KRY – výměna klíčů DH a ECDH přes nezabezpečený kanál

Ondřej Koumar, xkouma02

1 Implementace

Začal jsem implementací klasické Diffie-Hellman výměny než jsem se pustil do eliptických křivek. Nic, co by bylo překvapující:

- zpracování argumentů, zahájení komunikace přes schránky na `localhostu`, vygenerování privátních klíčů,
- výpočet veřejného klíče nad konečným tělesem,
- výměna veřejných klíčů a výpočet sdíleného tajného klíče.

Délka privátního klíče v bytech (a tím pádem také sdílených veřejných klíčů) je stejná jako délka společného domluveného modulu. Privátní klíče byly vygenerovány knihovnou `secrets`, která obstarává kryptograficky bezpečné generátory náhodných čísel. Interval, ve kterém se privátní klíč nachází, je $\langle 2, p-1 \rangle$.

Pro výpočty nad konečným tělesem jsem použil funkci `pow(base, exp, mod)`, která je dostupná v Pythonu bez jakýchkoliv knihoven. Konkrétně to v kódu vypadá takto (výpočet svého veřejného klíče je skoro identický):

```
def calculate_shared_key(public_key: int, private_key: int) -> int:
    return pow(public_key, private_key, p)
```

kde `public_key` je přijatý od druhé strany, `private_key` je vlastní.

Komunikace mezi klientem a serverem je implementována pomocí *TCP* schránek, které dávají mnohem větší smysl pro robustnější komunikaci mezi dvěma stranami než *UDP*. Vzhledem k důležitosti správné komunikace při výměně klíčů UDP protokol nepřipadal v úvahu. Pořadí operací (`send` a `recv`) u klienta a serveru musí být samozřejmě opačné, já na straně serveru zvolil prvně přijetí veřejného klíče od klienta, až poté poslání svého veřejného klíče klientovi. Klient tedy prvně posílá svůj veřejný klíč, poté přijme veřejný klíč serveru.

1.1 Eliptické křivky

Komunikace pro výměnu klíčů na jako bodů na eliptické křivce probíhá úplně stejně, jako bylo popsáno v posledním odstavci.

V rámci projektu jsem implementoval sčítání a násobení bodů na eliptických křivkách. Vycházel jsem z předpokladu, že eliptická křivka má rovnici $y^2 = x^3 + a \cdot x + b$. Pro tuto rovnici je definováno sčítání bodů $\mathcal{P}$ a $\mathcal{Q}$ následovně:

$$\begin{aligned} \mathcal{P} + \mathcal{Q} &= \mathcal{R} \\ (x_p, y_p) + (x_q, y_q) &= (x_r, y_r) \\ \lambda &= \frac{y_q - y_p}{x_q - x_p} \\ x_r &= \lambda^2 - x_p - x_q \\ y_r &= \lambda(x_p - x_r) - y_p \end{aligned}$$

Dále víme, že bod v nekonečnu je identitou. Takový bod získáme sečtením bodu a jeho inverze. Pro sčítání dvou stejných bodů je třeba použít jiný vzorec pro λ (jinak by se dělilo 0), který vypadá následovně:

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

Zbytek se pak počítá stejně. Důležité je, že dělení v našem případě bude multiplikativní inverze v konečném tělese, ne klasické dělení, které známe z klasické aritmetiky. Kód implementující sčítání bodů (chybí ošetření edge-cases, vizte kód v odevzdaných souborech):

```
# Specialni vzorec pro dva stejne body
if p == q:
    # Osetreni deleni nulou
    if y_p == 0:
        return self.infinity
    numerator = (3 * pow(x_p, 2, self.p) + self.a) % self.p
    denominator_inv = pow((2 * y_p) % self.p, -1, self.p)
    l = (numerator * denominator_inv) % self.p
else:
    numerator = (y_q - y_p) % self.p
    denominator_inv = pow(x_q - x_p, -1, self.p)
    l = (numerator * denominator_inv) % self.p

x_r = (pow(l, 2, self.p) - x_p - x_q) % self.p
y_r = (l * (x_p - x_r) - y_p) % self.p
return (x_r, y_r)
```

Multiplikativní inverze je počítána právě funkcí `pow(base, exp, mod)`, kde `exp` je -1, `mod` je velikost konečného tělesa.

Násobení bodů je třeba implementovat jinak než naivně, já zvolil implementaci algoritmem *double-and-add*. Počítáme $k \cdot \mathcal{P}$. Protože k je hodně-bitové číslo (řekněme 256 pro tento příklad), mohlo by se provést $\mathcal{O}(2^{256})$ operací, což je neproveditelné. Double-and-add využívá binární reprezentace k; $k = \sum_{i=0}^{m} b_i 2^i$. Násobení bodu se pak dá přepsat jako $\sum_{i=0}^{m} b_i(2^i \cdot \mathcal{P})$. Víme, že $2 \cdot \mathcal{P} = \mathcal{P} + \mathcal{P}, 4 \cdot \mathcal{P} = 2\mathcal{P} + 2\mathcal{P}, \ldots, 2^i \cdot \mathcal{P} = 2^{i-1}\mathcal{P} + 2^{i-1}\mathcal{P}$. Mezivýsledky sčítání můžeme kumulovat během iterací, kterých je lineární počet vzhledem k počtu bitů, na kterých je skalár uložen, logaritmický počet vzhledem k hodnotě skaláru v desítkové soustavě. Kód (chybí zde hodně ošetření edge-cases, ukazuji jen samotný double-and-add algoritmus):

```
k_temp = k
while k_temp > 0:
    # Pricitame pouze tam, kde bit je 1
    if k_temp & 1:
        result = self.point_add(result, temp)
    temp = self.point_add(temp, temp)
    k_temp >>= 1

return result
```

Další implementační detaily nepovažuji za natolik důležité, aby zde byly zmíněny nebo rozebrány více do podrobna.

2 Experimenty

Experimentoval jsem převážně s rychlostí výpočtu veřejných a sdílených klíčů. Očekával jsem, že operace na eliptických křivkách budou rychlejší, vzhledem k velikosti klíče (2048 vs 256 bitů), protože na eliptických křivkách se provede řádově míň operací, byť o něco náročnějších.

Má očekávání se splnila. V kódu mám implementovanou funkci `debug_message(msg)`, která vypisuje řetězec jí předaný na standardní výstup, pokud globální konstanta `DEBUG_MODE` je povolená. Každé generování klíčů je obehnáno časovači a pro účely zjištění náročnosti algoritmů jsem si vypisoval dobu trvání na standardní výstup (při kontrole kódu si jistě všimnete, že jsem si vypisoval skoro vše, snad to bude čitelné i přes všudypřítomné volání `debug_message()`).

Po několika měřeních jsem zjistil, že algoritmy na eliptické křivce jsou zhruba 20× rychlejší. Nicméně musíme vzít v potaz délku klíče, která je o dost menší. Pokud by klíče byly stejně dlouhé, věřím, že časy se dost vyrovnají.

3 Bezpečnost

Klasický algoritmus Diffie-Hellmann stojí na problému diskrétních logaritmů v multiplikativní konečné grupě, kde máme předem smluvené parametry p, g, kde p je velké prvočíslo, kterým modulujeme, g je generátor této cyklické grupy. Mějme parametry p, g, soukromý klíč a. Veřejný klíč A se spočítá jako

$$A = g^a \pmod{p}.$$

Problém je zjistit a.

Tento problém je výpočetně dost náročný, ale dnes se za bezpečnou variantu považuje použití alespoň 3072-bitového klíče.

ECDH používá stejného matematického problému, ale na základě jiných grup, a to konečných grup eliptických křivek. Známe parametry křivky, smluvený bod $\mathcal{P}$ a vypočítáme veřejný klíč $\mathcal{Q} = d \cdot \mathcal{P}$, kde d je soukromý klíč. Pro stejně velké klíče je diskrétní logaritmus na eliptických křivkách výrazně těžší. 256-bitová eliptická křivka použitá v tomto projektu má zhruba stejnou bezpečnost jako použití 3072-bitového klíče pro klasický Diffie-Hellman algoritmus. Kromě vyšší bezpečnosti je také podstatně rychlejší díky menším klíčům, navíc mezi klientem a serverem přenáší menší data. Na druhou stranu, má o něco složitější matematický základ.