Úvod do kryptografie

Jarek Kusák

March 2025

1 Úvod do kryptografie

1.1 Co je to kryptografie

Intuitivní vysvětlení: Kryptografie je věda o tom, jak posílat tajné zprávy tak, aby jim rozuměl jen ten, komu jsou určeny. Když si píšeš se svým kamarádem a nechceš, aby vám někdo četl zprávy, použiješ kryptografii.

Formální definice: Kryptografie je disciplína informatiky a matematiky, která se zabývá návrhem a analýzou technik pro zabezpečení komunikace v přítomnosti potenciálních útočníků. Kryptografie se dělí na několik oblastí: šifrování, autentizace, digitální podpisy, hashování a další.

1.2 Kryptografická primitiva

Intuitivní poznámka: Kryptografická primitiva jsou jako základní stavební kameny – jednoduché funkce, které se dají kombinovat pro stavbu složitějších systémů, jako jsou protokoly.

- **Symetrické šifry:**
  - **Intuitivně:** Odesílatel a příjemce mají stejný klíč (např. tajné heslo), kterým se zpráva zašifruje i dešifruje.
  - **Příklad:** AES (Advanced Encryption Standard).
- **Asymetrické šifry:**
  - **Intuitivně:** Každý má dva klíče – veřejný a soukromý. Co jeden zašifruje, druhý může rozšifrovat.
  - **Příklad:** RSA – veřejný klíč se používá pro šifrování, soukromý pro dešifrování.
- **Hešovací funkce (hash functions):**
  - **Intuitivně:** Vezme libovolně dlouhou zprávu a převede ji na krátký otisk (řetězec fixní délky).

  - **Vlastnosti:** Deterministická, odolná vůči kolizím, nelze z otisku zpětně získat původní zprávu.
  - **Příklad:** SHA-256.

- **Náhodné generátory (randomness generators):**
  - **Intuitivně:** Dávají náhodná“ čísla, důležitá např. pro generování klíčů.
  - **Rozdělení:** Hardwarové RNG (skutečná náhoda) vs. Pseudonáhodné generátory (PRNG, deterministické, ale vypadají náhodně).

1.3 Protokoly a role

Intuitivně: Protokol je jako scénář – předepisuje, co kdo kdy říká nebo dělá. Role jsou účastníci toho scénáře (např. Alice, Bob, útočník).

Příklad: Alice chce poslat Bobovi šifrovanou zprávu – protokol říká, že zašifruje zprávu Bobovým veřejným klíčem a pošle mu ji. Bob ji rozšifruje svým soukromým klíčem.

1.4 Kerckhoffsův princip

Intuitivně: Bezpečnost systému by měla záviset pouze na tajnosti klíče, ne na tajnosti algoritmu. Jinými slovy – i kdyby útočník znal přesný postup šifrování, neměl by být schopen zprávu rozluštit bez klíče.

Formálně: Kryptografický systém by měl být bezpečný, i když je vše kromě klíče veřejné.“

2 Kryptografické útoky

2.1 Druhy útoků

Intuitivně: Útočníci se snaží z různých dostupných informací zjistit něco tajného – buď obsah zprávy (plaintext), nebo přímo šifrovací klíč. Typ útoku se liší podle toho, co útočník zná nebo dokáže ovlivnit.

1. **Known ciphertext attack**
   - Známe pouze zašifrovaný text (ciphertext)
   - Cílem je zjistit odpovídající plaintext
2. **Known plaintext attack**
   - Známe páry (plaintext, ciphertext)
   - Cílem je zjistit klíč, který byl použit k zašifrování
3. **Chosen plaintext attack**

- Můžeme si sami zvolit plaintext a získat jeho šifrovanou podobu
- Cílem je zjistit klíč

4. **Rozlišovací útoky (distinguishing attacks)**
   - Chceme zjistit, jestli konkrétní ciphertext odpovídá nějakému typu plaintextu
   - Např. pokud šifra leakuje délku plaintextu

2.2 Jak měřit obtížnost útoku

Intuitivně: Obtížnost útoku se udává v bitech – čím více bitů, tím těžší je šifru prolomit. Udává se, kolik pokusů by útok vyžadoval.

- **Security level** (v bitech): pro prolomení šifry je potřeba $2^{\text{security level}}$ pokusů
- Pozor: security level $\leq$ velikost klíče

3 Narozeninové útoky

Intuitivně: Představme si třídu lidí – každý má narozeniny jen jednou v roce, ale i tak je poměrně vysoká pravděpodobnost, že dva lidé mají narozeniny ve stejný den. Podobně i u kryptografických funkcí může docházet ke kolizím, pokud výstupní prostor není dostatečně velký. Útočník pak může takovou kolizi využít, například k nalezení dvou různých vstupů, které mají stejný výstup (např. stejný hash).

Poznámka: Útočník *nepředpočítává* nonci. Nonce (number used once) má být náhodná hodnota, která se nikdy nezopakuje. Pokud je ale velikost prostoru možných hodnot malá (např. 2), pak se i při náhodném výběru může některá hodnota opakovat – a toho využívá narozeninový útok.

3.1 Challenge-response autentikace

- Máme n různých *nonce* (např. 64bitových = 2^{64} možných hodnot)
- Ptáme se: kolik pokusů m (např. výzev v challenge-response) potřebujeme, než se některý nonce zopakuje?
- Modelujeme to jako pravděpodobnost, že náhodná funkce f z $[m]$ do $[n]$ (m pokusů, n možností) je prostá:

$$P = \frac{\#\text{prostých funkcí}}{\#\text{všech funkcí}} = \frac{n!}{m! \cdot n^m} \approx e^{-\frac{m(m-1)}{2n}}$$

- Pro pravděpodobnost kolize $P[\text{kolize}] = \frac{1}{2}$ dostáváme:

$$\frac{m(m-1)}{2n} \approx \ln 2 \Rightarrow m \approx \sqrt{n}$$

- **Důsledek:** Pokud má funkce výstup dlouhý n bitů, tak k dosažení kolize s pravděpodobností 50 % stačí zhruba $2^{n/2}$ pokusů. To znamená, že **security level vůči kolizím je polovina délky výstupu**.

4 Welcome message útok

Intuitivně: Útočník využívá toho, že ví, jak vypadá první zpráva (např. "Welcome Bob!") – tedy známý plaintext. Pokud šifra není dost silná (např. krátké klíče), může si útočník předem připravit množství možných zašifrovaných verzí této zprávy pod různými náhodnými klíči a pak jen čekat, jestli se v síti objeví některá z nich.

4.1 Protokol

- Alice vygeneruje náhodný klíč k a předá ho Bobovi (např. side-channel komunikací)
- Alice pošle *welcome message* (např. známý text "Welcome!") zašifrovaný tímto klíčem k
- Následně posílá další šifrované zprávy pomocí stejného klíče k

4.2 Útok

- Útočník zná plaintext uvítací zprávy a zvolí a náhodných klíčů (např. $k_1, ..., k_a$)
- Pro každý klíč spočítá $c_i = \text{Encrypt}_{k_i}(\text{"Welcome!"})$ – vznikne množina šifrovaných zpráv A
- Poté poslouchá m relací a sleduje, zda se některý zachycený ciphertext shoduje s předpočítaným
- Pravděpodobnost, že se žádný netrefí:

$$P[\text{Im}(f) \cap A = \emptyset] = \left(1 - \frac{a}{n}\right)^m \approx e^{-\frac{ma}{n}}$$

- Pokud $m \cdot a \approx n$, pak je pravděpodobné, že k úspěchu dojde
- **Trade-off:** Útočník může snížit počet poslouchaných relací, pokud investuje více času do předvýpočtu, a naopak

Shrnutí: Útok je možný jen tehdy, pokud:

- Útočník zná plaintext první zprávy
- Klíče nejsou dost dlouhé (malý klíčový prostor)
- Aplikace používá předvídatelné chování (např. fixní welcome message)

5 Jednorázové klíče (One-Time Pad)

Intuitivně: Představ si, že chceš někomu poslat tajnou zprávu a máte k dispozici dokonale náhodný tajný papírek“ – pro každé písmeno jiný náhodný znak. Pokud tento papírek použijete jen jednou a oba ho máte, zpráva je zcela nerozluštitelná. To je princip One-Time Padu.

- Jedná se o **Vernamovu šifru**, známou jako **One-Time Pad**.
- Zpráva $x \in \{0,1\}^n$ je binární řetězec o délce n.
- Klíč k je také binární řetězec o délce n, náhodně generovaný: $k \in \{0,1\}^n$.
- Šifrování i dešifrování probíhá pomocí bitové operace `XOR`:

$$E(x,k) = x \oplus k \quad \text{a zároveň} \quad D(E(x,k),k) = (x \oplus k) \oplus k = x$$

- **Pozor:** Klíč **nesmí být nikdy znovu použit!**
  - Pokud by útočník zachytil dvě šifrované zprávy $x \oplus k$ a $x' \oplus k$, může je spojit:
    $$(x \oplus k) \oplus (x' \oplus k) = x \oplus x'$$
  - Tím získá vztah mezi dvěma zprávami a může na ně použít statistické útoky (např. frekvenční analýzu).
  - Tento typ chyby se skutečně stal – např. Sověti ve 2. světové válce.
- Pokud zašifrovaný text nedává žádnou informaci o plaintextu, říkáme, že šifra je **perfektně bezpečná**.

Věta: Pokud má šifra být perfektně bezpečná pro n-bitové zprávy, pak délka klíče musí být alespoň $k \geq n$.

Důkaz věty o perfektní bezpečnosti

Intuitivně: Pokud máme méně možných klíčů než zpráv, pak existuje šifrovaný text, ke kterému určitá zpráva nikdy nemohla být zašifrována – a útočník to tedy může vyloučit. Tím pádem už ví něco o původní zprávě – a šifra není perfektní.

- Množina všech možných plaintextů a ciphertextů má velikost 2^n.
- Množina všech možných klíčů má velikost 2^k.
- Pokud $k < n$, pak $2^k < 2^n$ – klíčů je méně než zpráv.
- Tedy při šifrování pomocí funkce $E(x,k) = y$ nemůže každá zpráva být převedena na každý ciphertext.

- To znamená, že existuje alespoň jeden ciphertext $y \in \{0,1\}^n$, ke kterému:

$$\exists x \in \{0,1\}^n, \exists k : E(x,k) = y$$

$$\text{ale zároveň } \forall x' \neq x, \forall k' : E(x',k') \neq y$$

- Tedy po zachycení šifry y víme, že x' není možný – máme informaci o zprávě.
- Rozdělení možných zpráv není rovnoměrné $\Rightarrow$ **šifra není perfektně bezpečná**.

Závěr: Aby šifra byla perfektně bezpečná, počet možných klíčů musí být alespoň stejně velký jako počet zpráv: $k \geq n$.

6 Dělení tajemství (Shamirovo prahové schéma)

Intuitivně: Někdy nechceme, aby tajemství znal jen jeden člověk – ale zároveň nechceme, aby ho mohl rekonstruovat kdokoliv. Pomocí tzv. *threshold scheme* rozdělíme tajemství tak, že ho lze obnovit jen ze l částí z celkových k. Méně než l částí nestačí k odhalení tajemství.

- Tajemství se šifruje více klíči – pro dešifrování je potřeba **všech k klíčů**.
- Díky komutativitě operace $\oplus$ nezáleží na pořadí dešifrování.

(k,l)-prahové schéma

- Známé jako **Shamir Secret Sharing**.
- Rozdělíme tajemství x na k částí tak, že:
  - Libovolných l částí umožní **rekonstruovat celé tajemství**.
  - Žádná množina $l-1$ nebo méně částí neprozradí vůbec nic o x.

Příklad $(k,2)$-schéma

Intuitivně: Představ si tajemství jako bod x na ose y. Vybereme náhodně přímku, která tímto bodem prochází ($f(t) = a \cdot t + b$). Pak každému účastníkovi pošleme jeden další bod na této přímce (např. $f(1), f(2), ...$). Dva body určují přímku - 2 účastníci mohou zjistit x. Jeden nestačí – má jen jeden bod - existuje nekonečně mnoho přímek, které jím procházejí.

- Hledáme lineární funkci $f(t) = a \cdot t + b$, kde:

$$f(0) = x, \quad f(1) \in_R \mathbb{Z}_p$$

- Náhodně zvolíme a (sklon), $b = x$

- Pak rozesíláme $x^1 = f(1), x^2 = f(2), ..., x^k = f(k)$
- Každé dvě hodnoty x^i, x^j určují jednoznačně přímku f
- Ale jedna hodnota x^i nestačí – všechna x jsou stále stejně pravděpodobná

Závěr: Přímku určí 2 body - tajemství lze obnovit ze 2 částí, ale z 1 části vůbec nic nevyčteme.

Obecné (k, l)-schéma

Intuitivně: Pro libovolné l-prahové schéma použijeme polynomy stupně $< l$. Polynomy mají tu krásnou vlastnost, že k jejich určení potřebujeme právě tolik bodů, kolik je jejich stupeň + 1.

- Zvolíme náhodný polynom f stupně $< l$ nad konečným tělesem $\mathbb{Z}_p$:

$$f(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_{l-1} t^{l-1}$$

- Hodnota $f(0) = x$ je tajemství.
- Hodnoty $f(1), f(2), ..., f(k)$ rozdáme účastníkům jako **podíly (shares)**.
- Když se sejde l účastníků - lze určit jednoznačně f pomocí Lagrangeovy interpolace - obnovíme $x = f(0)$.
- Když jich je méně než l:
  - Můžeme si vymyslet“ libovolná pokračování zbylých bodů - každé x je stejně pravděpodobné
  - Tím je zaručena perfektní bezpečnost

Shrnutí:

- Potřebujeme l částí - určují jedinečně f - známe $x = f(0)$
- Méně než l částí - není možné f rekonstruovat - x zůstává tajné
- Tohle dělení je **informace-teoreticky bezpečné**, pokud se vše děje nad konečným tělesem $\mathbb{Z}_p$

7 Symetrické šifry

Proudové šifry

Intuitivně: Proudové šifry generují pseudonáhodný proud bitů (keystream), který pak kombinujeme s daty pomocí `XOR`. Pokud máme správný klíč a stejný keystream, můžeme zprávu dešifrovat.

- Fungují jako Vernamova šifra, ale místo náhodného klíče používají **pseudonáhodný generátor**.
- Generátor vytváří keystream, který se `xoruje` s plaintextem:

$$y = x \oplus \text{keystream}$$

- Platí $E = D$, protože `XOR` je sama sobě inverzní.
- Změna jediného bitu v x způsobí změnu příslušného bitu v y (a naopak).
- **Nikdy nesmíme opakovat IV (inicializační vektor)** – jinak dojde k opětovnému použití stejného keystreamu, což může vést k prolomení šifry.

Blokové šifry

Intuitivně: Místo práce po jednotlivých bitech zpracovávají blokové šifry data po celých blocích pevné délky – např. 128 bitů. Ke každému bloku se aplikuje složitá transformace, která závisí na klíči.

- Každý blok má délku b bitů.
- Funkce šifrování:

$$E : \{0,1\}^b \times \{0,1\}^K \to \{0,1\}^b$$

  nebo $E_K : \{0,1\}^b \to \{0,1\}^b$ – to je permutace bloků závislá na klíči.
- Dlouhé zprávy se šifrují po blocích, případně se přidává padding.

Bezpečnost blokových šifer

Intuitivně: Cílem je, aby výstup šifry vypadal náhodně“. Dobrý útok je ten, který rozpozná výstup šifry od náhodného – my chceme, aby to nešlo.

- Formálně obtížné definovat, ale používáme model:
  - Verifikátor má přístup k černé skříňce“, která buď implementuje E_K, nebo náhodnou permutaci.
  - Může pokládat více dotazů a hádat, co je uvnitř.
  - Úspěšný útok: pokud se mu podaří říct s pravděpodobností $P \geq \frac{2}{3}$, co je uvnitř, s menší než exponenciální složitostí.
- Cíl: aby žádný útok neuspěl lépe než náhodný tip.
- **Shrnutí:** Dobrý šifrovací algoritmus vypadá náhodně“.
- Reálné šifry bývají sudé permutace (poznámka z přednášky).
- Tento model neřeší útoky typu **chosen-key** nebo **related-key**.

Iterované šifry

Intuitivně: Komplikovanou šifru si vyrobíme tak, že aplikujeme jednoduchou transformaci (tzv. kolo) opakovaně. Klíč pro každé kolo je jiný.

- Každé kolo provádí základní operace (viz SPN nebo Feistel).
- Používá se tzv. **key schedule**, který z hlavního klíče K vygeneruje podklíče $K_1, K_2, ...$

Substitučně-permutační sítě (SPN)

Intuitivně: V každém kole se vstup rozbije“ (substituce = zamíchání hodnot) a přehází“ (permute = zamíchání pozic). Tím se dosáhne *confusion* a *diffusion* – dvě hlavní zbraně proti statistickým útokům.

Během jednoho kola:

1. Vstup se xoruje s klíčem K_i (whitening)
2. Prochází S-boxy (malé invertibilní tabulky) – **confusion**
3. Výstupy z S-boxů se prohodí přes P-box – **diffusion**

- Všechny části jsou invertibilní – šifru lze zpětně dešifrovat.
- XOR a permutace komutují - pořadí můžeme měnit - inverze SPN je opět SPN.

Feistelovy sítě

Intuitivně: V této konstrukci rozdělíme zprávu na dvě poloviny. V každém kole upravujeme jednu polovinu pomocí nějaké funkce (nemusí být ani invertibilní!) a pak je prohodíme. Tajemství je v tom, že celý proces je stále inverzní – i když se používají jednoduché komponenty.

1. Rozdělíme vstup na levý a pravý blok: (L, P)
2. Pravý blok zašifrujeme pomocí funkce f a klíče K_i
3. Výsledek xorujeme s levým blokem
4. Prohodíme bloky a pokračujeme do dalšího kola

Dešifrování: Stačí projet stejný proces pozpátku a v opačném pořadí klíčů K_i.

Výhoda: Feistelovy sítě fungují i s neinvertibilními komponentami – což umožňuje použití libovolné funkce f v návrhu.

8 DES – Digital Encryption Standard

Intuitivně: DES byla historicky jedna z prvních moderních šifer široce nasazených pro komerční použití. Dnes už je považována za prolomenou, ale její struktura (Feistelova síť) inspirovala další šifry. Je to dobrý studijní příklad.

Obecné vlastnosti

- Vyvinuta na začátku 70. let firmou IBM pro NBS (Národní úřad pro standardy v USA)
- Do vývoje zasahovala i NSA
- Používá 56bitový klíč (technicky 64 bitů, ale 8 bitů je jen na paritu)
- Feistelova síť s 16 iteracemi
- NSA na poslední chvíli změnila S-boxy (podezřelé, ale dnes víme, že to ve skutečnosti zlepšilo odolnost vůči diferenciální kryptoanalýze)

Struktura DESu

- Pracuje se 64bitovými bloky, rozdělenými na dvě 32bitové poloviny
- 16 kol Feistelovy sítě
- Na začátku a na konci je P-box (permutace) navíc
  - Z kryptografického hlediska zbytečný, ale ztěžuje SW implementaci
- Funkce f:
  1. Rozšíří vstupních 32 bitů na 48 bitů (pomocí E-boxu)
     - Každý 4bitový blok vezme“ krajní bity od sousedních bloků → vznikne 6bitový vstup
  2. Výsledek se xoruje s podklíčem K_i
  3. 8 S-boxů: každý mapuje 6bitový vstup na 4bitový výstup (ztrátová operace)
  4. Výsledný 32bitový výstup projde přes P-box (permutaci bitů)

Generování podklíčů

- Z 56bitového klíče se vygeneruje 16 podklíčů $K_1, ..., K_{16}$ o délce 48 bitů
- Každý podklíč je permutací a podmnožinou hlavního klíče

Kritika DESu

- **Slabé klíče:**
  - Např. pokud $K = 0^{56}$, pak všechny $K_i = 0^{48}$
  - V takovém případě je šifrování identické s dešifrováním (Feistelova síť je symetrická)
  - Existuje více slabých klíčů – např. samé jedničky
- **Inverzní vstup produkuje inverzní výstup:**

$$E_K(\bar{x}) = \overline{E_K(x)}$$

- Doplňky se ve Feistelově funkci vyruší (kvůli xorování)
- **Krátké klíče:**
  - Už v 70. letech se vědělo, že je možné zkoušet všechny klíče (brute-force)
  - V roce 2012 existovala služba, kde jste si mohli koupit prolomení DESu na zakázku
- **Krátké bloky:** 64bitový blok - kolize jednou za 2^{32} bloků (tzv. *birthday paradox*)

Útoky na DES

- **Diferenciální kryptoanalýza:**
  - Potřebuje asi 2^{47} *chosen plaintexts*
- **Lineární kryptoanalýza:**
  - Stačí pouze 2^{43} *chosen plaintexts*
  - To už je považováno za efektivní prolomení

Pokusy o záchranu DESu

- V 90. letech se zkoušelo DES zachránit pomocí vícenásobného šifrování (tzv. *multiple encryption*)
- **2-DES:** nepomáhá
  - I když se klíč zdvojnásobí, existuje tzv. *meet-in-the-middle* útok
  - Security level zůstává ≤ 57 bitů
- **3-DES:**
  - Security level je až 113 bitů
  - Používá tři průchody DESem: $E_{K_1}(D_{K_2}(E_{K_1}(x)))$
  - Dodnes se někde používá, ale je pomalý

9 AES (Advanced Encryption Standard)

Intuitivně: AES je moderní standard šifrování, který je rychlý, bezpečný a dobře se implementuje v softwaru i hardwaru. Vyhrál soutěž mezi návrhy různých šifer a stal se novým standardem po DESu.

9.1 Historie

- V roce 1997 NIST (nástupce NBS) vyhlásil soutěž na novou šifru
- Přišlo 15 návrhů, které prošly několika koly veřejného hodnocení
- Kritéria: bezpečnost, rychlost, snadná implementace
- Vítězem se stal Rijndael (2001) – dnes známý jako AES

9.2 Struktura

- AES pracuje s 128-bitovými bloky
- Klíč může být dlouhý 128, 192 nebo 256 bitů
  - Počet kol: 10 (128), 12 (192), 14 (256)
- AES je SPN (substituční-permutační síť) s lineárními transformacemi
- Byte-orientovaný: každý blok jako matice 4×4 bajtů (celkem 128 bitů)

9.3 Kroky jednoho kola šifrování

1. **ByteSub:** každý bajt projde S-boxem (nelineární nahrazení, inverze v $GF(2^8)$ + afinní transformace)
2. **ShiftRow:** řádky matice se rotují (řádek 0 zůstává, řádek 1 rotace o 1 vlevo, atd.)
3. **MixColumn:** každý sloupec projde lineární transformací (difuze)
4. **AddRoundKey:** přičteme (XOR) kulový klíč
   - před 1. kolem navíc jeden AddRoundKey

Intuitivně: AES je jako vaření – každý krok zamíchá“ a přikoření“ zprávu jinak, takže výsledek je důkladně zamaskovaný a těžko se pozná původní text.

9.4 Optimalizace

- Pro 32-bitové procesory se často spojuje S-box a části MixColumn do 4 tabulek
- Může ale vzniknout zranitelnost – útočník může sledovat, co se načítá z cache

9.5 Inverzní šifrování

- Některé kroky AES komutují (pořadí lze změnit), např. AddRoundKey a MixColumn
- Inverzní kroky:
  1. InvMixColumn
  2. AddRoundKey (s upraveným klíčem)
  3. InvByteSub
  4. InvShiftRow

9.6 Rozvrh klíčů

- Pracuje po 32-bitových slovech
- Klíče se generují z původního klíče pomocí funkce f_i
- f_i používá rotace, S-boxy a přičítání konstant

Intuitivně: Abychom každý krok šifrovali trochu jinak, vezmeme původní klíč a rozmixujeme“ ho na několik menších klíčů – každý pro jednu iteraci.

9.7 Kritika AES

- Jednoduchá algebraická struktura – možná bude v budoucnu napadnutelná
- Málo kol – optimalizováno na rychlost
- Zarovnáno na byty – difuze je předvídatelná
- 128-bitový klíč není bezpečný proti kvantovým útokům (Grover)
- 128-bitové bloky mohou kolidovat po 2^{64} blocích (prakticky měníme klíč po 2^{32} blocích)

Intuitivně: AES je velmi bezpečná, ale protože je rychlá a strukturovaná, můžeme se bát, že jednoho dne někdo najde zkratku. A kvantové počítače jí taky moc nepřejí.

10 Módy blokových šifer

10.1 Použití blokových šifer

- jak šifrovat zprávy, jejichž délka není dělitelná velikostí bloku

Padding

- musí být reversibilní – nevíme totiž, jaká byla původní délka zprávy
- varianty:
  1. první byte paddingu je nějaká konstanta, pak doplníme samé nuly
  2. celý padding je konstanta P = délka paddingu
  3. celý padding jsou nuly až na poslední byte, který je P = délka paddingu
- jak ale poznáme, jestli jsme přijali nepoškozenou zprávu? (chceme kontrolovat formát paddingu)

Intuitivní vysvětlení: Padding si můžeš představit jako vyplnění poslední krabičky v řadě bonbónů – když nemáš dostatek bonbónů, dáš tam náhradní" výplň, aby to pasovalo do celého balení. Při rozbalení ale musíš poznat, co je bonbón a co jen výplň.

10.2 ECB (Electronic Code Book)

- každý blok zakódujeme stejnou funkcí a výsledky poslepujeme ve stejném pořadí dohromady
- velmi jednoduché (a velmi rozbité, nepoužívat)
  - leakujeme, jestli jsou nějaké bloky stejné
  - nemáme žádnou náhodnou vstupní hodnotu, takže při znalosti E_K jsme v loji
- zajímavé vlastnosti:
  - random access encryption i decryption
  - bit-flip v Y_i rozbije původní informace X_i, ale nijak neovlivní X_j pro $j \neq i$
  - pokud prohodíme Y_i a Y_j, prohodíme i X_i a X_j

Intuitivní vysvětlení: Je to jako šifrovat každou stránku knihy zvlášť bez ohledu na kontext – stejná stránka bude vždy vypadat stejně. Nepřítel tak může poznat opakující se obsah.

10.3 CBC (Cipher Block Chaining)

- první blok vstupu vyxorujeme s náhodným inicializačním vektorem IV a proženeme E_K
- každý další blok vstupu xorujeme s předchozím blokem výstupu a proženeme E_K

- zajímavé vlastnosti:
  - random access decryption (ale ne encryption)
  - bit-flip v Y_i změní celý X_i a jeden bit v X_{i+1}
  - vynechání/prohození dvou bloků ovlivní tyto bloky a 1 následující

Intuitivní vysvětlení: Je to jako řetězová reakce – každý nový blok závisí na předchozím. Pokud někdo změní prostřední článek, ovlivní tím zbytek.

Leaking

- narozeninový paradox: po $\approx 2^{b/2}$ blocích máme pravděpodobně $Y_i = Y_j$ pro $i < j$
- útočník si všimne, že se zopakovaly bloky Y_i a Y_j, a může z toho získat informace:

$$\begin{aligned} Y_i = Y_j &\Rightarrow E_K(X_i \oplus Y_{i-1}) = E_K(X_j \oplus Y_{j-1}) \\ X_i \oplus Y_{i-1} &= X_j \oplus Y_{j-1} \\ X_i \oplus X_j &= Y_{i-1} \oplus Y_{j-1} \end{aligned}$$

- dokud nešifrujeme moc velké zprávy, je tento leak docela v pohodě

Intuitivní vysvětlení: Když máš moc lidí na večírku, někdo bude mít narozeniny ve stejný den – stejně tak se začnou bloky opakovat a útočník si toho může všimnout. Když šifruješ opravdu hodně bloků, může se stát, že se náhodou některé výstupy opakují. A to je problém – protože XOR těchto opakovaných bloků ti může prozradit, jak se změnily vstupy. Útočník pak může vidět stín“ toho, co bylo uvnitř. Proto by se po určité době měl klíč změnit.

10.4 CTR (Counter Mode)

- proudová šifra – padding není potřeba
- nikdy nesmíme zopakovat IV!!
  - bylo by to jako zopakovat one-time pad
  - oproti CBC není potřeba, aby IV byl náhodný
- zajímavé vlastnosti:
  - bit-flip v Y_i udělá akorát bit-flip v X_i
  - random access encryption i decryption
  - možná paralelizace v obou směrech

Intuitivní vysvětlení: Je to jako mít číslovaný seznam a na každém čísle vygenerovat klíč. Pokud použiješ stejné číslo dvakrát, generuješ stejný klíč = problém.

Leaking v CTR

- pokud jsou všechny bloky různé:

$$\forall i, j (i \neq j) : C_i \neq C_j \Rightarrow C_i \oplus C_j \neq 0$$

- výpočet:

$$\begin{aligned} Y_i \oplus Y_j &= (X_i \oplus C_i) \oplus (X_j \oplus C_j) \\ &= (X_i \oplus X_j) \oplus (C_i \oplus C_j) \\ X_i \oplus X_j &\neq Y_i \oplus Y_j \end{aligned}$$

- leak pro každou dvojici bloků: $\frac{1}{2^b}$ informace
- celkový leak: $\binom{n}{2} \cdot \frac{1}{2^b}$ pro $n =$ počet bloků

Intuitivní vysvětlení: Je to jako zmenšit počet možností losováním bez vrácení – každým dalším tahem víme trochu víc, i když jen málo.

10.5 OFB (Output Feedback)

- také proudová šifra
- zajímavé vlastnosti:
  - nemá žádný random access
  - bit-flip v Y_i udělá akorát bit-flip v X_i
  - E_K je permutace – když má moc malé cykly, stream je periodický s malou periodou a jsme v loji

Intuitivní vysvětlení: Vygeneruješ si proud šifrovacích bitů dopředu a jen ho nakrajuješ“. Pokud je cyklus krátký, bude se opakovat jako nekonečná smyčka – což je bezpečnostní problém.

11 Padding Oracle Attack

- Ukážeme pro CTR s paddingem 2. typu (celý padding je konstanta P určující velikost paddingu).
- Dá se triviálně pozměnit a použít s CBC.
- Podívejme se na poslední blok:

- Předpokládejme $P \neq 1$
  - Budeme zkoušet všechny kombinace bit flips v posledním bajtu.
  - Pokud zpráva bude přijata $\Rightarrow$ poslední bajt změněného paddingu je 01.
  - Jelikož víme, které bit flippy vedou na 01 a CTR používá jen XOR, jednoduše zjistíme původní hodnotu P.
  - Pak změníme všechny padding bloky na $P+1$ a flipujeme bity v posledním bajtu plaintextu, než je zpráva přijata (tím získáme původní hodnotu posledního bajtu).
  - Poslední bajt plaintextu přidáme k paddingu, inkrementujeme a pokračujeme pro další bajty.
- Pokud $P = 1$
  - Všechny bit-flipy v předposledním bajtu zprávy vedou na korektní padding.

Intuitivní vysvětlení

Představ si, že dešifrovací algoritmus ti napoví, jestli jsi zprávu správně ukončil nebo ne. Útočník může postupně zkoušet změny v bajtech a sledovat, kdy dešifrovací algoritmus řekne: ano, padding sedí!“ Tímto způsobem může zjišťovat jednotlivé bajty zprávy odzadu.

Co je oráklum (intuitivně):

Oráklum je jako robot, kterému můžeš předložit šifrovanou zprávu a on ti řekne jen jednu věc: Ano, padding je v pořádku nebo Ne, padding je špatný. Zdánlivě nevinná odpověď nám ale může odhalit, co bylo ve zprávě.

Příklad: Představ si, že máš tajnou zprávu v obálce (zašifrovanou) a chceš zjistit, co v ní je. Máš robota, který ti pokaždé, když vyměníš poslední znak v obálce, řekne, jestli to pořád dává smysl (tedy jestli je padding validní). Když najdeš kombinaci, která dává smysl, víš, že jsi se přiblížil ke správnému konci zprávy. A takto můžeš postupně vydolovat celou původní zprávu.

12 Další zajímavé blokové šifry z finále AES

Serpent

- 128b bloky, 128–256b klíč, 32 iterací SPN + lineární transformace.
- Velmi konzervativní, ale velmi pomalá.

Twofish

- 128b bloky, 256b klíč, 16 iterací feistelovské sítě.
- S-boxy jsou vypočítávány z klíče – pomalá inicializace, změna klíče je náročná.

Intuitivní vysvětlení

Serpent a Twofish jsou další silní kandidáti, kteří bojovali o to být vybráni jako nový AES. Serpent sází na bezpečnost, ale je pomalejší, Twofish naopak využívá složité výpočty pro každý klíč, což ho zpomaluje při změně klíče.

13 Proudové šifry

- Historicky populární – snadno se implementují na hardwaru.
- Známe jich ale docela málo.
- Nelze na ně použít padding oracle attack.

13.1 LFSR (Linear-Feedback Shift Registers)

- Bitshiftujeme, než vyjedeme z registru ven do outputu.
- Po každém shiftu vyXORujeme všechny bity a výsledek přidáme na konec registru.
- Problém: prvních n bitů, které vyjdou z registru, je iniciační stav.
- Z dalších n bitů lze sestavit lineární rovnice pro klíč.
- Pro maximální periodu musí být soustava rovnic regulární.

Intuitivní vysvětlení

LFSR si představ jako trubku, kde na začátku něco vložíš a pak se to posouvá. Každý nový výstup ovlivňuje další hodnoty. Pokud se ale někdo podívá na dostatek výstupů, může se dopočítat zpět k tomu, co jsi na začátku vložil.

13.2 eSTREAM project

- Evropský projekt hledající nové proudové šifry.
- 2004–2008.
- Profile 2 (HW) – 3 šifry, např. Trivium, požadovaný security level ≥ 80.
- Profile 1 (SW) – 4 šifry.
  - Např. Salsa20 $\rightarrow$ z ní vznikla ChaCha20 (nejpoužívanější proudová šifra).

Intuitivní vysvětlení

Představ si, že Evropská unie vyhlásila soutěž: Najdi nejlepší proudovou šifru!“ Vítězové měli být použitelní buď na čipech (HW), nebo v aplikacích (SW). ChaCha20 je jako vítěz pro aplikace – jednoduchá, bezpečná a rychlá.

13.3 RC4

- Stav je klíčem vygenerovaná permutace pole S: $[0, 1, \ldots, 255]$.
- Používá indexy i a j.
- Krok:
  1. $i \leftarrow (i + 1) \mod 256$
  2. $j \leftarrow (j + S[i]) \mod 256$
  3. Prohodím $S[i] \leftrightarrow S[j]$
  4. Výstup: $S[(S[i] + S[j]) \mod 256]$
- Inicializace:
  - 256 kroků.
  - K j přičítám i-tý znak klíče (modulo 256).

Intuitivní vysvětlení

RC4 si můžeš představit jako hru s balíčkem 256 karet, který stále zamícháváš podle určitého klíče a pak z něj taháš náhodné“ karty – tedy bajty ke XORování s textem.

13.4 ChaCha20

- Následovník Salsa20.
- 20 rund (odtud ChaCha20).
- 256b klíč, 64b počítadlo bloku, 64b nonce (různé verze mají různá rozdělení bitů).
- Stav je zakódován do matice 4×4 pomocí 32bitových čísel.
- Inicializace:
  - 1. řádek – ASCII konstanty (např. `"expand 32-byte k"`).
  - 2. a 3. řádek – klíč.
  - 4. řádek – počítadlo a nonce.
- Runda:
  - ARX šifra – kombinace sčítání, XORů a rotací.
  - Po 20 rundách je výstup vyXORován s původním stavem.

Intuitivní vysvětlení

Představ si matici, do které zapíšeš klíč, nonce a další údaje. Potom s ní dvacetkrát zatočíš (provádíš rotace a XORy) a na závěr to smícháš s původním stavem. Výsledkem je proud bajtů, které vypadají náhodně.

14 Hashovací funkce

- $h : \{0,1\}^* \to \{0,1\}^b$
- V ideálním světě by byla dokonale náhodná, ale protože je deterministická, toho nelze dosáhnout.
- Požadavky na kryptografickou hashovací funkci:
  1. Nelze efektivně najít kolizi: tedy $x \neq x'$, ale $h(x) = h(x')$.
  2. Nelze k danému výstupu najít jiný vstup: známe-li $h(x) = y$, nedokážeme najít $x' \neq x$ takové, že $h(x') = y$.
  3. Nelze ji jednoduše invertovat: pro dané y nedokážeme najít žádné x takové, že $h(x) = y$.
- Pozorování: 1) $\Rightarrow$ 2) $\Rightarrow$ 3)

Rozdíl mezi běžnou a kryptografickou hashovací funkcí

Běžná hashovací funkce (např. v Pythonu) se používá pro rozdělení dat do tabulek. Rychlost a jednoduchost je důležitější než bezpečnost. Kolize nevadí, pokud jich není moc.

Kryptografická hashovací funkce se používá v bezpečnosti – např. pro digitální podpisy nebo ukládání hesel. Tady jsou kolize extrémně nebezpečné! Proto musí být hash funkce navržená tak, aby odolat útokům a byla co nejméně předvídatelná.

Intuitivní příklad

Představ si běžnou hashovací funkci jako šuplík na třídění ponožek podle barvy – občas do jednoho šuplíku vlezou dvě skoro stejné ponožky.

TODO: tohle není moc dobré vysvětlení, ale pointa skoro sedí Kryptografická hashovací funkce je ale jako sejf – když ti někdo dá jeho obsah, nesmíš být schopný poznat, co tam bylo předtím, natož vytvořit jiný obsah, který dá stejný výsledek.

14.1 Merkle–Damgårdova konstrukce

- Používáme kompresní funkci $f : \{0,1\}^b \times \{0,1\}^b \to \{0,1\}^b$
- Zachováváme si nějaký stav, který iterativně aktualizujeme:

  - počáteční stav je fixní hodnota (označený jako IV),
  - každý blok vstupu proženeme kompresní funkcí s aktuálním stavem,
  - na konci často přidáme reverzibilní padding a zakódujeme délku původní zprávy.
- Konstrukce se může v detailech lišit mezi implementacemi.
- Délka zprávy se většinou nepřidává jako samostatný blok, ale zakóduje se do paddingu.
- Vnitřní stav může být větší než vstupní blok x_i – v tom případě jde o asymetrickou konstrukci.

Intuitivní vysvětlení

Představ si, že máš deník a na každou stránku si vždy napíšeš, co se změnilo oproti předchozí stránce. Každá nová stránka obsahuje shrnutí té předchozí, a když přidáš další informaci, vznikne ti jakási historie. Na konci přidáš ještě informaci, kolik stran jsi napsal (délka zprávy). Díky tomu je těžké něco dodatečně změnit, aniž by si někdo všiml.

14.2 Důkaz: pokud je f bezkolizní, pak h je bezkolizní

Předpokládejme, že existuje kolize dvou různých vstupů pro funkci h. Pak mohou nastat dvě situace:

1. **Kolizní vstupy mají stejný počet bloků:**
   - Pokud jsme na konci dostali stejný stav, buď se lišil některý mezistav (pak máme kolizi ve funkci f),
   - nebo byly bloky různé, ale mezistavy stejné – pak z toho opět plyne kolize ve funkci f,
   - důsledkem je, že někde uvnitř jsme museli narazit na kolizi v f.
2. **Kolizní vstupy mají různý počet bloků:**
   - V posledním kroku jsme přihashovali různý počet bloků,
   - ale dostali jsme stejný výstup $\Rightarrow$ musela být kolize ve funkci f.

- Pokud bychom nepřihashovali délku zprávy n, důkaz by nefungoval.

Intuitivní vysvětlení

Je to jako kdybys měl seznam pokynů a každý další pokyn závisel na předchozím. Pokud bys na konci dostal stejný výsledek ze dvou různých sérií pokynů, musel bys někde uvnitř udělat stejné kroky. Pokud se série liší, a výsledek je stejný, musel ses nutně v nějakém kroku splést (kolize).

14.3 Length extension attack

- Máme hash $h(z)$, ale neznáme samotné z.
- Přesto můžeme vytvořit validní hash pro $z' = z \,||\, \text{něco}$.
- Pokud by na konci nebyla přihashována délka zprávy, útok by šel snadno:
  - Přidáme nové bloky a zhashujeme je jako pokračování,
  - tím získáme hash nějaké delší zprávy z', která se tváří jako originální pokračování.
- Pokud uživatel kontroluje hash hloupě“, můžeme útok provést i s přihashovanou délkou.

Intuitivní vysvětlení

Představ si, že si někdo uzamkne deník záznamem a podepíše ho. Ale ty víš, jak ten podpis vzniká a víš, co je poslední stránka – tak si můžeš přidat vlastní stránky a připojit podpis jako bys byl autor. Pokud si ten autor nezaznamenal, kolik stran původně měl, můžeš to udělat snadno.

14.4 Collision attack (útok hledáním kolizí)

- Pro b-bitovou hash funkci dokážeme s pravděpodobností najít kolizi s $\sim 2^{b/2}$ vstupy.
- To ale znamená i potřebu $\sim 2^{b/2}$ paměti.
- Existuje i varianta s konstantní pamětí:
  - Vezmeme náhodný vstup x_1, pak iterujeme $x_i = h(x_{i-1})$,
  - tím vznikne posloupnost hodnot, které se musí nutně cyklit (protože počet hodnot je konečný).
  - V grafovém pohledu:
    - vrcholy = b-bitové výstupy,
    - hrany = aplikace funkce h,
    - graf je konečný, ale posloupnost je nekonečná $\Rightarrow$ nutně vznikne cyklus (tzv. *lollipop graph*).

Intuitivní vysvětlení

Představ si, že máš bludiště, kde každá křižovatka vede jen jedním směrem. Ať vyrazíš kamkoli, jednou se nutně vrátíš na místo, kde už jsi byl – uděláš tedy cyklus. Tvoje práce je najít dvě různá místa, která tě do stejného cyklu dovedou (kolize).

14.5 Detekce cyklu – algoritmus se dvěma běžci (Floydův algoritmus)

- Jeden běžec (želva T) skáče po 1 kroku, druhý (zajíc H) po 2 krocích.
- Jakmile se potkají, víme, že jsme v cyklu.
- Zastavíme zajíce a vypustíme novou želvu T' od začátku.
- Želva T' a původní želva T se znovu potkají – to je začátek cyklu.
- Vzdálenost, kterou uběhnou mezi setkáními, je délka cyklu.

Intuitivní vysvětlení

Je to jako když běží dva lidi na ovále – jeden pomalu, jeden rychle. Nakonec se potkají. Pak pošleme někoho dalšího, kdo začne běžet od startu a čekáme, až se potká s tím pomalým – tím poznáme, kde začíná okruh, po kterém běhají.

14.6 Multi-Collisions

- Můžeme jeden vstup zahashovat více funkcemi a výsledky zřetězit (konkatenovat).
- Pokud ale alespoň jedna z těchto funkcí je typu Merkle-Damgård, výsledná síla hashování není výrazně vyšší.
- Příklad konstrukce:

$$\begin{aligned} x_1, x_1' &: f(IV, x_1) = f(IV, x_1') = y_1 \\ x_2, x_2' &: f(y_1, x_2) = f(y_1, x_2') = y_2 \\ x_3, x_3' &: \dots \end{aligned}$$

- Stačí najít k kolizí kompresní funkce $\Rightarrow$ máme 2^k různých zpráv, které se hashují stejně.
- Všechny možné kombinace x_i/x_i' zobrazí na stejný hash.
- Na těchto zprávách pak lze najít kolize i u jiných než M-D funkcí.
- Tento problém je znám jen pro collision-resistance; pro invertibility-resistance je konkatenace docela fajn řešení.

Intuitivní vysvětlení: Po nalezení k kolizních dvojic můžeme každou pozici zprávy volit dvěma způsoby, čímž získáme 2^k různých zpráv. Všechny mají stejný hash, protože kolidují v každém kroku. Jako bys skládal větu a na každém místě měl dvě synonyma — vznikne mnoho různých vět se stejným významem (hashem). Tímto způsobem lze podvrhnout množství falešných zpráv se stejným otiskem.

14.7 Parametrizované zprávy

- Chceme někomu podstrčit dokument, který se mu zdá být nevinný, ale jeho podpis přeneseme na škodlivou zprávu se stejným hashem.
- Klasický útok vytvoří kolizní zprávu, která je nesmyslná nebo "garbage".
- Vylepšení: připravíme dvě sémanticky podobné, ale textově odlišné varianty každé části zprávy.
  - b rozdílných míst $\Rightarrow 2^b$ kombinací
  - Vysoká šance, že některé kombinace dají stejný hash

Innocent / Evil Messages

- Máme 2 množiny zpráv – innocent (nevinné) a evil (škodlivé).
- Cílem je najít dvojici zpráv (jednu z každé množiny), které se hashují na stejný výstup.
- Pomocí parametrizace vytvoříme $2^{b/2}$ zpráv v každé množině.
- Problém: jednu z množin musíme držet v paměti.
- Můžeme si však předgenerovat a uložit jen jednu množinu a tu druhou postupně prohledávat.

Intuitivní vysvětlení: Je to jako kdybys chtěl vymyslet dvě různé pohádky, které končí stejnou pointou – místo toho, abys vymýšlel pohádky úplně od začátku, připravíš si kostru příběhu a na některých místech zaměníš synonyma nebo celé věty. Pak hledáš dvě varianty, které mají stejnou "pohádkovou pointu" (tj. hash).

14.8 Daviesova-Meyerova konstrukce

- Kompresní funkce postavená z blokové šifry.
- Předpokládáme, že velikost bloku je stejná jako velikost klíče = b.
- Definice: $f(a, b) := E_a(b) \oplus b$
  - $E_a(b)$ je výstup blokové šifry se vstupem b a klíčem a
  - $\oplus$ je XOR
- Pozor: a i b může kontrolovat útočník.

Esencialita XORování

- Kdybychom nepoužili XOR:

$$f(a,b) = E_a(b) = y$$
$$\text{kolize: } f(a',b') = y \Rightarrow b' = D_{a'}(y)$$

- Pak je snadné najít kolizi: zvolíme a' a spočítáme b'.

14.8.1 Bezpečnost Davies-Meyerovy konstrukce

Davies-Meyerova konstrukce definuje kompresní funkci $f(a,b) := E_a(b) \oplus b$ postavenou na blokové šifře E. Předpokládá se, že velikost bloku i klíče je b bitů.

Věta: Pokud f vznikne Davies-Meyerovou konstrukcí nad ideální (blokovou) šifrou E, pak každý útok používající méně než $q < 2^{b/2}$ volání na šifrování nebo dešifrování najde kolizi s pravděpodobností

$$\Pr[\text{kolize}] < \frac{q^2}{2^b}.$$

Poznámka: Ideální šifra je model, kde každý klíč a určuje nezávislou náhodnou permutaci na množině b-bitových bloků. Každý výstup $E_a(b)$ nebo $D_a(b)$ vypadá jako náhodný, a žádné dvě permutace nejsou korelované.

Důkaz. Z definice Davies-Meyerovy konstrukce:

$$f(x,y) = E_x(y) \oplus y, \quad \text{resp.} \quad f(x,t) = t \oplus D_x(t).$$

Chceme odhadnout pravděpodobnost, že pro q dotazů existují dva různé vstupy $(x,y) \neq (x',y')$, pro které platí:

$$f(x,y) = f(x',y').$$

Pro každou dvojici dotazů (je jich nejvýše $\binom{q}{2} \leq \frac{q^2}{2}$) odhadujeme pravděpodobnost, že dojde ke kolizi.

U ideální šifry je každý výstup $E_x(y)$ nezávislý a rovnoměrně náhodný. Tudíž i $f(x,y)$ je rovnoměrně náhodný a nezávislý pro každý nový dotaz (dokud nenarazíme na kolizi).

Pravděpodobnost kolize mezi dvěma konkrétními dotazy je tedy nejvýše:

$$\Pr[f(x,y) = f(x',y')] \leq \frac{1}{2^b - i} \leq \frac{1}{2^{b-1}},$$

kde i je pořadí dotazu (nejhorší případ pro maximální pravděpodobnost).

Celková pravděpodobnost, že existuje alespoň jedna kolizní dvojice, je pak odhadnuta násobením počtem dvojic:

$$\Pr[\exists\,\text{kolize}] \leq \frac{1}{2^{b-1}} \cdot \frac{q^2}{2} = \frac{q^2}{2^b}.$$

Intuitivní vysvětlení: Davies-Meyer využívá blokovou šifru jako stavební kámen hashovací funkce. Pokud šifra funguje jako černá skříňka“, která generuje náhodné výstupy (permutace), pak je výstup funkce f velmi blízký náhodné hodnotě. Kolize tedy nastávají podobně jako v klasickém paradoxu narozenin – až když máme kolem $\sqrt{2^b} = 2^{b/2}$ dotazů. To je základní hranice bezpečnosti pro konstrukce bez vnitřní struktury, jako je právě Davies-Meyer.

Proč nepoužívat DES

- DES je komplementární:

$$E_{\bar{K}}(\bar{x}) = \overline{E_K(x)}$$

- A tedy:

$$f(\bar{a}, \bar{b}) = E_{\bar{a}}(\bar{b}) \oplus \bar{b} = \overline{E_a(b)} \oplus \bar{b} = E_a(b) \oplus b = f(a, b)$$

- Výsledek: trivialita při hledání kolizí – úplně nevhodné pro bezpečné hashování.

Intuitivní vysvětlení: Představ si, že si šifrování a hashování hrají na schovávanou – XOR je jako další maska, kterou na výstup přidáš, aby to nebylo tak lehké odhalit. Když by ta maska chyběla, tak by se každý mohl snadno ”podívat pod kapotu” a najít kolizi.

14.9 Merkle Trees

1. Rozdělíme data do bloků a zahashujeme každý blok.
2. Bloky spojíme po dvojicích a zahashujeme znovu.
3. Tento proces opakujeme až ke kořeni.

- Výhoda: rychlá verifikace velkého objemu dat – stačí poslat list a cestu ke kořeni.
- Interně používá například Git.
- Problémy:
  - Neumíme rozlišit hash listu od hashe vnitřního vrcholu.
  - Útočník může tvrdit, že vnitřní uzel je list (což může být jiná sekvence se stejným kořenem).
  - Řešení: zaznamenáme si u každého uzlu, zda je to kořen/list a pevnou hloubku stromu.

Intuitivní vysvětlení: Merkle strom je jako rodokmen – každý list představuje dokument, rodiče vznikají spojením dvou dokumentů a nakonec máme jednoho "praotce" všech. Když ti někdo pošle dokument a jeho rodovou linii, můžeš si lehce ověřit, že opravdu patří do té rodiny (stromu).

Sakura

- Je to v podstatě Merkle strom, ale specifický pro SHA konstrukce.

Používané hashovací funkce

MD5

- Navrhl Ronald Rivest v roce 1992.
- Název: **Message Digest 5**.
- Jedná se o funkci **Merkle-Damgårdova typu**.
- Výstup má délku 128 bitů (což je dnes považováno za málo).
- Od roku 2004 umíme efektivně nacházet kolize (i na běžném laptopu do 10 sekund).
- Nikdy ale nebyl nalezen způsob, jak MD5 invertovat (najít vstup z výstupu).

Intuitivně: MD5 byla kdysi populární, ale dnes už ji lze lehce zlomit – tedy najít dvě různé zprávy, které mají stejný otisk. Funguje to jako razítko, které dává stejný otisk dvěma různým dokumentům. Zatím ale neumíme z otisku zpětně zjistit původní dokument.

SHA-1

- Navrhla NSA v roce 1995.
- Výstup o délce 160 bitů.
- Také Merkle-Damgårdova konstrukce s Davies-Meyerovou kompresní funkcí (založená na šifře SHACAL).
- Od roku 2017 umíme nalézt kolize, ale výpočetně je to velmi náročné.

Intuitivně: SHA-1 byl dlouho nejlepší dostupný" nástroj, i když vznikl u nedůvěryhodné NSA. Dnes už umíme najít kolize, ale je to drahé. Je jako zámek, který už někdo umí otevřít, ale potřebuje k tomu hodně času a nástrojů.

SHA-2 family

- Také od NSA.
- Výstup: 224 až 512 bitů (nejpoužívanější je 256 bitů).
- Bezpečnost považována za dostačující.
- Vylepšená SHA-1 – větší vnitřní stav, více kol, pomalejší.
- Zatím žádné známé útoky.

Intuitivně: SHA-2 je evoluce SHA-1 – větší, silnější, pomalejší. Jako když vylepšíš zámek na dveřích: je robustnější, ale trvá déle ho otevřít (i správným klíčem).

SHA-3

- Vznikla jako výstup soutěže NIST v roce 2015.
- Funguje na úplně jiném principu – tzv. **sponge construction** (houbová funkce“).

Intuitivně: SHA-3 je nová generace hashování – místo skládání dat do bloků nasává“ data jako houba a pak vymačkává“ výstup.

Sponge construction

- **Nasávací fáze:**
  1. Horní část stavu (r bitů) se XORuje se vstupem.
  2. Dolní část stavu (c bitů) zůstává nezměněná.
  3. Celý stav se zamíchá permutační funkcí Π.
- **Vymačkávací fáze:**
  1. Horní část stavu se použije jako výstup.
  2. Dolní část stavu zůstává.
  3. Opět se permutuje funkcí Π.

Intuitivně: Houba nasaje data (přimíchává vstup do stavu), pak je v ní zamícháno a nakonec se z ní postupně mačká výstup. Rychlost a bezpečnost se odvíjí od poměru r a c.

Keccak permutace (SHA-3 varianta)

- Vnitřní stav má 1600 bitů.
- Např. SHA3-224:
  - Cíl: bezpečnostní úroveň 224 $\Rightarrow$ použijeme $c = 448$, $r = 1152$.
- SHA3-512:
  - $c = 1024$, $r = 576$.

Intuitivně: Keccak je specifický způsob míchání vnitřního stavu. Čím víc nasáváš“ (větší r), tím rychleji, ale méně bezpečně. Čím víc kapacity (c), tím větší bezpečnost.

XOF (Extendable Output Functions)

- Funkce s výstupem libovolné délky.
- Využívají se jako pseudonáhodné generátory parametrizované seedem.
- Např.:
  - **SHAKE-128**: $c = 256$
  - **SHAKE-256**: $c = 512$

Intuitivně: SHAKE funguje jako generátor náhodných čísel – nasaješ semínko (seed) a pak můžeš donekonečna mačkat a získávat výstup. Hodí se, když potřebujeme hash proměnné délky.

Message Authentication Codes (MAC)

MAC je zkratka pro Message Authentication Code. Skládá se ze dvou funkcí:

- $\text{Sign}(X, K)$ – podepisuje zprávu X klíčem K
- $\text{Verify}(X, K, \text{sig})$ – ověřuje podpis zprávy X klíčem K

Cílem útočníka je vytvořit jinou zprávu, která má stejný podpis jako ta původní, tedy podvrhnout zprávu.

Způsoby implementace

- $\text{Sign}(X, K) := h(K \| X)$
  - jednoduchá implementace, ale zranitelná proti tzv. **extension attack**, pokud h je Merkle-Damgårdova funkce.
  - SHA3 je imunní vůči těmto útokům, standardizovaná varianta se jmenuje KMAC.

- $\text{Sign}(X, K) := h(X \| K)$
  - méně běžné, hůře analyzovatelné bezpečnostní vlastnosti.
- $\text{Sign}(X, K) := h(h(K) \| X)$
  - považováno za bezpečnější – dvakrát hashujeme klíč.

Intuitivně: MAC je jako podpis – chcete, aby jen člověk se správným klíčem mohl zprávu podepsat. Pokud je hash funkce zranitelná, útočník si může dopočítat, jak by vypadala zpráva s nějakým dalším obsahem. Představ si to jako lego: pokud víš, jak lego vypadá na konci, můžeš připojit další kostičku.

HMAC – Hash-based MAC

$$\text{HMAC}_h(X, K) := h(K_1 \| h(K_2 \| X))$$

kde:

$$K_1 = K \oplus C_1, \quad K_2 = K \oplus C_2$$

(C_1, C_2 jsou konstanty)

- Vnitřní hash je bezkolizní.
- Vnější hash pracuje s fixní délkou – chrání před extension útoky.

Intuitivně: HMAC je jako když dáš zprávu do krabičky, krabičku do další a až pak odešleš. Útočník nevidí obsah vnitřní krabičky a nemůže ji nahradit.

Kombinování šifer a MAC

Encrypt and MAC

- Šifrování a MAC jsou nezávislé – vytvoříme MAC, pak šifrujeme.
- Nevýhoda: pokud dvakrát pošleme stejnou zprávu, MAC je stejný (information leak).

MAC then Encrypt

- Nejprve vytvoříme MAC, pak zašifrujeme celé: zprávu i podpis.
- Výhoda: šifrovací funkce nevidí strukturu původní zprávy.
- Nevýhoda: může být zranitelné vůči **padding oracle** útokům.

Encrypt then MAC

- Nejprve zašifrujeme zprávu, pak podepíšeme.
- Pokud někdo změní šifrovanou zprávu, MAC to ihned pozná.

Intuitivně: Jako kdybys nejdřív zamknul dopis do trezoru (šifrování), a pak přilepil pečeť (MAC). Jakákoliv změna dopisu nebo trezoru způsobí rozbití pečeti.

MAC bez hashovací funkce: CBC-MAC

- Využívá blokovou šifru v CBC módu
- První blok xorujeme s IV, pak každý blok se šifruje na základě předchozího
- Výstupem je poslední blok ciphertextu

Problémy:

- Pokud IV není pevně dán, může útočník měnit první blok a vytvořit kolizi.
- Pokud unikne vnitřní stav, útočník může poskládat nové zprávy – tzv. **reassembly attack**.

Řešení: Na začátek zprávy přidáme její délku, aby nešlo zneužít prefix jiných zpráv.

Intuitivně: Představ si CBC-MAC jako domino – každé nové domino závisí na předchozím. Když ale někdo ví, kde jedno domino končí, může si vymyslet nové sady domín, které padnou stejně.

Shannonovsky bezpečný MAC

- předpokládáme one-time klíč $K \in_R \mathcal{K}$
- používáme rodinu 2-nezávislých hashovacích funkcí (klasické hashovací funkce z ADS1)
- množiny:
  - zdrojová množina $\mathcal{X}$, cílová množina $\mathcal{Y}$
  - zprávy $\in \mathcal{X}$, podpisy $\in \mathcal{Y}$
- MAC se definuje jako $\text{MAC}(X, K) := h_K(x)$
- pokud známe nějaký pár zpráva–podpis, neměli bychom z něj být schopni padělat jiné zprávy
- tedy: z libovolné dvojice zprávy a podpisu nepoznáme, jestli je to validní pár

Intuitivní vysvětlení: MAC (Message Authentication Code) má za úkol ověřit, že zpráva byla opravdu vytvořena někým, kdo zná tajný klíč. Pokud někdo MAC zná, ale nemá klíč, neměl by být schopen vytvořit jinou zprávu, která bude mít stejný MAC. Jako kdybys měl zapečetěný dopis a z pečetě jsi nepoznal, co je uvnitř, ani neuměl udělat jinou pečeť.

Pravděpodobnost padělání:

$$P_{h\in\mathcal{H}}[h(x') = s' \mid h(x) = s] = \frac{P[h(x) = s \wedge h(x') = s']}{P[h(x) = s]} = \frac{1}{|\mathcal{Y}|^2}$$

To znamená, že pokud máme 2-nezávislou hashovací funkci, tak pravděpodobnost, že nám náhodná funkce trefí hodnoty pro dva různé vstupy, je stejně malá jako pro dokonale náhodnou funkci.

Definice 2-nezávislých funkcí

- Rodina: $\mathcal{H} := \{h_K \mid K \in \mathcal{K}\}$
- Každá funkce $h_K : \mathcal{X} \to \mathcal{Y}$
- Pro všechna $x \neq x' \in \mathcal{X}$ a všechna $y, y' \in \mathcal{Y}$:

$$P_{h\in\mathcal{H}}[h(x) = y \wedge h(x') = y'] = \frac{1}{|\mathcal{Y}|^2}$$

Intuitivně: Funkce z této rodiny jsou tak náhodné, že chování na dvou různých vstupech je úplně nezávislé. Když víš, co vyšlo pro jeden vstup, nepomůže ti to vůbec k tomu uhodnout, co vyjde pro jiný.

Příklad: Lineární funkce

- $\mathcal{X}, \mathcal{Y} = \mathbb{Z}_p$
- Klíče: $\mathcal{K} = \mathbb{Z}_p^2$
- Hashovací funkce: $h_{a,b}(x) = ax + b \mod p$

Intuitivně: Vybíráme náhodně přímku $y = ax + b$ v konečném tělese. Dva různé body určují jednoznačně přímku, takže pokud máme dvě různé zprávy a dvě výstupní hodnoty, existuje přesně jedna přímka (funkce), která na ně sedí.

Polynomiální MAC

- Je trochu méně bezpečný než Shannonův MAC, ale dovoluje kratší klíče
- Pracuje nad konečným tělesem $\mathbb{F}$
- Zpráva: $X_1, \ldots, X_n \in \mathbb{F}$
- Klíč: dvojice $a, b \in \mathbb{F}$

- Podpis:

$$\text{Sign}(X_1, \ldots, X_n; a, b) := X_1 a^n + X_2 a^{n-1} + \cdots + X_n a + b$$

Výpočet pravděpodobnosti padělání:

$$P_{h \in \mathcal{H}}[h(x') = s' \mid h(x) = s] = \frac{n}{|\mathbb{F}|}$$

Intuitivně: Jako kdybys podepisoval zprávu pomocí polynomu – každý prvek zprávy je koeficient. Pokud někdo zná jeden podpis, je těžší, ale ne nemožné, napadnout druhý. Je to rychlé (výpočetně), ale o něco méně bezpečné.

GCM (Galois/Counter Mode)

- AEAD: Authenticated Encryption with Additional Data
- dovoluje přidat k šifrování ještě nějaká extra data (např. hlavičky v síťovém protokolu), která se nešifrují, ale ověřují
- využívá Galoisova pole a počítá MAC pomocí operací v $\text{GF}(2^{128})$

Intuitivně: GCM umožní nejen šifrovat zprávu, ale taky zajistit, že někdo nezmění její kontext (např. komu byla určena). Pro MAC používá speciální matematiku z Galoisových polí, takže místo obyčejných operací moduluje nad bity.

Poly1305

- Dnes velmi populární v kombinaci s ChaCha20
- Extrémně rychlý a efektivní pro moderní hardware

Intuitivně: Poly1305 je rychlá funkce pro výpočet MAC. Kombinuje výhody bezpečnosti a efektivity. V praxi se používá třeba v protokolech jako TLS nebo v šifrování spojení mezi servery a webovými prohlížeči.

15 Náhodné generátory

15.1 Požadavky

- Útočník nesmí být schopen předpovědět budoucí výstup, ani když zná předchozí výstupy.
- Statistická rovnoměrnost — výstup musí vypadat jako náhodný.
- Útočník by také neměl být schopen ovlivnit výstup.

15.2 Možná řešení

1. **PRNG (pseudo-random number generator)**
   - Např. AES v CTR módu.
   - Statisticky v pořádku, pokud vstupní posloupnost není příliš dlouhá.
   - Po nějaké době je nutné znovu vygenerovat klíč.
2. **Fyzická náhodnost (physical randomness)**
   - **Rádiový šum:** Útočník může poslouchat stejný šum.
   - **Termální šum na diodě nebo rezistoru:** Útočník může měnit teplotu a ovlivnit tak výstup (např. monotónní stringy).
   - **Radioaktivní záření:** Měření intervalů mezi β-částicemi (málo používané).
   - **Video → hash:** Např. snímání lávové lampy. Pokud vypneme světlo, dostaneme samé nuly.
   - **Kruhový oscilátor:** Útočník může ovlivnit napětí a tím i chování oscilátoru.
   - **Timing zařízení:** Např. klávesnice, disk, síť. Měříme velmi přesně a používáme nejnižší bity.
3. **Kombinace více zdrojů**
   - Samostatně fyzické nebo PRNG zdroje nemusí být spolehlivé.
   - Např. `/dev/random` využívá kombinaci více fyzických zdrojů + PRNG (např. ChaCha20).
   - `RDRAND` — používá oscilátor v procesoru. Potenciálně zranitelné při backdooru.

Intuitivní poznámka:

PRNG funguje podobně jako míchačka karet — pokud známe počáteční stav a způsob míchání, dokážeme předpovědět výsledek. Proto přidáváme fyzické šumy (např. šum z diody), které jsou pro útočníka těžko předvídatelné a měly by náš mix zarušit“.

16 Problémy a obnova náhodnosti

16.1 Obnova z kompromitovaného stavu

- **Entropy pooling:** Sbíráme entropii delší dobu, a pak ji najednou přimícháme do generátoru.
- **Počet bitů entropie** by měl být alespoň roven požadovanému security levelu.
- **Odhadování entropie:** Velmi těžké, jak poznat, kolik entropie jsme získali?

16.2 Inicializace po startu systému (boot)

- Čekání na dostatek entropie po zapnutí je pomalé.
- **Rollback útok:** Uložení stavu před vypnutím znamená možnost opakovaného použití stejné entropie (nebezpečné).
- Možné obrany: zálohy, snapshoty, detekce rychlého zapnutí/vypnutí.

Intuitivní vysvětlení:

Počítač si nemůže být jistý, že má dostatek náhodnosti“ hned po zapnutí. Představ si, že chceš losovat čísla, ale ještě nemáš promíchaný klobouk – musíš chvíli počkat, než do něj nasypeš dost různých papírků.

17 Fortuna (2003)

17.1 Generátor

- AES s 128bitovým počitadlem.
- Každé číslo počitadla šifrujeme tajným klíčem.
- Po 2^{16} blocích přegenerujeme klíč, ale počitadlo neresetujeme (pomáhá zabránit cyklení).

17.2 Akumulátor entropie

- 32 entropy poolů: $P_0, P_1, \ldots, P_{31}$
- Každý vzorek přidáváme do P_j podle $j \bmod 32$.
- Jakmile P_0 nasbírá dostatek vzorků (nejčastěji každých 100 ms), použijeme ho pro přegenerování klíče.
- V i-tém kroku použijeme navíc všechny pooly P_j, pro které platí $2^i \mid j$.

Intuitivní poznámka:

Pooly jsou jako nádržky se surovinami. Jakmile máme v nejmenší nádržce dost materiálu, můžeme začít vařit“ nový klíč a smícháme do něj i větší nádržky, pokud nastal jejich čas. Tím dosáhneme bezpečné obnovy i po kompromitaci.

Bezpečný kanál (symetrický)

Alice a Bob mají unikátní tajný klíč (jiný pro každý kanál). Každý směr komunikace je chráněn:

- symetrickým šifrováním (např. AES v CTR módu),
- MAC (Message Authentication Code), konkrétně používáme strategii **encrypt then MAC**,
- sekvenční číslo (32 bitů), abychom zabránili přehrávání zpráv.
- Po 2^{32} zprávách měníme klíče pomocí:
  - **KDF** – Key Derivation Function, založená na hashovacích funkcích.

Intuitivně: Vzniká tunel, kde každá zpráva je zašifrovaná, podepsaná a očíslovaná. I kdyby útočník zachytil zprávu, neumí ji změnit ani znovu použít.

18 Algoritmická teorie čísel – složitost aritmetiky

Operace	Složitost
$s + t$	$\mathcal{O}(b)$
$s \cdot t$	$\mathcal{O}(b^2)$ (lze i v $\mathcal{O}(b)$ s velkou konstantou)
s/t nebo $s \bmod t$	$\mathcal{O}(b^2)$
$s^t \bmod n$	$\mathcal{O}(b^3)$

Intuitivně: Čím delší čísla (více bitů), tím náročnější výpočty. Násobení je pomalejší než sčítání a mocnění modulo je vůbec nejpomalejší.

GCD – Euklidův algoritmus

- Implementujeme pomocí dělení se zbytkem: $a, b \to \gcd(a, b)$.
- V každé iteraci se jedno číslo zmenší (o 1 bit).
- Čas: $\mathcal{O}(b)$ iterací $\Rightarrow \mathcal{O}(b^3)$ celkem.
- Chytré implementace (binární GCD) dosahují $\mathcal{O}(b^2)$.

Značení: $x \perp y \iff \gcd(x, y) = 1$
Intuitivně: Pokud dvě čísla nemají žádného společného dělitele kromě 1, jsou "nesoudělná". Euklidův algoritmus to rychle zjistí.

Bézoutovy koeficienty a Počítání modulo N

Bézout: $\forall x, y \, \exists u, v : \; xu + yv = \gcd(x, y)$

- Pomocí rozšířeného Euklidova algoritmu.

Počítání modulo N:

$$\mathbb{Z}_N = \{0, 1, \ldots, N - 1\}$$

- Komutativní okruh – skoro těleso (nemusí mít inverze).
- Inverzi má $x \in \mathbb{Z}_N$ právě tehdy, když $x \perp N$.
- Množina prvků s inverzí:

$$\mathbb{Z}_N^* := \{x \in \mathbb{Z}_N \mid x \perp N\}$$

- Pokud je N prvočíslo $\Rightarrow \mathbb{Z}_N^* = \mathbb{Z}_N \setminus \{0\}$, čili $\mathbb{Z}_N$ je těleso.

Intuitivně: Číslo má inverzi modulo N, pokud není dělitelné N.

Důkaz existence inverze

Chceme zjistit, kdy má číslo $x \in \mathbb{Z}_N$ multiplikativní inverzi, tedy kdy existuje $y \in \mathbb{Z}_N$ takové, že:

$$x \cdot y \equiv 1 \pmod{N}$$

Tuto kongruenci lze přepsat jako rovnost v celých číslech:

$$x \cdot y - N \cdot t = 1$$

pro nějaké celé číslo t. To je přesně tvar Bézoutovy identity:

$$x \cdot y + N \cdot (-t) = 1$$

- Pokud $\gcd(x, N) = 1$, pak podle Bézoutovy věty existují celá čísla y, t, která splňují rovnici výše. Tedy takové y existuje a je multiplikativní inverzí čísla x modulo N.
- Pokud $\gcd(x, N) = d > 1$, pak levá strana rovnice $x \cdot y - N \cdot t$ je dělitelná d (protože x i N jsou dělitelné d), ale pravá strana je 1, která dělitelností $d > 1$ odporuje. Tedy takové y nemůže existovat — spor.

Závěr: Inverze existuje právě tehdy, když x a N jsou nesoudělná čísla, tedy $\gcd(x, N) = 1$.

Malá Fermatova věta

Pokud $x \perp p$ (tj. p nepatří mezi dělitele x), kde p je prvočíslo, pak:

$$x^{p-1} \equiv 1 \mod p$$

Z toho plyne:

$$x^{p-2} \equiv x^{-1} \mod p$$

Intuitivně: Tato věta pomáhá rychle spočítat inverzi modulo prvočíslo pomocí mocnění.

Eulerova věta

Eulerova věta je zobecněním Malé Fermatovy věty. Platí:

$$\text{Mějme } \varphi(N) := |\mathbb{Z}_N^*|, \text{ pak pokud } x \perp N, \text{ platí } x^{\varphi(N)} \equiv 1 \mod N$$

Důkaz (intuitivní) : Lagrangeova věta říká, že pokud máme konečnou grupu G a její podgrupu H, pak velikost $|H|$ dělí $|G|$.

- Nechť $x \in \mathbb{Z}_N^*$, tedy číslo nesoudělné s N.
- Uvažujme posloupnost mocnin: $x^0, x^1, \ldots, x^k$
- Jelikož máme konečně mnoho prvků, tato posloupnost se dřív nebo později začne opakovat. Nechť $x^k = 1$ je první opakování jedničky.
- Z toho plyne, že $x^0 = x^{k-i} \Rightarrow$ první opakování musí být právě jednička, jinak by se cyklus opakoval dřív.
- Posloupnost $\{x^0, x^1, \ldots, x^{k-1}\}$ tvoří uzavřenou množinu na násobení, tedy podgrupu H grupy $\mathbb{Z}_N^*$.
- Dle Lagrangeovy věty platí $k \mid \varphi(N)$
- Pak platí:
$$x^{\varphi(N)} = (x^k)^{\frac{\varphi(N)}{k}} = 1$$

Intuitivní vysvětlení: Pokud máme číslo x, které nemá s N žádné společné dělitele (kromě 1), pak po nějakém počtu násobení samo sebou (modulo N) musí "obejít kruh" a skončit opět v 1. Ten počet kroků je dán velikostí množiny všech čísel nesoudělných s N, tedy $\varphi(N)$.

Teorie čísel (pokračování)

Čínská zbytková věta (Chinese Remainder Theorem, CRT)

[title=Věta] Mějme $N_1, N_2, \ldots, N_k$ navzájem nesoudělná čísla a označme $N := \prod_i N_i$. Pak platí:

$$\mathbb{Z}_N \cong \mathbb{Z}_{N_1} \times \mathbb{Z}_{N_2} \times \cdots \times \mathbb{Z}_{N_k}$$

Funkce $f : \mathbb{Z}_N \to \mathbb{Z}_{N_1} \times \mathbb{Z}_{N_2}$, definovaná jako:

$$f(x) = (x \bmod N_1, x \bmod N_2)$$

je bijekce, tj. **injektivní** i **surjektivní**.

- Injektivita: Pokud $f(x) = f(y)$, pak $x \equiv y \mod N_1$ i $x \equiv y \mod N_2$
  Proto $N \mid (x - y) \Rightarrow x = y \mod N$
- Surjektivita: Z injektivity a stejné velikosti množin plyne, že f je i surjektivní.

Druhý důkaz (konstruktivní): Chceme najít x, takové že $f(x) = (a_1, a_2)$. Najdeme:

$$f(u_1) = (1, 0), \quad f(u_2) = (0, 1)$$

Pak:

$$x = a_1 u_1 + a_2 u_2$$

Díky tomu:

$$f(x) = f(a_1 u_1 + a_2 u_2) = f(a_1 u_1) + f(a_2 u_2) = (a_1, 0) + (0, a_2) = (a_1, a_2)$$

Jak najít u_1:

- víme, že $f(N_2) = (q, 0)$ pro nějaké $q \neq 0$
- pokud $q = 1$, máme hotovo: $u_1 = N_2$
- jinak najdeme q' takové, že $q \cdot q' \equiv 1 \mod N_1$, tedy:

$$u_1 = q' \cdot N_2$$

(pro u_2 symetricky)

Intuitivní vysvětlení: CRT říká, že pokud máme několik zbytků modulo různých (navzájem nesoudělných) čísel, existuje právě jedno číslo modulo jejich součinu, které má právě tyto zbytky. Prakticky: když víme, kolik je hodin na několika nezávislých ciferníících, můžeme určit přesný čas modulo jejich násobku“.

Výpočet Eulerovy funkce $\varphi(n)$

$\varphi(N) := |\mathbb{Z}_N^*| =$ počet čísel menších než N, která jsou s N nesoudělná

- **Pozorování:**
  1. Pokud je p prvočíslo, tak:
  $$\varphi(p) = p - 1$$
  protože všechna čísla $1, \ldots, p-1$ jsou s p nesoudělná.
  2. Pokud $N = p^k$, pak:
  $$\varphi(p^k) = p^k - \frac{p^k}{p} = (p-1) \cdot p^{k-1}$$
  protože každé p-té číslo je dělitelné p a ostatní nejsou.
  3. Pokud $x \perp y$, pak:
  $$\varphi(xy) = \varphi(x) \cdot \varphi(y)$$
  což lze odvodit z CRT.

Důkaz součinu: Zvolme funkci:

$$f : \mathbb{Z}_{xy} \to \mathbb{Z}_x \times \mathbb{Z}_y$$

Z definice CRT víme, že se jedná o bijekci. V obrazu je:

- $\varphi(x)$ řádků (indexů nesoudělných s x)
- $\varphi(y)$ sloupců (indexů nesoudělných s y)

Takže počet dvojic splňujících podmínku nesoudělnosti je $\varphi(x) \cdot \varphi(y)$.

Závěr: Pokud umíme rozložit číslo n na prvočinitele, umíme i efektivně spočítat $\varphi(n)$ pomocí výše uvedených pravidel.

Faktorizace

- Zatím nikdo nezná polynomiální algoritmus pro faktorizaci celých čísel.
- Známe ale sub-exponenciální algoritmy:
  - rostou rychleji než polynomy, ale pomaleji než exponenciály
  - například $\exp(\sqrt[\log n]{\log n})$ nebo $n^{\log \log n}$
- Na kvantových počítačích existují polynomiální algoritmy (např. Shorův algoritmus), ale:
  - zatím se je nikdo nepodařilo implementovat v praxi
  - konstanty algoritmů jsou obrovské (zatím se nehodí pro reálné použití)

Intuitivní vysvětlení: Faktorizace znamená rozložení“ čísla na jeho prvočítele (např. $15 = 3 \cdot 5$). Umět to efektivně je klíčové v kryptografii, ale pro velká čísla je to extrémně těžké — a právě na tom stojí bezpečnost RSA. Kvantové počítače by to ale jednou mohly zvládat rychle.

Testy prvočíselnosti

- Existují:
  - pravděpodobnostní testy
  - deterministické polynomiální testy (např. AKS, ale s exponentem kolem 6 — velmi pomalé)

Fermatův test

- Cíl: zjistit, zda je číslo N prvočíslo.
- Vezmeme náhodné $x \in \mathbb{Z}_N^*$ a spočítáme $x^{N-1} \mod N$
  - pokud $x^{N-1} \not\equiv 1 \mod N$, pak N **není** prvočíslo (tzv. Fermatův svědek)
  - pokud $x^{N-1} \equiv 1 \mod N$, pak N **může být** prvočíslo
- Test nemá false-negatives, ale může mít false-positives (někdy označí složené číslo za prvočíslo).

Eulerův svědek: $x \in \mathbb{Z}_N^*$ takové, že $x \perp N$ a $x^{N-1} \not\equiv 1 \mod N$

Carmichaelova čísla:

- Jsou to složená čísla, která **projdou Fermatovým testem pro všechna** $x \in \mathbb{Z}_N^*$
- Neexistuje pro ně Fermatův svědek, pouze Eulerův
- Je jich málo, ale existují nekonečně

Intuitivní vysvětlení: Fermatův test zkoumá, jestli se číslo N chová jako prvočíslo podle malé Fermatovy věty. Občas se ale složená čísla tváří jako prvočísla“, což je problém. Test je rychlý, ale někdy se splete. Carmichaelova čísla ho přechytračí“.

Pravděpodobnost chyby

- Mějme $H := \{x \in \mathbb{Z}_N^* \mid x^{N-1} \equiv 1 \mod N\}$ — množina všech nesvědků
- H je podgrupa $\mathbb{Z}_N^*$, a pokud N není Carmichaelovo číslo, pak platí:

$$|H| \leq \frac{1}{2}|\mathbb{Z}_N^*|$$

- Pravděpodobnost false-positive $\leq \frac{1}{2}$ — stačí test iterovat víckrát

Rabin-Millerův test

- Oproti Fermatovi odhalí i Carmichaelova čísla
- Algoritmus:
  1. Vybereme náhodné $x \in \mathbb{Z}_N$
  2. Pokud $\gcd(x, N) \neq 1$, pak máme složené číslo

3. Jinak spočítáme:

$$x^{N-1},\ x^{\frac{N-1}{2}},\ x^{\frac{N-1}{4}},\ \ldots\ x^{\frac{N-1}{2^k}} \mod N$$

4. Sledujeme, zda narazíme na hodnotu různou od ± 1
   - Pokud ano, N je složené
   - Pokud někdy výsledek $= -1$, pokračujeme
   - Pokud dojdeme až k 1, mohlo by být N prvočíslo

- Test nemá false-negatives

Věta Rabin: Pravděpodobnost false-positive $\leq \frac{1}{4}$

Věta Miller: (za předpokladu rozšířené Riemannovy hypotézy)
Pro složené N existuje svědek x tak, že $x \leq \mathcal{O}(\log N)$

Intuitivní vysvětlení: Miller-Rabin je chytřejší" verze Fermatova testu. Sleduje postupné odmocniny a když někde něco nehraje, víme, že číslo je složené. Je to velmi spolehlivý test, používá se v praxi např. při generování velkých prvočísel pro RSA.

18.1 Generování velkých prvočísel

- Cílem je generovat náhodné b-bitové prvočíslo.
- Postup: náhodně generujeme b-bitová čísla (nepočítáme leading zeros) a testujeme je na prvočíselnost.
- Tento proces je v nejhorším případě nedeterministický (může trvat dlouho).
- Hustota prvočísel kolem N je přibližně $\frac{1}{\ln N}$.
  - Tedy asi každé $\ln N$-té číslo je prvočíslo.
  - Pokud N je exponenciální vůči b, např. $N = 2^b$, pak $\frac{1}{\ln N} = \frac{1}{c \cdot b}$ pro nějakou konstantu c.
  - To znamená, že náhodné zkoušení prvočísel většinou funguje dobře.

Intuice: Prvočísel je sice méně než všech čísel, ale pořád je jich dost — takže když náhodně tipujeme, máme slušnou šanci na úspěch. Hustota říká, že když zkusíme třeba 1000 čísel, jedno z nich často prvočíslem bude.

18.2 Diskrétní logaritmus

- Věta: Množina $\mathbb{Z}_p^*$ (násobky modulo prvočíslo p bez nuly) je cyklická.
- Existuje tedy generátor g takový, že $g^0, g^1, \ldots, g^{p-2}$ pokryje celou $\mathbb{Z}_p^*$.
- Zobrazení $g^x \mapsto x$ je pak diskrétní logaritmus.

Jak ověřit, že g je generátor?

- Nejmenší t takové, že $g^t \equiv 1 \pmod{p}$, se nazývá řád prvku g.
- Pokud $t = p - 1$, pak je g generátor.
- Dle Lagrangeovy věty t musí dělit $p - 1$.
- Zrychlení: Stačí zkontrolovat, že $g^{\frac{p-1}{q}} \not\equiv 1 \pmod{p}$ pro všechny prvočíselné dělitele q čísla $p - 1$.

Intuice: Podobně jako u klasického logaritmu chceme zjistit, kolikrát musíme násobit g, abychom dostali nějaké číslo. Ale protože jsme v modulu, musíme být opatrní — nemusí to být jednoznačné. Tato úloha (najít x takové, že $g^x \equiv a \mod p$) je výpočetně těžká – na tom stojí bezpečnost mnoha kryptosystémů.

18.3 Diskrétní odmocniny a kvadratické zbytky

Odmocniny modulo prvočíslo

- Chceme řešit $x^2 \equiv a \mod p$.
- U $p = 5$: $0^2 = 0$, $1^2 = 1$, $2^2 = 4$, $3^2 = 4$, $4^2 = 1$.
- Vidíme, že 1 a 4 mají dvě odmocniny, 2 a 3 žádnou.
- 0 má právě jednu odmocninu – výjimka.
- Každé číslo má maximálně dvě odmocniny.
- Platí: $x^2 = (-x)^2 \Rightarrow$ počet odmocnin je sudý (kromě nuly).

Kvadratické zbytky

- Čísla s dvěma odmocninami jsou kvadratické zbytky.
- Čísla se sudým diskrétním logaritmem jsou vždy kvadratické zbytky.
- Pokud máme generátor g, pak $(g^k)^2 \equiv g^{2k}$ – odmocnina existuje.
- Čísla s lichým diskrétním logaritmem odmocninu nemají – nejsou kvadratické zbytky.

Eulerovo kritérium

- Věta: $x \neq 0$ je kvadratický zbytek (QR), pokud $x^{\frac{p-1}{2}} \equiv 1 \mod p$.
- Pokud dostaneme výsledek -1, pak x není kvadratický zbytek.
- Důkaz využívá vlastností mocnin generátoru g.

Intuice: Kvadratický zbytek je číslo, které je čtvercem něčeho“. V modulu se ale ne všechny čísla dají odmocnit. Eulerovo kritérium nám dá jednoduchý způsob, jak to zjistit — jen mocníme a podíváme se na výsledek.

18.4 Odmocniny modulo složené číslo

- Použijeme čínskou zbytkovou větu (CRT) – nejprve odmocniny modulo prvočíselné dělitele.
- Např. pro $n = p \cdot q$:
$$\mathbb{Z}_n \cong \mathbb{Z}_p \times \mathbb{Z}_q$$
- Počet odmocnin je pak 2^k nebo žádná, kde k je počet různých prvočíselných dělitelů.
- Pokud umíme faktorizovat n, umíme i odmocňovat.
- Pokud ne, neumíme. Právě tato vlastnost se používá v kryptografii (např. Rabinovo šifrování).

Intuice: Odmocniny u složených čísel děláme po částech – každá část (modulo prvočíslo) se řeší zvlášť a pak se složí dohromady. Pokud nevíme rozklad, nevíme, kde začít.

19 RSA šifra

Autoři: Rivest, Shamir, Adleman – 1978

Klíč

- Velká prvočísla p a q:
  - V praxi dnes potřebujeme alespoň 4096bitová prvočísla.
- Modul: $n = p \cdot q$
  - $\varphi(n) = (p-1)(q-1)$, protože $p \perp q$
  - n můžeme bezpečně posílat, protože faktorizace n je těžká.
- Šifrovací exponent: e takové, že $e \perp \varphi(n)$
- Dešifrovací exponent: d takové, že $d \cdot e \equiv 1 \mod \varphi(n)$
- Veřejný (šifrovací) klíč: (n, e)
- Soukromý (dešifrovací) klíč: (n, d)

Intuitivně:
Zvolíme dvě prvočísla p a q, které držíme v tajnosti. Spočítáme $n = pq$, a z toho určíme $\varphi(n)$. Vybereme náhodně číslo e, které nemá společného dělitele s $\varphi(n)$. K tomu dopočítáme d, což je inverze e modulo $\varphi(n)$. Komu pošleme (n, e), ten nám může bezpečně šifrovat zprávy.

Šifrování a dešifrování

- Zpráva je $x \in \mathbb{Z}_n$
- Šifrování: $E(x) = x^e \mod n$
- Dešifrování: $D(y) = y^d \mod n$

Korektnost:

- Musí platit $x \perp n$
- Pokud $gcd(x, n) \neq 1$, tak z něj zjistíme p nebo q
- Důkaz správnosti:

$$D(E(x)) = (x^e)^d = x^{ed} \equiv x^{k \cdot \varphi(n)+1} = x^{\varphi(n) \cdot k} \cdot x \equiv x \mod n$$

- Kde $d \cdot e \equiv 1 \mod \varphi(n)$ a využíváme Malou Fermatovu větu.

Intuitivně:
Šifrováním zprávu umocníme na e, dešifrováním znovu na d. Čísla e a d jsme vybrali tak, aby se po obou operacích dostala původní zpráva.

Efektivita

- Algoritmus je polynomiální, ale pomalý.
- V praxi se často používá hybridní šifrování – RSA se použije na šifrování klíče a zbytek dat se šifruje symetrickou šifrou.

Zrychlovací triky

- Veřejný klíč: zvolíme malé e (např. 3 nebo 7)
- Privátní klíč:
  Můžeme si pamatovat p a q a při dešifrování využít Čínskou zbytkovou větu:
  $$\text{aritmetika mod } n \equiv \text{aritmetika mod } p \text{ a mod } q$$
- Počítáme s menšími čísly – je to rychlejší

Důležité vlastnosti

- **Komutativita:**
$$D_{K_2}(D_{K_1}(E_{K_2}(E_{K_1}(x)))) = x$$
  - Klíče musí mít stejný n
  - Není bezpečné mít více klíčů se stejným n
- **Homomorfnost:**
$$E(x \cdot y) \equiv E(x) \cdot E(y)$$
- Vlastnost se moc nevyužívá, protože by mohla pomoci útočníkovi.

Slepé podpisy

- Bob má plaintext x
- Vybere náhodné $r \in_R \mathbb{Z}_n^*$ (tzv. blinding factor)
- Pošle Alici $x \cdot r^d$
- Alice spočítá:
$$(x \cdot r^d)^e = x^e \cdot r^{ed} = x^e \cdot r \text{ a pošle zpět}$$
- Bob vypočítá:
$$x^e = (x^e \cdot r) \cdot r^{-1}$$

Intuitivně:
Bob zamaskuje zprávu náhodným faktorem r a nechá Alici provést šifrování. Alice netuší, co podepisuje, protože nezná x.

Útoky na RSA

Špatná volba parametrů

- Pokud $x < n^{1/e}$, pak jde v $\mathbb{Z}$ spočítat odmocninu – to je polynomiální.
- Pokud známe $\varphi(n)$, můžeme dopočítat p a q:
$$n = pq, \quad \varphi(n) = (p-1)(q-1) = pq - p - q + 1$$
- Pokud $d < n^{1/4}$, lze d spočítat z e (malý soukromý exponent je problém).
- Pokud známe d i e, můžeme spočítat $\varphi(n)$ a následně faktorizovat n.

Shrnutí:
Bezpečnost RSA závisí na volbě dostatečně velkých náhodných prvočísel, bezpečných parametrů a skrytí dešifrovacího klíče.

Meet-in-the-middle attack

Mějme rovnici:

$$m^e \equiv c \mod n$$

a známe veřejný exponent e a šifrovaný text c. Snažíme se získat zprávu m.

Myšlenka: Zkusíme najít dvě malá čísla u a v, taková že:

$$u^e \equiv c \cdot v^{-e} \mod n$$

což znamená:

$$(u \cdot v)^e \equiv c \Rightarrow u \cdot v \equiv m$$

Intuitivně: Představ si, že místo toho, abychom zkoušeli všechna m zleva (brute-force), tak hledáme polovinu řešení zleva (u^e) a polovinu zprava ($c \cdot v^{-e}$) a hledáme, kde se setkají. To výrazně snižuje složitost z n na přibližně $\sqrt{n}$.

Poznámka: Funguje jen, pokud je zpráva m malá nebo má speciální strukturu.

Podobné zprávy

Pokud máme x' takové, že $x' = x + \delta$ a známe $c = x^e$, $c' = (x+\delta)^e$, pak můžeme sestrojit dva polynomy:

$$p(x) = x^e - c, \quad p'(x) = (x + \delta)^e - c'$$

Hledáme jejich společný kořen x:

$$x = \gcd(p(x), p'(x))$$

Intuitivně: Máme dvě rovnice, které mají stejný řešený kořen x, protože x' se liší od x jen o δ. Pokud známe rozdíl, můžeme pomocí gcd najít x.

Podobná prvočísla p a q

Mějme $q = p + 2\delta$, pak:

$$n = pq = p(p + 2\delta) = p^2 + 2p\delta$$

Zkusíme tedy různé hodnoty δ a počítáme:

$$n + \delta^2 = (p + \delta)^2$$

Intuitivně: Pokud jsou p a q blízko sebe, pak můžeme zkusit aproximaci odmocniny a najít p pomocí kvadratické rovnice. Poměrně efektivní útok na špatně zvolené klíče.

Více klíčů se stejným n

Pokud má více uživatelů stejný modul n a různé veřejné exponenty e_i, pak při šifrování stejné zprávy x dostáváme:

$$x^{e_1} \equiv c_1 \mod n_1, \quad x^{e_2} \equiv c_2 \mod n_2, \quad x^{e_3} \equiv c_3 \mod n_3$$

Pomocí čínské zbytkové věty (CRT) můžeme najít x.

Intuitivně: Pokud všichni dostanou stejnou zprávu, ale každý šifrovanou jiným klíčem, a tyto klíče sdílí stejný modul, můžeme zkombinovat tyto šifrované zprávy a rekonstruovat původní x.

Chyba při výpočtu

Při použití optimalizace pomocí CRT může při výpočtu dojít k chybě. Např. místo x^e dostaneme $x^e + c \cdot q$.

Pak útočník může vypočítat:

$$\gcd(c \cdot q, n) = q$$

a tím rozložit n.

Intuitivně: Pokud se při dešifrování někde udělá chyba (např. při umocňování modulo p), může tato chyba vést k úniku jednoho z tajných čísel p nebo q, protože rozdíl v hodnotě bude násobkem jednoho z nich.

19.1 Sémantická bezpečnost RSA

- Cílem je, aby ze zašifrovaného textu (ciphertextu) nešlo zjistit žádnou informaci o původní zprávě (plaintextu).
- Pokud umíme určit například lichost zprávy nebo její přibližnou velikost, umíme ji rozšifrovat.

Definice:

$$\text{parity}(y) := D(y) \bmod 2$$

$$\text{half}(y) := \begin{cases} 1 & \iff D(y) > \frac{n}{2} \\ 0 & \text{jinak} \end{cases}$$

Věta: Pokud máme orákulum pro `parity` nebo `half`, umíme dešifrovat v čase $\mathcal{O}(\text{poly}(b))$.

Intuice: Pokud známe pouze nejnižší bit zprávy (lichost) nebo první bit za desetinnou čárkou v podílu $\frac{x}{n}$, můžeme iterativně dopočítat celou zprávu. RSA je homomorfní, což nám umožňuje z ciphertextu vytvářet nové ciphertexty, které obsahují o trochu víc informace o zprávě.

Orákulum pro `half`

- Mějme $y \equiv x^e \mod n$, kde $x = \alpha \cdot n$ pro $\alpha \in [0, 1)$.
- Funkce `half` nám říká první bit za desetinnou čárkou v α.
- $half(y \cdot 2^e)$ je druhý bit za desetinnou čárkou.
- Homomorfismus RSA: $E(2x) = y \cdot 2^e$.

Postup: Iterací přibližujeme reálné číslo $\alpha = \frac{x}{n}$. Každý bit nám umožňuje zpřesnit odhad a po $\log(n) + 1$ krocích máme α' tak, že $|\alpha - \alpha'| < \frac{1}{n}$, tedy $x \approx n \cdot \alpha'$.

Orákulum pro `parity`

- Pomocí parity umíme simulovat orákulum pro half:

$$half(y) = parity(y \cdot 2^e)$$

- Homomorfismus: $E(2x) = y \cdot 2^e$
- Zjišťujeme paritu čísla $2x \mod n$.
- Pokud $x < \frac{n}{2}$, tak $2x$ zůstane menší než $n \Rightarrow$ sudé $\Rightarrow$ parity $= 0$
- Pokud $x > \frac{n}{2}$, tak $2x \mod n = 2x - n$, což je liché $\Rightarrow$ parity $= 1$

Intuice: Díky vlastnosti RSA, že je homomorfní pro násobení, můžeme získávat jednotlivé bity tajné zprávy. Pokud bychom měli černou skříňku (orákulum), která nám dá alespoň paritu, můžeme postupně získat celou zprávu.

19.2 Padding Schemes

Problém: RSA je deterministické. Pokud dvakrát zašifruji stejnou zprávu stejným klíčem, dostanu stejný ciphertext. Tím lze porovnávat zprávy a útočit.

Řešení: Používáme padding – přidání náhodných bitů do zprávy před šifrováním.

PKCS#1 v1.5

- Padding začíná sekvencí `00 02`, následuje náhodný blok a pak `00` a zpráva.
- Útočník, který má padding orákulum (řekne, zda padding odpovídá formátu), může iterací a manipulací ciphertextu zjistit tajnou zprávu.
- **Bleichenbacherův útok:** Po milionech dotazů na orákulum (které jen říká, zda má dešifrovaný text správný padding), lze získat celý klíč.

Intuice: Pokud orákulum pouze řekne, zda zpráva má správný tvar, tak i tato minimální informace může být zneužita – útočník může zprávu mírně měnit a sledovat, kdy orákulum začne odpovídat jinak.

PKCS#1 v2.0

- Používá 2-rundovou Feistelovu síť s hašovacími funkcemi.
- Je reverzibilní a zároveň nemá žádné rozeznatelné vzory jako `00`.
- Je odolná vůči Bleichenbacherovým útokům.

Intuice: Feistelova síť s haši zajistí, že výsledný padding je náhodně rozprostřený a nelze snadno zkoumat, zda má správný formát. Díky reverzibilitě lze padding znovu odstranit při dešifrování.

20 Diffie-Hellman

Protokol pro tajnou výměnu klíčů

- Založený na složitosti problému diskétního logaritmu.
- Poskytuje *forward security* – i když unikne privátní klíč, nelze dešifrovat staré zprávy.

Veřejné parametry: prvočíslo p a generátor g grupy $\mathbb{Z}_p^*$.

- Alice vybere náhodné $x \in_R \{0, \ldots, p-2\}$ a pošle $g^x \mod p$.
- Bob vybere náhodné $y \in_R \{0, \ldots, p-2\}$ a pošle $g^y \mod p$.
- Oba si vypočítají $g^{xy} \mod p$ jako sdílený tajný klíč.

Intuitivně: Každý z účastníků přispěje částí exponentu, ale jen oni znají celkový xy. Útočník zná jen g^x a g^y, z čehož bez znalosti x nebo y nelze g^{xy} efektivně spočítat.

Útoky na DH

Man-in-the-middle attack Útočník se tváří jako Bob vůči Alici a jako Alice vůči Bobovi. Oba komunikují ve skutečnosti s útočníkem, který si vytvoří dva různé klíče $g^{x'}$ a $g^{y'}$.

Řešení: podepisovat veřejné zprávy (asymetricky).

Útok na veřejné parametry Útočník může za g dosadit g^k, které generuje malou podgrupu $\mathbb{Z}_p^*$. Výpočet diskrétního logaritmu v malé grupě je snadný.

Řešení: validace parametrů.

Information leak Z výrazu $\left(\frac{g^x}{p}\right)$ lze odhalit lichost x, podobně i pro y. To může prozradit paritu xy.

Bezpečná volba parametrů Používáme **safe prime:** $p = 2q + 1$, kde q je také prvočíslo.

Zmenšení exponentů Používáme grupu řádu q, která je menší, ale stále bezpečná (např. 256 bitů). $p = kq + 1$, ale DH je stále těžké.

20.1 ElGamalův kryptosystém

Použití DH pro asymetrické šifrování.

- Veřejné parametry: p, g (stejné jako u DH)
- Privátní klíč: $k \in_R \{0, \ldots, p-1\}$
- Veřejný klíč: $h = g^k \mod p$

Šifrování zprávy x:

1. Náhodně zvolíme $t \in_R \{0, \ldots, p-2\}$
2. Spočítáme $s = h^t \mod p$
3. Spočítáme $y = x \cdot s$
4. Posíláme dvojici (g^t, y)

Dešifrování:

1. Spočítáme $s = (g^t)^k = g^{kt}$
2. Spočítáme $x = y \cdot s^{-1} \mod p$

Intuitivně: místo přímého šifrování jako v RSA, ElGamal zakóduje zprávu násobením s náhodným tajným klíčem, který je výsledkem DH výměny.

Bezpečnostní poznámka: podobně jako RSA, může leakovat informace (např. kvadratické zbytky). Řešení: používat zprávy pouze z množiny QR.

21 Eliptické křivky

- Jsou dobrým zdrojem malých grup s těžkým problémem diskrétního logaritmu a bez malých podgrup.
- Počítání na eliptických křivkách bývá navíc velmi efektivní.
- Problém diskrétního logaritmu na eliptických křivkách je těžší než v běžných grupech.

- Pokud by se podařilo zrychlit výpočet diskrétního logaritmu, eliptické křivky by mohly přežít díky své složitosti.

Definice křivky nad $\mathbb{R}$:

$$E : y^2 = x^3 + ax + b, \quad \text{za podmínky } 4a^3 + 27b^2 \neq 0$$

Tato podmínka zajišťuje, že křivka má tři různé průsečíky s osou x – tedy není singulární.

Intuitivně: Jedná se o hezké“ křivky bez hrotů a smyček, což umožňuje definovat na jejich bodech algebraické operace jako sčítání.

Grupová operace + na křivce

- Vytvoříme z bodů na křivce grupu, kde neutrální prvek je bod v nekonečnu“ $\mathcal{O}$.
- Sčítání bodů $P = (x_1, y_1)$ a $Q = (x_2, y_2)$:
  1. $x_1 \neq x_2$: spojíme P a Q přímkou, která protne křivku ve třetím bodě R – jeho zrcadlový obraz podle osy x je výsledkem $P + Q$.
  2. $x_1 = x_2, y_1 = -y_2$: přímka je svislá, takže $P + Q = \mathcal{O}$.
  3. $P = Q$: vezmeme tečnu v bodě P, ta protne křivku ve třetím bodě R, a opět vezmeme jeho zrcadlo.
- Výsledná struktura je abelovská grupa.

Intuitivně: Vždy, když nakreslíme přímku skrz dva body na křivce, ta přímka protne křivku ještě v jednom bodě – zrcadlení tohoto bodu je definováno jako výsledek sčítání.

Konečná tělesa

- Všechny výše uvedené operace fungují i nad konečnými tělesy $\mathbb{F}_p$ nebo $\mathbb{F}_{p^k}$, pokud $p > 3$.
- Tato tělesa používáme v praxi – místo reálných čísel, protože umožňují konečné reprezentace a výpočty.

22 Asymetrické podpisy

- Používají se dva klíče: soukromý pro podepisování a veřejný pro ověřování.
- Ověřovací funkce bere zprávu a podpis a rozhodne, zda k sobě patří.
- Podepisovací funkce může být náhodná (např. DSA), ověřovací musí být deterministická.

Cíle útočníka:

- Získání tajného klíče.
- *Existenční padělání:* schopnost podepsat libovolnou (byť nesmyslnou) zprávu.
- *Cílené padělání:* schopnost podepsat předem zvolenou zprávu.

Typy útoků:

- Máme veřejný klíč.
- *Known plaintext:* máme zprávy s podpisy.
- *Chosen plaintext:* útočník nechá někoho podepsat zprávu podle svého výběru.

22.1 RSA podpis

- Používá se obráceně“ než šifrování:

$$\text{sign} = x^d \mod n, \quad \text{ověření: } \text{sign}^e \mod n = x$$

Existenční padělání:

- Stačí zvolit náhodný podpis sign a spočítat $x = \text{sign}^e \mod n$.
- Výsledek je zpráva s platným podpisem, ale nesmyslná.

Cílené padělání:

- Útočník použije slepé podepisování.
- Řešení: nepodepisujeme zprávu, ale její hash ⇒ ztrácíme homomorfismus RSA.

22.2 ElGamal podpis

- Velmi odlišný od ElGamal šifry.
- Na zkoušce nebude vyžadován.

22.3 DSA – Digital Signature Algorithm

- Používá se i ve verzi s eliptickými křivkami (ECDSA).
- **NIKDY!** nepoužívat stejné číslo k pro dvě různé zprávy.
- Z podpisů dvou zpráv se stejným k lze zpětně dopočítat privátní klíč.

Intuitivně: Náhodné číslo k v podpisu funguje jako jednorázová sůl“. Pokud použijeme stejné k vícekrát, útočník může pomocí rovnic mezi dvěma podpisy spočítat tajný klíč. Je to podobné, jako kdybychom u hesla použili pořád stejnou sůl – dá se pak prolomit.

23 Kryptografické protokoly

- Typický postup kryptografického protokolu:
  1. Výměna *nonces* (náhodných čísel).
  2. Strana A vygeneruje *master secret*.
  3. Strana A pošle *master secret* zašifrovaný veřejným klíčem strany B.
  4. Strany A a B podepíší transkript protokolu svými tajnými klíči.
  5. Použijeme *KDF* (Key Derivation Function) pro vytvoření symetrických klíčů.
  6. Komunikace probíhá pomocí symetrické šifry a MAC.
  7. Po nějaké době přegenerováváme nové symetrické klíče pomocí KDF.
- Kroky 2 a 3 většinou probíhají pomocí Diffie-Hellman protokolu.

Intuitivní vysvětlení: Protokol si představ jako domluvu na tajném hesle. Nejdřív si vyměníme pár náhodných údajů, pak si tajně dohodneme hlavní klíč (*master secret*), podepíšeme si, že vše proběhlo správně, a z toho klíče uděláme šifrovací klíče. Pro jistotu je po čase obnovujeme.

23.1 Implementační pasti

- **Fuzzing**: Testování programu náhodnými daty pro odhalení chyb.
- **Používání jazyka C (nebo jiného low-level jazyka)**:
  - Špatné zacházení s pamětí může vést k chybám a bezpečnostním dírám.
- **Nepoužívání C**:
  - Na druhou stranu, absence nízkoúrovňové kontroly paměti může vést k jednoduchému exploitování aplikace.

Intuitivní vysvětlení: Programování v C je jako práce s ostrým nožem – můžeš s ním dělat skvělé věci, ale stačí malá chyba a ublížíš si. Pokud C nepoužiješ správně, hrozí bezpečnostní problémy.

23.2 Side Channels (Boční kanály)

23.2.1 Remote attacker (vzdálený útočník)

- **Timing attack**:
  - Měříme dobu odezvy při porovnávání dvou řetězců. Například delší porovnávání odhalí, že mají stejný prefix.

- **Ambiguous data**:
  - Skript ukrytý v komentáři, různé interpretace JSON/XML/UTF-8 formátu.
- **Format detection**:
  - Např. některý Java bytecode je validní JPEG soubor.
- **DOS** (Denial of Service):
  - Zaplavení serveru nesmyslnými požadavky.

23.2.2 In the same room (fyzická přítomnost)

- **Měření spotřeby energie procesoru**.
- **Měření elektromagnetického šumu**.
- **Sound-based keylogger**.
- **Měření teploty a ošoupanosti“ kláves**.

23.2.3 Physical access (fyzický přístup)

- **Coldboot attack**: Vytáhnutí RAM za běhu a čtení klíčů.
- **Trojan horse**: Instalace zadních vrátek na stroj.

23.2.4 Run my code

- **Stačí JavaScript na stránce**.
- **Meltdown**: Hardwarová chyba umožňující přístup k paměti jiných procesů.

Intuitivní vysvětlení: Boční kanály jsou triky, jak zjistit tajné informace tím, že neútočíme přímo na šifrování, ale využíváme vedlejší efekty (např. spotřeba elektřiny, délka výpočtu, zvuky).

23.3 Cache-based AES Attack

- **Chosen plaintext attack**: Útočník si volí šifrovaný plaintext.
- **AES**: Symetrická šifra s 128b klíčem a 10 rundami.
- **Tabulky**:
  - Rychlé šifrování pomocí lookup tabulek T_i.
  - AES provádí kombinaci několika předpočítaných operací.

- **Princip útoku**:
  - Fixujeme některé bajty a měříme časy přístupů.
  - Z měření usuzujeme, jak vypadají hodnoty tabulek T_i.

Intuitivní vysvětlení: Představ si, že AES ukládá pomocné výpočty do rychlých tabulek v paměti. Pokud víme, kam sahá v paměti podle vstupu, můžeme měřením času uhodnout části tajného klíče.

24 Hesla

Hesla jsou nejpoužívanější způsob zabezpečení v systémech. Bohužel ale představují i nejčastější slabinu celého bezpečnostního řetězce. Hlavním důvodem jsou uživatelé, kteří často volí jednoduchá, opakovaná nebo snadno uhodnutelná hesla.

Intuitivní vysvětlení:

Představte si, že máte za úkol chránit poklad. Pokud si nastavíte kombinaci zámku jako `1234`, tak se moc nenadřete, ale zároveň i útočník takovou kombinaci rychle uhodne. Hesla fungují úplně stejně — pokud jsou jednoduchá, nechrání systém dostatečně.

Uchovávání hesel

- Hesla se neukládají v prostém textu.
- Obvykle se hashují na straně serveru (server-side).

Intuitivní vysvětlení:

Ukládat hesla v prostém textu by bylo jako nechat klíče od trezoru položené na stole. Pokud útočník získá databázi hesel, získá i přímý přístup. Proto hesla nejprve projdou hashovací funkcí, která je převede na nevratnou podobu. Hash funguje jako otisk — z otisku zpět na heslo se prakticky nedostaneme.

Předpočítávání hashů hesel

- Pokud bychom chtěli crackovat hesla, můžeme si teoreticky vytvořit tabulku, kde budou hashe všech možných hesel (tzv. *lookup table*).
- Takové tabulky jsou ale extrémně náročné na paměť.

Intuitivní vysvětlení:

Představte si, že si dopředu spočítáte všechny možné kombinace a uložíte je do tabulky. Při útoku pak stačí jen hledat — je to rychlé, ale obrovské množství dat zabírá mnoho místa (stejně jako byste chtěli mít seznam všech možných hesel na papíře).

Chytřejší předpočítávání: řetízky (Chains)

- Místo uložení všech kombinací se vytváří řetězce hashů.
- Každý krok hashujeme a vzápětí pomocí tzv. *sampling function* převedeme zpět na nové heslo.
- Ukládáme jen první a poslední heslo v řetězci.

Intuitivní vysvětlení:
Představme si cestu v bludišti. Neuložíme si každý krok, ale jen začátek a konec. Když se pak dostaneme do bludiště, můžeme projít cestu znovu, dokud nenajdeme bod, který odpovídá tomu, co hledáme.

Crackování hesel pomocí řetězců

- Útočník hashuje hledané heslo a postupuje v řetězci, dokud nenarazí na konec.
- Pokud konec odpovídá známému záznamu, útočník může projít řetězec od začátku a najít původní heslo.

Intuitivní vysvětlení:
Je to podobné jako hledání osoby podle její trasy: když víme, kam došla, zkusíme projít trasu od začátku a podíváme se, kde začínala.

Trade-off: paměť vs. čas

- Pokud je délka řetězce N, máme paměťovou náročnost přibližně $\frac{\#\text{hesel}}{N}$.
- Časová náročnost je naopak přibližně N, protože musíme iterovat.
- Řetězce se ale mohou spojovat, což zhoršuje efektivitu.

Intuitivní vysvětlení:
Je to jako kompromis při balení — místo toho, abychom si zabalili všechno, bereme jen část, ale při hledání věcí to pak déle trvá.

Rainbow tables

- Pro každou hranu řetězce použijeme jinou samplovací funkci.
- To ztěžuje útok — útočník neví, jaká funkce byla použita v konkrétním kroku.
- Pokud by dva řetězce kolidovaly, pravděpodobně se potkají na hranách s jinou funkcí, a tím se nespojí.
- Paměťová náročnost je cca $\frac{\#\text{hesel}}{N}$, ale časová N^2.

Intuitivní vysvětlení:
Rainbow tables jsou jako barevné cesty v bludišti — každá cesta má jinou barvu. Pokud dvě cesty narazí na stejný bod, pravděpodobně jsou to jiné barvy a nespojí se. Útočník musí zkusit všechny barvy.

Ochrana proti brute-force útokům

- **Salting**:
  - Ke každému heslu přidáme náhodný string (salt).
  - Zabraňuje použití rainbow tables.
  - Není vidět, že dva uživatelé mají stejné heslo.
- **Iterace hashovací funkce**:
  - Hashování se opakuje vícekrát.
  - Zpomaluje útoky, protože brute-force se stává náročnějším.

Intuitivní vysvětlení:
Salting je jako kdyby ke každému heslu přidali tajné slovo. Iterace je jako kdybychom zamykali trezor vícero zámky — útočník musí prolomit všechny.

PBKDF2 (Password-Based Key Derivation Function 2)

- Používá pseudonáhodnou funkci (typicky HMAC) a iterace.
- Vstupem je salt a délka požadovaného klíče.
- Výstupem je klíč $B_1, B_2, \ldots, B_n$.
- Výpočet B_i:
$$B_i = U_1^i \oplus U_2^i \oplus \cdots \oplus U_c^i$$
kde
$$U_1^i = \mathrm{PRF}(\mathrm{passwd}, \mathrm{salt}||i)$$
a
$$U_{j+1}^i = \mathrm{PRF}(\mathrm{passwd}, U_j^i).$$

Intuitivní vysvětlení:
Je to jako opakované zamíchání karet — čím více mícháme (iterujeme), tím těžší je poznat původní pořadí (heslo).

Challenge-response autentikace

Účel: Ověření hesla, aniž by bylo nutné heslo (nebo jeho hash) posílat přímo přes síť.

Princip

- Klient se přihlásí a požádá o autentizaci.
- Server vygeneruje náhodnou hodnotu (*nonce*) a zároveň pošle i *salt* (pokud je potřeba).
- Klient vypočítá:

$$\text{HMAC}(h(\text{passwd} \,||\, \text{salt}),\ \text{nonce})$$

- Server porovná přijatou hodnotu HMAC s očekávaným výsledkem.

Výhody

- Znalost HMAC podpisu útočníkovi nepomůže zpětně získat heslo.
- Hodnota závisí na náhodném *nonce*, což zabraňuje tzv. *replay útokům* (kdy by útočník znovu poslal zachycenou starou zprávu).

Intuitivní vysvětlení:
Je to podobné jako kdyby vám kamarád vždy, když ho navštívíte, dal náhodné slovo a řekl: Řekni mi heslo zakódované podle tohoto slova." Každý den je jiné slovo, takže i když někdo slyší, co řeknete, zítra už mu to nebude k ničemu.

Nevýhoda

- Server si musí pamatovat hash hesla a salt.
- Pokud útočník získá hash ze serveru, může se tvářit jako klient a provést celý protokol.

Intuitivní vysvětlení:
Je to jako kdyby někdo ukradl vaše zašifrované heslo a věděl přesně, jak reagovat na jakékoli náhodné slovo od kamaráda — protože ví, jak heslo zakódovat.

Poznámka: Tento způsob je odolný proti *replay útokům*, ale není odolný proti krádeži hashovaného hesla ze serveru.

24.1 SCRAM (Salted Challenge-Response Authentication Mechanism)

- Řeší případ, kdy útočník ukradne přímo hash hesla ze serveru.
- **Setup**:
  1. Zahasujeme heslo pomocí PBKDF2:

$$P := \text{PBKDF2}(\text{passwd}, \text{salt}, \#\text{iterací})$$

2. Vytvoříme klientský klíč:

$$K_C := \text{HMAC}(P, \text{nějaký string})$$

3. Vytvoříme serverový klíč:

$$K_S := \text{HMAC}(P, \text{ten samý string})$$

- Server si pamatuje salt, klíč serveru a hash klíče klienta.
- Klient si nemusí pamatovat nic kromě svého hesla.

Intuitivní vysvětlení: Cílem SCRAM je, aby ani když někdo ukradne databázi se salted hashi hesel, nemohl jednoduše provádět útoky. Heslo je vícekrát přešifrované a server místo hesla ukládá jen odvozené klíče.

24.1.1 Průběh SCRAM autentizace

- Klient pošle login a nonce (náhodné číslo).
- Server odpoví svým nonce a slaním.
- Klient spočítá **client-proof**:

$$\text{client-proof} = K_C \oplus \text{HMAC}(h(K_C), \text{transkript})$$

- Server spočítá **server-proof**:

$$\text{server-proof} = \text{HMAC}(K_S, \text{transkript})$$

Intuitivní vysvětlení: Klient musí dokázat, že zná heslo, aniž by heslo posílal. Udělá to pomocí klíče a HMAC. Server si ověří správnost výpočtu. Takto lze bezpečně ověřit heslo přes síť.

24.1.2 Bezpečnostní poznámky

- Server má $K_C \oplus \text{HMAC}(h(K_C), \text{transkript})$ a hash klíče a transkript.
- Operace $\oplus$ je inverzní sama k sobě, takže K_C lze jednoduše dopočítat při validaci.
- Poměrně těžké zamezit timing side-channel útokům.

Intuitivní vysvětlení: Používáme operaci XOR, protože ji lze snadno zpětně spočítat. Ale musíme být opatrní na útoky, které by měřily dobu výpočtů.

24.2 Útoky na autentizaci

- **Phishing** – útočník napodobí server a vyláká heslo.
- **Útoky hrubou silou** – zkoušení všech možných hesel.
- **Únik hashů** – pokud někdo získá hashovaná hesla, může je crackovat offline.
- **Replay útoky** – někdo zachytí starou zprávu a znovu ji pošle.

Ochrany:

- Salting, iterování hashů (PBKDF2, bcrypt, scrypt).
- Používání nonce (náhodných čísel) v protokolech.
- MFA (vícefaktorová autentizace).
- Omezení počtu pokusů o přihlášení.

Intuitivní vysvětlení:
Je to jako kdyby ses chránil proti zlodějům tak, že:

- Náhodně měníš zámek (nonce).
- Máš super těžký zámek (iterovaný hash).
- Máš druhý zámek (MFA).
- Nepustíš nikoho, kdo klepe příliš často (limit pokusů).

25 DNSSEC

25.1 Úvod

- DNSSEC znamená **secure domain naming system**.
- DNS je strom doménových jmen.

25.2 Záznamy

- Nachází se ve vrcholech doménového stromu.
- Typy záznamů: A, AAAA, MX, ...
- Delegace subdomén probíhá pomocí NS záznamů (na zónách, nikoliv doménách).

Intuitivní vysvětlení

DNS je jako telefonní seznam internetu. Pokud chceme někoho najít (např. `www.example.com`), musíme se ptát od hlavního seznamu (root) směrem dolů až do cílové zóny.

25.3 Zabezpečení v DNSSEC

- **DNSKEY** záznam obsahuje veřejný klíč domény.
- Tajný klíč se používá k podepisování záznamů, což vytváří **RRSIG** záznamy.
- Podpis v RRSIG se váže ke dvojici (jméno + záznam).

Intuitivní vysvětlení

Představme si, že doména podepisuje všechny odpovědi razítkem. Pokud odpověď není podepsaná (nebo je podpis neplatný), víme, že ji někdo mohl podvrhnout.

25.4 DS záznamy v nadřazených doménách

- Ukládají hash veřejného klíče domény.
- DS záznam je podepsán rodičem (otcovskou doménou).

Intuitivní vysvětlení

Je to jako kdyby rodič schválil podpis svého dítěte - tím se vytvoří důvěryhodný řetězec.

25.5 Změna klíčů

- Během přechodu máme u domény dočasně dva klíče.
- Update klíče by jinak znamenal aktualizaci celé větve stromu.
- Řešení: Používáme dva typy klíčů:
  - **Key signing key (KSK)** – podepisuje DNSKEY záznamy.
  - **Zone signing key (ZSK)** – podepisuje běžné záznamy.

Intuitivní vysvětlení

Je to podobné jako v organizaci — jeden klíč má hlavní vedoucí (KSK) a druhý běžně používaní pracovník (ZSK).

25.6 Doplnění (PKI, chain-KDF, Trust models a Double Ratchet)

PKI a certifikační autority

- **PKI (Public Key Infrastructure)** představuje systém důvěry založený na certifikátech.
- Každý veřejný klíč je podepsán autoritou (Certificate Authority - CA), která garantuje, že klíč patří dané entitě.

- Certifikační autority (CA) jsou důvěryhodné třetí strany, jejichž certifikáty jsou často předinstalované v operačních systémech a prohlížečích.
- V případě DNSSEC tvoří tento řetězec důvěry hierarchickou strukturu:
  - Kořenový zónový klíč (root) → DS záznamy → doménové klíče.
- PKI je tedy velmi robustní model důvěry, ale spoléhá se na důvěru ve správu certifikátů (CA může být napadena nebo udělat chybu).

Intuitivní vysvětlení

PKI je jako síť notářů. Kořenový notář (root CA) je absolutně důvěryhodný. Každý další notář (CA) potvrzuje pravost jednotlivých podpisů. Když vidíme nějaký podpis, zkontrolujeme, kdo ho garantuje, a jestli tomu můžeme věřit.

Chain-KDF (řetězové odvozování klíčů)

- Chain-KDF zajišťuje bezpečné generování nových klíčů z existujícího tajemství.
- Pokud dojde k úniku starého klíče, nové klíče (vygenerované z KDF) zůstávají bezpečné.
- V DNSSEC může být chain-KDF použit při rotaci ZSK nebo KSK klíčů.

Intuitivní vysvětlení

Chain-KDF je jako výrobní linka, která ze starých klíčů vyrábí nové a nové klíče, přičemž každý nový klíč je nezávislý. I kdyby se někdo dostal ke starému klíči, ten další už mu nebude k ničemu.

Další modely důvěry

- **Trust on First Use (TOFU)** – první kontakt se považuje za důvěryhodný, následné změny klíče už jsou podezřelé.
  - Používá se například u SSH klíčů.
- **Web of Trust** – každý uživatel podepisuje klíče ostatních uživatelů, čímž se vytváří síť důvěry.
  - Používá se například v PGP (Pretty Good Privacy).
  - Uživatel si sám určuje, komu důvěřuje.

Intuitivní vysvětlení

Trust on First Use je jako když poprvé uložíme telefonní číslo — příště už čekáme, že bude stejné. Web of Trust je jako sociální síť — čím víc lidí (kterým důvěřujeme) říká, že někdo je důvěryhodný, tím spíš mu věříme i my.

Double Ratchet Protocol (např. Signal)

- Používá se například v Signal protokolu pro zabezpečené chaty.
- Kombinuje dvě odlišné ratchety:
  - **Diffie-Hellman ratchet** – při každé nové zprávě se vygeneruje nový klíč z DH výměny.
  - **Symmetric-key ratchet** – pro každou jednotlivou zprávu se klíč odvozuje z předchozího pomocí KDF.
- Tím je zajištěno:
  - **Perfect forward secrecy** – odcizení starých klíčů neumožňuje dešifrovat budoucí zprávy.
  - **Post-compromise security** – po kompromitaci se po několika zprávách obnoví bezpečnost.

Intuitivní vysvětlení

Double Ratchet je jako zámek, který se při každém otočení klíče automaticky mění. I kdyby někdo klíč zkopíroval, další otočení už mu nebude fungovat.

26 TLS

26.1 Úvod

- **TLS (Transport Layer Security)** je nástupce SSL.
- Verze: SSL 1 → SSL 2 → SSL 3 → TLS 1.0 → TLS 1.1 → TLS 1.2 → TLS 1.3.
- Streamový protokol zajišťující:
  - autentizaci,
  - zabezpečení,
  - integritu.
- Používá se jako mezivrstva většiny internetových protokolů (např. HTTPS nebo DTLS).

Intuitivní vysvětlení

TLS je jako šifrovaný obal kolem běžné komunikace — ostatní vidí, že si povídáme, ale nevidí, co si říkáme.

26.2 TLS 1.3

- **Výměna klíčů:** Diffie-Hellman na eliptických křivkách.
- **Autentizace:** pomocí RSA podpisu a certifikátu.
- **Šifrování:**
  - AEAD módy (Authenticated Encryption with Additional Data).
  - AES-GCM nebo ChaCha20-Poly1305.
- **Dohoda na parametrech:** verze TLS, skupina pro DH, autentikační mechanizmus a šifra jsou z předdefinovaného seznamu.

Intuitivní vysvětlení

TLS 1.3 zjednodušilo nastavení — klient a server si mohou vybrat pouze z povolených možností, což zabraňuje volbě slabých parametrů.

26.3 Record protokol

- Zajišťuje šifrování a MAC (Message Authentication Code).
- Přidává padding — kvůli ochraně proti odhalení velikosti zpráv.
- Obsahuje podprotokoly:
  - handshake (dohodnutí parametrů),
  - posílání dat,
  - alert (např. chyba nebo ukončení komunikace).

Intuitivní vysvětlení

Record protokol funguje jako bezpečný dopravce — stará se o to, aby zprávy dorazily neporušené a bez cizího zásahu.

26.4 Key schedule

- V TLS je potřeba hodně klíčů pro různé fáze.
- Používá HKDF (HMAC-based Key Derivation Function).
- Klíče jsou odvozeny (extrakce a expanze).
- **Exporter secret:** používá se k vygenerování finálního šifrovacího klíče.

Intuitivní vysvětlení

Představme si to jako výrobu různých klíčů v továrně z jedné suroviny (HKDF) podle potřeby, například klíč na šifrování dat nebo pro ověřování.

27 TLS – detailní mechanismy a PKI

27.1 TLS Handshake

- **Handshake** je proces, při kterém si klient a server dohodnou parametry šifrování a vygenerují sdílený klíč.
- Na začátku ještě nejsou použity žádné šifry – handshake probíhá v otevřené formě.

Průběh komunikace

- **Client Hello:** klient navrhne parametry spojení – např. algoritmy, křivky, PSK (předem sdílené klíče).
- **Server Hello:** server vybere konkrétní parametry (klíčovou výměnu, šifru, certifikát...).
- **Certificate:** server může poslat certifikát pro autentikaci.
- **Certificate Verify + Finished:** server podepisuje dosavadní zprávy a ukončuje handshake.
- **Client Finished:** klient po ověření certifikátu taktéž pošle zprávu ukončující handshake.

Možné alternativy

- Pokud se neautentikuje klient, některé zprávy (Certificate Request, Cert Verify) se neodešlou.
- Klient může posílat šifrovaná aplikační data hned po dokončení handshake.

Intuitivní vysvětlení

Handshake je jako úvodní domluva na tajném jazyce. Obě strany se dohodnou, jak budou komunikovat, předají si klíč k překladači“, a teprve poté začnou skutečně mluvit (šifrovaně).

27.2 Další zprávy a mechanismy

- **New session key:** server může navrhnout nový klíč (re-keying).
- **Resumption:** klient použije nonce a resumption key“ k vytvoření PSK a otevření více spojení najednou.
- **Key Update:** jedna strana může v průběhu komunikace změnit šifrovací klíč.
- **Hello Retry:** server odmítne první Client Hello a požádá o nový s lepšími parametry.

27.3 Rozšíření TLS

Zero-latency handshake

- Klient může posílat tzv. **early data** už při Client Hello.
- Výhoda: žádné čekání na handshake.
- Nevýhoda: tato data jsou zranitelná vůči replay útokům – útočník může early data zachytit a později znovu poslat.
- Řešení: používat jen pro idempotentní akce (např. HTTP GET).

SNI – Server Name Indication

- Umožňuje klientovi říci serveru, jakou doménu chce navštívit (v Client Hello).
- Server tak může poslat správný certifikát.

ALPN – Application-Layer Protocol Negotiation

- Pokud má server více protokolů (např. HTTP/1.1 a HTTP/2), klient navrhne seznam, server vybere.

Encrypted Client Hello

- Vylepšení TLS 1.3, kde se celé Client Hello zašifruje, aby útočník neviděl například SNI.
- Pomáhá chránit metadata spojení.

27.4 Internet PKI

- Veřejná infrastruktura klíčů (PKI) je založena na **certifikátech podle standardu ITU X.509**.
- Je překomplexní – vznikla pro OSI model, nikoliv pro internet.

Certifikační autority (CA)

- CA vydávají certifikáty pro jednotlivé subjekty.
- Dříve byly všechny CA komerční, dnes existují i non-profit (např. **Let's Encrypt**).
- Používají se tzv. **intermediate certifikáty**, které delegují ověřování jiným subjektům.
- Problém: uživatel nemá přehled, kolik entit má právo vystavovat certifikáty.

Intuitivní vysvětlení

CA je jako notář, který ověřuje totožnost. Pokud ale notář nechá ověřování dělat svým asistentům (intermediate certifikáty), může být těžké zjistit, kdo vlastně co zaručuje.

27.5 Obsah certifikátu

- jméno držitele
- alternativní jména
- hash veřejného klíče
- jméno podepisující CA
- podpis
- doba platnosti
- použití a rozšíření certifikátu

27.6 Zneplatňování certifikátů

- **CRL (Certificate Revocation List):**
  - seznam zneplatněných certifikátů vydaný CA.
  - často velmi objemný, zastarává.
- **Custom revocation list:**
  - například prohlížeče mají vlastní seznam důležitých certifikátů (např. CA).
- **OCSP (Online Certificate Status Protocol):**
  - dotazujeme se CA, zda je certifikát stále platný.
  - problém: útočník může blokovat OCSP dotazy.
  - druhý problém: CA tímto sleduje, co navštěvujeme.
- **OCSP Stapling:**
  - OCSP odpověď přikládá rovnou server (např. při TLS handshake).
  - šetří čas a brání sledování.
- **Limited lifetime:**
  - certifikát má krátkou životnost (např. měsíc), takže revokace není až tak nutná.
- **Certificate Transparency:**

- každá CA musí publikovat vydané certifikáty do veřejných logů.
- založeno na Merkleově stromu – nelze měnit historii.
- majitel domény může logy monitorovat a detekovat neautorizované certifikáty.

Intuitivní vysvětlení

Systém zneplatňování certifikátů je jako odvolání občanky. Problém je, že ne každý kontroluje, jestli je ještě platná. Certificate Transparency je jako veřejný registr občanek – kdokoliv se může podívat, jestli někdo nemá podezřelý duplikát.