný) vstup, který má stejný hash?*

Úloha 6 [5b]: *Existuje nějaká hodnota hashe, kterou funkce RIKSHA nevrátí pro žádný vstup? Nezapomeňte své tvrzení důkladně zdůvodnit.*

Kuba, Káťa a Lenka; `jakub.topfer@matfyz.cz`
e-mailová konference: `krypto@mam.mff.cuni.cz`