# Téma 2 – Principy kryptografie

## Asymetrické šifrování

V předchozích dílech jsme si představili blokové šifry. Pro ně platí, že existuje jeden univerzální klíč a kdo ho zná, může šifrovat i dešifrovat zprávy. Dnes si ukážeme koncept asymetrické kryptografie, tedy takové šifrování, kde existují dva oddělené klíče. *Veřejný klíč* je možné využít pouze pro zašifrování textu. Pro dešifrování je potřebné znát *soukromý klíč*.

### Jak to funguje

Asymetrickou šifru si můžeme představit jako dvě oddělené krabičky – jedna slouží pro šifrování a druhá pro dešifrování. Pro šifrování je nutná znalost veřejného klíče, pro dešifrování znalost soukromého klíče.

Jednoznačně nejrozšířenějším algoritmem pro asymetrické šifrování je RSA. Z dalších algoritmů můžeme jmenovat ještě třeba ElGamal.

Na rozdíl od zatím představených kryptografických komponent jsou algoritmy využívané v asymetrické kryptografii obvykle založeny na těžkých matematických problémech, jejichž obtížností se zabývala již celá řada matematiků.

### Modulární aritmetika

Než se pustíme do přesného popisu RSA, zavedeme si značení, se kterým budou naše úvahy jednodušší. Říkáme, že čísla $a$ a $b$ jsou *kongruentní* modulo $n$, pokud dávají po dělení číslem $n$ stejný zbytek. Tuto skutečnost zapisujeme $a \equiv b \pmod{n}$. Číslu $n$ říkáme *modul*.

Například platí $37 \equiv 82 \pmod{15}$. Můžete si rozmyslet, že vztah $a \equiv b \pmod{n}$ je ekvivalentní s tvrzením $n|a-b$. V našem příkladě skutečně $15|37-82$.

Pro odůvodnění korektnosti RSA budeme ještě potřebovat jedno užitečné tvrzení z teorie čísel.

Pro přirozené číslo $n$ označme jako *Eulerovu funkci* $\varphi(n)$ počet čísel menších nebo rovných $n$, která jsou s $n$ nesoudělná.

Například čísla nesoudělná s devítkou jsou všechna, která nejsou dělitelná třemi (1,2,4,5,7,8), takže $\varphi(9) = 6$. Podobně můžeme spočítat, že $\varphi(11) = 10$ nebo $\varphi(12) = 4$.

Obecně platí, že pokud $n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ je prvočíselný rozklad čísla $n$, tak

$$\varphi(n) = (p_1 - 1)p_1^{\alpha_1 - 1}(p_2 - 1)p_2^{\alpha_2 - 1} \dots (p_k - 1)p_k^{\alpha_k - 1}.$$

**Problém 1** [1b]: *Zdůvodněte, že vzorec pro výpočet Eulerovy funkce je správný.*

Nechť $a$ a $n$ jsou nesoudělná přirozená čísla. Potom platí

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Tomuto tvrzení se říká *Eulerova věta*.

Například již víme, že $\varphi(9) = 6$. Podle Eulerovy věty pro $a = 14$ a $n = 9$ tedy platí $14^{\varphi(9)} = 14^6 \equiv 1 \pmod 9$. Spočítejte si, že to je skutečně pravda.

**Problém 2** [1b]: *Platí Eulerova věta i pro přirozená čísla $a$ a $n$, jejichž největší společný dělitel je 2? Pokud ano, zdůvodněte. Pokud ne, najděte protipříklad.*

## RSA

Šifra RSA je založena na obtížnosti problémů faktorizace a diskrétního logaritmu. *Faktorizací* matematici nazývají rozklad čísla na prvočísla. V RSA se využívá skutečnosti, že pokud vynásobíme dvě velká prvočísla, je obtížné takto vzniklé číslo faktorizovat.

Logaritmus je obecně funkce, která vrací exponent, na který musíme umocnit základ, abychom dostali zadané číslo[1]. Jako *diskrétní logaritmus* se označuje ta samá úloha, jen počítaná v celých číslech (místo reálných) modulo nějaké pevně stanovené číslo. Najít hodnotu diskrétního logaritmu je považováno za výpočetně obtížný problém.

Označme si $\log_2$ logaritmus o základu 2. Potom $\log_2 32 = 5$, protože $2^5 = 32$. Pokud budeme počítat diskrétní logaritmus modulo 11, tak nám vyjde $\log_2 10 \equiv 5$, protože $2^5 \equiv 10 \pmod{11}$.

Abychom mohli použít RSA, musíme si nejdříve vygenerovat veřejný a soukromý klíč. To můžeme udělat v následujících krocích:

1. Najdeme dvě dostatečně velká a dostatečně náhodná prvočísla $p$ a $q$.
2. Spočítáme $n = pq$ a $\varphi(n) = (p-1)(q-1)$. Hodnota $n$ bude využívána jako modul pro všechny operace. Hodnota $\varphi(n)$ musí zůstat utajená a po dopočítání klíčů ji můžeme zapomenout.
3. Zvolíme *veřejný exponent* $e$ tak, aby $1 < e < \varphi(n)$ a $e$ a $\varphi(n)$ byla nesoudělná.
4. Dopočítáme *soukromý exponent* $d$ tak, aby $ed \equiv 1 \pmod{\varphi(n)}$.

Jako veřejný klíč použijeme dvojici $(n, e)$, jako soukromý klíč dvojici $(n, d)$.

Šifrování a dešifrování je nyní již jednoduché. Pokud chce kdokoli zašifrovat zprávu $m$, může to udělat pomocí veřejného klíče jako $c = m^e \pmod n$.

Dešifrovat zašifrovanou zprávu může pouze vlastník soukromého klíče umocněním zašifrované zprávy na $d$ opět modulo $n$, tedy $m = c^d \pmod n$.

Z definice veřejného a soukromého exponentu musí existovat nezáporné celé $h$, pro které $ed = 1 + h\varphi(n)$. S využitím Eulerovy věty tak můžeme dopočítat:

$$c^d = m^{ed} = m^{1+h\varphi(n)} = m\left(m^{\varphi(n)}\right)^h \equiv m\,(1)^h = m \pmod n.$$

Tím jsme dokázali korektnost algoritmu.

[1] Více informací o logaritmech můžeš najít například na `https://matematika.cz/logaritmy`.

**Problém 3** [2b]: *Mějme RSA s veřejným klíčem (493, 33) a soukromým klíčem (493, 353). Přišla vám zašifrovaná zpráva 93. Jaká je hodnota původního otevřeného textu?*

Znaky místo čísel můžeme šifrovat stejně jako v případě blokových šifer – můžeme využít například jejich ASCII reprezentaci.

**Problém 4** [2b+]: *Jaké jsou zvyklosti pro volby velikosti modulu a veřejného a privátního exponentu? Můžete vycházet z veřejně dostupných doporučení. Bonusové body jsou za vlastní průzkum. Například téměř každý server umožňující šifrovaný přístup přes HTTPS využívá certifikát s RSA klíčem. Podívat se, jaké hodnoty jsou využívány v praxi, tedy není vůbec obtížné.*

**Problém 5** [3b]: *Protože problém faktorizace je sice obtížný, ale nikoli neřešitelný, doporučuje se RSA klíče po nějaké době (třeba po několika letech) vyměnit. To vyžaduje vygenerovat dvě nová velká prvočísla. To je výpočetně náročné, a tak jsme se rozhodli vždy generovat jenom jedno velké prvočíslo a druhé nechat z minulého klíče. Je takový přístup bezpečný?*

### Vlastnosti RSA

**Problém 6** [3b+]: *Představte si, že znáte dvě zašifrované zprávy $c_1$ a $c_2$, které vznikly zašifrováním neznámých textů $m_1$ a $m_2$. Dokážete z nich spočítat zašifrovanou zprávu, která odpovídá $m_1 \cdot m_2$ nebo $m_1^2 \cdot m_2^3$? Jak by tomu bylo možné zamezit?*

## Řešení problémů z 2. čísla

### Úloha 1

**Zadání:**
*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?*

**Řešení:**
Na úvod se musíme omluvit, že úloha byla mnohem obtížnější, než jsme chtěli zadat. Každý kryptograf ví, že podle narozeninového paradoxu pro blok velikosti $n$ nastává kolize přibližně po $2^{n/2}$ pokusech. V našem případě tedy budeme muset provést přibližně $2^{32}$ výpočtů. V praxi žádný přesnější výpočet potřeba není. Pokud ale chceme najít přesné řešení, musíme se pustit do složitých výpočtů.

Celkem existuje $2^{64}$ různých hashů. Předpokládejme, že pravděpodobnost, že výpočtem získáme konkrétní hash, je rozdělená rovnoměrně. První hash se určitě nebude shodovat s žádným již spočítaným. Pravděpodobnost, že ani druhý hash se nebude shodovat s prvním, je $(2^{64} - 1)/2^{64}$. Pravděpodobnost, že $i$-tý vypočítaný

hash se nebude shodovat s žádným předchozím, vyjadřuje vzorec $(2^{64} - i)/2^{64}$. Takže pravděpodobnost, že mezi prvními $k$ hashi nebudou žádné dva stejné, můžeme vyjádřit jako

$$\mathrm{P}(k) = 1 \cdot \left(\frac{2^{64}-1}{2^{64}}\right) \cdots \left(\frac{2^{64}-(k-1)}{2^{64}}\right) = \frac{2^{64}!}{2^{64k}(2^{64}-k)!}.$$

Nás zajímá jev opačný, tedy potřebujeme najít nejmenší $k$ takové, aby platilo

$$1 - \mathrm{P}(k) \geq \frac{1}{2}.$$

Analyticky vyřešit tuto nerovnici je nad naše matematické schopnosti. Dokonce jenom vyčíslit hodnotu pro $k = 2^{32}$ je velmi obtížné.

Vzpomeneme si proto, že umíme programovat, a necháme počítač, aby úlohu spočítal za nás. Následující skript vychází ze vzorce uvedeného výše. Pro jednotlivá $k$ postupně počítá pravděpodobnost kolize a testuje, zda je již dostatečně velká.

```
#/usr/bin/env python3
pst = 1
k = 1
h2na64 = pow(2,64)
while True:
    pst *= (h2na64-(k-1))/h2na64
    if pst <= 0.5:
        print(k, pst)
        break
    k += 1
```

Na mém nepříliš výkonném počítači jsem se výsledku dočkal přibližně za půl hodiny. Hledané $k$ vyšlo 5056937540. Tato hodnota je ovlivněna skutečností, že Python standardně zaokrouhluje na 16 desetinných míst (počítat s větší přesností by bylo možné, ale výpočetně náročnější). Skutečnému řešení se ale bude velmi blížit.

Platí $\log_2(5056937540) \doteq 32{,}2$, takže vidíme, že původní odhad $2^{32}$ byl poměrně dobrý.

## Problém 2

**Zadání:**
*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.*

**Řešení:**
Správnému řešení tohoto problému byli blízko Dr.MM Beáta Hroncová i Mgr.MM Tomáš Sourada.

Hlavním problémem prostého hashování hesel je skutečnost, že pokud mají dva uživatelé stejné heslo, mají i stejný hash hesla. To výrazně usnadňuje potenciálnímu útočníkovi, který hash či hashe získá, nalezení původního hesla (tomuto postupu se říká *crackování*). Pokud použijeme dobrou hashovací funkci, je nejefektivnější možností, jak najít původní heslo, zkoušet zahashovat různá hesla a porovnávat výsledek se získanými hashi. Když má útočník k dispozici mnoho hashů, může výsledek svého snažení zkusit hledat najednou v celém seznamu získaných hashů. Navíc hashe častých hesel zahashovaných pomocí obvyklých hashovacích funkcí je možné dokonce i vygooglit, což se týká dokonce i českých hesel. Zkuste třeba najít, jaké heslo má hash (spočítaný neznámou funkcí) 29d847ffce86b63c39a756a25b198751.

Řešením tohoto problému je takzvané *solení* hesel. To spočívá v tom, že pro každého uživatele vygenerujeme unikátní náhodný řetězec nazývaný *sůl* a uložíme si dvě hodnoty: sůl a hash hesla spojeného se solí. Pokud nám nějaký útočník ukradne data z databáze, získá hashe i soli a může tedy opět crackovat hesla, ale pro každý hash bude muset dělat výpočty zvlášť a hlavně nebude moct využít žádné předpočítané hodnoty.

Toto řešení je možné i dále vylepšovat. Například k heslu nemusíme před hashováním přidávat pouze sůl, ale můžeme připojit i nějakou konstantu specifickou pro celou aplikaci, která není uložená nikde v databázi. Té se říká pro změnu *pepř*.

Ještě si můžeme říct něco o crackování hesel. Obvyklým postupem je, že si útočník vezme nějaký seznam častých hesel a ty zkouší postupně hashovat. Tento seznam se ještě pokouší nějak sám doplnit, například k němu přidá všechna jména a sportovní kluby, které jsou pro oblast, ze které očekává uživatele, relevantní. Když mu to nestačí, může si vzít třeba i seznam všech slov v předpokládaném jazyce. Pokud ani to nepomůže, obvykle přijdou na řadu pravidla, jak hesla ze seznamu modifikovat. Může se například pokusit za každé slovo z jazyka přidat několik čísel nebo měnit velikost počátečních písmen. Tato znalost může být užitečná, až si budete vymýšlet nějaké své heslo. To by obecně mělo být dostatečně dlouhé (za naprosté minimum lze považovat 8 znaků, ale delší bude lepší) a pro útočníka by mělo být obtížné ho pomocí pravidel vytvořit. Pokud použijete čtyři slova a vložíte mezi ně nějaké číslo, pravděpodobně útočník s crackováním neuspěje.

## Úloha 3

**Zadání:**

*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?*

**Řešení:**

Konstanta ve funkci `pad` slouží jako alespoň částečná obrana před výpočtem hashe z prodlouženého textu bez znalosti původního (*length extension attack*).

Pokud bychom padding nepoužívali, mohl by útočník na základě znalosti hashe spočítat hash původní jemu neznámé zprávy doplněné o nějaký další text. Takový útok nemá příliš mnoho praktických uplatnění, ale je to něco, co je obecně spíše nežádoucí.

## Úlohy 4, 5 a 6

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

**Zadání úlohy 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?*

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

K těmto úlohám zatím nepřišlo žádné řešení, a tak jsme se je rozhodli ponechat minimálně do dalšího čísla otevřené a budeme se nadále těšit na vaše řešení.

Ke čtvrté úloze napovíme, že k vyřešení stačí pochopit, jak funguje funkce RIKSHA. Měli byste ji zvládnout zcela bez programování.

Naopak u páté úlohy se bez programování neobejdete. Doplníme informaci, že text se skládal pouze z malých a velkých písmen anglické abecedy.

Náročnost šesté úlohy odpovídá jejímu bodovému hodnocení. Pokud ale chcete pouze získat pět bodů, tak existují i snazší cesty.

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