Masarykova univerzita
Fakulta informatiky

Návrh implementácia algoritmov RSA a HMAC pomocou whitebox kryptografie

Diplomová práca

Marián Čečunda

Brno, 2014

Prehlásenie

Prehlasujem, že táto diplomová práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovávaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Marián Čečunda

Vedúci práce: RNDr. Petr Švenda, Ph.D.

Poďakovanie

V prvom rade ďakujem Petrovi Švendovi za vedenie práce, jeho rady a komentáre k práci, diskusie o problémoch a za všetok čas, ktorý mi venoval.

Ďalej by som rád poďakoval priateľom, rodine a kolegom za ich podporu, pomoc a ústretovosť počas vypracovávania práce.

Zhrnutie

Táto práca sa zaoberá problematikou kryptografie v nedôveryhodnom prostredí. Popisuje a analyzuje možnosti implementácie niekoľkých hašovacích funkcií pomocou siete vyhľadávacích tabuliek a ich prípadné použitie v implementácií mechanizmu HMAC, pre nedôveryhodné prostredie. Ďalej je v práci navrhnutý spôsob na white-box implementáciu mechanizmu HMAC, najprv všeobecne, neskôr s použitím konkrétnej hašovacej funkcie. V rámci práce vznikla aj implementácia navrhnutého HMAC mechanizmu.

Práca skúma aj možnosti implementácie kryptosystému RSA vhodnej do nedôveryhodného prostredia a popisuje problémy spojené s jeho transformáciou na vyhľadávacie tabuľky.

Kľúčové slová

Hašovacie funkcie, HMAC, RSA, White-box kryptografia, White-box implementácia

Obsah

1 **Úvod** . . . 1
2 **White-box kryptografia a hašovacie funkcie** . . . 3
2.1 *White-box útočník* . . . 5
2.1.1 Vyhľadávacie tabuľky . . . 6
2.2 *Hašovacie funkcie* . . . 6
2.2.1 Kryptografické hašovacie funkciie . . . 7
2.3 *HMAC* . . . 10
3 **Prehľad a analýza hašovacích funkcií** . . . 11
3.1 *Operácie používané hašovacími funkciami* . . . 11
3.2 *BLAKE* . . . 14
3.3 *JH* . . . 16
3.4 *Skein* . . . 19
3.5 *Keccak* . . . 23
3.6 *Grøstl* . . . 24
3.7 *ECHO* . . . 27
3.8 *Shabal* . . . 28
3.9 *Fugue* . . . 30
3.10 *Diskusia* . . . 31
4 **White-box HMAC** . . . 33
4.1 *Návrh white-box HMAC protokolu* . . . 34
4.1.1 Predpoklady a prostredie protokolu . . . 34
4.1.2 WB HMAC Protokol . . . 35
4.1.3 Analýza protokolu . . . 37
4.2 *HMAC ako sieť vyhľadávacích tabuliek* . . . 38
4.2.1 White-box implementácia JH . . . 39
4.2.2 White-box JH a White-box HMAC . . . 44
4.3 *Diskusia* . . . 47
5 **White-box RSA** . . . 49
5.1 *Obfuskácia exponentu* . . . 50
5.1.1 Nedostatok obfuskácie . . . 51
5.1.2 Výpočtové nároky obfuskácie . . . 51
5.2 *White-Box umocňovanie* . . . 52
5.2.1 Algoritmy na umocnenie . . . 52
5.2.2 Modulárne násobenie . . . 53
5.3 *RSA pomocou RNS a MM* . . . 54

5.4 *Diskusia* 56
6 **Záver** 57
6.1 *Budúci výskum* 58
A **Popis implementácie a obsah CD** 62
A.1 *HMAC pre nedôveryhodné prostredie* 62
A.2 *Obfuskácia exponentu z kryptosystému RSA* 62

1 Úvod

Úlohou modernej kryptografie je navrhnúť algoritmy na zaistenie dôvernosti, integrity, autenticity dát a nepopierateľnosti. Tie sa potom dajú využiť na návrh protokolov na ochranu komunikačného kanála pred odpočúvaním.

Od polovice minulého storočia vzniklo veľké množstvo symetrických, asymetrických kryptosystémov, ale aj digitálnych podpisov či hašovacích funkcií, ktoré pomáhajú pri zaistení spomenutých cieľov. Pri ich návrhu sa počítalo s tým, že koncovým bodom sa dá pri výpočte dôverovať. Teda že výpočet prebieha v bezpečnom prostredí a útočník ma prístup, v rôznych kombináciách, iba k vstupu a výstupu[1]. Vo väčšine týchto kryptografických mechanizmov sa časom našli nejaké bezpečnostné nedostatky, ktoré viedli k ich, aspoň čiastočnému, prelomeniu. Samozrejme existujú aj návrhy mechanizmov, ktoré sú v zamýšľanom prostredí buď výpočtovo alebo preukázateľne bezpečné.

Útočník môže mať fyzický prístup k zariadeniu a viesť útok tzv. postranným kanálom. Ako takýto postranný kanál (z anglického *side channel*) môže slúžiť meranie času výpočtu, alebo spotreby energie. Útočník sa sledovaním postranných kanálov snaží o použitom kľúči niečo dozvedieť resp. kľúč odhaliť. Obrana proti takémuto útoku typicky znamená dodatočné nároky na hardware resp. zníženú efektivitu výpočtu.

Kryptografické algoritmy sú často navrhnuté pre použitie v štandardnom, black-box, prostredí. V tomto modele sa predpokladá, že komunikujúcim entitám a ich výpočtovému prostrediu môžeme dôverovať. Tento predpoklad nemusí byť v reálnom svete splnený. V súčastnosti sa stretneme aj s prípadom, kedy má útočník prístup k zariadeniu, a navyše aj kontroluje samotný výpočet. V takomto prípade tradičné kryptosystémy nemôžu splniť svoju úlohu.

Práve týmto problémom sa zaoberá tzv. *white-box* kryptografia, alebo kryptografia v nedôveryhodnom prostredí. V práci je tento prístup popísaný spolu s návrhom white-box implementácie vybraných kryptografických primitív.

1. known plaintext, known ciphertext, chosen plaintext, chosen ciphertext attack

V kapitole 2 si podrobnejšie predstavíme white-box kryptografiu. Vhodnosť vybraných hašovacích funkcií na implementáciu do nedôveryhodného prostredia je analyzovaná v kapitole 3. Kapitole 4 popisuje návrh white-box implementácie mechanizmu na výpočet hašovaného autentizačného kódu. V kapitole 5 je najprv popísaný existujúci spôsob ochrany kľúča kryptosystému RSA vo white-box prostredí, potom je popísaný útok na takúto ochranu a nakoniec sú uvedené problémy tradičných prístupov transformácie mechanizmov do white-box prostredia a ich použitie na ochranu kryptosystému RSA.

2 White-box kryptografia a hašovacie funkcie

Použitie kryptografických algoritmov sa postupne rozširuje aj na osobné počítače alebo chytré telefóny. K zraniteľnosti kryptosystémov často prispieva otvorený charakter týchto platforiem. Predpoklad o dôveryhodnosti koncových bodov kryptosystému nemôže byť splnený ak výpočet prebieha na zariadení, ktoré je pod plnou kontrolou útočníka. Pár nasledujúcich odstavcov sa budeme venovať činnostiam útočníka, ktoré môžu viesť až k odhaleniu kľúča za takýchto podmienok.

Ak spolu chcú dve entity komunikovať šifrovane, potrebujú šifrovací kľúč. Tento kľúč by nemal byť ľahko uhádnuteľný, preto je vhodné, ak je zvolený náhodne. Náhodnosť sa dá merať entropiou a tento šifrovací kľúč by mal mať vysokú entropiu. Na druhej strane algoritmus na šifrovanie resp. dešifrovanie má nízku entropiu, pretože skompilovaný do programu sa bude skladať len z obmedzeného počtu inštrukcií.

V [18] autori ukázali jednoduchý spôsob na lokalizáciu kľúča v skompilovanom programe. Na obrázku 2.1 je znázornená binárna implementácia programu so zabudovaným tajným kľúčom. Zatiaľ čo časti s nízkou entropiou na ľavej a pravej strane majú nejakú štruktúru, časť v strede, s vysokou entropiou, sa javí ako šum. Preto môžme s relatívne vysokou istotou tvrdiť, že tajný kľúč sa bude nachádzať v časti súboru odpovedajúcej tejto časti obrázku. Na určenie presnej polohy a hodnoty kľúča je potrebný hlbší rozbor, ale tento prístup nám cestu k odhaleniu kľúča výrazne skrátil.

Obr. 2.1: Grafické znázornenie binárnych dát alebo pamäte programu (bit s hodnotou 0 ako čierna bodka, bit s hodnotou 1 ako biela).

Bielenie pomocou kľúča (z anglického *key whitening*) je jednou z techník, ako ľahko zvýšiť bezpečnosť blokových šifier. Myšlienkou

techniky je skombinovanie kľúča s dátami na vstupe alebo výstupe šifrovacieho a dešifrovacieho algoritmu pomocou operácie XOR a zabrániť útočníkovi získavať navzájom si odpovedajúce páry zašifrovaných a nezašifrovaných správ. Medzi blokové šifry, ktoré využívajú túto techniku, patrí okrem iných aj Twofish alebo AES.

Ak je tento prístup použitý v prostredí, kde má výpočet pod kontrolou útočník, môže ho veľmi jednoducho využiť na zistenie kľúča následovne. Konkrétne AES, pred týmto záverečným pridávaním kľúča zmení hodnoty všetkých bajtov vnútorného stavu pomocou S-boxu. Keďže je tento S-box verejne známy, útočník ho môže pomocou jednoduchého hex editoru nájsť v spustiteľnom súbore a všetky hodnoty zmeniť na nuly. Výstup takto modifikovaného algoritmu potom bude kľúč použitý v poslednom kole spracovávania dát. U AES algoritmu sa z tohto kľúča dá ľahko odvodiť pôvodný šifrovací kľúč. Tento postup predviedli autori v [12].

Na týchto príkladoch vidíme, že dôveryhodnosť všetkých koncových bodov kryptosystému nie je samozrejmá. Ak má útočník prístup k zariadeniu na ktorom beží výpočet, alebo k samotnej softvérovej implementácií, môže jednoducho analyzovať kód, príslušnú pamäť, prerušiť systémove volania, modifikovať binárny spustiteľný súbor alebo jeho výpočet.

V podobných situáciách nám štandardný model útočníka nestačí. Potrebujeme vytvoriť nový model silnejšieho útočníka. Jeden takýto model ponúka *white-box kryptografia*. Oproti tradičnému útočníkovi dáva jeho white-box náprotivku (tzv. *white-box útočníkovi*) viac možností ako napadať rôzne kryptosystémy. Detailnejšie si white-box útočníka predstavíme v časti 2.1.

Ako rýchly úvod do problematiky white-box kryptografie môže poslúžiť [21], viac podrobnejších informácií je možné nájsť v [20] alebo v [13][1].

V praxi sa white-box model dá použiť napríklad v DRM (Digital Rights Management) systémoch, všeobecne v akomkoľvek systéme, kde nemôžeme dôverovať zariadeniam koncových užívateľov. Svoje white-box riešenia ponúka na trhu niekoľko spoločností napr. SafeNet (Sentinel), Irdeto (Cloakware Security Kernel), Arxan (TransformIT), whiteCryption (Cryptanium), alebo Microsemi

1. Alebo aj na `whiteboxcrypto.com`

(WhiteboxCRYPTO).

V tieto produkty obsahujú implementácie rôznych kryptografických mechanizmov ako DES, 3DES, AES, RSA, ECC, ECDSA, DH, ECDH, SHA a pravdepodobne používajú kombináciu akademických white-box návrhov, obfuckácie kódu a proprietárnych riešení. Väčšina týchto implementácií, z pochopiteľných dôvodov, nie je zverejnená. V časti 5.1 bude jeden z patentovaných návrhov analyzovaný.

2.1 White-box útočník

Ak chceme dokázať, že kryptosystém je bezpečný, potrebujeme opis schopností útočníka, proti ktorému nás má tento kryptosystém chrániť. Tradične sa v kryptografii bránime proti útočníkom, majúcim prístup k vstupom a výstupom kryptografických algoritmov. V rôznych variantách potom môžu mať prístup iba k výstupom (known ciphertext attack), k vstupom aj výstupom (known plaintext attack), môžu byť schopní do kryptosystému vložiť zašifrované dáta a získať ich v dešifrovanej podobe (chosen ciphertext attack), alebo naopak do systému vložiť dáta a získať ich v zašifrovanej podobe (chosen plaintext attack). Útočník nemá prístup k samotnému výpočtu, ale, podľa Kerckhoffovho princípu, pozná algoritmy kryptosystému. Pri konkrétnych útokoch môže útočník naviac využiť aj známe chyby v implementácií, volať funkcie s nesprávnymi parametrami, alebo narušiť kontext výpočtu.

White-box kryptografia počíta s útočníkom, ktorý naviac úplne kontroluje výpočet algoritmu. Konkrétne [11][21]:

- Útočník môže pozorovať výpočet algoritmu:
  - vidí spracovávané inštrukcie,
  - je schopný prečítať používanú pamäť,
  - sleduje chod algoritmu.
- Útočník môže za behu meniť výpočet algoritmu:
  - meniť hodnoty v pamäti,
  - nechať prebehnúť len časť algoritmu,

– menit' podmienky v algoritme,
– menit' riadiace premenné v cykloch,
– vyvolávat' chyby.

2.1.1 Vyhl'adávacie tabul'ky

Najbezpečnejší spôsob ako sa proti white-box útočníkovi bránit' je transformovanie celého algoritmu na jednu vel'kú vyhl'adávaciu tabul'ku. Dosiahli by sme rovnakú úroveň zabezpečenia ako majú algoritmy v black-box prostredí. Problémom tohto prístupu je vel'kost' takejto tabul'ky. Aj pre jednoduché algoritmy by bola obrovská a preto nepoužitel'ná.

Namiesto toho môžme algoritmus transformovat' na siet' vyhl'adávacích tabuliek závislých na kl'úči. Celková funkcionalita algoritmu sa nezmení, ale tajný kl'úč je zabudovaný v sieti tabuliek.

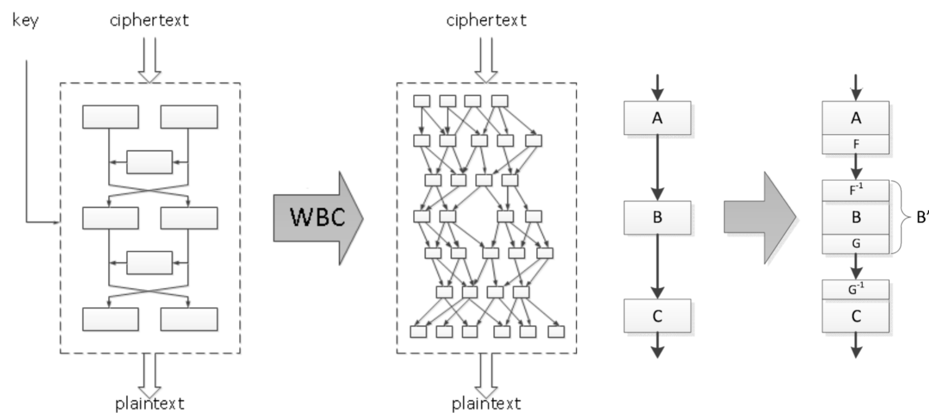
V procese transformácie by sme do siete mali vložit' aj náhodnost' na ochranu kl'úča, medzi jednotlivými vyhl'adávaniami v sieti. Na to sa tzv. vstupno-výstupné kódovania.

Každú tabul'ku modifikujeme nasledovne. Pre každú dvojicu tabuliek zvolíme náhodnú bijekciu a spolu s jej inverznou bijekciou budeme kódovat' data medzi týmito dvoma tabul'kami. Z pôvodnej tabul'ky a dvoch kódovaní vytvoríme novú tabul'ku a zo všetkých nových tabuliek budeme mat' novú siet'.

2.2 Hašovacie funkcie

Pojem hašovanie resp. hašovacia funkcia [16] má v informatike dva významy. Jeden súvisí so zrýchlením vyhl'adávania dát vo vel'kých množinách na spôsob kl'úč (haš) - hodnota (dáta). V tejto práci nás budú naopak zaujímat' kryptografické hašovacie funkcie. Obe spomínané hašovania majú spoločnú vlastnost'. Mapujú l'ubovol'ne dlhý vstup na výstup pevnej dĺžky.

V kryptografii hašovacie funkcie pomáhajú pri zaistení integrity dát, napríklad pri vytváraní digitálneho podpisu. Kvôli nízkej rýchlosti väčšiny asymetrických šifier sa nepodpisuje celá správa, iba jej odtlačok - haš. Okrem toho sa s hašovacími funkciami môžme stret-



Obr. 2.2: Transforácia na vyhľadávacie tabuľky (vľavo) a vstupno-výstupné kódovania (vpravo), prevzaté z [21]

núť aj pri počítaní autentizačných kódov správ (MAC[2]). MAC algoritmus spočíta tento kód pomocou kľúča a dát. Príjemca správy, ktorej súčasťou je autentizačný kód dát, je schopný zistiť, či bola správa zmenená a overiť pôvod tejto správy. Jednou variantou takýchto autentizačných kódov sa budeme podrobnejšie zaoberať v časti 2.3.

V tejto časti si ďalej popíšeme požiadavky kladené na hašovacie funkcie v kryptografii a niekoľko typov ich konštrukcií.

2.2.1 Kryptografické hašovacie funkciie

Aby bola hašovacia funkcia vhodná na kryptografické použitie, musí mať ešte niekoľko dodatočných vlastností [16][3]. Jednou z nich je *jednosmernosť*.

Funkcia $h(x) = h$ je *jednosmerná*, ak pre dané x je jednoduché spočítať h, a zároveň pre dané h je zložité spočítať také x, aby $h(x) = h$.

Druhou požadovanou vlastnosťou hašovacích funkcií je *bezkolíznosť*. U nej rozlišujeme dva typy. Funkcia $h(x)$ je *slabo bezkolizná* ak pre dané x je zložité nájsť y, také že $h(x) = h(y)$. Funkcia $h(x)$ je *silno*

2. z anglického *message authentication code*
3. Pre jednoduchosť sú v popisoch vlastností, a ďalej v práci, použité pojmy „jednoduché“ a „zložité“. Prvý z nich vo význame „výpočtovo jednoduché“ alebo „vypočitateľné v polynomiálnom čase“, druhý zasa „výpočtovo zložité“ alebo „nevypočitateľné v polynomiálnom čase“

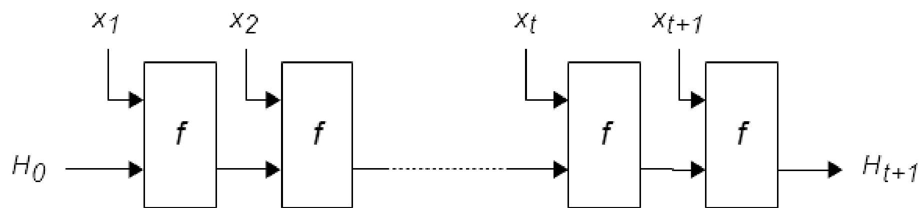
bezkolizná ak je zložité nájsť x a y, také že *h(x) = h(y)*.

V tejto časti budeme potrebovať aj pojem kompresnej funkcie. Funkcia *h(x)* je *kompresná*, ak má vstup ľubovoľnej, ale pevnej dĺžky M a výstup pevnej dĺžky N, kde $M > N$.

Podľa vnútorného fungovania sa hašovacie funkcie dajú rozdeliť na niekoľko typov. Jedným z nich je tzv. Merkle-Damgård konštrukcia, ktorá využíva fakt, že akákoľvek kompresná funkcia odolná voči kolíziám môže byť rozšírená na hašovaciu funkciu rovnako odolnú voči kolíziám. Myšlienkou rozšírenia je rozdelenie vstupu hašovacej funkcie na t blokov dĺžky n, čo je dĺžka vstupu kompresnej funkcie[4]. Následne sa za správu pridá blok obsahujúci dĺžku pôvodnej správy. Výsledný haš správy x sa potom spočíta takto:

$$h(x) = H_{t+1} = f(H_t \parallel x_{t+1})$$
$$H_i = f(H_{i-1} \parallel x_i), 1 \leq i \leq t+1$$

kde f je kompresná funkcia, x_i sú bloky správy dĺžky n, H_0 je reťazec núl dĺžky n. Rovnaké rozšírenie je znázornené aj na obrázku 2.3. Pred



Obr. 2.3: Merke-Damgård konštrukcia hašovacej funkcie

vytvorením samotného hašu správy, môže posledná hodnota H_{t+1} prejsť ešte tzv. *finalizáciou*, v ktorej stav hašovacej prejde záverečnou úpravou.

Medzi hašovacie funkcie využívajúce túto konštrukciu patria napríklad kandidáti na SHA-3 štandard Grøstl, JH alebo predchádzajúce verzie štandardu SHA-2, SHA-1 či v minulosti často používaná MD5.

Ďalším typom konštrukcie hašovacej funkcie je tzv. špongiová [1]. Jej základom je permutácia alebo transformácia *f*, ktorá pracuje

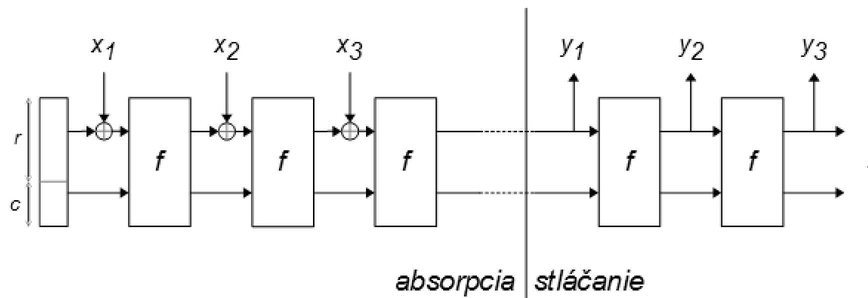
4. S prípadným zarovnaním posledného bloku.

s *b* bitmi vnútorného stavu funkcie. Takáto funkcia potom môže mať ľubovoľne dlhý vstup aj výstup.

Hašovanie takýmto typom funkcie má dva fázy: *absorpčnú* a *stláčaciu*[5]. V prvej, absorpčnej fáze, sa vstupná správa rozdelí na bloky x_i dĺžky r[6]. Každý z nich sa postupne kombinuje operáciou XOR s prvými *r* bitmi vnútorného stavu funkcie. Toto kombinovanie sa prekladá aplikáciou permutácie *f* na vnútorný stav.

Po vyčerpaní všetkých blokov správy funkcia prechádza do nasledujúcej fázy. Vo fáze stláčania sa *r* bitov vnútorného stavu vráti ako časť výstupu y_i funkcie a stav funkcie sa znova zamieša permutáciou *f*. Haš správy dostaneme zreťazením výstupných blokov, ich počet si volí užívateľ hašovacej funkcie.

Vnútorný stav špongiovej hašovacej funkcie tvorí $b = r + c$. Posledných *c* bitov stavu nie je nikdy priamo ovplyvnených vstupom a nikdy nie sú súčasťou výstupu. Pre názornosť je špongiová konštrukcia znázornená aj na obrázku 2.4



Obr. 2.4: Špongiová konštrukcia hašovacej funkcie

Príkladom hašovacej funkcie používajúcej špongiovú konštrukciu môže byť napríklad víťaz súťaže o SHA-3 štandard, algoritmus Keccak, ďalej Fungue, CubeHash alebo Luffa.

5. z anglického *absorb* a *squeeze*
6. S prípadným zarovnaním posledého bloku.

2.3 HMAC

Hašovaný autentifikačný kód [14] (HMAC[7]) je typ autentizačného kódu správy. Takisto sa používa na kontrolu integrity a autentizácie správ posielaných cez nedôveryhodný komunikačný kanál. Bol navrhnutý s ohľadom na päť hlavných požiadaviek:

- používať dostupné hašovacie funkcie bez dodatočnej modifikácie,
- zachovať pôvodný výkon použitej hašovacej funkcie,
- jednoduché použitie kľúčov,
- za rozumných predpokladov na použitú hašovaciu funkciu, poskytnúť ľahko pochopiteľnú analýzu sily tohto mechanizmu.
- ak bude potrebná rýchlejšia alebo bezpečnejšia hašovacia funkcia, umožniť jej jednoduchú výmenu.

Tento mechanizmus spočíta autentizačný kód pomocou tajného kľúča K a hašovacej funkcie H, s veľkosťou bloku B bajtov, nasledovným spôsobom. Najprv definujeme dva reťazce:

ipad = B krát opakujúci sa bajt 0x36
opad = B krát opakujúci sa bajt 0x5C

Ak to je potrebné, tak pridaním nulových bajtov zarovnáme kľúč K na veľkosť B bajtov. Potom spočíta autentizačný kód dát (označme ako *text*) takto:

HMAC = *H(K XOR opad, H(K XOR ipad, text))*

Bezpečnosť takto navrhnutého mechanizmu závisí na vlastnostiach použitej hašovacej funkcie a veľkosti tajného kľúča.

7. z anglického *keyed-hash message authentication code*

3 Prehľad a analýza hašovacích funkcií

Na to, aby sme z HMAC-u transformovali na sieť vyhľadávacích tabuliek, potrebujeme hašovaciu funkciu vhodnú na implementáciu pomocou takýchto tabuliek. V tejto kapitole je popis niekoľkých hašovacích funkcií z druhého a tretieho kola súťaže na SHA-3 štandard.

Po popise každej funkcie sú krátko zhrnuté dôvody, prečo je či nie je vhodná na použitie pre implementáciu HMAC-u do nedôveryhodného prostredia.

V tabuľke 3.1 sú zhrnuté niektoré vlastnosti hašovacích funkcií z druhého kola súťaže o štandard SHA3. Ide o typ konštrukcie hašovacej funkcie, veľkosť vnútorného stavu v bitoch, veľkosť spracovávaného bloku správy a počet kôl na spracovanie jedného bloku správy.

Tieto údaje nepovedia o vhodnosti funkcie na white-box implementáciu veľa, môžu ale napovedať, čo konkrétna funkcia so vstupným blokom robí. Málo kôl s veľkým blokom naznačuje, že operácie vo vnútri budú zložité a transformácia na sieť vyhľadávacích tabuliek bude veľmi zložitá.

Naopak, veľký počet kôl naznačuje veľa jednoduchších operácií, teda sieť by bola zložená z veľkého množstva malých tabuliek. Malé tabuľky je vhodné spájať do väčších, prípadne ich ku väčším pripájať, aby invertovanie operácií bolo zložitejšie.

3.1 Operácie používané hašovacími funkciami

Hašovacie funkcie používajú na spracovanie dát širokú škálu rôznych operácií. V tejto časti si predstavíme ako ich môžme, resp. prečo ich nemôžme transformovať na vyhľadávacie tabuľky a spomenieme výhody a nevýhody takýchto transformácií.

Uvažujme o vyhľadávacia tabuľka, ktorá ako vstup dostáva n bitov (ako jedno alebo viacero slov) a jej výsledok má m bitov. Akúkoľvek operáciu táto tabuľka so vstupnými slovami vykoná, musí vrátiť výsledok pre všetky možne vstupy. Tých je celkom 2^n. Každý z výstupov má m bitov, preto veľkosť celej tabuľky bude $m \cdot 2^n$ bitov.

Veľkosť tabuľky rastie exponenciálne. Pre 33 bitový vstup bude

Hašovacia funkcia	**Typ konštr.**	**Veľkosť stavu** (bity)	**Počet kôl**	**Blok** (bity)
BLAKE	MD	512/1024	14/16	512/1024
BMW	MD	512/1024	-	512/1024
CubeHash	Š	1024	16	256
ECHO	MD	2048	8/10	1536/1024
Fugue	Š	960/1152	2 + (39)	32
Grøstl	MD	512/1024	10/14	512/1024
Hamsi	MD	512/1024	-	32/64
JH	MD	1024	42	512
Keccak	Š	1600	24	ľubovoľný
Luffa	Š	256	-	256
Shabal	MD	1024 + r*32	1	512
SHAvite-3	DM	1024	12	512
SIMD	DM	512/1024	-	512/1024
Skein	MMO	256/512/1024	72/72/80	256/512/1024

Tabuľka 3.1: Prehľad hašovacích funkcií z druhého kola súťaže o SHA3 štandard. MD - Merkle-Damgård, Š - Špongiová konštrukcia, DM - Davies-Meyer mód blokovej šifry, MMO - Matyas-Meyer-Oseas mód blokovej šifry

mať m GB. V práci sú tabuľky so vstupom dĺžky n a výstupom dĺžky m označované ako tabuľky typu $typun \times m$.

Pri množstve opakujúcich sa operácií v hašovacích funkciách je tabuľka so šestnástimi bitmi vstupu a ôsmimi bitmi výstupu (64 kB) v podstate nepoužiteľná.

Exclusive-OR (XOR) a ostatné logické operátory

Tvorba tabuľky na kombináciu dvoch slov pomocou operácie XOR je pomerne priamočiara. Pre vstup dĺžky $2n$ (slová A, B dĺžky n) bude tabuľka na výstupe mať jedno slovo C dĺžky n, ktoré bude výsledkom $C = A\ XOR\ B$. Analogicky by sme vytvorili tabuľku aj pre ostatné logické operátory.

Počítanie tejto operácie pomocou tabuľky je dosť neefektívne, ale väčšinou nutné na ochranu výpočtu. Je žiadúce aby sa takéto tabuľky spájali s väčšími tabuľkami iných operácií.

WB implementácia AES v [11] používa takúto tabuľku s osembitovým vstupom (dva štvorbitové slová) a štvorbitovým výstupom. Veľkosť jednej takejto tabuľky je 128 bajtov.

Permutácie, rotácie a posuny

Permutácie, bitové rotácie a bitové posuny slov by sa na vyhľadávaciu tabuľku transformovali veľmi jednoducho. Vhodnejšie ale je, ak sú súčasťou inej, väčšej tabuľky.

Zahrnúť bitový posun alebo rotáciu či permutáciu do tabuľky je možné, ak sa týka len slova, s ktorým tabuľka pracuje. V prípade modifikácie väčších slov môže pomôcť spojenie viacerých tabuliek, to ale môže spôsobiť natiahnutie vstupu a neprijateľné zväčšenie tabuľky.

Ak sa stav hašovacej funkcie posúva alebo permutuje po blokoch, môže sa takáto transformácia prejaviť na prepojení jednotlivých tabuliek. Napríklad permutácia štvorbitových blokov v sieti, kde tabuľky pracujú s osembitovými vstupmi.

Modulárne sčítanie a odčítanie

Transformovať modulárne sčítanie (taktiež odčítanie) na vyhľadávaciu tabuľku nie je problém, kým sú vstupné slova krátke (spolu do 12 bitov).

Problém s použitím modulárneho sčítania v hašovacích funkciách je vo veľkosti použitých slov. 32 a 64 bitové slová sú pomerne bežné a v celku ich ako vstup pre jednú tabuľku použiť nemôžme.

Rozdelením operácie sčítania resp. odčítania na menšie bloky s pridaním bitu prenosu z predchádzajúceho bloku by sme si mohli pomôcť. Bit prenosu by ale v takejto sieti bol slabým miestom, pretože môže obsahovať len jednú z dvoch hodnôt.

Sčítanie dvoch dlhých slov by sme teda mohli rozdeliť na sčítanie štvorbitových blokov a menším tabuľkám by sme pridali po jednom bite (bit prenosu) na vstup a výstup. Potrebovali by $5 \cdot 2^9$ bitov (320 bajtov) pamäte.

Na takéto sčítanie dvoch 32 bitových slov by sme potrebovali 8 menších tabuliek, čo je 2,5 kB. V prípade 64 bitových slov by to bolo 5 kB.

Tieto veľkosti tabuliek sú priateľnejšie, kvôli spomínanému problému s bitom prenosu by sme sa sčítaniu a odčítaniu mali radšej vyhnúť.

Ostatné

Hašovacie funkcie často používajú aj nelineárnu transformáciu pomocou S-boxov. Tá už v podstate je vyhľadávacou tabuľkou.

Ostatné používané operácie si analyzujeme pri popise jednotlivých hašovacích funkcií.

3.2 BLAKE

Popis funkcie

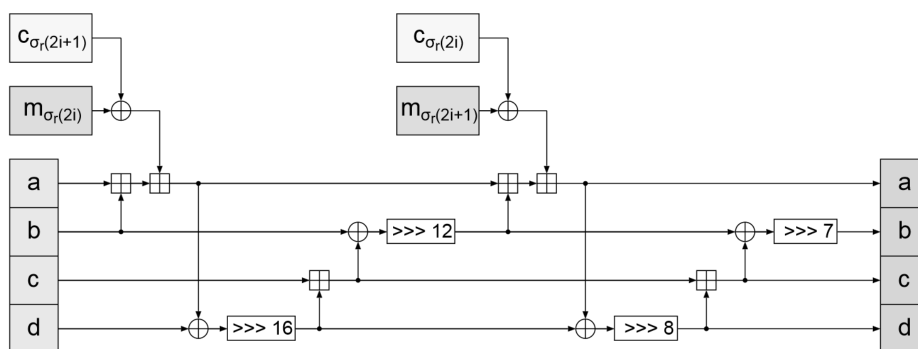
Funkcia BLAKE [3] je hašovacia funkcia typu Merkle-Damgård, vychádzajúca z prúdovej šifry ChaCha. BLAKE dokáže vytvárať 224, 256, 384, alebo 512 bitové haše dát. Jej vnútorný stav má 512 resp. 1024 bitov, podľa zvolenej dĺžky hašu.

Jeho kompresná funkcia spracováva správu po šestnástich slovách. Vnútorný stav sa používa ako matica typu 4×4 šestnástich slov. Inicializuje ju pomocou reťazovej hodnoty, soli, čítača a konštánt.

Kompresná funkcie ho modifikuje pomocou funkcie G_i znázornenej na obrázku 3.1. Ako vstup potrebuje štyri slová (označme *a, b, c, d*), spracuje dve slová správy a pozostáva z nasledujúcich krokov:

$$
\begin{aligned}
a &= a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})\\
d &= (d \oplus a) \ggg 16\\
c &= c + d\\
b &= (b \oplus c) \ggg 12\\
a &= a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})\\
d &= (d \oplus a) \ggg 8\\
c &= c + d\\
b &= (b \oplus c) \ggg 7
\end{aligned}
$$

Znázornené sčítanie je modulárne[1], σ_r je jedna z desiatich permutácií, *c* je jedno z konštantných slov, a *m* je jedno slovo správy.



Obr. 3.1: Funkcia G_i pracujúca s 32 bitovými slovami, prevzaté z [3]

V každom kole zavolá kompresná funkcia funkciu G_i 8-krát. Ako vstup jej dáva postupne 4 stĺpce a 4 diagonály matice stavu. Výsledkom kompresnej funkcie bude reťazová hodnota pre ďalší blok správy.

1. podľa veľkosti slova, popísané nižšie, modulo 2^{32} alebo 2^{64}

Po poslednom bloku správy je 8 slovný haš vytvorený kombináciou slov poslednej reťazovej hodnoty, soli a stavu.

Verzia funkcie BLAKE s 256 resp. 224 bitovým hašom používa 32 bitové slová. Vnútorný stav a blok správy majú teda po 512 bitov. Kompresná funkcia má 14 kôl.

512 resp. 384 bitová verzia používa 64 bitové slová teda 1024 bitový stav a blok správy. Jeden blok spracuje v 16 kolách. Tieto verzie naviac vo funkcii G_i používajú rotácie o 32, 25, 16, resp. 11 bitov.

Analýza

Funkcia BLAKE sa vo veľkej miere spolieha na modulárne sčítanie. Navyše slová správy a stavu používa v celku.

Kombinácia týchto dvoch faktov spôsobuje, že vyhľadávacie tabuľky na jednotlivé operácie potrebujú veľa pamäte, čo je pre potencionálnu white-box implementáciu problém.

Množstvo volaní funkcie G_i túto veľkosť ešte viac znásobí. Ak by tabuľka na jedno modulárne sčítanie mala 2,5 kB resp. 5 kB, tak ako je ukázané v časti 3.1, funkcia BLAKE by na všetky použité sčítania 32 bitových slov, pri spracovaní jedného bloku správy potrebovala 1680 kB pamäte resp. v prípade 64 bitových slov 3840 kB pamäte. Na všetky operácie XOR by bolo treba ďalších 672 kB resp. 1344 kB pamäte.

Dobrou myšlienkou vo funkcii BLAKE je, že spracovanie bloku správy rozdeľuje do viacerých veľmi podobných spracovaní po dvojiciach slov. Na druhej strane, náväznosť jednotlivých operácií vrámci funkcie G_i zabraňuje ďalšej palatalizácií.

Ak by sme vedeli transformovať funkciu G_i na vyhľadávacie tabuľky, spojenie niekoľkých takýchto sietí do väčšej siete na spracovanie bloku správy by bolo veľmi jednoduché.

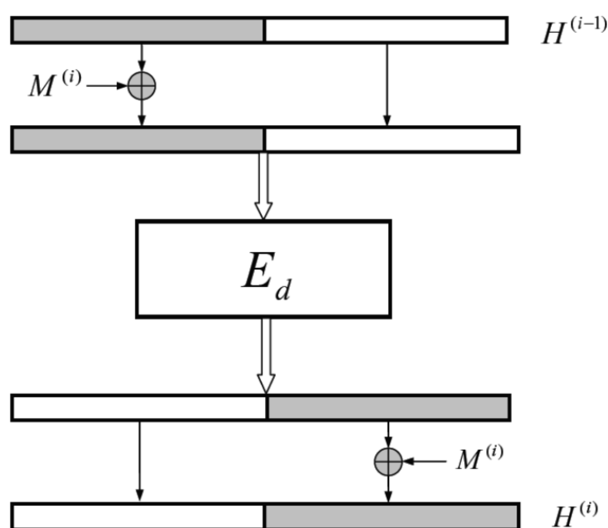
3.3 JH

Popis funkcie

Funkcia JH [19] je hašovacia funkcia typu Merkle-Damgård. Výsledný haš môže mať dĺžku 224, 256, 384, alebo 512 bitov a jej vnú-

torný stav má 1024 bitov.

Jej kompresná funkcia F_8 spracováva správu po 512 bitových blokoch. Každý z nich najprv operáciou XOR skombinuje s prvou polovicou vnútorného stavu, modifikuje ho pomocou funkcie E_8, a na záver ho opäť operáciou XOR skombinuje s druhou polovicou vnútorného stavu. Tento postup je znázornený na obrázku 3.2.



Obr. 3.2: Kompresná funkcia funkcie hašovacej funkcie JH, prevzaté z [19]

Funkcia E_8 zoskupí bity stavu do štvoríc, modifikuje ich v štyridsiatich dvoch kolách funkciou R_8 a rozdelí štvorice na pôvodne miesta. Každá aplikácia R_8 má tri fázy:

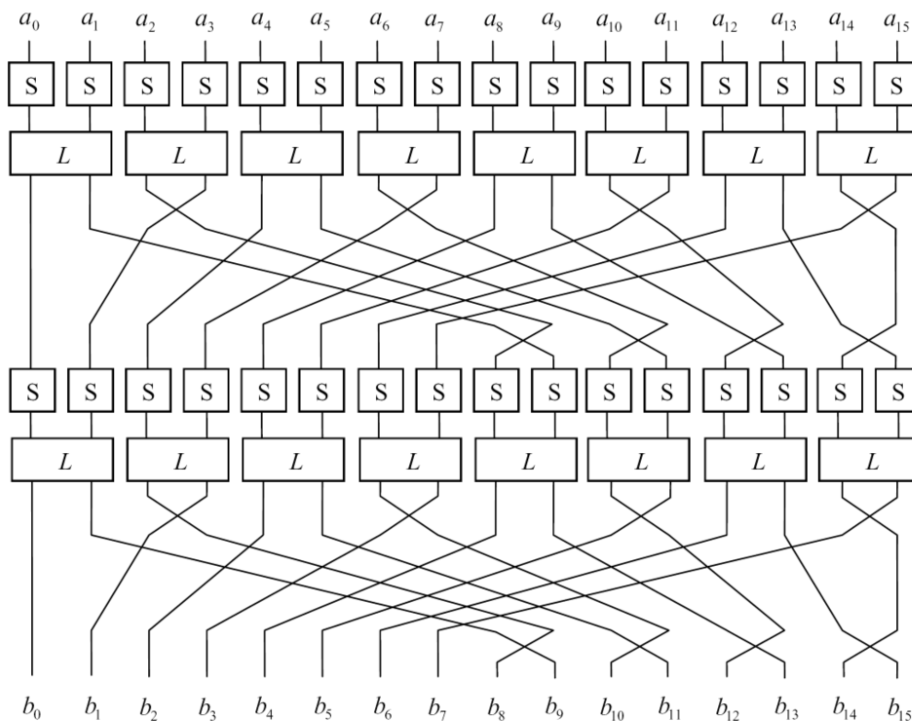
- Modifikácia každej štvorice bitov S-boxom. JH používa dva rôzne S-boxy, o výbere pre konkrétnu štvoricu bitov rozhoduje konštanta kola.
- Aplikácia lineárnej transformácie na každý bajt stavu (Zodpovedá aplikácií MDS kódu, viac v [19]).
- Permutácia vnútorného stavu zložená z troch jednoduchších permutácií.

Ilustrácia dvoch kôl funkcie E_8 je na obrázku 3.3.

Zoskupenie bitov prebieha tak, že vnútorný stav sa rozdelí na štyri 256 bitové časti. Prvú štvoricu bitov budú tvoriť prvé bity z každej časti. Druhú štvoricu budú tvoriť sto dvadsiate ôsme bity. Tretia štvorica vznikne z druhých bitov, štvrtá zo sto dvadsiatych deviatych bitov.

Takto vznikne spolu 256 štvoríc, ktoré po 42 kolách funkcia E_8 analogicky rozdelí naspäť do jedného reťazca 1024 bitov vnútorného stavu.

Haš správy tvorí posledných 224, 256, 384, resp. 512 bitov vnútorného stavu po spracovaní posledného bloku správy.



Obr. 3.3: Ilustrácia dvoch kôl funkcie E_8, prevzaté z [19]

Analýza

Všetky operácie, ktoré hašovacia funkcia JH používa sa dajú transformovať na vyhľadávacie tabuľky. Jej nevýhodou je, že finalizácia

spočíva v odhalení časti vnútorného stavu. Veľkosť funkcie JH transformovanej na sieť vyhľadávacích tabuliek, na spracovanie 512 bitového bloku správy, je približne 2,6 MB.

Návrh použitia funkcie JH na implementáciu white-box HMAC-u je popísaný v kapitole 4.

3.4 Skein

Popis funkcie

Funkcia Skein[7] bola navrhnutá tak, aby podporovala tri rôzne veľkosti vnútorných stavov: 256, 512, 1024. Ľubovoľná dĺžka výstupu je poskytovaná všetkými troma variantami.

Základné stavebné bloky tejto hašovacej funkcie sú:

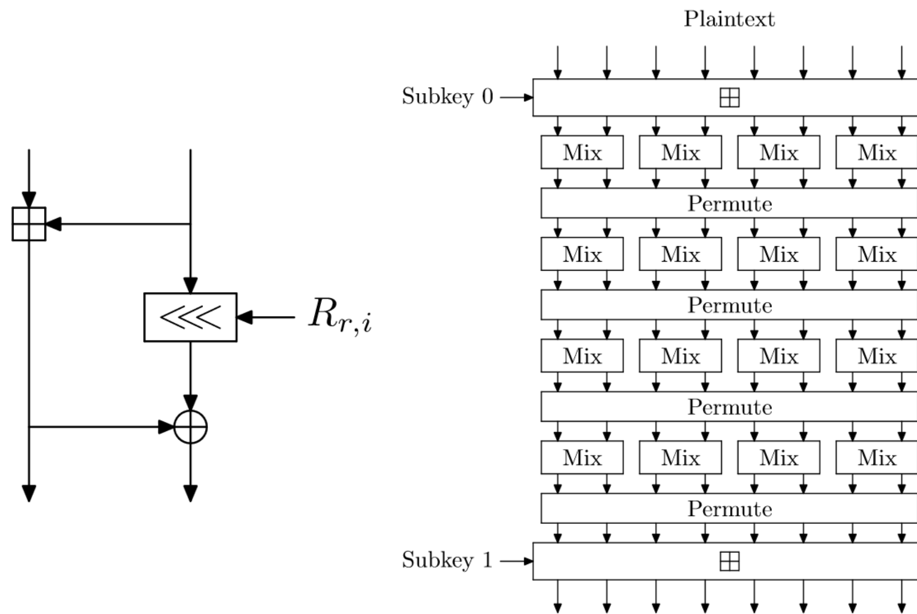
- *Treefish*. Vylepšiteľná bloková šifra[2], jadro funkcie Skein.
- *Unique Block Iteration* (UBI). UBI je spôsob reťazenia ktorý využíva Treefish na vytvorenie kompresnej funkcie. Myšlienka UBI pochádza z tzv. Matyas-Meyer-Oseas módu blokovej šifry [16].
- *Systém voliteľných argumentov*, vďaka ktorému môže Skein podporovať rôzne voliteľné rozšírenia svojej funkcionality.

Bloková šifra Treefish používa 256, 512, alebo 1024 bitov dlhý blok dát, kľúč je rovnako dlhý ako blok a tzv. *tweak* je dlhý 128 bitov pre všetky tri dĺžky bloku. Návrh šifry sa drží myšlienky, že veľa jednoduchých kôl je bezpečnejších ako menej zložitých.

Na obrázku 3.4 vľavo je znázornená jednoduchá, nelineárna, zmiešavacia funkcia *Mix*. Jej vstupom aj výstupom sú dve 64 bitové slová a skladá sa z sčítania, rotácie o konštantu a operácie XOR.

Na 3.4 obrázku vpravo sú znázornené štyri zo 72 kôl blokovej šifry Treefish-512. Jedno kolo sa skladá zo štyroch operácií *Mix* a jednej permutácie ôsmich 64 bitových slov. Podkľúč sa k stavu pričíta každé štyri kolá. Jednotlivé podkľúče sa počítajú z slov kľúča, čítača a dvoch tweak slov. Treefish-256 a Treefish-1024 sa líšia v dĺžke

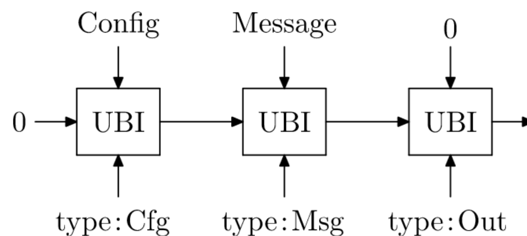
2. Tweakable block cipher. `http://en.wikipedia.org/wiki/Block_cipher#Tweakable_block_ciphers`



Obr. 3.4: Komponenty blokovej šifry Treefish, vľavo funkcia MIX, vpravo štyri kolá šifry Treefish-512, prevzaté z [7].

spracovávaného slova, šírke permutácií, počte použitých MIX funkcií a počte kôl.

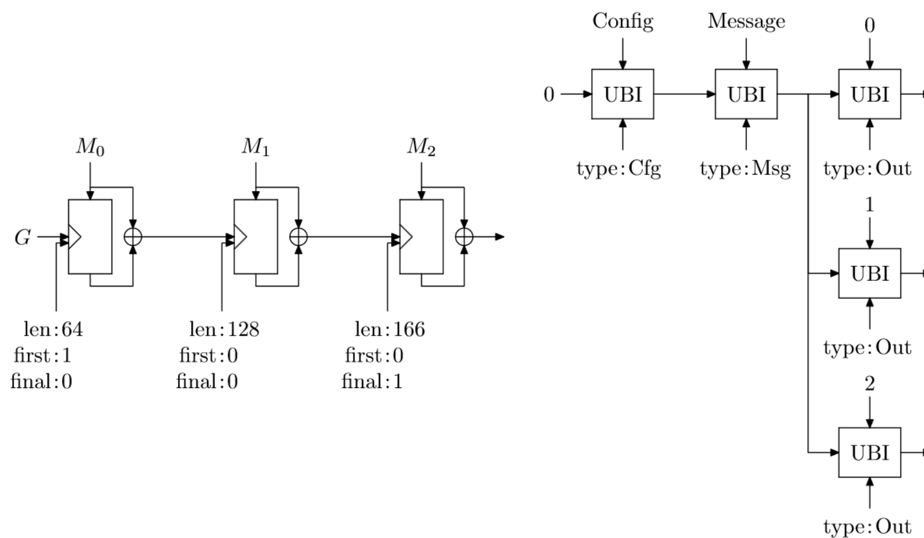
Hašovanie pomocou funkcie Skein je zreťazenie niekoľkých UBI blokov. Obrázok 3.5 znázorňuje tri povinné vyvolania UBI reťazenia. Spracovanie konfigurácie, dát a generovanie výstupu.



Obr. 3.5: Hašovanie pomocou funkcie Skein, prevzaté z [7].

Hašované dáta spracováva Skein po blokoch pomocou UBI. kombinuje vstupné bloky s dĺžkou doposiaľ spracovaného vstupu. Na obrázku 3.6 vľavo je znázornené hašovanie správy. Ako tweak je po-

užitý počet spracovaných bajtov a dva príznaky prvého a posledného spracovávaného bloku. Tweak takisto kóduje typ spracovávaných dát, čo nie je znázornené na obrázku, ale bude vysvetlené neskôr. Z obrázka 3.6 je tiež vidieť, že tweak je pre každý spracovávaný blok hašovaných dát iný, preto je každý z týchto blokov spracovávaný jedinečnou variantou kompresnej funkcie.



Obr. 3.6: Skien, vľavo reťazenie UBI pri spracovaní správy, vpravo generovanie hašu, prevzaté z [7].

Skein umožňuje použiť aj voliteľné volania UBI. Každé z nich dostáva ako vstup aj typ – tweak spracovávaných dát a Skein ich postupne spracováva. Bolo navrhnutých niekoľko takýchto tweakov:

- Kľúč. Využiť sa dá pri počítaní autentizačných kódov správ, alebo ak Skein používame ako funkciu na odvodenie kľúča.
- Konfigurácia. Povinný 32 bajtový reťazec zahrňujúci hlavne dĺžku výstupu.
- Personalizácia. Pre vytvorenie rôznych funkcií na odlišné použitia.

- Verejný kľúč. Vhodný na použitie spolu s podpisovaním správy. Jedna správa potom pre rôzne verejné kľúče vyprodukuje rôzne haše.
- Identifikátor na odvodenie kľúča. Môžme použiť hlavný kľúč a na odvodenie iných šifrovacích kľúčov.
- Nonce. Pri použití funkcie Skein ako prúdovej šifry.
- Správa. Dáta ktoré majú byť zahašované (až $2^96 - 1$ bajtov).
- Výstup. Ukončenie hašovania. Môže sa opakovať, výsledný haš môže mať až 2^64 bitov.

Tento zoznam by mohol byť jednoducho rozšírený o ďalšie voliteľné argumenty. Rôzne implementácie si môžu vybrať, ktoré z nich budú používať, a ktoré nie.

Analýza

Transformácia funkcie Skien na vyhľadávacie tabuľky je vďaka jej jednoduchému a modulárnemu návrhu triviálna. Jediné čo potrebujeme, je transformovať operáciu MIX.

Kvôli veľkosti vstupu z nej nemôžme spraviť jednú tabuľku, preto ju transformujeme na sieť vyhľadávacích tabuliek.

Vstupom funkcie MIX sú dve 64 bitové slová, ktoré sa sčítajú a skombinujú operáciou XOR. Tieto dve operácie na sieť tabuliek môžme transformovať, na všetky sčitavacie a XOR tabuľky funkcie MIX budeme potrebovať približne 2 MB.

Ako vstup pre operáciu XOR by sme podľa rotácií druhého vstupného slova museli správne vyberať bity, ktoré v konkrétnej tabuľke použijeme.

Ďalšie tabuľky sú potrebné na pridávanie podkľúčov k stavu funkcie Treefish – 760 kB pre jednú Treefish-512.

Matyas-Meyer-Oseas mód blokovej šifry na konci spracovania bloku kombinuje operáciou XOR výsledok šifry so správou – 16kB pre 512 bitový blok.

Sieť spracúvajúca jeden 512 bitový blok správy bude potrebovať približne 2,7 MB[3].

3. 1024 bitová verzia by potrebovala približne dvakrát toľko.

Nevýhodou funkcie Skein, je že na vytvorenie hašu púšťa ďalšiu iteráciu šifry Treefish. Čo znamená pridanie ďalšej 3,3 megabajtovej siete vyhľadávacích tabuliek.

3.5 Keccak

Popis funkcie

Funkcia Keccak [2] je hašovacia funkcia špongiového typu. Preto môže vytvárať haše variabilnej dĺžky. Jej vnútorný stav tvorí 25 slov dĺžky 2^l, pre $0 \leq l \leq$. Tie sú usporiadané do matice typu 5×5. Počet bitov stavu (*b*) funkcie Keccak je dohromady $b = 25 \times 2^l$.

Správa je *absorbovaná* po r-bitových blokoch a výstup je počas *stláčania* sa haš tiež vytvára po r-bitových častiach.

Kapacita funkcie Keccak $c = b - r$, uvádza aká časť stavu nebude nikdy priamo ovplyvnená vstupom a vrátená na výstup. Ňou môžme ovplyvniť úroveň bezpečnosti – kompromis medzi rýchlosťou spracovávania a kapacitou.

Spracovanie jedného vstupného bloku prebehne v $12 + 2l$ kolách. V jednom kole sa stav a vstupný blok zamieša operáciou XOR a funkciou kola R, ktorá sa skladá z niekoľkých operácií:

- θ modifikuje každý stĺpec (5 bitov) vnútorného stavu nasledovne. Ku každému bitu stĺpca pridá operáciou XOR súčet parít dvoch susedných stĺpcov,
- ρ je bitová rotácia každého z slova vnútorného stavu,
- π je zafixovaná permutácia dvadsiatich piatich slov vnútorného stavu,
- χ je jediná nelineárna transformácia v R, ktorou sa modifikuje každý riadok (päť bitov) stavu pomocou S-boxu,
- ι ku každému slovu stavu pridá operáciou XOR konštantu, vytvorenú pomocou lineárneho registra zo spätnou väzbou[4]. Jej cieľom je rozbiť symetriu ostatných operácií.

4. LFSR - `http://en.wikipedia.org/wiki/Linear_feedback_shift_register`

Spomínané operácie sú aplikované na stav v tomto poradí:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

Výsledným hašom sú prvé bity stavu. Ak je potrebným viacero *stlačení*, vykoná sa medzi nimi ďalší blok permutácií.

Východzie hodnoty pre parametre funkcie Keccak sú $l = 6$, $r = 1600 - c$ a $c = 576$.

Analýza

Dobrou návrhovou myšlienkou funkcie Keccak je, že sa do veľkej miery spolieha na permutácie, ktoré nás vo white-box implementáciách skoro nič nestoja. Pre transformáciu na tabuľky je vhodné, že pracuje na úrovní bitov, nie bajtov a používa pomerne malé slová (5 bitov).

Funkcia Keccak má najväčší vnútorný stav spomedzi všetkých hašovacích funkcií.

Operácie χ, ι, a π by sa do white-box implementácie dali transformovať jednoducho.

Rotácia ρ síce nerotuje slova o rovnakú vzdialenosť, ale funkcia R pracuje s jednotlivými bitmi stavu, preto táto rotácia nebráni transformácií na tabuľky.

Problematická je operácia θ. Jej tabuľka by musela byť typu 11×1 (pre každý stĺpec 5 tabuliek). Pri množstve použití tejto operácie by tabuľky dohromady potrebovali približne 9,4 MB na spracovanie jedného bloku.

Práve množstvo pamäte potrebnej je dôvodom, prečo hašovacia funkcia Keccak nie je vhodná na white-box implementáciu.

3.6 Grøstl

Popis funkcie

Funkcia Grøstl [8] je hašovacia funkcia typu Merkle-Damgård. Dokáže vytvárať haše rôznej dĺžky, od 8 po 512 bitov v osembitových krokoch. Vnútorný stav má veľkosť 512 bitov (pre haš menší ako 256 bitov) alebo 1024 bitov (pre haš väčší ako 256 bitov).

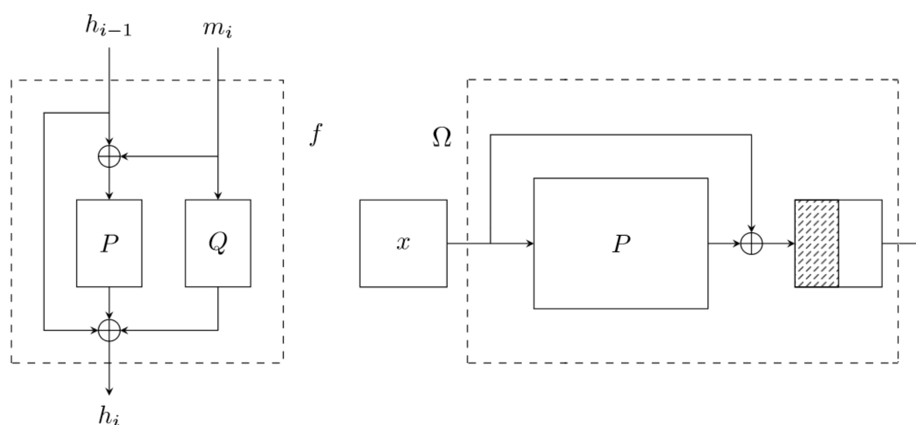
Podľa veľkosti vnútorného stavu spracováva kompresná funkcia f správu po 512 resp. 1024 bitových blokoch. Spracovanie jedného bloku je definované ako:

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h,$$

a je znázornená na obrázku 3.7 vľavo. Finalizácia Ω je definovaná ako:

$$\Omega(x) = tnunc_n(P(x) \oplus x),$$

kde $trunc_n()$ vracia posledných n bitov vstupu. Na obrázku 3.7 je finalizácia znázornená vpravo.



Obr. 3.7: Grøst - spracovanie bloku správy, prevzaté z [8]

Základom kompresnej funkcie sú permutácie P a Q. Sú to podobné permutácie, líšia sa len v hodnotách ktoré v jednotlivých krokoch používajú. Podľa veľkosti vnútorného stavu ho obe transformujú v desiatich resp štrnástich kolách funkciou R.

Funkcia kola, R, pozostáva zo štyroch transformácií založených za operáciách šifry AES.

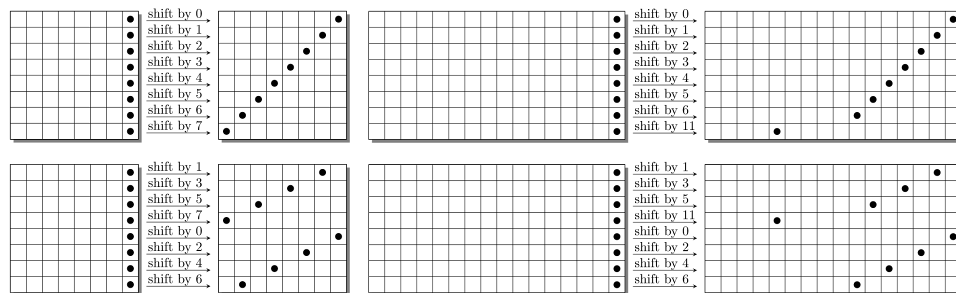
$$R = MixBytes \circ ShiftBytes \circ SubBytes \circ AddRoundConstant$$

Stav spracovávajú ako maticu typu 8×8 resp. 8×16 bajtov.

AddRoundConstant je jednoduché pridanie konštanty ku každému bajtu stavu. P a Q používajú rôzne konštanty pre rôzne veľkosti vnútorného stavu.

SubBytes aplikuje transformáciu pomocou S-boxu na každý bajt stavu. Použitý S-box je rovnaký ako v šifre AES.

ShiftBytes rotuje riadky matice o rôzne konštanty, podľa veľkosti stavu. Varianty jednotlivých posunov sú znázornené na obrázku 3.8.



Obr. 3.8: Grøst - ShiftBytes, permutácia P hore, Q dole, 512 bitový stav vľavo, 1024 bitový stav vpravo, prevzaté z [8]

MixBytes je, podobne ako v šifre AES, vynásobenie každého stĺpca matice stavu cyklickou (z anglického *circulant*) maticou $B = circ(02, 02, 03, 04, 05, 03, 05, 07)$

Analýza

Operácia MixBytes by sa mala dať rozdeliť na menšie operácie podobne ako vo white-box implementácií šifry AES by vzniklo 8 resp. 16 tabuliek typu 8×64 (2 kB) a ich výsledky by sa skombinovali pomocou ďalších $16 \cdot 7$ resp. $16 \cdot 15$ tabuliek XOR.

Ostatné operácie, používané kompresná funkcia f (XOR, transformáciu pomocou S-boxov, rotácie stavu o rôzny počet bajtov) sa dajú veľmi ľahko transformovať na vyhľadávacie tabuľky. Tie by sme spojili s tabuľkami operácie MixBytes.

Na spracovanie jedného 1024 bitového bloku správy by Grøstl potreboval približne 13,5 MB.

Problémom je, že útoky na white-box implementáciu šifry AES by sa zrejme dali prispôsobiť aj na white-box transformáciu hašovacej funkcie Grøstl.

3.7 ECHO

Popis funkcie

Funkcia ECHO [4] je hašovacia funkcia typu Merkle-Damgård. Jej kompresná funkcia využíva symetrickú šifru AES.

Správu spracováva po 1536 bitových blokoch, ak má mať výsledný haš dĺžku od 128 po 256 bitov resp. po 1024 bitových blokoch, ak má mať výsledný haš dĺžku od 256 po 512 bitov.

Vnútorný stav hašovacej funkcie tvorí matica rozmerov 4×4 tvorená z 128 bitových slov. Jednotlivé slová v tejto matici môžeme vnímať ako vnútorné stavy šifry AES.

V ôsmich (pre haš menší ako 256 bitov) resp. desiatich (pre haš väčší ako 256 bitov) kolách, sa stav hašovacej funkcie modifikuje funkciou kola BigRound(). Tá používa na modifikáciu stavu tri jednoduchšie funkcie, vychádzajúce zo šifry AES:

- *BigSubWords(S, Salt, C)* modifikuje každé 128 bitové slovo vnútorného stavu *S* dvomi šifrovaniami pomocou AES: nové slovo stavu je výsledkom $AES(AES(w_i, k1), k2)$, w_i je slovo vnútorného stavu S, kde k_1 je kľúč odvodený z vnútorného čítača C, a ako k_2 sa použije soľ (Salt).

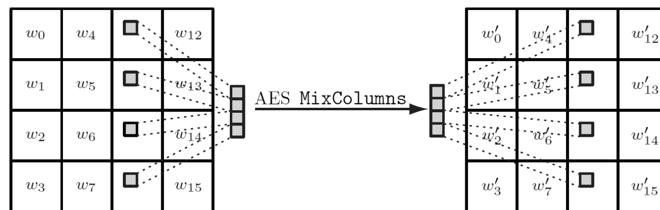
- *BigShiftRows(S)* je, podobne ako u šifry AES, rotácia jednotlivých riadkov matice 4×4 o 0, 1, 2, resp. 3 pozície doľava.

- *BigMixColumns(S)* používa štyri 128 bitové stĺpce ako 64 stĺpcov širokých jeden bajt. Na každý z nich potom aplikuje operáciu MixColumns() zo šifry AES. Táto operácia je znázornená na obrázku 3.9.

Po konci posledného kola sa zo stavu vytvorí reťazová hodnota, ktorá sa použije pri spracovaní nasledujúceho bloku správy. Haš celej správy vzniká skrátením poslednej reťazovej hodnoty.

Analýza

Základným stavebným kameňom, resp. vzorom funkcie ECHO, je nepochybne šifra AES. V tomto prístupe je skrytých niekoľko dobrých



Obr. 3.9: Operácia BigMixColumns() používaná hašovacou funkciou ECHO, prevzaté z [4]

myšlienok pre white-box kontext, zároveň je však masívne používanie šifry AES dôvodom, prečo je funkcia ECHO pre HMAC do nedôveryhodného prostredia prakticky nepoužiteľná.

Výhody funkcie ECHO pre white-box kontext:

- Využitie existujúcich white-box implementácií šifry AES.
- Skoro každé použitie šifry AES prebehne s iným kľúčom. Pre white-box implementáciu to znamená, že časti siete tabuliek zodpovedajúce jednotlivým použitiam šifry budú iné.
- Ak by sme od white-box HMAC-u nepožadovali aby bol kompatibilný s black-box HMAC-om, veľmi jednoducho by sa kľúč z mechanizmu HMAC dal skombinovať s AES kľúčmi, čo by prípadnému útočníkovi výrazne sťažilo analýzu.

Nevýhodou funkcie ECHO a zároveň najväčším dôvodom prečo je pre white-box kontext nepoužiteľná, je počet použití šifry AES. V každom z ôsmych kôl sa šifra AES použije 32 krát. White-box implementácia z [11] ma 770 kB. Spolu s operáciami MixColumns() z poslednej časti funkcie kola, by sieť tabuliek na spracovanie jedného bloku správy pomocou funkcie ECHO mala takmer 200 MB.

3.8 Shabal

Popis funkcie

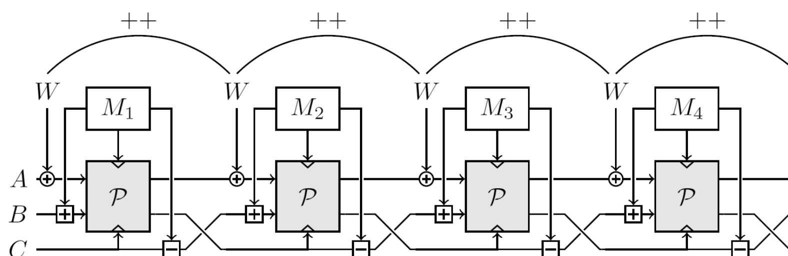
Shabal [5] je založený na príncípe Merkle-Damgård. Vnútorný stav o veľkosti 1024 + 32*r* bitov[5] je rozdelený na 3 rôzne časti (A,B,C), kde

5. odporúčaná hodnota *r* je 12

A má 32*r* bajtov a B, C sú 512 bitové slová.

Správa je spracovávaná po 512 bitových blokoch, pomocou čítača, modulárneho sčítania a odčítania, operácie XOR a permutácie s použitím kľúča. Postup hašovania je znázornený na obrázku 3.10.

V troch kolách finalizácie sa potom použije posledný vkladaný blok správy a čítač sa nebude meniť. Shabal umožňuje vytvárať haše dlhé 192, 224, 256, 384 alebo 512 bitov.



Obr. 3.10: Shabal - spracovanie správy, prevzaté z [5]

Analýza

Funkcia Shabal nemôže byť transformovaná na sieť vyhľadávacích tabuliek z dvoch hlavných dôvodov.

Vo white-box implementáciách symetrických šifier sa permutácia prejaví na prepojení tabuliek. Keby sme to isté spravili s permutáciou, ktorú používa Shabal ako jeden zo základných prvkov, hrozilo by, že s nej útočník získa informácie o vnútornom stave. Výsledok použitej permutácie totiž závisí na vstupe, ktorý dostane, a tým je stav hašovacej funkcie.

Permutáciu by sme mohli transformovať na tabuľku, tomu však bráni dĺžka jej vstupu, preto by transformovanie na tabuľku bolo nepraktické.

Druhou prekážkou je modulárne sčítanie a odčítanie dvoch 512 bitových slov. Tieto operácie nemôžme previesť v jednej tabuľke kvôli jej potencionálnej veľkosti. Viac v časti 3.1.

3.9 Fugue

Popis funkcie

Fugue [9] svojím správaním pripomína špongiový návrh hašovacej funkcie. Vstupnú správu spracováva po štvorbajtových blokoch. Výsledný haš môže mať dĺžky 224, 256, 384 alebo 512 bitov. V tejto časti bude pre jednoduchosť popísaná len varianta s 256 bitovým výstupom. Ostatné verzie sa líšia len inými hodnotami parametrov.

Vnútorný stav funkcie tvorí matica 30-tich štvorbajtových stĺpcov. S každým spracovávaným štvorbajtovým slovom sa táto matica modifikuje funkciou kola. Po poslednom spracovanom bloku prebehne finalizácia a výstupom funkcie bude 8 pevne daných stĺpcov matice.

Transformácia stavu v jednom kole používa nasledujúce kroky[6]:

- *TIX(I)* sa skladá z nasledujúcich krokov. $S_4+=S_0$; $S_0=I$; $S_{14}+=S_0$; $S_{20}+=S_0$; $S_8+=S_1$, kde S_i je i-tý stĺpec stavu a I je spracovávané slovo správy.

- *ROR3* rotuje stav funkcie doprava o tri stĺpce.

- *CMIX* sa skladá z nasledujúcich krokov. $S_0+=S_4$; $S_1+=S_5$; $S_2+=S_6$; $S_{15}+=S_4$; $S_{16}+=S_5$; $S_{17}+=S_6$;, kde S_i je i-tý stĺpec stavu.

- *SMIX* najprv modifikuje všetky bajty prvých štyroch stĺpov pomocou S-boxu. Tieto stĺpce modifikuje operáciou Super-Mix(), ktorá sa skladá z násobenia a sčítania matíc a následnej rotácie riadkov výsledku.

Transformácia kola vykoná sériu týchto operácií:

$$TIX(I);\ 2\times(ROR3;CMIX;SMIX).$$

.

Finalizácia používa podobné operácie na záverečnú modifikáciu stavu pred vytvorením hašu správy.

6. sčítanie dvoch štvorbajtových vektorov je to isté čo XOR dvoch 32 bitových slov

Analýza

Fugue by vo white-box kontexte mal jeden hlavný problém. V transformácií *SMIX* sa dvakrát používa násobenie matíc. V čase výberu skúmania hašovacích funkcií sa mi nepodarilo nájsť algoritmus násobenia matíc vhodný na transformáciu do siete vyhľadávacích tabuliek. Naivná transformácia by potrebovala 7 MB na jedno násobenie dvojice matíc. Veľkosť siete by veľmi rýchlo rástla s každým spracovávaným štvorbajtovým slovom.

Toto násobenie dvoch matíc sa dá nahradiť násobením vektoru a matice, stále by však výsledná tabuľka musela mať 16 bajtov vstupu a výstupu a tiež by potrebovala veľa pamäte.

Operácia XOR a S-box sa dá transformovať na tabuľku jednoducho, tie by sa ale bez spojenia z väčšími tabuľkami dali ľahko invertovať.

Permutácie a rotácie stavu by sa prejavili na prepojení tabuliek v sieti.

3.10 Diskusia

Predstavili a analyzovali sme si niekoľko hašovacích funkcií. Rôzne prístupy k hašovaniu sa na počte tabuliek v prípadnej white-box implementácií prejavia odlišne. V analýzach jednotlivých funkcií sme videli, ako sú jednotlivé hašovacie funkcie vhodné na white-box implementáciu.

Veľkosť vnútorného stavu, počet kôl a ostatné údaje z tabuľky 3.1 nám o vhodnosti danej funkcie na implementáciu do nedôveryhodného prostredia povedia málo, preto je potrebné hašovacie funkcie analyzovať podrobnejšie.

Niekoľko operácií, ktoré sa v hašovacích funkciách používajú (sčítanie, odčítanie, násobenie s maticami), by po transformovaní na vyhľadávacie tabuľky potrebovali príliš veľa miesta. Najväčšou prekážkou bola veľkosť slov, s ktorými tieto operácie pracovali. V rámci práce sa nepodarilo nájsť iné algoritmy na ich výpočet, ktoré by veľkosť vyhľadávacej tabuľky by zmenšili. Záleží teda na použitých operáciách, ale veľkosť používaných slov v hašovacej funkcií môže napovedať viac o jej vhodnosti na white-box implementáciu.

Na druhej strane logické operácie, S-boxy, permutácie, posuny, ro-

tácie či rôzne transformácie bajtov sa na vyhľadávacie tabuľky transformujú veľmi jednoducho. Hašovacia funkcia ideálna na white-box implementáciu by mala používať hlavne tieto operácie.

Hašovať dáta pomocou existujúcej white-box implementácie symetrickej šifry je tiež zaujímavou, ale v súčasnosti nepraktickou, myšlienkou funkcií Grøstl a ECHO.

Je vhodné ak aj finalizácia kompresnej funkcie je nejakou transformáciou časti vnútorného stavu, nie len jednoduchým orezaním. Tabuľky finalizácie pomôžu skryť stav hašovacej funkcie tesne pred skončením hašovania.

Z funkcií analyzovaných v tejto kapitole sa najvhodnejšia na white-box implementáciu zdá byť funkcia JH. V nasledujúcej kapitole je pomocou nej navrhnutá white-box implementácia machanizmu HMAC.

Funkcie Skein a BLAKE používajú modulárne sčítanie, V ktorom môže spôsobiť problém bit prenosu (vysvetlené v časť 3.1) preto ich použite označíme ako možné, ale nevhodné.

Funkcie Grøstl a Keccak by potrebovali viac pamäte, ich použitie je teda nepraktické. White-box implementácia funkcií ECHO a Fugue je veľmi nepraktická, tiež kvôli pamäťovým nárokom.

Implementácia pomocou vyhľadávacích tabuliek funkcie Shabal je nemožná, kvôli permutáciám závislých na vstupe.

4 White-box HMAC

Mechanizmus HMAC popísaný v predchádzajúcej kapitole (v časti 2.3) používa tajný kľúč. V nedôveryhodnom prostredí budeme od podobného mechanizmu požadovať, aby sa tento kľúč na nedôveryhodnom zariadení neobjavil v otvorenej podobe, a aby výsledný autentizačný kód bol kompatibilný z black-box HMAC-om.

Utajenie kľúča, napríklad hašovaním, samo o sebe neposkytuje oveľa väčšiu bezpečnosť. Potencionálny útočník by síce nepoznal kľúč, ale aj tak by mohol vytvárať resp. overovať autentizačné kódy akýchkoľvek dát.

V časti 4.1 si predstavíme naivný výpočet autentizačného kódu použitím jednoduchého protokolu. Nebude využívať techniky white-box v zmysle [11] a bez ďalších zmien, nezabráni útočníkovi získať kľúč. Bude nám slúžiť ako ukážka požadovaných resp. nežiadúcich vlastnosti mechanizmu HMAC v nedôveryhodnom prostredí.

Hašovacia funkcia transformovaná na sieť vyhľadávacích tabuliek nám umožní počítať autentizačné kódy takým spôsobom, aby získanie použitého kľúča bolo výrazne zložitejšie. Rozbor a návrh takejto modifikácie mechanizmu HMAC je popísaný v časti 4.2

Ďalšia vlastnosť, ktorú by v ideálnom prípade mal mať HMAC v nedôveryhodnom prostredí, sa podobá na jednu z vlastností white-box implementácií symetrických šifier. Symetrický kryptosystém obsahuje dva algoritmy. Jeden na šifrovanie správ, druhý na ich dešifrovanie. Jeho white-box implementácia potom modifikuje jeden z nich a ten poskytne nedôveryhodnému zariadeniu.

Podľa toho, ktorý z algoritmov kryptosystému bol modifikovaný, môže potencionálny útočník vykonávať len tú z operácií, na ktorú bol pôvodný algoritmus určený. Pri návrhu symetrického white-box kryptosystému dostaneme túto vlastnosť takpovediac zadarmo.

Od white-box implementácie mechanizmu HMAC by sme mohli očakávať aby umožnila nedôveryhodnej strane autentizačné kódy len vytvárať alebo len overovať. Dosiahnuť toto správanie s white-box implementáciou HMAC-u nebude také triviálne ako so symetrickým kryptosytémom.

Problém spočíva v jednosmernosti hašovacích funkcií resp. mechanizmu HMAC. Pri overovaní správnosti autentizačného kódu sa

používa rovnaký algoritmus ako pri jeho vytváraní. Ak by sme teda nejakú white-box implementáciu algoritmu, na výpočet HMAC-u, bez ďalších zmien poskytli nedôveryhodnej entite, umožnilo by jej to autentizačné kódy aj vytvárať aj overovať. Konkrétnejšie je odstránenie symetrie popísané v časti 4.2.2.

White-box implementáciu mechanizmu HMAC, s obmedzením len na jeden smer výpočtu (vytváranie alebo overovanie), by sme v praxi mohli využiť napríklad ako náhradu tradičného (black-box) digitálneho podpisu.

4.1 Návrh white-box HMAC protokolu

V tejto časti bude popísaný jednoduchý komunikačný protokol medzi dvoma entitami.

Prvú z nich budeme považovať za dôveryhodnú, nazveme ju *DE* (dôveryhodná entita). Druhej entite tohto protokolu nebudeme dôverovať. Označíme ju ako *NE* (nedôveryhodná entita). Toto nedôveryhodné prostredie, v ktorom sa komunikácia medzi entitami bude odohrávať je podrobnejšie popísané v časti 2.1.

V praxi by v úlohe DE-y mohol vystupovať napríklad server, poskytujúci nejakú službu, plne pod našou kontrolou. V úlohe NE-y si môžeme predstaviť počítače alebo chytré telefóny užívateľov, resp. potencionálnych útočníkov.

Navrhovaný protokol nám umožní vytvárať autentizačné kódy správ v nedôveryhodnom prostredí. Najprv si ho, pre jednoduchosť, predstavíme v základnej verzii. Následne uvedieme niekoľko nedostatkov rozšírení na zvýšenie bezpečnosti protokolu. Na záver sa zameriame na vlastnosti protokolu, vhodné i nevhodné do white-box prostredia.

4.1.1 Predpoklady a prostredie protokolu

Pri popise protokolu nebudeme uvažovať o tretej strane, ktorá má prístup ku kanálu medzi DE-ou a NE-ou a odpočúva ich komunikáciu. Hlavný dôvod tohto predpokladu je, že white-box útočník je oveľa silnejší, ako jeho black-box náprotivok. Zabezpečovať dôveryhodnosť komunikácie je zbytočné, keďže útočník, ktorý má pod

kontrolou výpočet na NE-e, môže ľahko prijaté dáta komukoľvek sprístupniť.

Cieľom navrhovaného protokolu je umožniť NE-e, aby poskytla DE dôkaz o integrite a autenticite dát bez znalosti akéhokoľvek tajného kľúča.

O útočníkovi predpokladáme, že má výpočet na strane NE plne pod kontrolou (white-box útočník z časti 2.1). Jeho cieľom bude odhalenie tajného kľúča.

4.1.2 WB HMAC Protokol

Výpočet autentizačného kódu pomocou mechanizmu HMAC pozostáva z dvoch hašovaní. Prvý blok oboch hašovaní je známy po zarovnaní kľúča na dĺžku bloku použitej hašovacej funkcie.

Myšlienka navrhovaného protokolu je jednoduchá. Spočíva vo presunutí spracovávania kľúča do dôveryhodného prostredia. Hašovanie dát sa potom môže vykonať aj v prostredí, ktoré má pod kontrolou útočník. Zdá sa, že NE nespracováva žiadne citlivé údaje, preto nebude na hašovanie využívať white-box implementaciu hašovacej funkcie.

Rozdelenie spracovávanie kľúča a spracovávania dát vyzerá na prvý pohľad lákavo, v časti 4.1.3 si ale ukážeme, že zabrániť útočníkovi aby získal kľúč, a zároveň mu umožniť autentizačné kódy vytvárať resp. overovať nebude až také jednoduché.

Základná verzia navrhovaného white-box protokol pre výpočet autentizačného kódu dát sa skladá z týchto krokov:

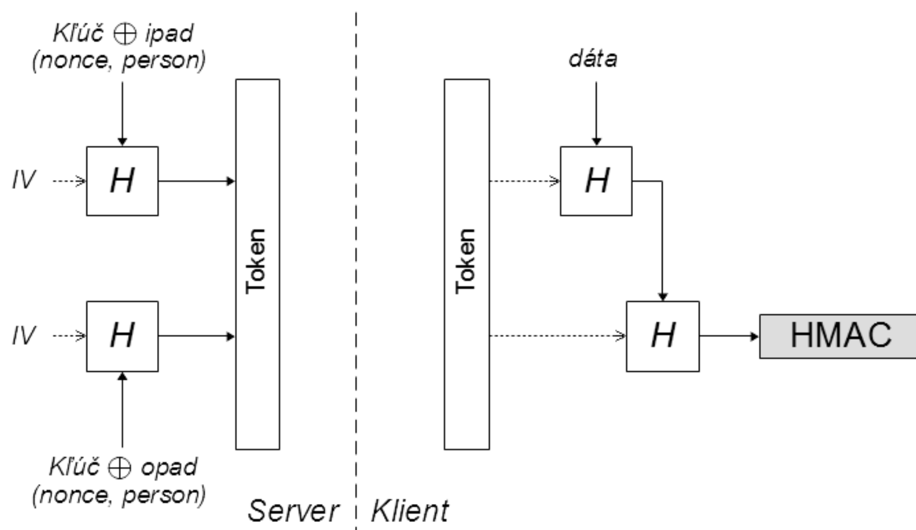
- DE vytvorí token T a odošle ho NE,
- Prijatý token T použije NE na výpočet HMAC-u ľubovoľných dát, pripojí k nemu dáta a výsledok odošle serveru na overenie,
- DE overí správnosť prijatého HMAC-u.

Pri vytváraní tohto tokenu T server využije tajný kľúč tak, aby bolo nemožné resp. bolo výpočtovo náročné ho získať späť. Jednosmernosť DE dosiahne použitím hašovacej funkcie. Zarovnanie kľúča pred jeho hašovaním zaistí, že hašovacia funkcia blok skutočne spracuje. Aby bola NE schopná pokračovať v hašovaní dát, musí token T

obsahovať vnútorný stav hašovacej funkcie po skončení spracovávania tajného kľúča.

NE teda od DE dostane vnútorný stav hašovacej funkcie. Týmto stavom si inicializuje svoju inštanciu tej istej hašovacej funkcie a spracuje ňou dáta, ktorých autentizačný kód chce vytvoriť. Po zahašovaní všetkých dát, pošle výsledok spolu s dátami naspäť DE-e na overenie platnosti.

Aby z tohto výpočtu vznikol autentizačný kód kompatibilný s tradičnou black-box variantou HMAC-u, NE by potrebovala vykonať hašovanie dvakrát. Token T by musel obsahovať dva časti: stavy po zahašovaní $(Key \oplus opad)$ a $(Key \oplus ipad)$. Vytváranie autentizačného kódu pomocou tohto protokolu je znázornené na obrázku 4.1.



Obr. 4.1: Vytváranie autentizačného kódu (HMAC) pomocou HMAC protokolu

Overovanie HMACu je pre DE už jednoduché. Z uloženého tokenu T a dát spočíta kontrolný $\overline{HMAC}$ a porovná ho s HMAC-om prijatým od NE.

4.1.3 Analýza protokolu

Ako už bolo spomenuté má niekoľko problémov. V tejto časti si ich postupne popíšeme.

Pred začiatkom komunikácie NE nepozná tajný kľúč, preto bez DE nemôže vytvoriť HMAC žiadnych dát. Môže tak urobiť až po prijatí tokenu T od DE-y. Token T obsahuje vnútorný stav hašovacej funkcie po spracovaní jedného bloku – tajného kľúča.

Všeobecne hašovacie funkcie nezaručujú, že sa jeho invertovaním nebude dať získať pôvodný vstup. Jednosmernosť je zaistená až finalizáciou. Útočník teda invertovaním spracovania jedného bloku hašovacou funkciou dokáže z odhaleného vnútorného stavu získať použitý tajný kľúč.

Akákoľvek modifikácia vnútorného stavu pred prenosom, a príslušná zmena algoritmu na strane klienta, nemá vo white-box prostredí význam. Útočník môže pôvodný vnútorný stav získať sledovaním výpočtu a zachytením hodnoty v momente, keď ju algoritmus na strane klienta vráti naspäť na pôvodný stav.

DE nemá žiadnu kontrolu nad tým, aké dáta NE s tokenom T spracuje. Ak chceme zaistiť, aby NE mohla spracovať iba konkrétne dáta, musí tieto dáta poznať aj DE ešte pred odoslaním tokenu T. DE a NE môžu mať napríklad prístup k rovnakému zdroju dát (čipová karta), a tieto dáta využiť pri kontrole správnosti HMAC-u.

V niektorých situáciách môže byť žiadúce, aby každá iterácia navrhovaného protokolu na rovnakých dátach skončila iným HMAC-om. Výpočet autentizačného kódu môžme upraviť tak, že DE spracuje jeden dodatočný dátový blok, obsahujúci napríklad náhodné číslo alebo časovú známku (alebo ich skombinuje s kľúčom). NE bude potom tokeny používať na výpočet jednorázových[1] HMAC-ov.

V prípade, že chceme protokol použiť v prostredí s viacerými nedôveryhodnými entitami, musí DE vlastniť identifikátory každej z nich. Tie potom použije pri spracovávaní dodatočného dátového bloku pre konkrétnu entitu, podobne ako pri použití náhodných čísiel. Zaistiť unikátnosť tokenov T pre jednotlivé entity je možné aj použitím tzv. soli.

Mohli by sme sa pokúsiť výpočet autentizačného kódu zviazať

1. jednorázovo vytvorených, ale viacnásobne overiteľných

s konkrétnym hardvérom. Táto varianta nebude vo white-box kontexte fungovať z podobného dôvodu ako varianta s modifikovaným stavom funkcie. Útočník môže modifikovať aj vstupné hodnoty algoritmu.

4.2 HMAC ako sieť vyhľadávacích tabuliek

Jedným z hlavných nedostatkov protokolu v predchádzajúcej časti bolo, že útočník mohol invertovať postup vykonaný dôveryhodnou entitou a získať tak pôvodne použitý kľúč. Počas výpočtu autentizačného kódu preto musíme stav hašovacej funkcie v nedôveryhodnom prostredí chrániť, napríklad sieťou vyhľadávacích tabuliek.

Aby sme mohli transformovať výpočet autentizačného kódu pomocou mechanizmu HMAC na sieť vyhľadávacích tabuliek, musíme na sieť vyhľadávacích tabuliek transformovať aj použitú hašovaciu funkciu. S funkciou JH sa o to pokúsime v časti 4.2.1.

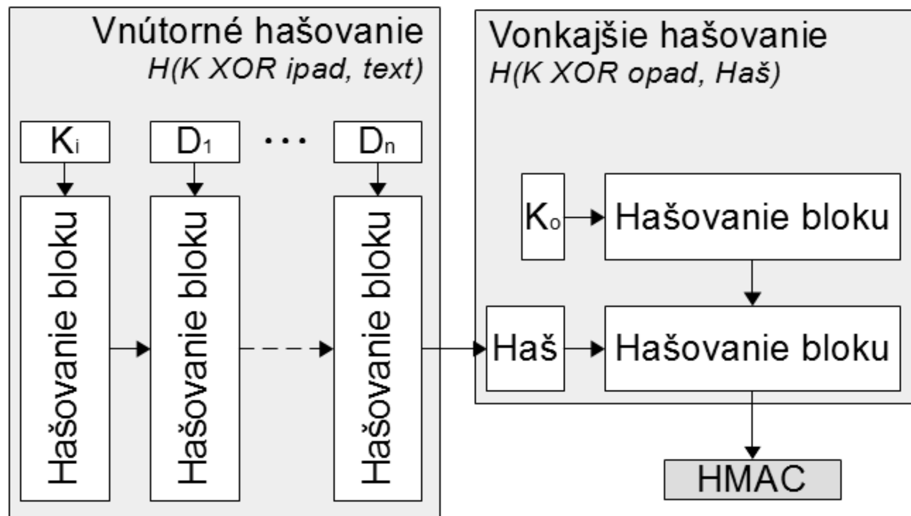
V mechanizme HMAC hašovanie prebieha dvakrát, preto budeme potrebovať dve siete tabuliek na hašovanie. Jednú na výpočet hašu s dátami (označenými ako *text*) *Haš = H(K XOR ipad, text)*, túto sieť nazveme *vnútorné hašovanie*. Druhú sieť, na výpočet *H(K XOR opad, Haš)*, nazveme *vonkajšie hašovanie*. V oboch sieťach je kľúč označený ako K.

Tieto dve siete spojíme do jednej siete na výpočet HMAC-u následovne. Výsledok *vnútorného hašovania* použijeme ako vstup *Haš* pre *vonkajšie hašovanie*, tak ako je HMAC navrhnutý. Na obrázku 4.2 je takýto postup znázornený spolu s použitím siete tabuliek na spracovanie jedného bloku dát resp. kľúča.

Kľúč použitý v mechanizme HMAC sa pred samotným hašovaním musí zarovnať na dĺžku vstupného bloku použitej hašovacej funkcie. Hašovanie začne spracovaním tohto bloku. Potom ako prebehne jeden cyklus hašovania bloku so zarovnaným kľúčom, začne hašovanie dát.

Cieľom implementácie mechanizmu HMAC v nedôveryhodnom prostredí je skryť cyklus so spracovávaním kľúča pred potenciálnym útočníkom. Cyklus so spracovávaním kľúča preto transformujeme na sieť vyhľadávacích tabuliek.

Výstup takejto siete ešte nemôžme dať užívateľovi resp. útoční-



Obr. 4.2: HMAC pomocou sietí vyhľadávacích tabuliek.

kovi k dispozícií, aby s ním pokračoval v hašovaní dát. Keďže pozná algoritmus hašovania, mohol by operácie z jedného kola invertovať.

Aby sme stav hašovacej funkcie ochránili, vytvoríme druhú sieť vyhľadávacích tabuliek na spracovanie bloku správy. Táto sieť bude mať oproti prvej sieti dlhší vstup. K bitom stavu po spracovaní predchádzajúceho bloku pridáme ďalších N bitov nasledujúceho bloku hašovaných dát, kde N je dĺžka bloku hašovacej funkcie.

Pre každý blok hašovanej správy vygenerujeme sieť vyhľadávacích tabuliek, každú s inými vstupno-výstupnými kódovaniami. Z tohto plynie jedna nevýhoda tohto prístupu k white-box HMAC-u: pri generovaní musíme poznať dĺžku hašovanej správy.

Medzi spracovávaním jednotlivých blokov dát a medzi tabuľkami v celej sieti bude vnútorný stav chránený pomocou vstupno-výstupných kódovaní.

4.2.1 White-box implementácia JH

Hašovacia funkcia JH je popísaná v časti 3.3. V tejto časti si podrobnejšie predstavíme spôsob, ako ju môžme využiť na implementáciu mechanizmu HMAC do nedôveryhodného prostredia.

Najprv vytvoríme sieť vyhľadávacích tabuliek na spracovanie

bloku dát funkciou JH, potom predstavíme spôsob ako túto sieť upraviť, aby sme ňou mohli vytvárať hašované autentizačné kódy.

Aby sme transformovali funkciu F_8 na sieť vyhľadávacích tabuliek, potrebujeme z každej operácie spraviť vyhľadávaciu tabuľku. Funkcia F_8 a ňou ďalej volané funkcie používajú počas hašovania 4 základné operácie:

- substitúcia pomocou S-boxu,
- lineárna transformácia bajtov,
- permutácia,
- XOR.

S-box, XOR a lineárnu transformáciu zameníme za tabuľky v nasledujúcich častiach. Permutácia sa prejaví na prepojení jednotlivých tabuliek, preto ju nebudeme transformovať na vyhľadávaciu tabuľku.

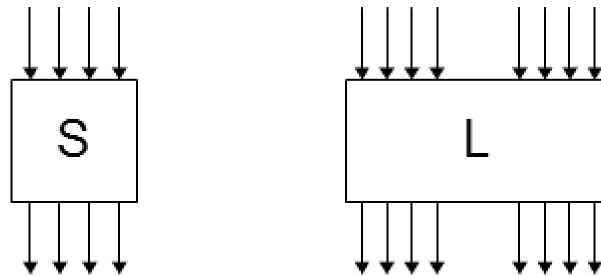
JH využíva dva rôzne S-boxy, oba sú typu 4×4. O tom, ktorý z nich sa v určitom kole, na určitej pozícií použije rozhodujú konštanty kôl. Tie sa generujú pomocou funkcie R_6 so vstupom nastaveným na nulový vektor. S-boxy už v podstate sú vyhľadávacie tabuľky, podľa konštanty kola budeme rozhodovať, ktorý z nich vyberieme, na ktoré miesta v sieti tabuliek.

Typy tabuliek

Transformáciu pomocou S-boxu a príslušnú tabuľku označíme ako S. Pre vstupné štvorbitové slovo A tabuľka S vráti výstupné štvorbitové slovo B, ako výsledok operácie $B = S_0(A)$ resp. $B = S_1(A)$ podľa konštanty kola. Jedna takáto tabuľka bude potrebovať $4 \cdot 2^4$ bitov, resp. 8 bajtov pamäte. Na obrázku 4.3 je tabuľka s S-boxu znázornená vľavo.

Z lineárnej transformácie, použitej vo funkcii JH, vytvoríme tabuľku typu 8×8. Túto tabuľku označíme ako L. Pre vstupné štvorbitové slová A a B vráti tabuľka L výstupné štvorbitové slová C a D ako výsledok operácie $(C, D) = L(A, B)$. Táto tabuľka bude potrebovať $8 \cdot 2^8$ bitov, resp. 256 bajtov pamäte. Na obrázku 4.3 je tabuľka lineárnej transformácie znázornená v strede.

Operáciu XOR akokoľvek dlhých slov, rozdelíme na viacero XOR operácií dvoch štvorbitových slov. Tieto XOR operácie na menších slovách transformujeme na tabuľky, označíme ich ako $\oplus$. Ako vstup budú požadovať dve štvorbitové slová A a B. Jej výstupom bude štvorbitové slovo C, výsledok operácie $C = A \oplus B$. Jedna tabuľka $\oplus$ bude mať $4 \cdot 2^8$ zaberať bitov, resp. 128 bajtov pamäte. Na obrázku 4.3 je tabuľka operácie XOR znázornená vpravo.



Obr. 4.3: Operácie použite v hašovacej funkcii JH transformované do tabuliek. S-box a lineárna transformácia.

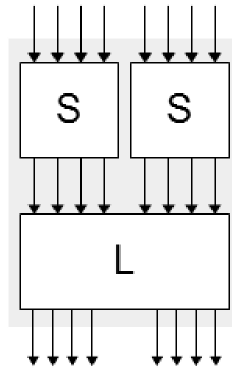
V jednotlivých kolách funkcie E_8, môžme vždy dve transformácie pomocou S-boxov a jednu lineárnu transformáciu zjednotiť do jednej tabuľky, označme ju *RT* (z anglického *round transformation*). Na obrázku 4.4 je táto tabuľka znázornená. Ako vstup bude požadovať dve štvorbitové slová A a B. Výstupom tabuľky budú dve štvorbitové slová C a D ako výsledok výrazu: $(C, D) = L(S(A), S(B))$. Jedna *RT* tabuľka bude potrebovať $8 \cdot 2^8$ bitov resp. 256 bajtov pamäte.

Všimnime si na obrázku 4.5, ako sa vo funkcii JH zoskupujú bity stavu do štvoríc na začiatku funkcie E_8, potom ako sa k prvej polovici stavu pridal blok správy. Pred prvým vykonaním R_8 sa ku každej štvorici bitov pridajú dva bity správy operáciou XOR.

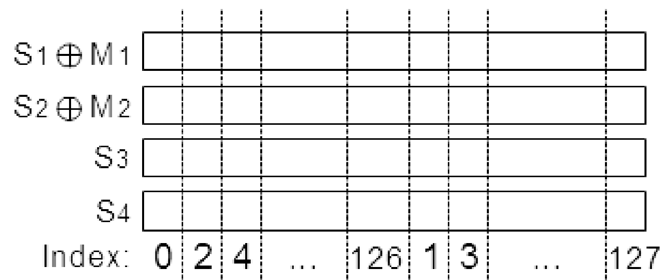
Rovnako môžme bity správy pridávať k stavu v poslednom kole funkcie E_8 po poslednej lineárnej transformácií[2]. Rušenie zoskupenia je podobné zoskupovaniu zo začiatku.

Vo white-box implementácií funkcie F_8 môžme pridávanie správy k stavu pomocou operácie XOR na začiatku resp. konca E_8 spojiť

2. Z ohľadom na permutáciu.



Obr. 4.4: Tabuľka RT



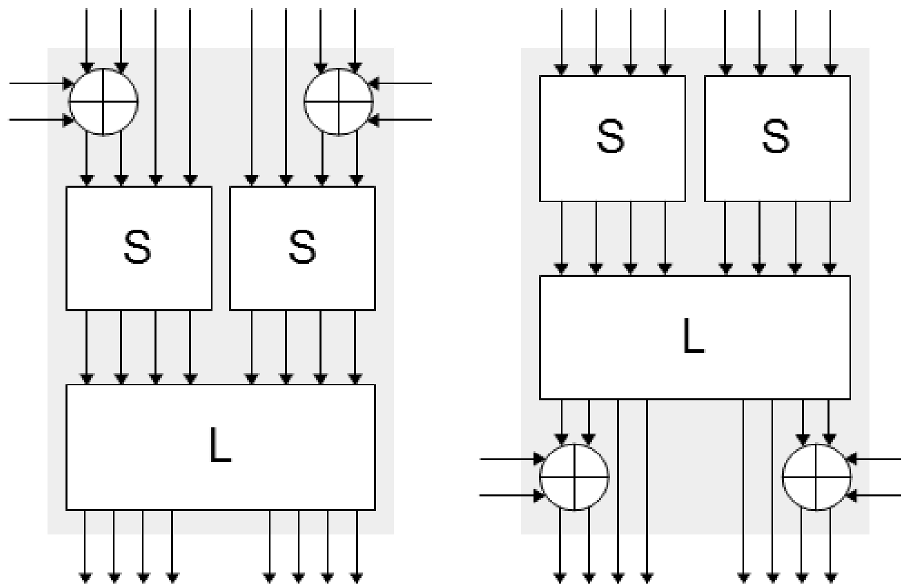
Obr. 4.5: Zoskupovanie štvoríc bitov zo začiatku funkcie E_8 spolu s pridávaním bloku správy z funkcie F_8, index určuje polohu štvorice v stave pred transformáciou pomocou R_8

s vyhľadávacími tabuľkami prvého resp. posledného vykonávania R_8.

Tabuľkám *RT* z prvého kola pridáme 4 bity vstupu. Budú to tie 4 bity správy, ktoré by sa v black-box variante skombinovali s bitmi pôvodného vstupu *RT* tabuľky. Operáciou XOR skombinujeme tieto bity so zodpovedajúcimi bitmi pôvodného vstupu. Ďalej transformácia prebieha tak, ako v tradičnej *RT* tabuľke. Nový typ tabuľky označme ako *RTs*. Takáto tabuľka bude potrebovať $8 \cdot 2^12$ bitov resp. 4 kilobajty pamäte a je znázornená na obrázku 4.6 vľavo.

Podobne rozšírime tabuľky *RT* z posledného kola funkcie E_8. Tentoraz však operáciu XOR použijeme až za lineárnou transformáciou. Bity správy, ktoré týmto tabuľkám dáme naviac na vstup musíme

vyberat' s ohľadom na poslednú permutáciu, ktorá po použití týchto tabuliek ešte bude musiet' prebehnúť. Nový typ tabuľky označme ako *RTe*. Táto *RT* tabuľka bude rovnako ako predchádzajúca potrebovat' $8 \cdot 2^{1}2$ bitov resp. 4 kilobajty pamäte a je znázornená na obrázku 4.6 vpravo.



Obr. 4.6: Tabuľky RT z prvého (vľavo) a posledného (vpravo) kola funkcie JH s pridaným vstupom

Počet tabuliek

Funkcia JH má 1024 bitov dlhý vnútorný stav. Prvé pridávanie bloku správy sa pred prvým kolom vykoná vrámci rozšírených *RTs* tabuliek. Na pridanie 512 bitového bloku správy k vnútornému stavu funkcie budeme potrebovat' 128 *RTs* tabuliek.

V nasledujúcich kolách funkcie E_8 budeme používat' na transformáciu stavu už len tabuľky *RT*. Konkrétne 128 tabuliek v jednom kole.

Na záverečné pridávanie bloku správy použijeme v poslednom kole 128 tabuliek typu *RTe*.

Typ tabuľky	počet × veľkosť	pamäť
RT (prvé kolo)	128×4 kB	512 kB
RT (2. až 41. kolo)	$(40 \times 128) \times 256$ B	1280 kB
RT (posledné kolo)	128×4 kB	512 kB
Spolu	**5376 tabuliek**	**2304 kB pamäte**

Tabuľka 4.1: Počty a veľkosť jednotlivých typov tabuliek vo funkcii F_8 transformovanej do siete vyhľadávacích tabuliek

Spracovanie ľubovoľného bloku dát M funkciou F_8, kompresnou funkciou hašovacej funkcie JH, transformovanou na výpočet pomocou vyhľadávacích tabuliek je znázornené na obrázku 4.9, počty jednotlivých typov tabuliek a potrebná pamäť na celú sieť vyhľadávacích tabuliek je zhrnutá v tabuľke 4.1

Ak by sme takouto sieťou chceli spracovávať dlhú správu, postup by bol nasledovný. Správa sa zarovná na násobok 512 bitov a rozdelí sa na n 512 bitových blokov. Pre každý blok správy vygenerujeme sieť tabuliek na výpočet funkcie F_8. Výsledok siete spracúvajúcej jeden blok správy, bude počiatočným stavom siete, ktorá spracováva nasledujúci blok. Do takto vzniknutej siete ešte pridáme vstupno-výstupné kódovania medzi dvojice tabuliek.

4.2.2 White-box JH a White-box HMAC

Sieť z predchádzajúcej časti spracováva 512 bitov vstupných dát. Ak ju použijeme vo white-box HMAC-u dáta, ktoré bude spracovávať budú začínať blokom kľúča nasledovaným niekoľkými blokmi dát.

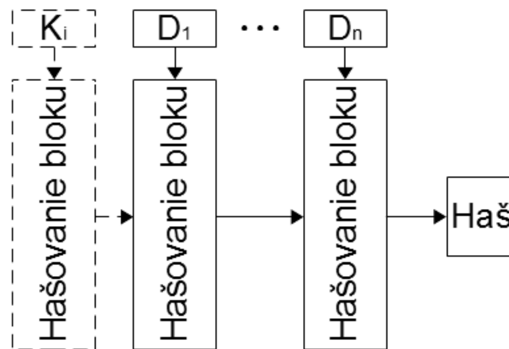
Prvý spracovávaný 512 bitový blok s kľúčom bude teda pri generovaní tabuliek známy a výpočet s ním bude vždy rovnaký.

Vďaka tomu sieť tabuliek mechanizmu HMAC nemusí obsahovať tabuľky, ktoré budú spracovávať kľúč. Do tabuliek spracujúcich prvý blok dát za kľúčom zabudujeme stav hašovacej funkcie po spracovaní bloku kľúča. Zmenšíme tým veľkosť celej siete, ale hlavne zabránime potencionálnemu útočníkovi sledovať výpočet s kľúčom.

Tabuľky z prvého kola na spracovanie prvého bloku dát budú používať ako vstup iba bity správy. Každú *RTs* tabuľku z tohto kola teda upravíme tak, aby pôvodný vstup stavu z predchádzajúceho kola bol konštantný a odpovedal príslušnej štvorici bitov stavu po

spracovaní vopred daného kľúča.

Odstránenie spracovávania bloku kľúča zo siete tabuliek je znázornené na obrázku 4.7.



Obr. 4.7: HMAC s konštantným kľúčom, prerušovane znázornené časti vo výslednej sieti nebudú.

Podobne, *RTe* tabuľky zo siete na spracovanie posledného dátového bloku môžu vracať len bity výsledného hašu. Toto zmenšenie sa ale neprejaví na veľkosti celej siete tak výrazne.

Na kompletný výpočet autentizačného kódu budeme potrebovať dve siete s funkciou JH. Spojíme ich do jednej tak ako bolo znázornené na obrázku 4.2.

Zarovnanie správy

Doteraz sme zarovnanie správy pred hašovaním len párkrát spomenuli, ale jeho následky pre white-box HMAC nie sú zanedbateľné. Každá hašovacia funkcia správu najprv zarovná na násobok veľkosti spracovávaného bloku, okrem iného tak, že sa na koniec správy pridá blok so zakódovanou dĺžkou správy.

V mechanizme HMAC sa dokonca hašuje dvakrát. Výsledkom bude, že do siete vyhľadávacích tabuliek potrebujeme pridať ďalšie tabuľky na spracovanie dvoch blokov dát.

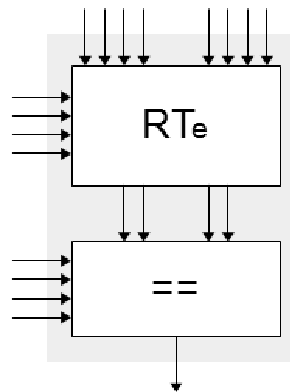
Čiastočným riešením by mohlo byť obmedzenie použitia siete tabuliek iba na správy s vopred daným počtom blokov, čo by nám umožnilo zmenšiť tabuľky, ktoré spracovávajú posledný blok so zarovnaním – to by bolo stále rovnaké.

Obmedzenie symetrie HMAC-u

Na začiatku kapitoly sme popísali vlastnosť white-box symetrických šifier. Podľa toho či na sieť vyhľadávacích tabuliek transformujeme algoritmus šifrovania alebo dešifrovania, bude vytvorená sieť umožňovať iba šifrovanie alebo dešifrovanie.

V prípade mechanizmu HMAC by podobná vlastnosť znamenala, že vytvorenou sieťou tabuliek môže užívateľ dáta len podpisovať, alebo len overovať.

Sieť tabuliek na podpisovanie by vytvárala HMAC a sieť na overovanie by mohla mať vstup rozšírený o autentizačný kód, ktorým sa má výsledok hašovania porovnať. V takto rozšírenej sieti by sa vstupný kód použil ako vstup pre posledné kolo hašovania posledného dátového bloku – *RTe* tabuľky. Tým by sme mohli pridať ďalšie štyri bity vstupu, s ktorými by porovnali svoj pôvodný výstup a vracali by len jeden bit, či porovnávané hodnoty boli rovnaké alebo nie. Takáto *RTe* tabuľka je znázornená na obrázku 4.8



Obr. 4.8: *RTe* tabuľka s pridaným vstupom na porovnávanie správnosti autentizačného kódu.

Problém s týmto prístupom je v tom, že posledné *RTe* tabuľky by mali už 16 bitov vstupu a 128 takýchto tabuliek by potrebovalo 1 MB pamäte.

Zabudovanie tohto vstupu do tabuľky by nemalo význam. Celá sieť by potom kontrolovala, či vytváraný HMAC je zhodný z HMAC-om vytvoreným s konkrétnym kľúčom a konkrétnych dát.

4.3 Diskusia

V tejto kapitole sme najprv navrhli jednoduchý protokol na výpočet autentizačných kódov pomocou hašovacej funkcie. Korekciou jeho nedostatkov sme dostali spôsob ako vytvoriť white-box implementáciu mechanizmu HMAC pomocou všeobecnej hašovacej funkcie.

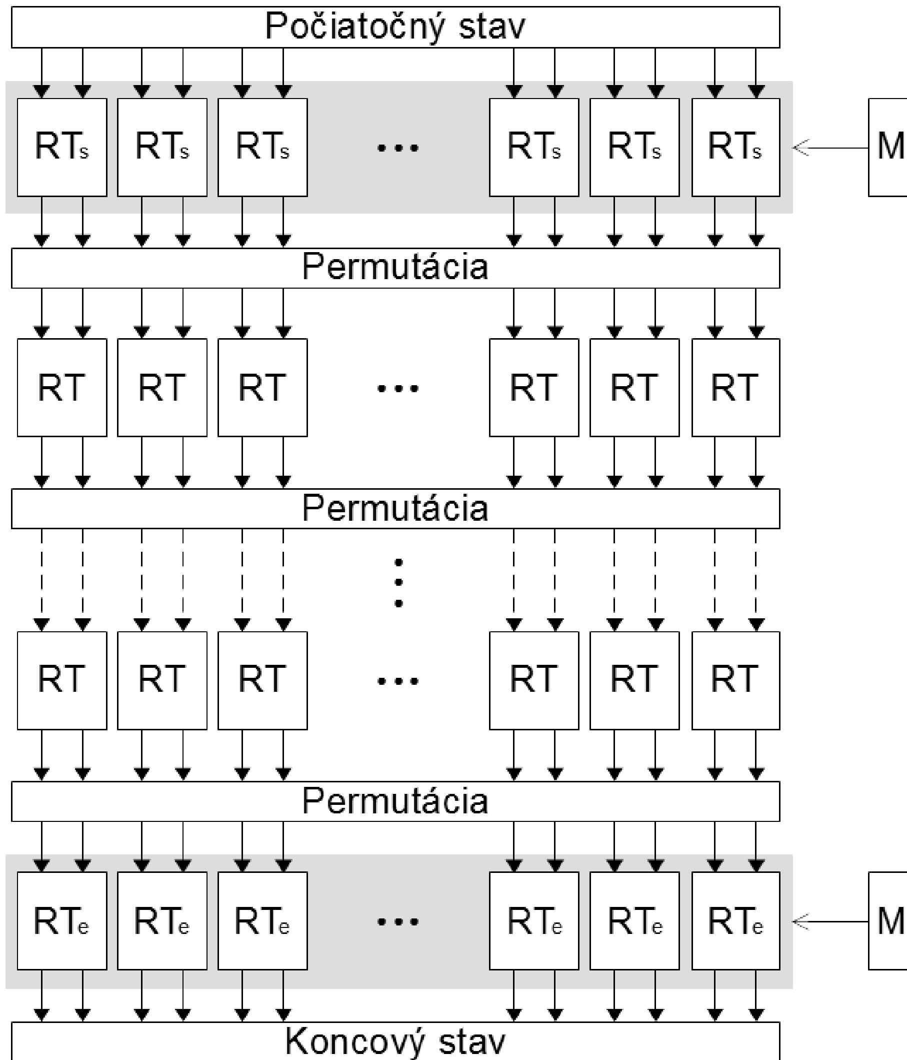
Na základe analýzy z kapitoly 3 sme vybrali hašovaciu funkciu JH ako najvhodnejšiu pre white-box implementáciu a transformovali sme ju na sieť vyhľadávacích tabuliek. Výslednú sieť sme použili na návrh white-box implementácie mechanizmu HMAC s použitím funkcie JH.

Vnútorný stav hašovacej funkcie je v takejto sieti chránený štvorbitovými vstupno-výstupnými kódovaniami. V celej sieti ich je celkom $41 \cdot 256$. Naviac môže byť celá sieť chránená ešte externými vstupno-výstupnými kódovaniami podobne ako white-box implementácia šifry AES v [11]. Pred invertovaním všetkých operácií chránia sieť tabuliek iba tieto kódovania.

Tabuľky v sieti nie sú závislé na kľúči. Pokiaľ trváme na kompatibilite s tradičným black-box HMAC-om, tak ani nemôžu byť, pretože hašovanie nepoužíva kľúč. Takáto závislosť by ale bezpečnosť celej siete zvýšila.

Všimnime si, že pri generovaní siete tabuliek mechanizmu HMAC stále potrebujeme vedieť, koľko blokov dát sa bude pomocou generovanej siete spracovávať. Buď môže byť generátor interaktívny a na požiadanie bude generovať časti siete na spracovanie jednotlivých blokov (so špeciálnou požiadavkou na posledný blok vytvorí časť siete, ktorá bude mať ako výstup výsledný haš), alebo jednoducho obmedzíme dĺžku správ, ktoré budú môcť byť so sieťou spracovávané, vygenerovaním siete s konkrétnym počtom častí.

Počas trvania práce sa nepodarilo nájsť existujúcu implementáciu mechanizmu HMAC do nedôveryhodného prostredia, preto je dosť možné, že návrh v tejto práci je prvý svojho druhu.



Obr. 4.9: F_8 transformovaná do siete vyhľadávacích tabuliek

5 White-box RSA

Veľmi rozšírený asymetrický kryptosystém RSA [16] sa dá použiť na šifrovanie dát alebo na vytváranie digitálnych podpisov. Jeho bezpečnosť je založená na predpoklade, že faktorizovať veľké čísla je ťažká úloha.

Súkromný a verejný kľúč sa v RSA kryprosystéme generuje pomocou veľkých prvočísel (odlišných, štandardne dvoch, označme p, q) nasledovne:

- Spočítame $N = pq$,
- pomocou Eulerovej funkcie $\Phi(N)$,
- zvolíme e nesúdeliteľné s $\Phi(N)$ z intervalu $(1, \Phi(N))$,
- pomocou Euclidovho algoritmu dopočítame d také, že $ed \equiv 1 mod \Phi(N)$ opäť z intervalu $(1, \Phi(N))$.

Verejným kľúčom bude dvojica *(N, e)*, privátnym číslo *d*.

Šifrovanie a dešifrovanie, resp. vytváranie a overovanie digitálnych podpisov správy *m* potom prebieha ako modulárne umocňovanie s príslušným exponentom:

- šifrovanie: $c = m^e \ mod \ N$
- dešifrovanie: $m = c^d \ mod \ N$
- podpis: $sig = m^d \ mod \ N$
- overenie podpisu: $m = sig^e \ mod \ N$

Od white-box implementácie kryptosystému budeme požadovať, aby sa pri rôznych operáciách súkromný ani verejný kľúč (exponent) neobjavil pri výpočte v pamäti v otvorenej podobe.

Každá z operácií kryptosystému RSA je v podstate len jedným umocnením. Pre dosiahnutie zmienenej vlastnosti musíme upraviť toto modulárne umocnenie. V nasledujúcich častiach je popísaných niekoľko prístupov na úpravu RSA.

5.1 Obfuskácia exponentu

Spoločnosť Irdeto má patentovaný [17] spôsob na ochranu exponentov v modulárnom umocňovaní, nie len v kryptosystéme RSA. Ochrana spočíva v skrytí alebo obfuskácií pôvodných hodnôt exponentov dôveryhodnou stranou (serverom). Nedôveryhodný klienti budú upravené exponenty používať rovnako ako v tradičnom black-box RSA, nebudú ale poznať pôvodnú hodnotu exponentu.

Pôvodne exponenty sú chránené ich zväčšením nasledujúcim postupom.

- Nájde sa hodnota λ taká, že $x^{\lambda} \equiv 1 \; mod \; N$ pre všetky x z konečnej algebraickej štruktúry. Takouto hodnotou môže byť $\Phi(N)$.
- λ sa použije na zväčšenie pôvodného exponentu. Nový exponent y vznikne zo starého exponentu a ako $y = a + b\lambda$, kde b je kladné číslo.

Pričítaním hodnoty λ k pôvodnému exponentu ho môžme natiahnuť na ľubovoľnú dĺžku. Exponent $y = a + b\lambda$ má K bitov ak spĺňa[1]:

$$\lceil \frac{2^{K-1}-d}{\Omega} \rceil \leq b \leq \lfloor \frac{2^{K}-1-d}{\Omega} \rfloor$$

Pomocou tejto obfuskácie môžme dosiahnuť aby exponent obsahoval dopredu určený bitový reťazec, napr. identifikátor hardvéru. V prípade white-box prostredia nám táto vlastnosť nepomôže, útočník totiž môže identifikátor použitý na porovnanie s takýmto kľúčom zmeniť, aby program pracoval aj s podvrhnutým kľúčom.

Obfuskovaný exponent y môže byť súčinom viacerých y_i, teda $y = \prod_{i=1}^{m} y_i = a + b\lambda$. Umocnenie obfuskovaným exponentom môžme nahradiť za niekoľko umocnení jednotlivými y_i. Jednotlivé y_i pritom nemusia byť rovnako dlhé.

Správu x spracujeme takýmto exponentom postupne takto $z_1 = x$ a $z_{i+1} = z_i^{y_i}$ potom $z_{m+1} = x^y$. Takéto rozdelenie je podrobnejšie popísané v samotnom patente [17].

Technika podobná vstupno-výstupným kódovaniam sa dá použiť aj pri umocňovaní. Namiesto správy x sa bude programu na umocňovanie posielať βx, kde β je ľubovoľná hodnota. Výstup programu

1. dostaneme to z $2^{K-1} \leq d + b\Omega \leq 2^K - 1$

bude $(\beta x)^y$, pre získanie výsledku umocnenia ho bude treba vynásobiť hodnotou β^{-y}.

5.1.1 Nedostatok obfuskácie

Spôsob skrývania pôvodných hodnôt exponentov z predchádzajúcej časti bol analyzovaný v [15]. Autor objavil vážny nedostatok, ktorý umožní získať pôvodné hodnoty exponentov.

Ak pomocou obfuskácie s predchádzajúcej časti skryjeme hodnoty exponentov e, d z kryptosystému RSA, dostaneme nové hodnoty exponentov e^* a d^*.

Útočníkovi stačí poznať jednu z dvojíc (e, d),(e, d^*), (e^*, d), alebo (e^*, d^*), pomocou Millerovho algoritmu[2] bude schopný faktorizovať N. Z rozkladu čísla N jednoducho spočíta hodnotu λ. Jej odčítavaním od obfuskovanej hodnoty (e^* resp. d^*) zistí jej pôvodnú hodnotu.

Použitie obfuskácie exponentov z predchádzajúcej časti je teda možné iba v situáciách kedy útočník pozná len jeden z dvojice exponentov. Teda len šifrovanie dát pre dôveryhodný server nedôveryhodným klientom a overovanie podpisu dôveryhodného servera nedôveryhodným klientom.

V druhých dvoch prípadoch by sme museli zaistiť, aby útočník nepoznal svoj verejný kľúč, čo by už nebolo v súlade so základným princípom asymetrickej kryptografie.

5.1.2 Výpočtové nároky obfuskácie

White-box implementácia symetrických šifier, oproti ich black-box variantám, potrebuje oveľa viac pamäte na uloženie všetkých vyhľadávacích tabuliek. Prístup k white-box implementácií RSA algoritmu, ktorý zvolila spoločnosť Irdeto si svoju daň vyberá na časovej náročnosti výpočtu s obfuskovaným exponentom.

Na získanie predstavy, aké náročné bude umocňovanie s obfuskovaným exponentom bol vykonaný jednoduchý experiment[3]. V tabuľke 5.1 sú zhrnuté priemerné časy výpočtov s neobfuskovanými a obfuskovanými exponentmi.

---

2. Millerov algoritmus na faktorizáciu je pravdepodobnostný. Vstupom je e, d, N a výstupom p/q/fail.
3. Experiment sa vykonával na procesore s frekvenciou 2 GHz a 4 GB pamäte

Kľúč	$\times 1$	$\times 5$	$\times 10$	$\times 100$	$\times 500$
1024	1,3	32	63	703	3599
2048	8,3	235	510	5004	25871
4096	48,8	1689	3293	33521	164664

Tabuľka 5.1: Priemerné časy dešifrovania s obfuskovaným exponentom v milisekundách.

V experimente sa s každým vygenerovaným kľúčom dešifrovalo 100 správ[4]. Podľa dĺžky kľúča (označme n) mali správy vždy jednu z piatich rôznych dĺžok: $\frac{1}{5}n$, $\frac{2}{5}n$, $\frac{3}{5}n$, $\frac{4}{5}n$, n.

V stĺpci označenom ako $\times 0$ sú priemerné časy dešifrovania správy neobfuskovaným kľúčom. V stĺpcoch $\times 5$, $\times 10$, $\times 100$, $\times 500$ sú priemerné časy dešifrovania s obfuskovanými exponentmi s dĺžkou zodpovedajúcou päť, desať, sto resp. päťsto krát väčšou ako bol pôvodný exponent.

5.2 White-Box umocňovanie

Aby transformácia kryptosystému RSA bola použiteľná mala by byť sieť vyhľadávacích tabuliek rozumne malá. Dosiahnuť to môžme použitím efektívneho algoritmu na umocňovanie, ktorý používa málo násobení. Zároveň potrebujeme transformovať modulárne násobenie a druhú mocninu na vyhľadávacie tabuľky rozumnej veľkosti.

5.2.1 Algoritmy na umocnenie

V [16] je popísaných niekoľko algoritmov na umocňovanie. *Right-to-left binary exponentiation* (zovšeobecnený *Square-and-multiply*) a *Left-to-right binary exponentiation* sú pravdepodobne najefektívnejšie algoritmy, použiteľné pre white-box umocňovanie. Počet násobení a druhých mocnín je logaritmicky závislý na veľkosti použitej grupy. V prípade kryptosystému RSA a súčasne používaných kľúčoch je to však stále veľmi veľa.

4. S exponentom dĺžky 4096 bitov so 100 a 500 násobným predĺžením sa experiment konal iba na desiatich správach.

Algoritmy *Left-to-right k-ary exponentiation* (aj modifikovaná verzia), *Sliding window exponentiation* a *Simultaneous multiple exponentiation* sú, čo sa počtu násobení a umocnení týka, efektívnejšie, ale ako vstup požadujú aj základ umocnenia. Spoliehajú sa na dopredu spočítané hodnoty umocnenia základu, ktoré v ďalších krokoch využijú na zrýchlenie výpočtu.

Pri generovaní siete tabuliek na výpočet umocnenia ešte nevieme, aké správy sa budú s vytvorenou sieťou spracovávať, preto tieto algoritmy nemôžme použiť.

5.2.2 Modulárne násobenie

Tradičný algoritmus modulárneho násobenie $x \times y \ mod \ m$ sa dá rozdeliť na na dva kroky: vynásobiť vstupné hodnoty *x* a y, zistiť zvyšok po delení výsledku číslom m.

Existuje veľké množstvo rôzne efektívnych návrhov na modulárne redukcie. Veľmi rozšírená Barrettova redukcia [16] nahradí delenie v redukcií za násobenie. Zo zanedbateľnými nákladmi na výpočet parametra μ pred redukciou pomôže, no stále je na výpočet redukovanej hodnoty potrebných veľa násobení.

Autori v [6] navrhli tri metódy na modulárnu redukciu. Všetky si dopredu spočítajú výrazne viac hodnôt ako Barrettova redukcia. Najlepšia z nich stále potrebuje na redukciu *2k*-bitového čísla $\frac{k}{2}$ sčítaní *k*-bitových čísel.

Metóda na redukciu v [10] sa nazýva *skladanie*. Zjednoduší redukciu *2n*-bitového čísla *N* pomocou jedného násobenia dopredu spočítanej hodnoty *m′* a $\frac{n}{2}$ bitov čísla *N* a jedného sčítania dvoch $\frac{3n}{2}$-bitových slov na modulárnu redukciu $\frac{3n}{2}$-bitového slova. Na white-box implementáciu redukcie 1024 bitových slov použitých v RSA kryptosystéme je to stále málo.

Sieť vyhľadávacích tabuliek na modulárne násobenie by kvôli počtu násobení nemala byť veľká. Tradičný (pero-papier) algoritmus nebude fungovať kvôli množstvu násobení a sčítaní.

Modulárne násobenie sa dá počítať v systéme zbytkov (RNS, Residue number systems)[16], kde je číslo x reprezentované pomocou n-tice $v(x) = (x_1, ..., x_k)$, kde $x_i \ = \ mod \ m_i$ a súčin všetkým m_i je M a $gcd(m_i, m_j) = 1$ pre všetky $i \neq j$.

Pre sčítanie a násobenie dvoch čísel $v(x) = (v_1, v_2, ..., v_k)$ a $v(y) =$

$(u_1, u_2, ..., u_t)$ potom platí:

$$v(x + y) = (w_1, w_2, ..., w_t), w_i = v_i + u_i \, mod \, m_i$$
$$v(x \times y) = (z_1, z_2, ..., z_t), z_i = v_i \times u_i \, mod \, m_i$$

Problém je vo výbere čísel m_i. Viac malých čísel znamená ľahkú implementáciu pomocou siete tabuliek, ale nízku bezpečnosť RSA. Naopak veľké čísla zvýšia bezpečnosť kryptosystému, ale znemožnia vytvorenie sete vyhľadávacích tabuliek.

Poslednou uvedenou variantou násobenia bude Montgomeryho násobenie, pomocou ktorého spočítame výsledok modulárneho násobenia bez explicitného kroku redukcie. V časti 5.3 resp. v [15] je zdôvodnené, prečo je transformácia tohto násobenia na vyhľadávacie tabuľky v kombinácií s RNS resp. CRT ťažká.

5.3 RSA pomocou RNS a MM

V [15] sa autor pokúsil transformovať umocňovanie na sieť vyhľadávacích tabuliek pomocou systému zvyškov (RNS)[5] resp. čínskej zvyškovej vety (CRT) a Montgomeryho modulárneho násobenia (MM).

V RNS je číslo x, podľa bázy $\mathcal{B} = (p_+, ..., p_k)$ k nesúdeliteľných čísel, reprezentované ako n-tica $(x_1, ..., x_k)$, kde $x_i \;\; = \;\; mod \; p_i$ pre $1 \leq i \leq k$.

Výhodou RNS je, že sčítanie a násobenie sa počítajú jednoducho a môžu byť vykonávané paralelne.

Montgomeryho modulárne násobenie čísel a a b spočítame nasledovne:

$$abR^{-1} \, mod \, N = \frac{ab + N(-N^{-1}ab \, mod \, R)}{R},$$

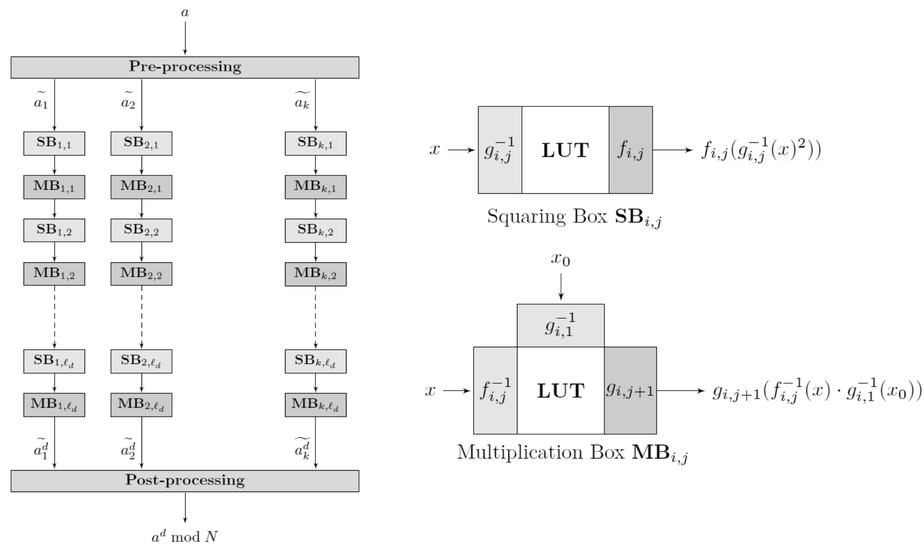
kde R je číslo nesúdeliteľné s N, také že $ab < NR$. Pravá strana rovnice nepočíta v modulárnej aritmetike, preto ju môžme počítať pomocou RNS.

Pomocou MM a RNS teda môžme jedno umocnenie veľkým exponentom rozdeliť na niekoľko menších. Umocňovanie počítal autor pomocou *square-and-multiply-always* algoritmu, ktorý naviac nezávisí na Hammingovej váhe exponentu.

5. Residue Number System

Za predpokladu, že sa Montgomeryho súčin dá transformovať na vyhľadávaciu tabuľku, môžeme umocnenie transformovať na sieť vyhľadávacích tabuliek tak, ako je znázornené na obrázku 5.1. Postup by bol nasledovný.

- vstup a sa pripraví na počítanie MM $\tilde{a}$ a rozdelí sa na k častí $\tilde{a}_i = \tilde{a} \;\; mod\, p_i$, pre $1 \leq i \leq k$
- každé $\tilde{a}_i$ spracuje sieť tabuliek na výpočet umocnenia $\tilde{a}_i^d \; mod\, p_i$
- pomocou CRT spojíme výsledky $\tilde{a}_i^d \; mod\, p_i$ do výsledku MM $\tilde{a}^d$, a ten potom transformujeme na $a^d \; mod\, N$



Obr. 5.1: White-box implementácia umocňovania – square-and-multiply-always algoritmus, vľavo sieť tabuliek, vpravo tabuľky s vstupno-výstupnými kódovaniami, prevzaté z [15]

Predpoklad o existujúcej tabuľke na výpočet MM sa ukázal ako kritický. Z hodnôt a_i a b_i totiž nie je možné získať unikátnu hodnotu $(-N^{-1}ab \; mod\, R)$.

Autor [15] skúmal aj rozšírenie tejto implementácie o ďalšiu bázu RNS, tiež neúspešne. MM spolu s RNS by sa hodilo na efektívnu hardvérovú implementáciu RSA, ale vytvoriť white-box implementáciu nie je jednoduché.

5.4 Diskusia

V tejto kapitole sme si predstavili a analyzovali niekoľko prístupov k implementácií kryptosystému RSA vhodnej do nedôveryhodného prostredia. Z rôznych dôvodov, uvedených v predchádzajúcich častiach, sa všetky ukázali ako nevhodne resp. neúspešné. White-box implementácia kryptosystému RSA a ostatných asymetrických kryptosystémov by sa pre spomenuté dôvody mala riešiť iným spôsobom ako transformáciou na vyhľadávacie tabuľky.

6 Záver

Cieľom práce bolo preskúmať možnosti implementácie kryptosystému RSA a mechanizmu HMAC do nedôveryhodného prostredia, tieto informácie vyhodnotiť a ich základe navrhnúť a implementovať prípadné white-box riešenia.

V práca sú najprv zhrnuté základné informácie o white-box kryptografii, motivácia na jej použitie, nasleduje niekoľko informácií o hašovacích funkciách a mechanizme HMAC, ktoré slúžia hlavne ako súhrn pojmov používaných v práci.

Neskôr sú v práci analyzované niektoré hašovacie funkcie zo súťaže o SHA-3 štandard, ktorá sa ukázala ako dobrý zdroj rôznych návrhov hašovacích funkcií. Z analýzy pochopiteľne vyplynulo, že nie všetky z funkcií sú na implementáciu pomocou vyhľadávacích tabuliek.

Vhodnosť pre white-box prostredie sa hodnotila podľa možnosti previesť použité operácie na vyhľadávacie tabuľky a podľa ich prípadnej veľkosti a tiež veľkosti celej siete tabuliek. Problémy spojené s transformáciou hašovacích funkcií a zopár dobrých myšlienok jednotlivých funkcií je v diskusii 3.10 na konci kapitoly.

V ďalšej kapitole detailnejšie predstavený návrh vytvárania autentizačných kódov pomocou hašovacej funkcie JH, ktorá z analýzy vyšla ako najúspešnejšia. Vychádzajúc z jednoduchého protokolu na výpočet autentizačných kódov je v tejto časti práce navrhnutá white-box implementácia mechnizmu HMAC s použitím funkcie JH. Zahŕňa jednotlivé typy tabuliek ich prepojenie v rámci siete a dizajn celej siete tabuliek. Kapitola je zakončená diskusiou 4.3 s niekoľkými problémami navrhovaného mechanizmu.

Posledná kapitola sa venuje spôsobom ako použiť v nedôveryhodnom prostredí kryptosystém RSA. Po úvodnom predstavení kryptosystému, je popísaný zaujímavý prístup k skrývaniu exponentov. Práce na tejto časti práce usmernil autor [15]. Vo svojej správe totiž zmienené skrývanie prelomil a tým jeho použitie výrazne obmedzil. Naviac dokázal, že RSA pomocou vyhľadávacích tabuliek (za použitia systému zvyškov a Montgomeryho násobenia) nevieme zostrojiť.

V kapitole o RSA sa ďalej preskúmané iné algoritmy na umoc-

ňovanie a násobenie a sú tam zhrnuté dôvody, prečo sa modulárne násobenie na sieť vyhľadávacích tabuliek nepodarilo transformovať. V rámci práce boli kontaktované aj niektoré zo spoločnosti spomenuté v úvode, aby poskytli svoje prístupy k white-box implementácií RSA. Bohužiaľ, žiadna z nich neodpovedala.

6.1 Budúci výskum

Zmienené spôsoby transformácie samozrejme nie sú jedinou možnosťou, ako vhodne implementovať kryptografické mechanizmy do nedôveryhodného prostredia. Najmä implementácia umocňovania pomocou vyhľadávacích tabuliek sa zdá byť nepraktická. Je pravdepodobné, že niekto príde s iným prístupom k white-box implementácií asymetrických kryptosystémov. Táto otázka zostáva zatiaľ nezodpovedaná.

Veľkosť navrhnutého mechanizmu na počítanie autentizačných kódov je najmenšia, akú sme spomedzi skúmaných funkcií mohli dosiahnuť, ale stále veľmi nepraktická. Preto sa ponúka možnosť nájsť, alebo vytvoriť, hašovaciu funkciu, ktorej transformácia na vyhľadávacie tabuľky bude potrebovať menej pamäte, prípadne na podpisovanie pomocou hašovacích funkcií použiť iný mechanizmus ako HMAC.

Literatúra

[1] BERTONI, Guido, Joan DAEMEN, Michaël PEETERS a Gilles VAN ASSCHE. *Cryptographic sponge functions*. 2011. Dostupné z: `http://sponge.noekeon.org/CSF-0.1.pdf`

[2] BERTONI, Guido, Joan DAEMEN, Michaël PEETERS a Gilles VAN ASSCHE. *The Keccak reference*. 2011. Dostupné z: `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`

[3] AUMASSON, Jean-Philippe, Luca HENZEN, Willi MEIER a Raphael C.-W. PHAN. *SHA-3 proposal BLAKE*. 2010. Dostupné z: `https://131002.net/blake/blake.pdf`

[4] BENADJILA, Ryad, Olivier BILLET, Henri GILBERT, Gilles MACARIO-RAT, Thomas PEYRIN, Matt ROBSHAW a Yannick SEURIN.

[5] BRESSON, Emmanuel, Anne CANTEAUT, Benoît CHEVALLIER-MAMES, Christophe CLAVIER, Thomas FUHR, Aline GOUGET, Thomas ICART, Jean-François MISARSKY, María NAYA-PLASENCIA, Pascal PAILLIER, Thomas PORNIN, Jean-René REINHARD, Céline THUILLET, Marion VIDEAU. *Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition*. 2008. Dostupné z: `https://www.shabal.com/wp-content/uploads/Shabal.pdf`

[6] CAO, Zhengjun, Ruizhong WEI a Xiaodong LIN. *A Fast Modular Reduction Method*. 2014. Dostupné z: `https://eprint.iacr.org/2014/040.pdf`

[7] FERGUSON, Niels, Stefan LUCKS, Bruce SCHNEIER, Doug WHITING, Mihir BELLARE, Tadayoshi KOHNO, Jon CALLAS a Jesse WALKER. *The Skein Hash Function Family*. 2010. Dostupné z: `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`

[8] GAURAVARAM, Praveen, Lars R. KNUDSEN, Krystian MATUSIEWICZ, Florian MENDEL, Christian RECHBERGER, Martin

SCHLÄFFER a Søren S. THOMSEN. *Grøstl - a SHA-3 candidate.* 2011. Dostupné z: http://www.groestl.info/Groestl.pdf

[9] HALEVI, Shai, William E. HALL a Charanjit S. JUTLA. *The Hash Function "Fugue 2.0"*. 2012. Dostupné z: http://researcher.watson.ibm.com/researcher/files/us-csjutla/fugue_2012.pdf

[10] HASENPLAUGH, William, Gunnar GAUBATZ a Vinodh GOPAL. *Fast Modular Reduction*. 2007. Dostupné z: https://www.lirmm.fr/arith18/papers/hasenplaugh-FastModularReduction.pdf

[11] CHOW, Stanley, Philip A. EISEN, Harold JOHNSON a Paul C. VAN OORSCHOT. *White-Box Cryptography and an AES Implementation*. 2002. Dostupné z: http://www.cs.colorado.edu/~jrblack/class/csci7000/s05/project/oorschot-whitebox.pdf

[12] KERINS, Tim a Klaus KURSAWE. *A Cautionary Note on Weak Implementations of Block Ciphers*. 2006. Dostupné z: https://www.cosic.esat.kuleuven.be/wissec2006/papers/10.pdf

[13] KLINEC, Dušan. *White-box attack resistant cryptography*. 2013. Dostupné z: https://is.muni.cz/auth/th/325219/fi_m/thesis.pdf

[14] KRAWCZYK, Hugo, Mihir BELLARE a Ran CANETTI. *HMAC: Keyed-Hashing for Message Authentication*. 1997. Dostupné z: https://tools.ietf.org/html/rfc2104

[15] LEPOINT, Tancrède. Towards Asymmetric White-Box Cryptography. 2011. Dostupné z: https://www.cryptoexperts.com/tlepoint/pub/removedLep11-towardswbc.pdf

[16] MENEZES, Alfred J., OORSCHOT, Paul C. van, VANSTONE, Scott A. *Handbook of Applied Cryptography*. 5. vydanie. CRC Press, 2001. 816 s. ISBN:0-8493-8523-7. Dostupné z: http://cacr.uwaterloo.ca/hac/

[17] MICHIELS, Wilhelmus P. A. J. a Paulus M. H. M. A. GORISSEN. *Exponent obfuscation* [patent]. Uděleno 2011. Dostupné z: http://www.google.com/patents/US20110064215

[18] SHAMIR, Adi a Nicko VAN SOMEREN. *Playing hide and seek with stored keys*. 1998. Dostupné z: https://www.cs.jhu.edu/~astubble/600.412/s-c-papers/keys2.pdf

[19] WU, Hongjun. *The Hash Function JH*. 2011. Dostupné z: http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf

[20] WYSEUR, Brecht. *White-Box Cryptography*. 2009. Dostupné z: https://www.cosic.esat.kuleuven.be/publications/thesis-152.pdf

[21] WYSEUR, Brecht. *White-box cryptography: Hiding keys in software*. 2012. Dostupné z: http://whiteboxcrypto.com/files/2012_misc.pdf

A Popis implementácie a obsah CD

V rámci práve vznikla aj implementácia dvoch mechanizmov z práce v jazyku Java. V tejto prílohe sú stručne popísané triedy v jednotlivých balíčkoch. Na CD sú všetky triedy implementácií, použité knižnice a referenčná implementácia hašovacej funkcie JH.

A.1 HMAC pre nedôveryhodné prostredie

V balíku *cz.muni.referenceJH* je referenčná implementácia hašovacej funkcie *JH* prepísaná do Javy a rozšírená o metódu na vybratie stavu hašovacej funkcie. Tá sa využíva pri generovaní stavu po spracovaní kľúčov použitých mechanizmom HMAC.

Triedy white-box implementácie mechanizmu HMAC je v balíku *cz.muni.wbhmac*. Najdôležitejšie sú *WBHMAC.java* a *WBJH.java*. Inštancie týchto dvoch tried sú výsledkom generácie HMAC-u, implementovanej v jedinej triede balíka *.generator*, v triede *Generator.java*.

Triedy, ktoré modelujú jednotlivé tabuľky, resp. skupiny tabuliek z white-bo vrámci kôl funkcie JH sú v balíkoch *.tables* a *.rounds*. Triedy tabuliek, kvôli lepšej čitateľnosti kódu iba zaobaľujú polia bajtov.

Balík *cz.muni.wbhmac.test* obsahuje triedu s niekoľkými testami implementácie operácií hašovacej funkcie JH.

Implementácia nezahŕňa ochranu pomocou vstupno-výstupných kódovaní, ani obmedzenie symetrie mechanizmu HMAC, ktoré je popísané v práci.

A.2 Obfuskácia exponentu z kryptosystému RSA

Balík *cz.muni.wbrsa* obsahuje dva generátory obfuskovaných kľúčov pre kryptosystém RSA. *KeyGenerator.java* na generovanie a obfuskovanie kľúčov pre čistú Javu, *KeyGeneratorBC.jeva* na obfuskáciu kľúčov pre použitie v knižnici *Bouncy Castle*. V triede *Sample.java* je vzorový kód ako oba generátory použiť.