ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Matějka** Jméno: **Jonatan** Osobní číslo: **435642**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra informační bezpečnosti**

Studijní program: **Informatika**

Studijní obor: **Počítačová bezpečnost**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Odhalení klíče AES sledováním běhu programu

Název diplomové práce anglicky:

Recovery of the AES key by monitoring a program's flow

Pokyny pro vypracování:

Nastudujte algoritmus šifry AES a jeho implementační aspekty na architektuře Intel x86. Identifikujte typické bloky šifry v knihovnách.Navrhněte algoritmus, který by sledováním toku kódu a jeho přístupů do paměti dokázal odhalit použití šifry AES a rozpoznat její klíč.Vytvořte aplikaci, která realizuje tento algoritmus nad cílovým procesem.Ověřte funkčnost této aplikace vůči běžným šifrovacím knihovnám, např. OpenSSL.Diskutujte své výsledky a možná rozšíření aplikace.

Seznam doporučené literatury:

Dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Josef Kokeš, katedra informační bezpečnosti FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **31.01.2019** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: ______________

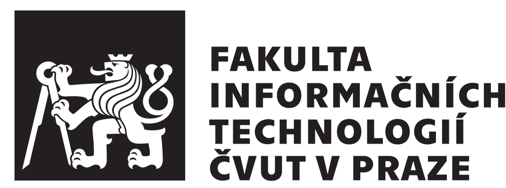
Ing. Josef Kokeš podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC



Diplomová práce

Odhalení klíče AES sledováním běhu programu

Bc. Jonatan Matějka

Katedra informační bezpečnosti
Vedoucí práce: Ing. Josef Kokeš

1. května 2019

Poděkování

Za vložený čas a úsilí, cenné rady, upřímný zájem o zvolené téma a vždy pohotovou korespondenci děkuji vedoucímu práce Ing. Josefu Kokešovi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 1. května 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Jonatan Matějka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Matějka, Jonatan. *Odhalení klíče AES sledováním běhu programu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Šifra AES je nejpoužívanější symetrická bloková šifra. Denně se s ní setkáváme v zabezpečených komunikačních protokolech a při skladování dat. V jedné oblasti nás ale použití této šifry netěší – v nevyžádaném software. Může to být ransomware, který proti naší vůli šifruje naše cenná data a požaduje výkupné za jejich dešifrování, nebo klient botnetu, který se šifrovaně domlouvá na dalším cíli útoku. V takových situacích by bylo vhodné mít k dispozici nástroj, který by šifrovaná data odhalil.

V této práci jsme pro tento účel navrhli algoritmus. Sledováním přístupů do S-Boxu při běhu programu jsme schopni odhalit klíče a data použitá při šifrování. Algoritmus jsme následně implementovali pro systém Microsoft Windows a architekturu Intel x86. Výsledný program nalezne použité klíče a data v programech provádějících AES šifrování pomocí kryptografických knihoven a v běžných uživatelských aplikacích.

Klíčová slova AES, šifra, šifrování, S-Box, odhalení klíče

Abstract

The AES cipher is the most widely used symmetric block cipher. It is used daily in secure communication protocols and in data storage. There is a certain area where the usage of this cipher doesn't please us – unwanted software. It might be ransomware encrypting our precious data and demanding money for decryption. It might be a botnet client using secure communication to coordinate the next attack. In these situations we would find it handy to have a tool to reveal these encrypted data to us.

In this thesis we propose an algorithm to serve that purpose. By observing accesses to the S-Box made during the program's run we're able to recover keys and data used for the encryption. This algorithm was implemented as a Microsoft Windows application running on Intel x86 architecture. The tool has been successfully tested against a set of applications using different cryptographic libraries and common user applications.

Keywords AES, cipher, encryption, S-Box, key recovery

Obsah

Úvod **1**

1 Šifra Rijndael **3**

1.1 Stavební prvky 3
1.2 Kroky algoritmu 6

2 Šifra AES a její implementace **11**

2.1 Naivní implementace 11
2.2 T-Tabulky 12
2.3 Bit slicing 14
2.4 AES-NI 15

3 Rozpoznání šifry AES **17**

3.1 Odhalení knihovny 17
3.2 Nalezení konstant 18
3.3 Rozpoznání struktury kódu 19

4 Návrh detekčního programu **23**

4.1 Paměťový breakpoint 23
4.2 Rekonstrukce klíče 25
4.3 Rekonstrukce dat 29

5 Implementace detekčního programu **35**

5.1 Ladicí rozhraní 38
5.2 Nalezení S-Boxů 38
5.3 Paměťový breakpoint 38
5.4 Zpracování přístupů 40

6 Testování **41**
6.1 Knihovny 41
6.2 Běžné programy 45
6.3 Vliv na výkon 50

Závěr **53**

Literatura **55**

A Obsah přiloženého CD **61**

Seznam obrázků

1.1 Násobení polynomů v konečném tělese šifry Rijndael 4
1.2 Uspořádání bajtů ve stavu o šesti sloupcích 4
1.3 S-Box šifry Rijndael 5
1.4 Počet rund v závislosti na délce klíče a velikosti bloku 6
1.5 Tabulka rundovních konstant 7
1.6 Expanze 224bitového klíče 8
1.7 Ilustrace funkce AddRoundKey 8
1.8 Ilustrace funkce ShiftRows 9
1.9 Ilustrace průběhu jedné rundy 10

2.1 Pseudokód šifrovací funkce 12
2.2 Bity nařezaných stavů uspořádané do vektorového registru 15
2.3 Vektorový registr připravený pro zpracování AES-NI instrukcí . . 15
2.4 Ukázka programu v jazyce C používajícího instrukce AES-NI . . . 16

3.1 Vytvoření otisku z grafu toku kódu a nalezení shody 21

4.1 Závislost známých sloupců při expanzi 128bitového klíče 26
4.2 Proces uspořádávání dvou po sobě jdoucích stavů šifry 32
4.3 Pokračování obrázku 4.2 32

5.1 Rozhraní knihovny LibAesSniffer, třídy Memory a SBoxFinder . . 36
5.2 Rozhraní knihovny LibAesSniffer, třída Reassembler 37

6.1 Záznam testování knihovny OpenSSL 42
6.2 Záznam testování knihovny CryptoPP 43
6.3 Záznam testování kryptografického rozhraní Microsoft Windows . 44
6.4 Zkouška odhalení šifry při vytváření ZIP archivu 46
6.5 Zkouška odhalení šifry v SSH spojení 47
6.6 Zkouška odhalení šifry v HTTPS spojení 49
6.7 Vliv použití našeho nástroje na dobu běhu programu 50

Úvod

Šifra AES (Advanced Encryption Standard) byla standardizována v roce 2001 Národním institutem standardů (NIST) a technologie Spojených států amerických. V následujících letech se stala nejpoužívanější symetrickou blokovou šifrou. Drtivá většina dnešní šifrované internetové komunikace je zabezpečena díky šifře AES. Při šifrování datových médií první volbou opět bývá šifra AES.

Šifrování nemusí vždy sloužit dobrým účelům. Nechtěný software může proti naší vůli zašifrovat naše cenná data a požadovat výkupné za jejich dešifrování. Skupina infikovaných zařízení může šifrovaně komunikovat s řídicím uzlem a provádět koordinované útoky na jiná do internetu vystavená zařízení. I pro tyto účely je populární volbou šifra AES. Pro zkoumání takových programů by bylo užitečné mít k dispozici nástroj, který by dokázal rozpoznat použití této šifry. Například automatické odhalení použitých klíčů a šifrovaných dat by značně urychlilo analýzu použitého komunikačního protokolu ve zkoumaném programu.

Cílem této práce je navrhout a implementovat takový nástroj použitelný k těmto účelům. Nástroj bude sledovat běh zkoumaného programu a z jednotlivých přístupů do paměti rozpozná použitý klíč a šifrovaná data.

Pro vytvoření takového nástroje bude nejprve nutné důkladně prozkoumat algoritmus šifry AES a různé způsoby jeho implementace. Na základě takto získaných poznatků navrhneme algoritmus, který ze zaznamenaných přístupů do paměti rekonstruuje klíč a data použitá během šifrování. Námi navržený algoritmus následně implementujeme v jazyce `C++` jako aplikaci pro operační systém Microsoft Windows na architektuře Intel x86.

Správná funkčnost vytvořené aplikace ověříme na sadě jednoduchých programů. Tyto testovací programy budou provádět šifrování pomocí známých šifrovacích knihoven (například OpenSSL), které použití šifry AES umožňují.

Kapitola 1

Šifra Rijndael

V roce 1997 vypsal Národní institut standardů a technologie Spojených států amerických veřejnou soutěž na novou symetrickou blokovou šifru. Šifry, které do soutěže vstoupily, musely podporovat 128bitovou velikost bloku a délky klíče o 128, 192 a 256 bitech. Jedním z účastníků byla i šifra Rijndael navržená belgickými kryptografy Joanem Daemenem a Vincentem Rijmenem. [1] V roce 2001 byl tento návrh vybrán jako vítězný a navržená šifra byla s drobnými úpravami standardizována pod názvem Advanced Encryption Standard, neboli AES. [2]

Jediný rozdíl mezi navrženým Rijndaelem a standardizovaným AES je v možných velikostech bloku a délkách klíče. Rijndael podporuje všechny velikosti a délky v rozsahu mezi 128 a 256 bity, které jsou násobkem 32[1]. AES stanovuje jedinou možnou velikost bloku na 128 bitů. Délky klíčů také omezuje dle původního zadání na 128, 192 a 256 bitů. AES je tedy pouhou podmnožinou šifry Rijndael. [3, s. 31]

1.1 Stavební prvky

Před uvedením samotného šifrovacího algoritmu prozkoumáme matematické koncepty, které budou v rámci šifry využívány.

1.1.1 Konečné těleso

Šifra pracuje s klíčem i daty na úrovni bajtů. Všechny bajty jsou z pohledu šifry polynomy, prvky konečného tělesa GF(2^8). Jednotlivé bity $(b_7, b_6, ..., b_0)$, seřazené od nevíce významného bitu po nejméně významný, jsou koeficienty polynomu $p(x) = \sum_{n=0}^{7} b_i x^i$. Sčítání bajtů je definováno jako sčítání těchto

[1] Původní návrh požadavků na šifru vyžadoval i velikosti bloku o 192 a 256 bitech. Od požadavku se později upustilo, nicméně autoři Rijndaelu viděli proměnnou šířku bloku jako výhodu, a tak ji v návrhu šifry ponechali. [3, s. 2]

polynomů – sčítáme jednotlivé koeficienty modulo 2, neboli provádíme XOR těchto dvou bajtů. Operace násobení odpovídá násobení polynomů modulo ireducibilní polynom $m(x) = x^8 + x^4 + x^3 + x + 1$. [4, s. 10–13]

$$\begin{aligned}
&(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = \\
&= (x^{13} + x^{11} + x^9 + x^8 + x^7) + \\
&+ (x^7 + x^5 + x^3 + x^2 + x) + \\
&+ (x^6 + x^4 + x^2 + x + 1) = \\
&= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 = \\
&= x^{11} + x^4 + x^3 + 1 = \\
&= x^7 + x^6 + 1
\end{aligned}$$

Obrázek 1.1: Násobení polynomů v konečném tělese použitém šifrou Rijndael.

1.1.2 Stav

Bajty zpracovávané šifrou Rijndael jsou uspořádané do matic o čtyř řádcích a různých počtech sloupců. Prvky matice $\mathbb{M}_{n\times m}$ jsou indexovány číslem z množiny $\{0, 1, ..., nm - 1\}$.[2] Prvku na i-tém řádku a j-tém sloupci přísluší index $jn + i$. Tyto matice se nazývají stavem šifry. Stav je základní struktura, nad kterou jsou prováděny jednotlivé kroky šifrovacího algoritmu. [4, s. 9–10]

$$\begin{pmatrix}
p_0 & p_4 & p_8 & p_{12} & p_{16} & p_{20} \\
p_1 & p_5 & p_9 & p_{13} & p_{17} & p_{21} \\
p_2 & p_6 & p_{10} & p_{14} & p_{18} & p_{22} \\
p_3 & p_7 & p_{11} & p_{15} & p_{19} & p_{23}
\end{pmatrix}$$

Obrázek 1.2: Uspořádání bajtů ve stavu o šesti sloupcích.

1.1.3 S-Box

Součástí většiny symetrických šifer bývá takzvaný S-Box – nelineární funkce, speciálně navržená tak, aby korelace vstupů s výstupy byla co nejmenší [3, s. 35–36]. Účel této funkce je zajištění konfúze v rámci šifry. Nejčastěji bývá

[2] Stejně jako v návrhu a standardu šifry je v této práci dodržováno indexování od nuly.

zadána nebo implementována jako substituční tabulka[3]. V této tabulce dohledáme výstup pro zadaný vstup.

V případě šifry Rijndael je substituční funkce $S_{RD} : \mathrm{GF}(2^8) \mapsto \mathrm{GF}(2^8)$ zadána jako složení dvou funkcí $S_{RD} = f(g(b))$, kde:

$$g(b) = \begin{cases} 0 & \text{pro } b = 0 \\ b^{-1} & \text{pro } b \neq 0 \end{cases}$$

$$f(b) = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Z této definice samozřejmě lze dopočítat samotnou substituční tabulku (obrázek 1.3). Původní návrh [1] žádné tabulky neobsahuje, vypočtené hodnoty nalezneme až ve standardu šifry AES [4].

Při dešifrování je potřeba aplikovat inverzi této funkce, proto většinou předpočítáme dopředný i inverzní S-Box.

S_{RD}	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
10	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
20	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
30	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
40	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
50	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
60	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
70	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
80	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
90	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A0	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B0	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D0	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E0	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F0	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Obrázek 1.3: S-Box šifry Rijndael.

[3] Anglicky Substitution-Box, odtud název S-Box.

1.2 Kroky algoritmu

Šifrovací algoritmus se skládá z několika dílčích kroků, které jsou opakovány v závislosti na délce klíče a velikosti bloku, respektive na počtu jejich sloupců N_k a N_b. Každý sloupec obsahuje 32 bitů. Pro zašifrování bloku dat je nutné provést následující kroky:

1. Expanze klíče
2. Prvotní součet dat a klíče
3. Substituce bajtů stavu
4. Posun řádků stavu
5. Promíchání sloupců stavu
6. Přičtení rundovního klíče ke stavu
7. Opakování kroků 3 až 6 dle délky klíče a velikosti bloku
8. Substituce bajtů stavu
9. Posun řádků stavu
10. Přičtení rundovního klíče ke stavu

Před spuštěním algoritmu naplníme stav daty k zašifrování. Aplikací výše uvedených kroků získáme stav obsahující zašifrovaná data. Skupina kroků 3 až 6 se nazývá *runda*. Krokům 8–10 se také říká *poslední runda*. Pro 128bitový blok ($N_b = 4$) a 128bitový klíč ($N_k = 4$) je počet provedení všech rund $N_r = 10$. S každým přidaným sloupcem klíče nebo bloku dojde k navýšení počtu provedených rund N_r o jednu. Přesná definice pro počet opakování je $N_r = \max(N_k, N_b) + 6$. [3, s. 33–34, 41–42]

Při dešifrování se provádí inverzní kroky v opačném pořadí. Nejprve je potřeba expandovat klíč, stejně jako při šifrování. Stav naplníme zašifrovanými daty a postupujeme od posledního kroku k prvnímu. Provádíme opačné operace až k samotnému začátku algoritmu, kde stav obsahuje dešifrovaná data.

N_r		Sloupce bloku				
		4	5	6	7	8
Sloupce klíče	4	10	11	12	13	14
	5	11	11	12	13	14
	6	12	12	12	13	14
	7	13	13	13	13	14
	8	14	14	14	14	14

Obrázek 1.4: Počet rund v závislosti na délce klíče a velikosti bloku.

1.2.1 Expanze klíče

Prvním krokem je vždy expandování klíče (KeyExpansion). Tato operace je nezávislá na datech, a proto je většinou prováděna pouze jednou pro každý klíč. Účelem tohoto kroku je prodloužit původní klíč natolik, aby šel rozdělit na $N_r + 1$ částí o velikosti bloku. Tyto části se nazývají rundovní klíče a je jich vždy o jednu více než rund, aby mohlo dojít ke kroku součtu dat s prvním rundovním klíčem, který předchází všem rundám. Hodnotu i-tého sloupce prodlouženého klíče Wi lze definovat následujícím předpisem,

$$W_i = \begin{cases} K_i & \text{pro } i < N_k \\ W_{i-N_k} + r(s(W_{i-1})) + rcon_{i/N_k} & \text{pro } i \geq N_k, i \equiv 0 \pmod{N_k} \\ W_{i-N_k} + s(W_{i-1}) & \text{pro } i \geq N_k, N_k > 6, i \equiv 4 \pmod{N_k} \\ W_{i-N_k} + W_{i-1} & \text{v ostatních případech} \end{cases}$$

kde K jsou sloupce původního klíče [5]. Funkce r, s a konstanty $rcon$ mají následující definici:

$$r(w) = r\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_0 \end{pmatrix}$$

$$s(w) = s\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} S_{RD}(w_0) \\ S_{RD}(w_1) \\ S_{RD}(w_2) \\ S_{RD}(w_3) \end{pmatrix}$$

$$rcon_i = \begin{pmatrix} 2^{i-1} \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{ pro } i \in \mathbb{N}^+$$

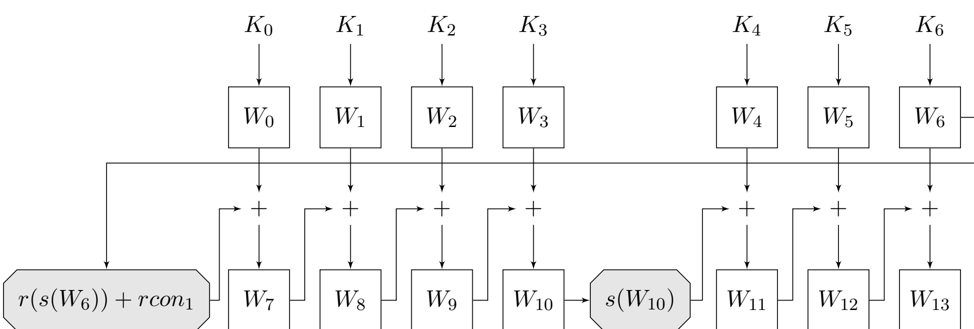
Funkce r a s ve standardu najdeme pod názvy RotWord a SubWord. Rundovní konstanty $rcon$ se většinou předpočítají do formy tabulky, podobně jako v případě S-Boxu. Například pro šifrování bloků o velikosti 128 bitů si vystačíme s deseti konstantami (tabulka 1.5).

i	01	02	03	04	05	06	07	08	09	0A
$rcon_i$	01	02	04	08	10	20	40	80	1B	36

Obrázek 1.5: Tabulka rundovních konstant $rcon$. Obsahuje dostatečný počet předpočítaných hodnot pro použití při šifrovaní 128bitových bloků. Uvedeny jsou pouze první bajty sloupců, zbylé bajty jsou dle definice vždy nulové.

Počet potřebných sloupců N_{ek} expandovaného klíče W se odvíjí od počtu rund a velikosti bloku:

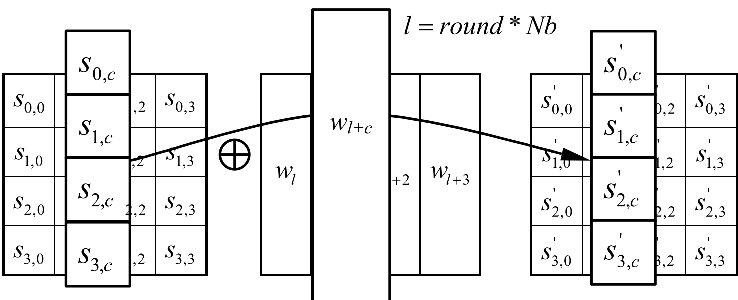
$$N_{ek} = (N_r + 1)N_b = (\max(N_k, N_b) + 7)N_b$$



Obrázek 1.6: Výpočet prvních 14 sloupců expandovaného klíče. Délka původního klíče je 224 bitů.

1.2.2 Přičtení rundovního klíče

V předchozím kroku jsme získali dostatečně dlouhý expandovaný klíč, který jsme rozdělili na potřebný počet rundovních klíčů. Další operací bude přičtení jednotlivých bajtů rundovního klíče k bajtům stavu šifry (AddRoundKey). Při prvotním přičtení klíče sečteme bajty dat s bajty nultého rundovního klíče a výsledek přiřadíme jednotlivým bajtům stavu. V každé následující i-té rundě pouze přičteme ke vstupnímu stavu $i+$ první rundovní klíč. Z povahy sčítání (XOR) je tato operace symetrická, při dešifrování postupujeme od posledního rundovního klíče k nultému.



Obrázek 1.7: Ilustrace funkce AddRoundKey ze standardu šifry AES [4, s. 19].

1.2.3 Substituce

Operace substituce (SubBytes) nahradí každý bajt stavu S hodnotou dohledanou v S-Boxu, neboli:

$$S'_i = S_{RD}(S_i)$$

Tento krok je velice podobný funkci SubWord, prováděné při expanzi klíče. Při dešifrování provádíme inverzí operaci (InvSubBytes) za použití inverzního S-Boxu:

$$S_i = S_{RD}^{-1}(S'_i)$$

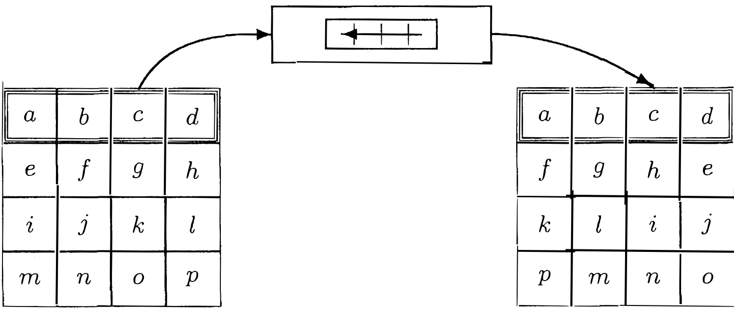
1.2.4 Posun řádků

Tento krok (ShiftRows) mění uspořádání bajtů ve stavu. Jedná se o jediné místo v celém algoritmu, kdy se namísto sloupců stavu pracuje s řádky. Na jednotlivé řádky stavu aplikujeme operaci rotace směrem doleva o daný počet prvků. Pro stavy s nejvýše šesti sloupci platí, že i-tý řádek posuneme doleva o i prvků. Z pohledu bajtů provádíme:

$$S'_i = S_{(i+4(i \bmod 4)) \bmod 4N_b}$$

Poslední řádek stavu se sedmi a více sloupci posuneme o jeden prvek dále, totéž platí pro předposlední řádek stavu s osmi sloupci.

Pro účely dešifrování existuje také inverzní operace (InvShiftRows), kdy dochází k rotaci řádků směrem doprava.



Obrázek 1.8: Ilustrace funkce ShiftRows z knihy *The Design of Rijndael* [3, s. 38].

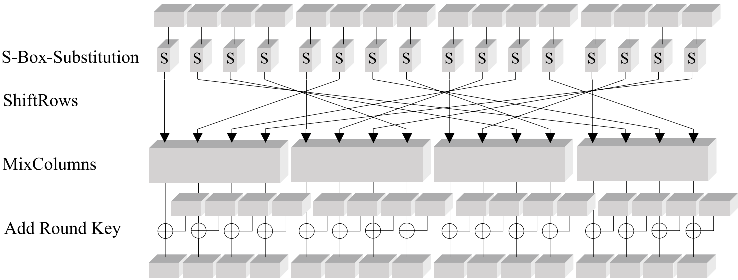
1.2.5 Promíchání sloupců

Poslední operací je mísení bajtů v jednotlivých sloupcích (MixColumns). Každý sloupec stavu šifry je zleva vynásoben určenou maticí o rozměrech 4×4 a výsledkem je sloupec nového stavu:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix}$$

Obdobně zpracujeme i zbylé tři sloupce. Během dešifrování provádíme násobení inverzní maticí (InvMixColumns):

$$\begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix} = \begin{pmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{pmatrix} \begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix}$$



Obrázek 1.9: Ilustrace průběhu jedné rundy z knihy *Cryptography in C and C++* [6, s. 246].

Kapitola 2

Šifra AES a její implementace

Zbytek práce se věnuje pouze šifře AES, neboli speciálnímu případu šifry Rijndael. Ze všech možných konfigurací poskytnutých v návrhu šifry Rijndael si vybereme pevnou velikost bloku o 128 bitech a délku klíče o 128, 192 nebo 256 bitech. Algoritmus šifry AES lze implementovat mnoha způsoby. Přístupy se liší v závislosti na zvolené cílové architektuře a požadavcích na efektivitu výpočtu.

Tato kapitola se zaměřuje na detaily softwarových implementací pro architekturu 32 a 64bitových procesorů z rodiny x86, také známé pod názvy *IA-32* a *Intel 64*. [7]

2.1 Naivní implementace

Prvním krokem při šifrování je expanze klíče. Tuto operaci je nutné provést pouze jednou pro každý klíč. Při běžném použití šifrování bývá výrazně více dat než klíčů. Neexistuje tedy žádná výrazná motivace tuto funkci optimalizovat. Implementace tak zpravidla zcela přímočaře kopíruje textový popis funkce. Obdobný přístup lze aplikovat i na zbytek operací a z každého kroku vytvořit funkci, do které vstupuje stav, popřípadě i rundovní klíč, a vystupuje nový stav. Řídicí funkce pak obsahuje cyklus, ve kterém dochází k volání těchto dílčích funkcí. Výsledek může vypadat obdobně jako na obrázku 2.1.

První optimalizací takové implementace může být připravení vyhledávacích tabulek[4] pro násobky potřebné při míchání sloupců. Vedle S-Boxu a tabulky rundovních konstant přibude tabulka pro násobení v tělese – násobky dvou a tří pro MixColumns; devíti, jedenácti, třinácti a čtrnácti pro InvMixColumns. [8, 9]

[4]Anglicky *lookup table*.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Obrázek 2.1: Pseudokód šifrovací funkce ze standardu šifry AES [4, s. 15].

2.2 T-Tabulky

Dalšího zrychlení je možné dosáhnout díky využití vlastností konkrétní architektury. Na procesorech, které umožňují efektivní práci s 32bitovými slovy, můžeme pracovat s celými sloupci stavu namísto jednotlivých bajtů. Operace přičtení rundovního klíče k aktuálnímu stavu přímo vybízí k použití instrukcí pro práci s čtyřbajtovými slovy:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix} + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Také můžeme zkusit kroky algoritmu sloučit. Dosaďme do předchozího výrazu za vstupní stav výstupní stav operace promíchání sloupců:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix} + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Předřazením operace posunu řádků pozměníme indexy bajtů vstupního stavu:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} S_0 \\ S_5 \\ S_{10} \\ S_{15} \end{pmatrix} + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Sloučením s krokem substituce dostáváme předpis pro celou rundovní operaci nad jedním výstupním sloupcem stavu:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} S_{RD}(S_0) \\ S_{RD}(S_5) \\ S_{RD}(S_{10}) \\ S_{RD}(S_{15}) \end{pmatrix} + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Maticové násobení lze rozepsat jako lineární kombinaci sloupců:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = S_{RD}(S_0) \begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \end{pmatrix} + S_{RD}(S_5) \begin{pmatrix} 3 \\ 2 \\ 1 \\ 1 \end{pmatrix} + S_{RD}(S_{10}) \begin{pmatrix} 1 \\ 3 \\ 2 \\ 1 \end{pmatrix} + S_{RD}(S_{15}) \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix} + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Mějme funkce $T_{0,\ldots,3} : \mathrm{GF}(2^8) \mapsto \mathrm{GF}(2^8)^4$ odpovídající prvním čtyřem sčítancům z předešlého výrazu:

$$T_0(b) = S_{RD}(b) \begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \end{pmatrix}, \ldots, T_3(b) = S_{RD}(b) \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$

Dosazením do předpisu rundovní operace získáme zjednodušený výraz:

$$\begin{pmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = T_0(S_0) + T_1(S_5) + T_2(S_{10}) + T_3(S_{15}) + \begin{pmatrix} K_0 \\ K_1 \\ K_2 \\ K_3 \end{pmatrix}$$

Stejnou úpravu můžeme provést i pro ostatní sloupce, výsledek se bude lišit pouze v indexech stavů a klíče.

Funkce $T_{0,\ldots,3}$ implementujeme jako čtyři vyhledávací tabulky – *T-Tabulky*. Veškeré násobení v tělese nahradíme dohledáním slova v tabulce. Kde dříve býval S-Box a tabulky dvojnásobků a trojnásobků, budou čtyři T-Tabulky. Přestože provádíme samostatnou substituci při expanzi klíče a v poslední rundě, můžeme se S-Boxu beztrestně zbavit, protože potřebné hodnoty můžeme dohledat i v jednotlivých T-tabulkách. [1, s. 17–18]

Před provedením podobné úpravy i pro dešifrování musíme nejprve přeházet kroky dešifrování tak, aby zachovávaly stejné pořadí jako při šifrování. Díky linearitě kroků to udělat můžeme, nicméně musíme náležitě upravit rundovní klíče – aplikovat na ně inverzní operaci míchání sloupců. [4, s. 23–25]

Po přeuspořádání kroků v dešifrovacím algoritmu nám nic nebrání ve vytvoření obdobných dešifrovacích *D-Tabulek*. Opět se můžeme zbavit tabulek s násobky, ale inverzní S-Box musíme zachovat, protože žádná z D-Tabulek neobsahuje původní hodnoty inverzního S-Boxu.

Výsledný program bude potřebovat výrazně více paměti. K původním dvěma S-Boxům a tabulce rundovních konstant (522 bajtů) jsme přidali tabulky pro usnadnění násobení (v součtu 2058 B), abychom je nakonec vyměnili za ještě větší tabulky (dohromady 8458 B). Pokud nás tyto nároky trápí, můžeme ze čtyř T-Tabulek vybrat pouze jednu a její výstup vždy patřičně rotovat [1, s. 17–18]. Takovou úpravou se dostaneme na celkových 2314 bajtů potřebné paměti.

Na architekturách, které umožňují nezarovnaný přístup ke slovům v paměti, můžeme tabulky přes sebe překrýt. Výsledkem bude dvojnásobně velká T-Tabulka, kde pro každý vstupní bajt dostaneme osmibajtové slovo, které následně vymaskujeme na požadované čtyřbajtové slovo z původní tabulky [10]. Takový program si pro uložení konstant bude nárokovat 4362 bajtů paměti.

2.3 Bit slicing

Při realizaci algoritmu se také lze vydat naprosto jiným směrem a v programu imitovat hardwarovou implementaci. Stav šifry rozdělíme na jednotlivé bity[5], a operace, které by byly prováděny hradly, budeme simulovat pomocí příslušných instrukcí (XOR, AND, OR, . . .) [11, s. 5–9]. Zbavíme se tak veškerých tabulek a po všech krocích algoritmu nám zůstane jen dlouhá posloupnost jednoduchých instrukcí. Tento přístup má své nesporné výhody:

- Na šifrování je spotřebováno vždy stejné množství cyklů. Program je tak odolný vůči útokům postranním kanálem založených na měření doby vyzvednutí dat z určitých adres v paměti[6]. [12, s. 13]

- Výsledný program potřebuje výrazně méně paměti pro uložení potřebných konstant.

Zmíněné přínosy jsou vyváženy horším výkonem programu. Rychlosti tabulek využívajících implementací bylo dosaženo až s příchodem vektorových rozšíření (MMX, SSE, SSE2, SSE3, SSSE3, SSE4) pro architekturu x86 [13]. Do osmi 128bitových registrů nařežeme osm nezávislých stavů šifry, které zpracováváme paralelně. Tuto úpravu však můžeme provést jen u určitých režimů blokové šifry. Například v režimu čítače (CTR) nebo kódové knihy (ECB) nejsou jednotlivé bloky na sobě závislé, a tak je můžeme zpracovávat paralelně. Naopak pro mód řetězení bloků (CBC) je tato implementace nevhodná, pokud nechceme šifrovat několik nezávislých textů naráz [11, s. 6].

[5] Odtud anglický název *bit slicing*.

[6] Anglicky *cache timing attack*.

<table>
<tr><td colspan="16">row 0</td><td>...........</td><td colspan="9">row 3</td></tr>
<tr><td colspan="4">column 0</td><td colspan="4">column 1</td><td colspan="4">column2</td><td colspan="4">column 3</td><td>...........</td><td colspan="4">column 0</td><td>......</td><td colspan="4">column 3</td></tr>
<tr><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td><td>...........</td><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td><td>......</td><td>block 0</td><td>block 1</td><td>...</td><td>block 7</td></tr>
</table>

Obrázek 2.2: Nařezané bity z osmi různých stavů uspořádané do jednoho z osmi 128bitových vektorových registrů [11, s. 6].

2.4 AES-NI

V průběhu času přibývala další rozšíření, která navyšovala počet vektorových registrů i jejich velikost. Na nejnovějších procesorech (AVX-512) můžeme pracovat až s 32 registry po 512 bitech [7]. S takovýmito prostředky by jistě šlo dosáhnout ještě lepší paralelizace pomocí bit slicingu. Tato technika ale byla překonána rozšířením AES-NI, které přináší speciální instrukce realizující konkrétní části algoritmu AES:

- AESENC a AESENCLAST provádí výpočet jedné rundy, respektive poslední rundy.
- AESDEC a AESDECLAST počítají rundy při dešifrování.
- AESKEYGENASSIST vypočítá hodnoty potřebné pro expanzi klíče. Dva vstupní sloupce nahradí (SubWord), posune (RotWord) a přičte rundovní konstantu. Dalším výstupem jsou i dva pouze substituované sloupce pro účely expanze 256bitového klíče.
- AESIMC provede inverzní operaci promíchání sloupců (InvMixColumns), nutnou pro úpravu rundovních klíčů před dešifrováním (stejně jako při použití T-Tabulek).

Použití těchto instrukcí skýtá podobné výhody jako bitslicing – bezpečnost, malé paměťové požadavky (nyní i pro uložení kódu), ale hlavně rychlost [14]. Software využívající šifru AES si většinou za běhu ověří, jaké rozšíření má k dispozici, a na základě toho se rozhodne, jakou implementaci použije.

Byte #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit Position	**127-120**	**119-112**	**111-103**	**103-96**	**95-88**	**87-80**	**79-72**	**71-64**	**63-56**	**55-48**	**47-40**	**39-32**	**31-24**	**23-16**	**15-8**	**7-0**
	127 - 96				95 - 64				64 - 32				31 - 0			
State Word	X3				X2				X1				X0			
State Byte	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Obrázek 2.3: Bajty stavu AES v 128bitovém registru připravené na zpracování AES-NI instrukcí [7].

```
#include <wmmintrin.h>

/* Note - the length of the output buffer is assumed to be a multiple of 16 bytes */

void AES_ECB_encrypt(const unsigned char *in,  //pointer to the PLAINTEXT
                     unsigned char *out,       //pointer to the CIPHERTEXT buffer
                     unsigned long length,     //text length in bytes
                     const char *key,          //pointer to the expanded key schedule
                     int number_of_rounds)     //number of AES rounds 10,12 or 14
    {
    __m128i tmp;
    int i,j;

    if(length%16)
        length = length/16+1;
    else
        length = length/16;

    for(i=0; i < length; i++){
        tmp = _mm_loadu_si128 (&((__m128i*)in)[i]);
        tmp = _mm_xor_si128 (tmp,((__m128i*)key)[0]);
        for(j=1; j <number_of_rounds; j++){
            tmp = _mm_aesenc_si128 (tmp,((__m128i*)key)[j]);
        }
        tmp = _mm_aesenclast_si128 (tmp,((__m128i*)key)[j]);
        _mm_storeu_si128 (&((__m128i*)out)[i],tmp);
        }
    }
```

Obrázek 2.4: Ukázka programu v jazyce C používajícího instrukce AES-NI. Vestavěné (*intrinsic*) funkce zpřístupněné hlavičkovým souborem `wmmintrin.h` jsou pouze jednoduché obaly samotných instrukcí [15, s. 28–29].

Kapitola 3

Rozpoznání šifry AES

Nyní už víme, jakých různých podob může nabývat program šifrující pomocí AES. Popsané implementace jsou natolik výrazné, že jsme schopni v cílovém programu detekovat jejich použití a následně odhalit šifrovaná data nebo tajný klíč. Analýzu programů můžeme provádět staticky, bez spuštění programu, nebo dynamicky, sledováním běhu spuštěného procesu. V této a následujících kapitolách se zaměříme na použití šifry AES v programech pro běžné operační systémy – rodiny systémů unix a Microsoft Windows.

3.1 Odhalení knihovny

AES je velmi rozšířená symetrická bloková šifra, existuje tedy velké množství různých kryptografických knihoven, které nabízí její hotové implementace [16]. Knihovnu můžeme s programem propojit staticky, kdy při překladu dojde k sloučení kódu programu s kódem knihovny. Druhou možností je dynamické propojení, kdy se do programu zakomponuje pouze informace o tom, které části (funkce, proměnné) knihovny budou za běhu použity. Při spouštění programu se operační systém postará o nalezení potřebné knihovny a načtení do paměti spouštěného procesu. V načtené knihovně najde požadované části a jejich adresy předá programu. Ten s těmito informacemi již může knihovnu začít používat. Spojení s knihovnou probíhá během překladu v kroku zvaném *linkování*. Mluvíme tak o statickém a dynamickém linkování.

3.1.1 Dynamicky linkovaná knihovna

V případě připojení cizího kódu až za běhu programu je odhalení kryptografické knihovny snadné. S pomocí statické analýzy zjistíme seznam dynamicky slinkovaných knihoven. U systémů Windows prohledáme strukturu *import directory table* v PE souboru [17]. Na unixových systémech prozkoumáme v ELF souboru v sekci `.dynamic` záznamy s příznakem DT_NEEDED [18].

Jakmile v seznamu nalezneme knihovnu poskytující šifrování, můžeme dále staticky zkoumat program a hledat části kódu, které knihovnu využívají, s očekáváním, že nás dovedou k tajnému klíči.

Také se můžeme uchýlit k dynamické analýze, která nám nabízí širší možnosti. Danou knihovnu můžeme nahradit nebo podvrhnout upravenou verzi, která bude potřebné údaje zaznamenávat. Systém Windows přesvědčíme k načtení jiné knihovny například umístěním knihovny přímo do složky se spouštěným programem [19]. Na unixu použijeme proměnnou prostředí LD_LIBRARY_PATH nebo LD_PRELOAD, do které nastavíme cestu k adresáři obsahující naší verzi knihovny, respektive cestu ke knihovně samotné [20]. Spuštěný proces také lze ladit a přímo sledovat jednotlivá volání funkcí. Výsledek je vždy stejný – program nám dobrovolně předá klíč i data v argumentech volané knihovní funkce.

3.1.2 Staticky linkovaná knihovna

Při použití statické knihovny je naše práce značně ztížena. Výslednou aplikaci nelze jednoduše rozdělit zpět na části, které do překladu vstupovaly. Pokud ale máme použitou knihovnu k dispozici ve zkompilované podobě, můžeme se pokusit nalézt její části v daném programu. Nejprve si připravíme otisky[7] funkcí, které budeme v programu chtít dohledat. Takovým otiskem může být například posloupnost bajtů, které se ve funkci nachází [22], multimnožina systémových volání z dané funkce [23], nebo vybrané rysy grafu toku kódu[8] [21]. Otisky následně porovnáme s částmi zkoumaného programu a dle míry podobnosti rozhodneme, zdali se jedná o konkrétní knihovní funkci.

Tímto způsobem můžeme nalézt v programu část kódu, která odpovídá šifrovací funkci z původní knihovny. Tuto část můžeme opět staticky zkoumat, upravovat, nebo za běhu programu ladit. Zkoumání provádíme za účelem nalezení argumentů funkce, které skrývají klíč a data.

3.2 Nalezení konstant

V případě, že program obsahuje vlastní implementaci šifry, nebo natolik pozměněný kód z kryptografické knihovny, že není možné ho rozpoznat pomocí otisků, můžeme hledat pro šifru specifické konstanty – S-Boxy a T-Tabulky. Tyto tabulky musí být přítomny v každé efektivní implementaci, která se nespoléhá čistě jen na bit slicing nebo AES-NI instrukce. Existují nástroje, které pomocí statické analýzy program prozkoumají a nahlásí vše, co vypadá jako charakteristická konstanta nějaké šifry [24].

Od nalezených tabulek je dlouhá cesta k odhalení klíče a dat. Využitím statické analýzy kódu můžeme odhalit instrukce, které k těmto tabulkám

[7] Anglický termín pro tuto techniku je *function fingerprinting* [21].

[8] Anglicky *control-flow graph*, zkratka CFG.

přistupují. Z okolí těchto instrukcí musíme zjistit, o jaký krok šifry se jedná. Dynamickou analýzou nad spuštěným procesem můžeme zkoumat, jaký stav šifry vstupuje do tohoto kroku a z něj dopočítat původní použitý klíč a data.

Při tomto přístupu můžeme narazit na problém při hledání instrukcí, které s konstantami pracují. Statickou analýzou nemusíme odhalit potřebné instrukce, a tak bude potřeba se uchýlit k dynamické analýze a trasování programu. Během trasování je kód programu vykonáván po jednotlivých instrukcích. Před provedením každé instrukce zkontrolujeme, s jakými daty pracuje a zdali nepřistupuje k námi sledovaným tabulkám. Na této informaci následně stavíme při další analýze programu.

Pokud nejsou konstanty přítomny v samotném programu, není tento postup použitelný. K takové situaci může dojít snadno, například generováním obsahu tabulek až za běhu programu. Dále zabalením programu, kdy je původní aplikace zkomprimována [25]. K rozbalení a odhalení tabulek opět dojde až po spuštění. V aplikaci také může být použitá neznámá dynamická knihovna obsahující potřebné kryptografické funkce. V takovém případě budeme nejprve zkoumat ji a následně aplikujeme dříve popsaný postup pro práci s knihovnami.

3.3 Rozpoznání struktury kódu

Poslední možností, která nám zbývá, je zkoumání samotného kódu. To je velmi náročná úloha, navíc těžko automatizovatelná. Statickou analýzou spolehlivě nalezneme pouze instrukce AES-NI, a to ještě za předpokladu, že program nebyl zabalen nebo *obfuskován*. Obfuskace je způsob úpravy programu za účelem znesnadnění statické analýzy při zachování funkcionality. V takovém případě je vhodnější se věnovat dynamické analýze kódu již běžících procesů.

3.3.1 Zachycení AES-NI instrukcí

Zaměříme se na detekci AES-NI rozšíření pro architekturu x86. Abychom nemuseli program krokovat po jednotlivých instrukcích, hodilo by se nám program zastavit až v momentě vykonání speciální instrukce z tohoto rozšíření. Procesory, které tyto instrukce neznají, jsou pro náš účel ideální – provedou nezajímavé části kódu a v pravém okamžiku skončí chybou [26, s. 6–7]. Stejného výsledku můžeme na některých procesorech dosáhnout vypnutím AES-NI rozšíření [27, s. 42]. Také můžeme deaktivovat veškeré instrukce, které používají vektorové registry (do této skupiny spadají i AES-NI instrukce) [28].

Program spustíme ve virtuálním stroji. Když se program zeptá svého virtuálního stroje (hosta), jestli podporuje instrukční sadu AES-NI, je tento dotaz předán programu, který virtualizaci zajišťuje (hostiteli). Ten samozřejmě odpoví, že ano. Tato odpověď neodpovídá skutečnosti, takže v momentě, kdy program bude chtít vykonat AES-NI instrukci, dojde k vyvolání výjimky a předání řízení hostiteli. Na straně hostitele nasimulujeme průběh instrukce

tak, aby host nic nepoznal, a předáme řízení zpět. V případě instrukce AESENC nebo AESDEC si zaznamenáme operandy (stav šifry a rundovní klíč), ze kterých dopočítáme původní klíč a data.

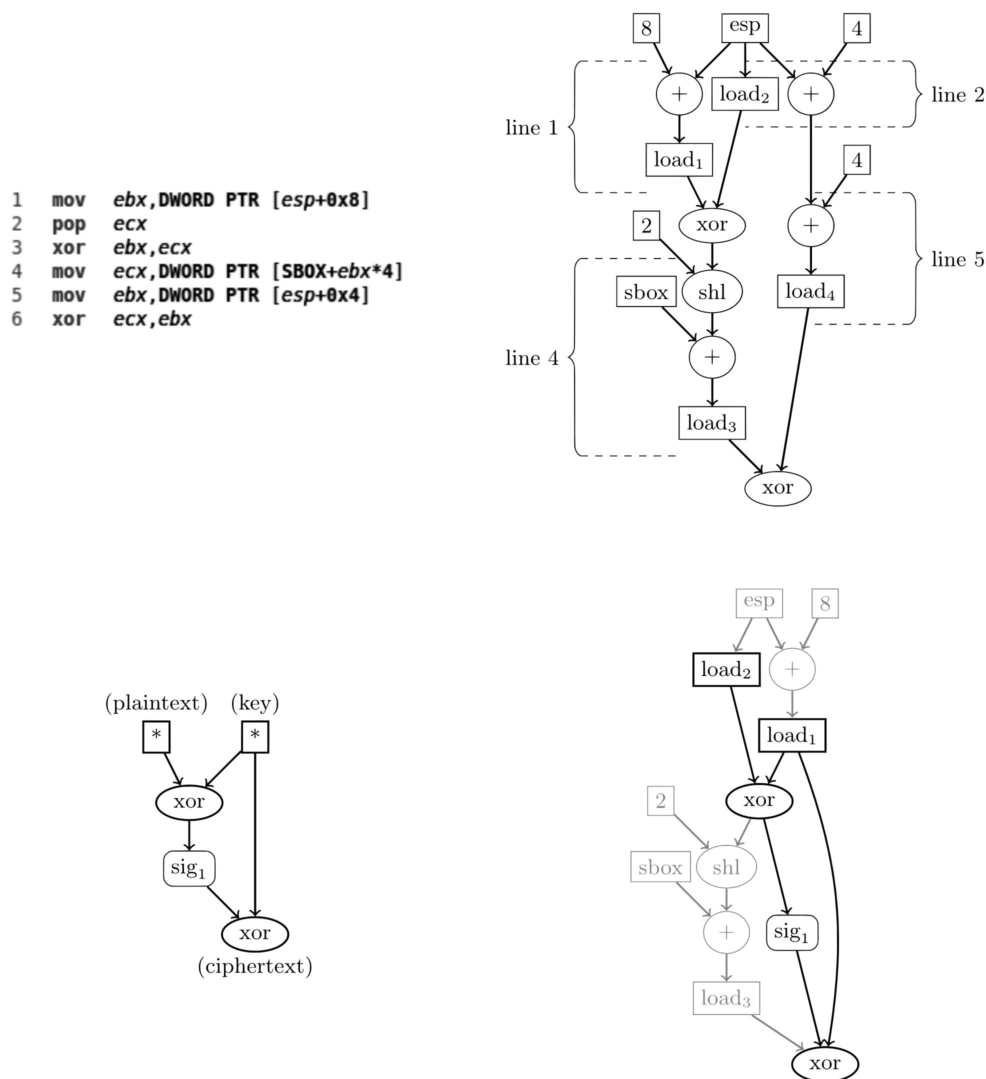
3.3.2 Trasování

V programu může být šifra implementována i bez použití AES-NI instrukcí. V takovém případě nám nezbývá nic jiného než program trasovat. Krokováním programu po jednotlivých instrukcích vznikne naprosto přesný záznam o tom, co program vykonal. Uložený běh programu jistě obsahuje veškeré potřebné informace k odhalení šifry. Většina zaznamenaných informací je ale pro nás naprosto bezcenná. Ze záznamu nejprve vyřadíme části, kdy program prováděl kód knihoven, o kterých s jistotou víme, že k šifrování neslouží [29, s. 6]. Dále se můžeme zbavit opakovaných instrukcí nalezením cyklů, které byly v programu provedeny [30, 31]. Stále nám ale zbyde obrovské množství dat, které musíme zpracovat.

Pro každý cyklus můžeme zkoumat, jaké hodnoty do něj vstupují a jestli hodnoty z něj vystupující odpovídají nějakému kroku šifry [30, 31]. Také si můžeme připravit záznamy běhu různých implementací šifry, spočítat jejich průnik a hledat ho v běhu trasovaného programu [29, s. 10–11].

Tato technika se velice podobá vytváření otisků funkcí. Od otisků použitelných pro hledání v bězích programů vyžadujeme výraznou robustnost, abychom dohledali i různé implementace šifry. Tuto vlastnost může poskytovat například graf toku dat – orientovaný acyklický graf, kde vrcholy obsahují proměnné nebo aritmetické operace a hrany určují vstupy pro danou operaci. Postupným zjednodušováním tohoto grafu získáme otisk, který následně hledáme v zaznamenaném běhu pomocí grafového izomorfismu. [32]

Nezanedbatelnou nevýhodou trasování je jeho náročnost. Pro vykonání jediné instrukce původního programu musíme několikrát předávat řízení systému a programu, který trasování provádí. Oproti standardnímu běhu programu trvá trasování o několik řádů déle [33].



Obrázek 3.1: Vrchní řádek: instrukce ukázkové šifry a z instrukcí vytvořený graf toku kódu. Spodní řádek: zjednodušený otisk kódu a nalezený grafový izomorfismus v průběhu jiného programu. [32]

Kapitola **4**

Návrh detekčního programu

Náš přístup k odhalení šifry AES bude vycházet z dříve popsaného rozpoznávání konstant. Tento přístup selhával v případech, kdy S-Box nebo T-Tabulky nebyly v programu přítomny před spuštěním. Tuto nevýhodu odstraníme tím, že program spustíme a za běhu budeme průběžně hledat tabulky v paměti procesu.

Předpokládáme, že zkoumaný program vždy bude používat S-Boxy nebo T-Tabulky. Programy, které využívají instrukce AES-NI nebo bit slicing, můžeme spustit ve virtualizovaném prostředí, kde tyto instrukce zakážeme [34] a vynutíme použití záložní implementace. Nepředpokládáme existenci programů, které neobsahují zpětně kompatibilní implementaci (využívající tabulky) a ke svému běhu vyžadují vektorová nebo AES-NI rozšíření.

Další nevýhodou přístupu založeném na hledání konstant bylo nesnadné využití nalezené tabulky pro odhalení použitého klíče a dat. Naše řešení bude založené na zaznamenávání přístupů ke konkrétním hodnotám S-Boxu nebo T-Tabulek. Z těchto přístupů následně vypočítáme klíč a data použitá při šifrování.

4.1 Paměťový breakpoint

Pro zaznamenávání přístupů budeme potřebovat *paměťový breakpoint* – funkcionalitu, která nám umožní zastavit běžící proces v momentě čtení nebo zápisu na konkrétní adresu v paměti. Ze zastaveného procesu zjistíme, s jakou adresou pracoval, a poznamenáme si, která hodnota tabulky na této adrese leží. Při trasování jistě zjistíme všechny přístupy do tabulky, ale za cenu, že program po každé vykonané instrukci zastavíme. Naším cílem bude program zastavovat co nejméně, aby mohl běžet co nejrychleji.

Při ladění programu používáme breakpointy, které nám umožňují zastavit program před vykonáním instrukce na určité adrese. Toho je nejčastěji dosaženo pomocí takzvaného softwarového breakpointu – speciální instrukce (INT3 [7]), která je na danou adresu umístěna místo původní instrukce. Jakmi-

le procesor vykoná tuto instrukci, dojde k výjimce, která je předána ladícímu programu. Ten vykonávanou instrukci změní na její původní podobu a tím umožní laděnému programu pokračovat v běhu. Tento postup však pro paměťový breakpoint použít nemůžeme, protože neexistuje žádná speciální hodnota, jejíž čtení by způsobilo zastavení programu.

4.1.1 Hardwarový breakpoint

Ladit program můžeme také pomocí hardwarových breakpointů. Procesory architektury x86 obsahují registry (DR0–7) vyhrazené pro účely ladění. První čtyři registry DR0–3 obsahují námi zvolené adresy breakpointů. Úpravou bitové masky v registru DR7 můžeme nastavit počet bajtů, na kterých breakpoint leží (1, 2, 4 nebo 8 bajtů), při které operaci nad danou adresou dojde k vyvolání výjimky (čtení, zápis nebo spuštění[9]) a zda je daný breakpoint aktivní. [35]

Díky těmto registrům můžeme vytvořit malé paměťové breakpointy – celkem můžeme pokrýt 32 bajtů paměti. To nám ale na 256bajtové S-Boxy nestačí, o T-Tabulkách nemluvě.

4.1.2 Stránkování

Procesory z rodiny x86 podporují stránkování paměti – vlastnost, která operačním systémům umožňuje spouštět procesy v oddělených paměťových prostorech. Každý proces si myslí, že má k dispozici celý (virtuální) paměťový prostor. Ve skutečnosti jsou jednotlivé části tohoto prostoru namapovány na různé části fyzického adresového prostoru bez jakéhokoliv vztahu mezi adresami v těchto prostorech. Různé části virtuální paměti také mohou být uloženy na disku a k jejich načtení do fyzické paměti dojde až při přístupu. Velké množství částí není namapováno nikam a přístup na příslušné adresy způsobí výjimku. Stránkování také umožňuje odebrat procesu právo na zápis, čtení, nebo spuštění kódu z dané části. [36]

Tyto části se označují jako stránky a na procesorech architektury x86 mohou nabývat různých velikostí – 4 KiB, 2 MiB, 4 MiB a 1 GiB. Výchozí velikost stránek jsou 4 KiB, jak na systémech Microsoft Windows [37], tak na unixech [38]. Odstraněním patřičných práv na dané stránce vytvoříme paměťový breakpoint o velikosti 4 KiB. Proces provede nad danou adresou akci, na kterou nemá práva, a dojde tak k vyvolání výjimky. Tuto výjimku zachytí operační systém a buďto program ukončí, nebo rozhodnutí deleguje jinému, ladícímu procesu. V tuto chvíli si můžeme poznamenat, s jakou adresou se pracovalo, obnovit původní práva a předat řízení dříve zastavenému procesu.

Paměťové breakpointy využívající stránkování jsou vhodné pro naše použití. Zatímco S-box zabere pouze jednu šestnáctinu stránky, čtyři T-Tabulky

[9] Při standardním nastavení procesoru můžeme vybírat mezi těmito událostmi: pouze čtení instrukce / pouze zápis / zápis nebo čtení dat. [35]

v ideálním případě zaplní celou stránku. Běžně se stává, že tabulka není zarovnána přesně dle hranic stránky a dochází tak k přístupům do míst stránky, kde žádná tabulka není. V takovém případě nic zaznamenávat nebudeme a programu umožníme přístup provést.

4.2 Rekonstrukce klíče

První přístupy do S-Boxu budou provedeny během expanze klíče a budou odpovídat bajtům vstupujícím do funkce SubWord (viz sekce 1.2.1). Tímto způsobem odhalíme každý čtvrtý (AES-128, AES-256) nebo šestý (AES-192) sloupec prodlouženého klíče. Budeme předpokládat, že:

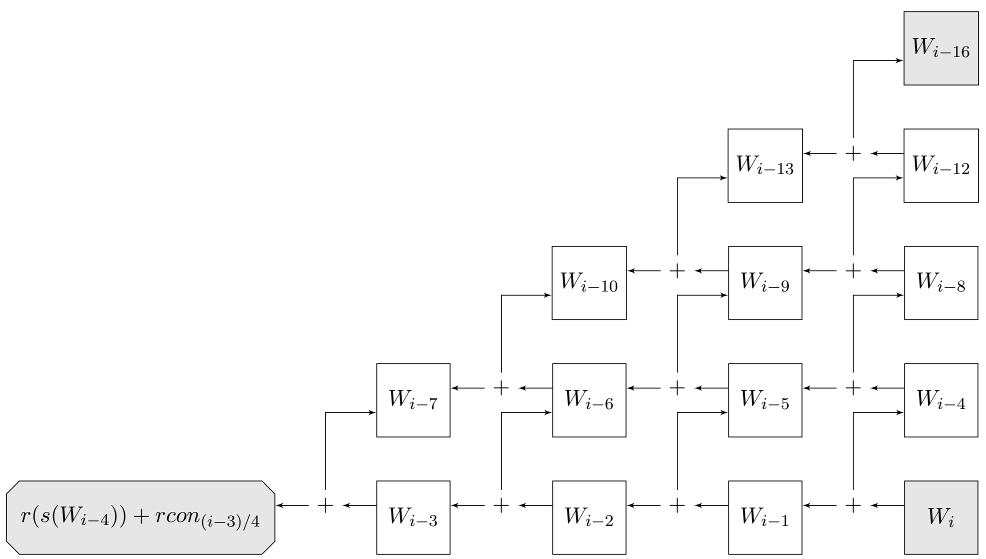
- Během expanze klíče se k tabulkám přistupuje pouze prostřednictvím funkce SubWord.
- Volání této funkce jsou uspořádána stejně jako do ní vstupující sloupce.
- Přístupy uskutečněné v rámci různých volání SubWord nejsou navzájem promíchány.

Pořadí jednotlivých nahrazovaných bajtů ve slově zaručené nemáme. Každé volání funkce SubWord může způsobit přístupy v různém pořadí. Takové chování většinou není programátorem zamýšleno, ale použitím agresivních optimalizací při překladu kódu (rozbalení cyklů, přeházení pořadí instrukcí) k němu může jednoduše dojít.

Prvním krokem k úspěšné rekonstrukci klíče ze zachycených přístupů do tabulky S-Boxu bude uspořádání bajtů do sloupců. K tomu využijeme závislostí, které mezi sloupci existují. Jako závislost označíme výraz vytvořený opakovaným použitím předpisu pro výpočet expandovaného klíče, který obsahuje pouze známé sloupce, neboli slova vstupující do funkce SubWord. Tyto výrazy nalezneme zvlášť pro každou délku klíče.

4.2.1 128bitový klíč

V případě šifry AES-128 budou závislost tvořit sloupce s indexy z množiny $\{x \mid x = 3 + 4k, k \in \mathbb{N}\}$. Rozepíšeme, na kterých předchozích částech klíče závisí hodnota sloupce W_i, kde i patří do dříve zmíněné množiny.



Obrázek 4.1: Závislost známých sloupců při expanzi 128bitového klíče. Šedé uzly tvoří výslednou rovnici. Bílé uzly byly expandovány nebo vykráceny.

$$
\begin{aligned}
W_i = & W_{i-4} + W_{i-1} = \\
= & W_{i-8} + W_{i-5} + W_{i-5} + W_{i-2} = \\
= & W_{i-8} + W_{i-2} = \\
= & W_{i-12} + W_{i-9} + W_{i-6} + W_{i-3} = \\
= & W_{i-16} + W_{i-13} + W_{i-13} + W_{i-10} + W_{i-10} + W_{i-7} + W_{i-7} + \\
& + r(s(W_{i-4})) + rcon_{(i-3)/4} = \\
= & W_{i-16} + r(s(W_{i-4})) + rcon_{(i-3)/4}
\end{aligned}
$$

Při úpravách jsme využili vlastností sčítání v GF(2^8), které jsme aplikovali na jednotlivé bajty sloupce. Sečtením sloupce se sebou samým tak vznikne nulový sloupec. Členy W_{i-4}, W_{i-8} a W_{i-12} jsme expandovali, abychom se zbavili jiných členů s indexy mimo námi zadanou množinu.

Pro použití námi nalezené závislosti potřebujeme alespoň pět po sobě jdoucích známých sloupců. Z definice expanze klíče pro délku klíče 128 bitů vyplývá, že do funkce SubWord vstoupí 10 slov z expandovaného klíče. Můžeme tedy připravit šest navzájem odlišných rovnic, které vyjádří veškeré vazby mezi známými sloupci.

$$
\begin{aligned}
W_{19} &= W_3 + r(s(W_{15})) + rcon_4 \\
&\vdots \\
W_{39} &= W_{23} + r(s(W_{35})) + rcon_9
\end{aligned}
$$

Pokud data získaná sledováním přístupů do S-Boxu nesplňují podmínky zadané touto soustavou rovnic, nemůže se jednat o přístupy prováděné během expanze klíče funkcí SubWord, nebo došlo k porušení námi dříve stanovených předpokladů. Tohoto pozorování využijeme při hledání konkrétního uspořádání nasledovaných bajtů do sloupců expandovaného klíče.

Pro taková uspořádání bajtů, pro která je tato soustava podmínek splněna, vypočítáme zbytek klíče. Pomocí předpisu pro expanzi klíče jsme schopni kompletně rekonstruovat použitý klíč.

Součtem dvojic známých sloupců $(W_i, W_{i+4}), i \in \{3, 7, 11, \dots, 35\}$ získáme sloupce W_{i+3}. Z nově získaných sloupců znovu utvoříme dvojice a tento výpočet ještě dvakrát zopakujeme. Po těchto výpočtech „zprava doleva" nám zůstanou neznámé pouze sloupce $\{W_0, W_1, W_2, W_4, W_5, W_8, W_{40}, \dots, W_{43}\}$. Hodnotu posledních čtyř sloupců zjistíme jednoduše z definice, jelikož ostatní hodnoty potřebné pro výpočet již známe. Sloupce $W_i, i \in \{0, 4, 8\}$ získáme součtem $W_i = W_{i+4} + r(s(W_{i+3})) + rcon_{(i+4)/4}$. Poslední zbývající neznámé s indexy $i \in \{1, 2, 5\}$ opět dopočítáme z dříve zjištěných hodnot: $W_i = W_{i+3} + W_{i+4}$. Vždy postupujeme v takovém pořadí, abychom měli k dispozici všechny potřebné sloupce.

4.2.2 192bitový klíč

Expandovaný klíč pro šifru AES-192 obsahuje jiné závislosti mezi známými sloupci. Tentokrát se budeme snažit získat výraz obsahující sloupce pouze s indexy z množiny $\{x \mid x = 5 + 6k, k \in \mathbb{N}\}$.

$$
\begin{aligned}
W_i =& W_{i-6} + W_{i-1} = \\
=& W_{i-12} + W_{i-7} + W_{i-7} + W_{i-2} = \\
=& W_{i-12} + W_{i-2} = \\
=& W_{i-18} + W_{i-13} + W_{i-8} + W_{i-3} = \\
=& W_{i-24} + W_{i-19} + W_{i-19} + W_{i-14} + W_{i-14} + W_{i-9} + W_{i-9} + W_{i-4} = \\
=& W_{i-24} + W_{i-4} = \\
=& W_{i-30} + W_{i-25} + W_{i-10} + W_{i-5} = \\
=& W_{i-36} + W_{i-31} + W_{i-31} + W_{i-26} + W_{i-16} + W_{i-11} + W_{i-11} + \\
& + r(s(W_{i-6})) + rcon_{(i-5)/6} = \\
=& W_{i-36} + W_{i-26} + W_{i-16} + r(s(W_{i-6})) + rcon_{(i-5)/6} = \\
=& \cdots + W_{i-36} + W_{i-16} + W_{i-12} + W_{i-36} + W_{i-26} + W_{i-24} = \\
=& W_{i-36} + W_{i-24} + W_{i-12} + r(s(W_{i-6})) + rcon_{(i-5)/6}
\end{aligned}
$$

Na předposledním řádku jsme přičetli dvě trojice sloupců, jejichž součet je nulový sloupec. Tyto trojice odpovídají rovnostem na šestém a třetím řádku v upravovaném výrazu. Díky této úpravě jsme získali výraz obsahující pouze sloupce jejichž hodnota je nám známa.

Při expanzi klíče o 192 bitech do funkce SubWord vstoupí osm slov. Nalezená závislost vyžaduje sedm po sobě jdoucích známých sloupců. Soustava rovnic všech vazeb mezi známými sloupci proto bude obsahovat pouze dvě rovnice:

$$W_{41} = W_5 + W_{17} + W_{29} + r(s(W_{35})) + rcon_6$$
$$W_{47} = W_{11} + W_{23} + W_{35} + r(s(W_{41})) + rcon_7$$

Soustavu opět použijeme pro rozhodnutí, zdali se jedná o expanzi klíče pro AES-192, a jak správě do sloupců uspořádat jednotlivé bajty z napozorovaných přístupů. Při výpočtu zbývajících částí klíče postupujeme podobně jako v předchozím případě.

4.2.3 256bitový klíč

Expanze klíče pro AES-256 se výrazně odlišuje od předchozích případů. Do funkce SubBytes vstupuje nejen každé osmé slovo, ale také všechna slova s indexem z množiny $\{x \mid x = 11 + 8k, k \in \mathbb{N}\}$. Celá množina indexů známých sloupců je díky tomu $\{x \mid x = 7 + 4k, k \in \mathbb{N}\}$. Nejprve nalezneme závislost sloupce W_i na ostatních známých sloupcích, kde $i = 7 + 8k, k \in \mathbb{N}$.

$$\begin{aligned} W_i =& W_{i-8} + W_{i-1} = \\ =& W_{i-16} + W_{i-9} + W_{i-9} + W_{i-2} = \\ =& W_{i-16} + W_{i-2} = \\ =& W_{i-24} + W_{i-17} + W_{i-10} + W_{i-3} = \\ =& W_{i-32} + W_{i-25} + W_{i-25} + W_{i-18} + W_{i-18} + W_{i-11} + W_{i-11} + s(W_{i-4}) \\ =& W_{i-32} + s(W_{i-4}) \end{aligned}$$

Podobně odhalíme závislost i pro $i = 11 + 8k, k \in \mathbb{N}$.

$$\begin{aligned} W_i =& W_{i-8} + W_{i-1} = \\ & \dots \\ =& W_{i-32} + W_{i-25} + W_{i-25} + W_{i-18} + W_{i-18} + W_{i-11} + W_{i-11} + \\ & + r(s(W_{i-4})) + rcon_{(i-3)/8} \\ =& W_{i-32} + r(s(W_{i-4})) + rcon_{(i-3)/8} \end{aligned}$$

Naplníme soustavu rovnic závislostmi mezi známými sloupci.

$$
\begin{aligned}
W_{39} &= W_7 + s(W_{35}) \\
W_{47} &= W_{15} + s(W_{43}) \\
W_{55} &= W_{23} + s(W_{51}) \\
W_{43} &= W_{11} + r(s(W_{39})) + rcon_5 \\
W_{51} &= W_{19} + r(s(W_{47})) + rcon_6
\end{aligned}
$$

Obě předchozí soustavy obsahovaly všechny známé sloupce. V tomto případě do funkce SubBytes vstoupilo 13 slov, do soustavy se ale dostalo pouze 11 navzájem různých proměnných. Hodnotu sloupců W_{27} a W_{31} známe, ale neumíme ji vyjádřit pomocí jiných známých sloupců. Tyto podmínky stále můžeme použít pro odlišení přístupů do S-Boxu během expanze 256bitového klíče od jiných operací.

Pro určení konkrétního uspořádání bajtů ve sloupcích nemáme dostatek informace. V obou nevázaných sloupcích můžeme volit z $4! = 24$ permutací bajtů. Ze vstupních dat tak můžeme rekonstruovat nejméně $24^2 = 576$ různých klíčů, pro které bude soustava závislostí splněna. K tomuto problému s neuspořádanými klíči se vrátíme během rekonstrukce dat, kdy využijeme informace předané během šifrování pro nalezení správného uspořádání klíče.

Jakmile budeme mít k dispozici správné uspořádání, opět využijeme definici expanze klíče pro dopočítání všech ostatních sloupců celého klíče, obdobně jako v předchozích případech.

4.3 Rekonstrukce dat

Prováděním jednotlivých rund šifry bude docházet k přístupům do S-Boxu nebo T-Tabulek v rámci kroku substituce (SubBytes), viz sekce 1.2.3. Pro odhalení šifrovaných dat si stanovíme následující předpoklady:

- Data jsou šifrována klíčem, který byl při aktuálním běhu programu expandován a rekonstruován dříve popsaným postupem.
- Veškeré přístupy do tabulek jsou prováděny pouze prostřednictvím funkce SubBytes.
- Volání dané funkce jsou uspořádána stejně jako jednotlivé rundy.
- Přístupy provedené během různých rund nejsou navzájem promíchány.

Obdobně jako při rekonstrukci klíče nemáme zaručené pořadí jednotlivých přístupů provedených v rámci jedné rundy. Každé volání funkce SubBytes může způsobit přístupy v různém pořadí, ať už z vůle programátora, či překladače.

4.3.1 Runda z pohledu substitučního kroku

První stav šifry, který do funkce SubBytes vstupuje, je pouhý součet dat s prvním rundovním klíčem. Z funkce vystoupí stav, z jehož bajtů patřičným způsobem (posun řádků, promíchání sloupců, přičtení rundovního klíče) připravíme vstupní stav pro další volání funkce SubBytes. Takto můžeme pohlížet na proces, který probíhá po zbytek rund. Z definice šifry AES přesně víme, jak tento proces vypadá. Díky předpokladům také známe použitý klíč. Pokud před šifrováním došlo k expanzi více klíčů, vyzkoušíme všechny odhalené klíče a uvidíme, který z nich bude ke zpracovávaným přístupům pasovat. Poslední chybějící informací je uspořádání bajtů v jednotlivých stavech šifry.

Zaměříme se na vztah vstupního stavu S některého z volání funkce SubBytes a výstupního stavu T z předchozího volání. Pro správně uspořádané stavy platí, že:

$$S = \text{MixColumns}(\text{ShiftRows}(T)) + K$$

Kde K je patřičný rundovní klíč. Nám předané stavy ale uspořádané být nemusí, proto tento vztah přeformulujeme a zrelaxujeme:

$$\forall x, x \in \text{InvMixColumns}(S + K) \iff x \in T$$

Rovnost jsme vyměnili za existenci bijektivního zobrazení mezi dvěma množinami. V tuto chvíli bychom mohli začít prohledávat všech 16! možných permutací stavu S a zkoumat, jestli danou podmínku splňuje. My si tuto zbytečně náročnou práci ušetříme tím, že danou podmínku budeme ještě více relaxovat a zaměříme se na jednotlivé sloupce.

$$\forall x, x \in \text{InvMixColumns}(S_i + K_i) \implies x \in T$$

Díky této úpravě můžeme řešit tento problém pro každý ze sloupců S_i nezávisle. V součtu prohledáme $4 \times \frac{16!}{(16-4)!}$ variací. Pro každý výběr takto upravených sloupců zkontrolujeme, že jsou všechny jeho bajty obsaženy ve stavu T. Pokud nenajdeme jediný výběr sloupců, pro který by byla tato podmínka splněna, s jistotou víme, že se nejedná o přístupy provedené při šifrování daným klíčem.

Tuto podmínku budeme postupně skládat a aplikovat na vztahy mezi stavy od poslední rundy k první. Řekněme, že dříve popsaná podmínka se stavy S a T se týkala dvou posledních rund. Navážeme podmínkou se stavy U a V, kde U je vstupní stav, neboli InvSubBytes(T), a V je výstupní stav třetího volání od konce. Patřičný rundovní klíč označíme písmenem L.

$$\forall x, x \in \text{InvMixColumns}(U_i + L_i) \implies x \in V$$

Dosadíme za proměnnou U. Pro stručnost se omezíme pouze na levou stranu implikace.

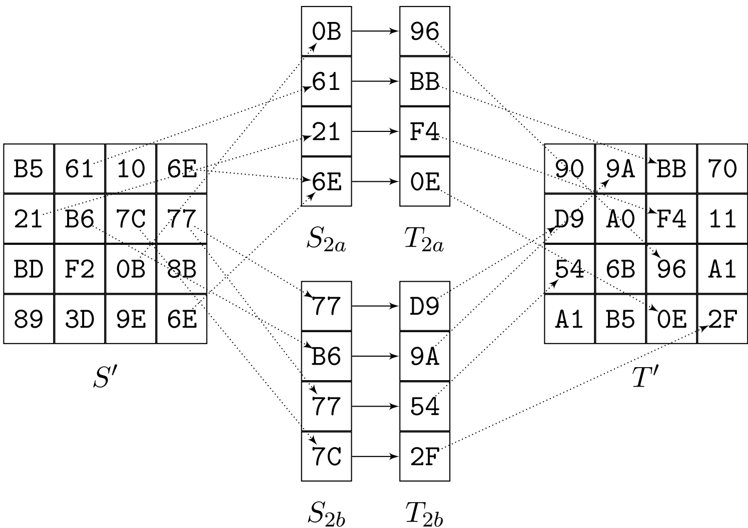
$$\text{InvMixColumns}(\text{InvSubBytes}(T)_i + L_i)$$

Stav T vyjádříme pomocí stavu S:

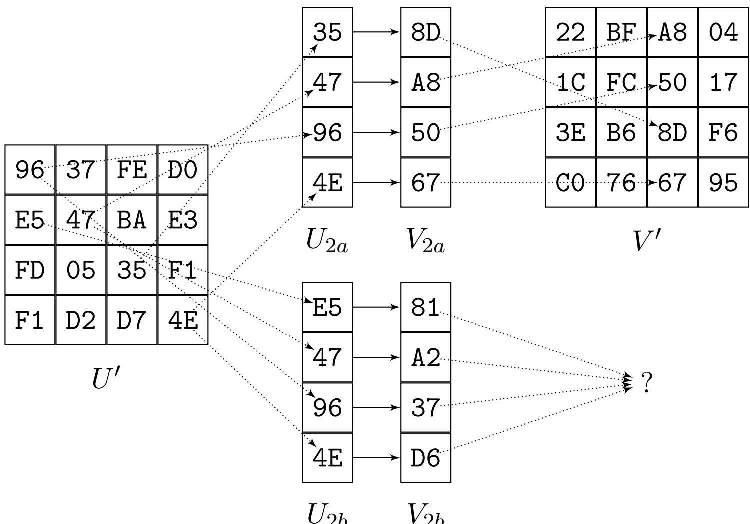
$$\text{InvMixColumns}(\text{InvSubBytes}(\text{InvShiftRows}(\text{InvMixColumns}(S + K))_i + L_i)$$

Získali jsme podmínku vztahující se na stavy S a V. Pro n rund tímto procesem získáme $n - 1$ podmínek, které nám umožní zredukovat výběr možných uspořádání bajtů do stavů. Správné uspořádání stavu musí splňovat všechny tyto podmínky. Po nalezení správného uspořádání bajtů vstupního stavu prvního volání funkce SubBytes k tomuto stavu přičteme první rundovní klíč a získáme šifrovaná data. Také můžeme uspořádat poslední výstupní stav, dokončit zbývající kroky šifry (posun řádků a přičtení posledního rundovního klíče) a získat tak data v zašifrované podobě.

Při odhalování dešifrovaných dat vyměníme vstupní a výstupní stavy funkce InvSubBytes a obrátíme pořadí rund. Dále pokračujeme stejně jako při odhalování šifrovaných dat. Opět můžeme získat zašifrovaná i rozšifrovaná data.



Obrázek 4.2: Zleva doprava: Neuspořádaný vstupní stav S' poslední rundy, dva způsoby výběru sloupce S_2, sloupec T_2 získaný přičtením patřičného sloupce rundovního klíče a inverzním promícháním, neuspořádaný výstupní stav T' předposlední rundy.



Obrázek 4.3: V předchozím obrázku 4.2 jsme určili bajty `96` a `D9` ze stavu T jako dva možné kandidáty pro první bajt třetího sloupce. Inverzní substitucí získáváme bajty `35` a `E5` na stejných pozicích stavu U. Ty použijeme při výběru sloupce U_2. Patřičnou úpravou získáváme sloupec V_2, ale pouze v jednom z případů nacházíme potřebné bajty ve stavu V. Nyní víme, že pouze vrchní volby sloupců z obou obrázků splňují naše podmínky.

Na obrázcích 4.2 a 4.3 je předvedena část popsaného procesu. K ukázce byl použit jeden z testovacích vektorů ze standardu šifry AES [4, s. 35–36]. Jedná se o šifrování dat {`0011...EEFF`} pomocí AES-128 s klíčem {`0001...0E0F`}. Námi použité stavy odpovídají řádkům s těmito jmény:

$$S = \texttt{round[10].start}$$
$$T = \texttt{round[9].s_box}$$
$$U = \texttt{round[9].start}$$
$$V = \texttt{round[8].s_box}$$

4.3.2 Uspořádání klíče

Ne vždy se nám podaří z přístupů při expanzi klíče rekonstruovat jediný konkrétní klíč. Například u 256bitových klíčů vždy rekonstruujeme větší množinu klíčů, která odpovídá daným přístupům. Abychom nemuseli dříve popsaný postup zkoušet pro všechny klíče z této množiny, budeme aplikovat stejné podmínky, ale pouze na čtvrté sloupce stavů a rundovních klíčů.

$$\forall x, x \in \text{InvMixColumns}(S_3 + K_3) \implies x \in T$$

Důvod pro toto omezení je ten, že o čtvrtých sloupcích klíčů máme nejvíce informací. Například u klíče pro AES-256 známe hodnoty bajtů čtvrtého sloupce všech rundovních klíčů mimo první a poslední klíč – tyto okrajové klíče ale nejsou v podmínkách nikdy využity. U AES-128 známe všechny čtvrté sloupce kromě posledního. U AES-192 známe čtvrté sloupce každé třetí rundy, pro zbylé rundy musíme daný sloupec dopočítat.

Konkrétní uspořádání bajtů do sloupce klíče znát nemusíme, proto při řešení podmínek budeme vybírat nejen variace do čtvrtého sloupce stavu, ale i permutace do čtvrtého sloupce rundovního klíče. V lepším případě jsme během rekonstrukce klíče omezili možný výběr permutací pomocí závislostí. V nejhorším případě budeme muset prohledat $\frac{16!}{(16-4)!} \times 4!$ možností[10].

Tímto způsobem jsme schopni zredukovat množinu klíčů, které mohou odpovídat nějakým přístupům. Pokud nedojde ke splnění podmínek, určitě se nejedná o klíč použitý pro dané šifrování. Při splnění podmínek vyzkoušíme rekonstrukci dat se zredukovanou množinou možných klíčů a pokusíme se tak odhalit dvojici šifrovaných dat a použitého klíče.

[10] V případě klíče pro AES-192 to může být až $4! \times$ více, protože některé sloupce jsou vypočítávány ze dvou jiných sloupců.

Kapitola 5

Implementace detekčního programu

Pro implementaci byl zvolen jazyk C++. Jádro je tvořeno platformově nezávislou knihovnou LibAesSniffer, která má na starosti nalezení S-Boxů v paměti a rekonstrukci klíče a dat z přístupů do těchto substitučních tabulek. Samotné zajištění přístupu do paměti sledovaného procesu a zaznamenávání přístupů do tabulek je vyčleněno do platformově závislého programu, který knihovnu využívá. Tento program může nabývat různých podob v závislosti na operačním systému a zamýšleném způsobu použití, například:

- Běžný uživatelský program, který provádí ladění sledovaného programu.
- Rozšíření existujícího ladicího nástroje.
- Knihovna, která je načtena do sledovaného programu.
- Rozšíření operačního systému.

Varianta zvolená pro tuto práci je běžný uživatelský program StandaloneAesSniffer, který provádí ladění sledovaného programu, na systému Microsoft Windows. Jedná se o konzolovou aplikaci, které pomocí argumentů na příkazové řádce předáme jméno programu, který chceme sledovat. Aplikace spustí cílový program a po dobu jeho běhu vypisuje na standardní výstup informace o nalezených tabulkách, klíčích a datech.

Například sledování šifrování v průběhu spojení se vzdáleným ssh serverem pomocí klienta PuTTY uskutečníme následujícím příkazem:

```
standaloneAesSniffer.exe putty.exe -ssh fray1.fit.cvut.cz
```

```
class Memory {
public:
  struct Chunk {
    size_t length;
    uintptr_t address;
    std::vector<uint8_t> data;
  };
  virtual bool HasNext() const = 0;
  virtual const Chunk& Next() = 0;
  virtual void Restart() = 0;
};

class SBoxFinder {
public:
  using OnSBoxDiscovered = std::function<void(
      uintptr_t address, int size, bool inverse)>;
  SBoxFinder(Memory& memory, OnSBoxDiscovered
      onSBoxDiscovered, bool thisProcess = false);
  void Start();
  void Scan();
  void Stop();
};
```

Obrázek 5.1: Část rozhraní uveřejněného knihovnou LibAesSniffer. Třída Memory umožňuje procházet paměťový prostor procesu. Funkce Next vrátí část dat z paměti, včetně jejich adresy a velikosti. Opakovaným voláním získáme veškerá čitelná data procesu. Pomocí funkce HasNext si můžeme ověřit, zdali už jsme prošli celý prostor, a případně pomocí Restart vymazat informaci o pozici a začít znovu od začátku. Třída je abstraktní a její implementaci má na starosti uživatel (platformově závislý program).

Instance konkrétní implementace je předána knihovně v rámci konstrukce objektu SBoxFinder. Tento objekt uživateli pomocí funkce onSBoxDiscovered oznamuje adresy, velikosti a typy nalezených tabulek. Volání funkce Scan způsobí jednorázové prohledání paměti, Start vytvoří nezávislé vlákno pro kontinuální prohledávání. V případě, kdy se samotná knihovna nachází v paměťovém prostoru sledovaného procesu, vyjmeme tabulky uložené v knihovně z výsledků hledání pomocí parametru thisProcess.

```
class Reassembler {
public:
  struct SBoxAccess {
    uint8_t index;
    std::chrono::system_clock::time_point time = std::
        chrono::system_clock::now();
  };
  using History = std::vector<SBoxAccess>;
  struct DataEvent {
    enum class Type {
      UnknownAccess,
      Key,
      Encryption,
      Decryption
    } type;
    std::vector<uint8_t> key;
    std::vector<uint8_t> dataIn;
    std::vector<uint8_t> dataOut;
    History forwardHistory;
    History inverseHistory;
  };

  Reassembler(bool aes128, bool aes192, bool aes256);
  void ReportAccess(int index, int size, bool inverse);
  void ClearAccessHistory();
  virtual void OnDataEvent(const DataEvent& event)
     const = 0;
};
```

Obrázek 5.2: Zbývající část rozhraní knihovny LibAesSniffer – třída Reassembler. Při vytváření instance určíme varianty šifry AES, které nás zajímají. Voláním funkce ReportAccess předáváme knihovně jednotlivé přístupy do sledovaných tabulek. Jakmile knihovna odhalí expanzi klíče nebo (de)šifrování dat, je uživateli tato informace předána voláním jím implementované funkce OnDataEvent.

Předaná struktura DataEvent obsahuje typ nastalé události, použitý klíč, vstupní a výstupní data a seznam jednotlivých přístupů – jejich indexy a časy provedení přístupu. Pokud knihovna nerozpozná určitou posloupnost přístupů, předá je uživateli také pomocí této funkce a jako typ uvede UnknownAccess. Po skončení běhu sledovaného programu uživatel zavolá metodu ClearAccessHistory, kterou vynutí předání zbývajících nerozpoznaných posloupností.

5.1 Ladicí rozhraní

V programu je použito systémové rozhraní pro práci s procesy [39], jejich ladění [40] a práci s pamětí [41]. Nejprve využijeme funkci CreateProcess s příznakem DEBUG_PROCESS pro spuštění sledovaného procesu a zaregistrování našeho procesu jako debuggeru pro spuštěný proces. To nám v budoucnu umožní použít funkce WaitForDebugEvent a ContinueDebugEvent pro sledování a zpracovávání událostí v daném procesu (například zastavení programu při nalezení breakpointu nebo výpadku stránky).

5.2 Nalezení S-Boxů

Prohledávání paměti je zajištěno funkcí VirtualQueryEx, pomocí které získáme kompletní přehled o využití paměti v daném procesu – adresy a rozsahy stránek včetně jejich oprávnění. Obsah těchto stránek překopírujeme do paměťového prostoru našeho programu pomocí ReadProcessMemory. V tuto chvíli práci předáme knihovně LibAesSniffer, které překopírovaná data prohledá za účelem nalezení S-Boxů a T-Tabulek. Tyto tabulky mohou nabývat různých podob (viz sekce 2.2). Nebudeme proto hledat celé předpočítané tabulky, ale pouze násobky hodnot S-Boxu v různých rozestupech. V T-Tabulkách najdeme například trojnásobky hodnot S-Boxu v čtyřbajtových rozestupech. Pokud použijeme překryté T-Tabulky s osmibajtovými slovy, rozestupy se prodlouží na osm bajtů.

Hledat budeme jedno, dvoj a trojnásobky hodnot S-Boxu a jedno, devíti, jedenácti, třinácti a čtrnáctinásobky inverzního S-Boxu v intervalech po jednom, dvou, čtyřech a osmi bajtech. Takto jednoduše pokryjeme celou škálu substitučních tabulek. Adresy a velikosti nalezených tabulek jsou předány zpět platformově závislé části programu za účelem zahájení sledování přístupů.

Knihovna neustále prohledává paměť sledovaného procesu v odděleném vlákně. Díky tomu objeví i tabulky, které nejsou k dispozici před spuštěním programu – například pokud byl program zabalen. Tabulky také mohou být generovány až za běhu programu, nebo mohou být umístěny v oddělené dynamicky linkované knihovně. I tyto tabulky odhalí.

5.3 Paměťový breakpoint

Přístupy do nalezených tabulek jsou sledovány pomocí paměťového breakpointu popsaného v sekci 4.1.2. Systémové rozhraní umožňuje manipulaci s přístupovými právy stránek pomocí funkce VirtualProtectEx. Přidáním příznaku PAGE_GUARD dočasně odebereme veškerá práva [42]. Při přístupu k takto označené stránce dojde k zastavení programu, odebrání příznaku a obnovení předchozích práv. Tuto událost zachytíme ve funkci WaitForDebugEvent, kde se dozvíme, na kterou adresu a jakým způsobem (čtení, zápis, spuštění) se

snažilo vlákno sledovaného procesu přistoupit. Adresu přístupu porovnáme s adresami a rozměry tabulek a v případě zásahu tabulky předáme tuto informaci knihovně.

Nyní musíme provést jediný krok sledovaného programu. Pokud bychom obnovili příznak PAGE_GUARD a obnovili běh procesu, došlo by k pokusu o vykonání stejné instrukce a opětovnému zastavení programu. Abychom předešli tomuto nekonečnému cyklu, musíme příznak stránky nastavit až po provedení dané instrukce.

V registru příznaků FLAGS pro dané vlákno aktivujeme *Trap Flag*. Pro tuto akci použijeme dvojici funkcí GetThreadContext a SetThreadContext. Tato změna způsobí, že po spuštění zastaveného programu bude provedena jediná instrukce a následně dojde k okamžitému zastavení. Tuto událost opět odchytíme, obnovíme příznak PAGE_GUARD a program spustíme. Trap Flag nemusíme rušit, podobně jako příznak stránky je odstraněn automaticky při jeho použití. Zastavením programu po provedení instrukce byl automaticky deaktivován.

5.3.1 Smíšené stránky

Popsaným postupem vždy získáme první přístup do sledované stránky pro každou instrukci. To je ve většině případů dostačující. Pokud jsou však tabulky umístěny přímo v kódovém segmentu vedle samotných instrukcí programu [43], může dojít k různým nepříjemným situacím. Zdrojem komplikací jsou instrukce umístěné ve sledované stránce. Když dojde k pokusu o jejich vykonání, je program zastaven a aplikujeme výše popsaný postup. Prováděné instrukce mohou dále přistupovat k paměti. Pokud je přistupováno mimo sledované stránky, vše funguje dle očekávání a přístup způsobený předchozím načtením instrukce je ignorován, neboť spouštěný kód není součástí žádné tabulky.

Pokud instrukce přistupuje k jiné sledované stránce, jejím vykonáním dojde k další události a shození příznaku na dané stránce. Aplikací dříve popsaného postupu získáme adresu do paměti, kam daná instrukce přistoupila. V případě, že adresa ukazuje dovnitř některé z tabulek, předáme ji knihovně. Následně nesmíme zapomenout obnovit oba příznaky PAGE_GUARD.

Poslední variantou je instrukce přistupující do stránky, ve které se sama nachází. Vykonáním takové instrukce nedojde k očekávané události, protože v době provádění instrukce je příznak na dané stránce neaktivní. Nezbývá nám jiná možnost, než tuto instrukci dekódovat a adresu cíle vypočítat z registrů aktuálního vlákna. Adresu opět porovnáme vůči seznamu nalezených S-Boxů a případně zaznamenáme. Pro účely dekódování instrukcí byla použita knihovna Zydis [44].

5.4 Zpracování přístupů

Práci s přístupy má na starosti platformově nezávislé jádro – knihovna LibAesSniffer. Ta obsahuje dvě fronty, jednu pro přístupy do S-Boxu a druhou pro přístupy do inverzního S-Boxu. Přístupy do T-Tabulek jsou jednoduše přepočítány na ekvivalentní přístupy do S-Boxů. Jakmile fronta obsahuje dostatečné množství přístupů do dopředného S-Boxu, dojde k pokusu o rekonstrukci klíče.

Nejméně přístupů potřebujeme pro AES-192 klíč, kdy z expandovaného klíče o 52 sloupcích do funkce SubWord vstupuje každé šesté slovo – celkem osm slov, neboli 32 přístupů. Expandovaný klíč pro AES-128 má 44 sloupců a substituované je každé čtvrté slovo, kromě posledního, které už není využito k výpočtu jiných sloupců. Pro rekonstrukci využijeme 10 sloupců, 40 přístupů. AES-256 používá expandovaný klíč o 60 sloupcích a k substituci dochází u každého čtvrtého slova, s výjimkou prvního a posledního případu – celkem potřebujeme 13 slov, 52 přístupů.

Jakmile máme k dispozici rekonstruovaný klíč pro danou variantu šifry, můžeme přejít k rekonstrukci dat. Při šifrování 128bitovou variantou je krok SubBytes proveden 11krát, dojde tedy k 160 přístupům do dopředného nebo inverzního (v případě dešifrování) S-Boxu. AES-192 způsobí 192 přístupů, AES-256 224 přístupů.

Pokud je k dispozici více přístupů než je pro rekonstrukci nutné, je vždy použita nejčerstvější část. Při úspěšné rekonstrukci jsou přebytečné přístupy zahozeny. O nalezeném klíči, datech i zahozených přístupech je uživatel informován. Ještě než sledovaný program dokončí substituční krok poslední rundy šifrování, uživatel už zná šifrovaná data společně s použitým klíčem.

Kapitola 6

Testování

Pro účely testování jsme připravili virtuální stroj s 32bitovou verzí Microsoft Windows 7, Service Pack 1. Aby došlo k úspěšnému odhalení šifry, pozměnili jsme chování instrukce CPUID. Příznaky prozrazující dostupnost AES-NI instrukcí, případně některých vektorových rozšíření, jsme vymaskovali. Programům spuštěným na operačním systému hosta jsme přítomnost těchto rozšíření tímto způsobem zatajili. Pro virtualizaci jsme využili program Oracle VM VirtualBox[11] a změny chování instrukce CPUID jsme dosáhli pomocí příkazu `vboxmanage modifyvm --cpuidset`.

6.1 Knihovny

Funkčnost vytvořeného programu jsme ověřili na běžně používaných kryptografických knihovnách poskytujících šifru AES. Pro tyto účely jsme vybrali následující knihovny:

- OpenSSL[12]
- CryptoPP[13]
- Botan[14]
- Wincrypt[15]

OpenSSL je notoricky známá knihovna, jednoduše použitelná programy napsanými v jazyce `C`. CryptoPP a Botan jsou knihovny určené pro použití skrze jazyk `C++`. Wincrypt je kryptografické rozhraní poskytované systémem Microsoft Windows.

[11] https://www.virtualbox.org/
[12] https://www.openssl.org/
[13] https://www.cryptopp.com/
[14] https://botan.randombit.net/
[15] https://docs.microsoft.com/en-us/windows/desktop/api/wincrypt/

Pro každou z těchto knihoven byl vytvořen jednoduchý program, který provede šifrování bloku {0011...EEFF} 128bitovým klíčem {0001...0E0F}, 192bitovým klíčem {0001...1617} a 256bitovým klíčem {0001...1E1F}. Zašifrované bloky následně dešifruje stejnými klíči.

6.1.1 OpenSSL

Test s knihovnou OpenSSL verze 1.0.2q jsme provedli na dříve zmíněném virtuálním stroji se skrytými instrukcemi AES-NI a vektorovými rozšířeními SSE a SSSE3. Využitím našeho programu jsme úspěšně odhalili všechny použité klíče a šifrovaná i dešifrovaná data.

```
Found SBox at address 0x0000000068D929C0, size 2048,
    region 0x0000000068D92000 - 0x0000000068D94000
Found SBox at address 0x0000000068D931C0, size 256,
    region 0x0000000068D93000 - 0x0000000068D94000
...
Recovered key: 000102030405060708090A0B0C0D0E0F
First access: 2019-04-08 08:07:08.581
Last access: 2019-04-08 08:07:09.066

Unknown access!
SBox: 0020406080A0C0E0
InvSBox:
First access: 2019-04-08 08:07:10.206
Last access: 2019-04-08 08:07:10.206

Recovered encryption data:
Input data: 00112233445566778899AABBCCDDEEFF
Output data: 69C4E0D86A7B0430D8CDB78070B4C55A
Used key: 000102030405060708090A0B0C0D0E0F
First access: 2019-04-08 08:07:10.206
Last access: 2019-04-08 08:07:12.784
...
```

Obrázek 6.1: Část záznamu pořízeného při sledování běhu programu využívajícího knihovnu OpenSSL. Velikosti nalezených S-Boxů odpovídají standardním S-Boxům a překrytým T-Tabulkám. Osm nerozpoznaných přístupů mezi expanzí klíče a šifrováním je důsledkem techniky přednačtení tabulky do keše (cache prefetch).

6.1.2 CryptoPP

Knihovnu CryptoPP verze 8.0.0 jsme otestovali na virtuálním stroji se skrytými rozšířeními AES-NI a SSE2. Při inicializaci knihovny dochází k výpočtu překrytých T-Tabulek ze standardních S-Boxů. Kvůli pozdnímu odhalení takto vygenerovaných tabulek se našemu nástroji nepodařilo rekonstruovat první šifrovaná data. Ostatní šifrovaná i dešifrovaná data včetně klíčů jsme odhalili úspěšně.

```
Found SBox at address 0x0000000000B599B0, size 256,
   region 0x0000000000B59000 - 0x0000000000B5A000
Found InvSBox at address 0x0000000000B59AB0, size 256,
   region 0x0000000000B59000 - 0x0000000000B5A000

Recovered key: 000102030405060708090A0B0C0D0E0F
First access: 2019-04-08 08:35:50.487
Last access: 2019-04-08 08:35:50.644

Unknown access!
SBox: 00010203 ... FCFDFEFF
InvSBox:
First access: 2019-04-08 08:35:50.691
Last access: 2019-04-08 08:35:54.519

Found SBox at address 0x0000000000B5F0B1, size 2048,
   region 0x0000000000B5F000 - 0x0000000000B60000
Found InvSBox at address 0x0000000000B5F8C0, size 2048,
   region 0x0000000000B5F000 - 0x0000000000B61000
...
Recovered decryption data:
Input data: 69C4E0D86A7B0430D8CDB78070B4C55A
Output data: 00112233445566778899AABBCCDDEEFF
Used key: 000102030405060708090A0B0C0D0E0F
First access: 2019-04-08 08:35:57.253
Last access: 2019-04-08 08:35:57.425
...
```

Obrázek 6.2: Část záznamu z testování knihovny CryptoPP. Po startu jsme nalezli pouze standardní S-Boxy. Nepodařilo se nám rekonstruovat první šifrovaná data, protože jsme nenalezli dostatečně rychle novou za běhu vypočítanou překrytou T-Tabulku. Záznam obsahuje 256 nerozpoznaných přístupů, které odpovídají postupnému průchodu celého S-Boxu.

6.1.3 Botan

V případě knihovny Botan verze 2.9.0 jsme virtuálnímu stroji kromě AES-NI zatajili i vektorové rozšíření SSSE3. Podobně jako u CryptoPP dochází při inicializaci k výpočtu T-Tabulek, takže jsme opět nestihli rekonstruovat první šifrování. Zbylé šifrování, dešifrování i klíče jsme opět odhalili úspěšně.

6.1.4 Wincrypt

Program využívající kryptografické rozhraní Microsoft Windows jsme kromě dříve zmíněného virtuálního stroje vyzkoušeli i na dvou dalších verzích operačního systému – Windows 8.1 a Windows 10 (Build 17134) – v obou případech v 64bitové variantě. Všem virtuálním strojům jsme skryli instrukce AES-NI. Knihovna operačního systému vždy obsahovala všechny tabulky již při spuštění, díky čemuž jsme úspěšně odhalili veškeré klíče i data.

```
...
Recovered key: 00010203 ... C1D1E1F
First access: 2019-04-08 10:49:31.547
Last access: 2019-04-08 10:49:31.860

Unknown access!
SBox: 6D68DE36
InvSBox:
First access: 2019-04-08 10:49:31.860
Last access: 2019-04-08 10:49:31.860

Recovered encryption data:
Input data: 00112233445566778899AABBCCDDEEFF
Output data: 8EA2B7CA516745BFEAFC49904B496089
Used key: 00010203 ... C1D1E1F
First access: 2019-04-08 10:49:31.938
Last access: 2019-04-08 10:49:35.266
...
```

Obrázek 6.3: Část záznamu pořízeného při sledování programu využívající kryptografické rozhraní systému Windows 7 a 8.1. Úspěšně jsme rekonstruovali klíče i (de)šifrovaná data. Po expanzi 256bitového klíče obsahuje záznam čtveřici nerozpoznaných přístupů, které odpovídají poslednímu slovu expandovaného klíče. Dle standardu šifry ale poslední sloupec 256bitového klíče již do funkce SubWord nevstupuje. Pravděpodobně jedná o chybu v implementaci, neboť při testování na stroji s Windows 10 k těmto přístupům již nedocházelo.

6.2 Běžné programy

Následně jsme ověřili funkčnost našeho nástroje i na sadě běžně používaných programů, ve kterých dochází k šifrování pomocí AES.

6.2.1 7-Zip

Šifru AES můžeme využít při vytváření archivů chráněných heslem. Notoricky známý formát ZIP umožňuje využití AES-256 šifrování v módu čítače (CTR) [45]. Abychom mohli ověřit odhalená zašifrovaná data, umístíme do archivu soubor obsahující samé nulové bajty bez použití jakékoliv komprese. Ve výsledném archivu tak budou vidět jednotlivé zašifrované čítače.

Pro účely vytváření ZIP souborů jsme vybrali konzolová variantu aplikace 7-Zip[16], verze 19.00. Archiv s heslem „secret" jsme vytvořili ze souboru s 32 nulovými bajty (32zero.bin) příkazem:

```
7za a -mem=AES256 -m0=Copy -psecret 32zero.zip 32zero.bin
```

Příkaz jsme spustili pod dohledem programu standaloneAesSniffer ve virtuálním stroji se skrytými instrukcemi AES-NI. Odhalili jsme expanzi 256bitového klíče a dvě šifrování čítačů. Výsledná šifrovaná data jsme úspěšně nalezli v souboru 32zero.zip (obrázek 6.4).

6.2.2 PuTTy

Šifra AES je jednou z doporučených šifer pro použití v protokolu SSH [46]. Tento protokol je často využíván při vzdáleném přístupu k unixovým strojům. Jako sledovaný program jsme vybrali PuTTy[17] ve verzi 0.70. Program PuTTy je nejznámějším SSH klientem pro systém Windows. Tento program jsme spustili a sledovali mimo virtuální stroj, protože nepoužívá instrukce AES-NI ani bitslicing.

Provedli jsme zkušební spojení se serverem `fray1.fit.cvut.cz`, obě strany se dohodly na použití AES-256 v režimu čítače (CTR). Odhalili jsme expanzi dvou 256bitových klíčů a mnoho šifrování. Síťovou komunikaci jsme zachytávali nástrojem Wireshark[18]. K šifrovanému obsahu zachycených paketů jsme přičetli příslušné odhalené zašifrovaných čítače. Dešifrovaná data odpovídala struktuře SSH paketů (obrázek 6.5).

[16] `https://www.7-zip.org/`
[17] `https://www.putty.org/`
[18] `https://www.wireshark.org/`

```
Recovered key: D8BB5BFD ... ECF2FC09
First access: 2019-04-08 13:25:39.521
Last access: 2019-04-08 13:25:39.581

Recovered encryption data:
Input data: 01000000000000000000000000000000
Output data: BD60E065069F22A02559EAA58FDA5958
Used key: D8BB5BFD ... ECF2FC09
First access: 2019-04-08 13:25:39.671
Last access: 2019-04-08 13:25:39.686

Recovered encryption data:
Input data: 02000000000000000000000000000000
Output data: 737145FD109E6F70FE8691303971FFA7
Used key: D8BB5BFD ... ECF2FC09
First access: 2019-04-08 13:25:39.686
Last access: 2019-04-08 13:25:39.701
```

```
  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 50 4B 03 04 33 00 01 00 63 00 0A 6B 88 4E 00 00
00000010: 00 00 3C 00 00 00 20 00 00 00 0A 00 0B 00 33 32
00000020: 7A 65 72 6F 2E 62 69 6E 01 99 07 00 02 00 41 45
00000030: 03 00 00 07 0A 42 31 D6 A9 72 43 A4 72 19 A1 F3
00000040: 0B E3 D9 5D 15 BD 60 E0 65 06 9F 22 A0 25 59 EA
00000050: A5 8F DA 59 58 73 71 45 FD 10 9E 6F 70 FE 86 91
00000060: 30 39 71 FF A7 68 40 76 94 B7 52 40 93 D6 A7 50
```

Obrázek 6.4: Nahoře: část záznamu sledování běhu programu 7za, expanze klíče a šifrování dvou čítačů. Dole: začátek souboru 32zero.zip, zvýrazněna je oblast obsahující zašifrované čítače.

```
Recovered encryption data:
Input data: A49335094600AEAB81A8F7A28905F9B5
Output data: D075402F215B647B4A9FE805CFA99A3B
Used key: 3B92EA24 ... BA7ABAF6
First access: 2019-04-09 11:07:19.410
Last access: 2019-04-09 11:07:19.433
...
Recovered encryption data:
Input data: 3B341BE29B55E23679227F2745397A93
Output data: 9E1971B7496DDAA20803A84A2B52997B
Used key: EA55588E ... 0069D2D6
First access: 2019-04-09 11:07:19.462
Last access: 2019-04-09 11:07:19.486
```

```
Time                       Protocol Info
2019-04-09 11:07:19.458057 SSHv2    Client: Encrypted packet
SSH Protocol
    SSH Version 2 (encryption:aes256-ctr ...)
        Encrypted Packet: d07540333005647b4a9fe805cfa8e930...
        Decrypted Packet: 0000001c115e0000000000000001730b...

Time                       Protocol Info
2019-04-09 11:07:19.460721 SSHv2    Server: Encrypted packet
SSH Protocol
    SSH Version 2 (encryption:aes256-ctr ...)
        Encrypted Packet: 9e1971ab5833daa20903a84a2b53eaa4...
        Decrypted Packet: 0000001c115e000001000000000173df...
```

```
packet_length: 0000001C
padding_length: 11
payload:
    protocol: 5E (SSH_MSG_CHANNEL_DATA)
    recipient channel: 00000100
    data:
        length: 00000001
        value: 73
```

Obrázek 6.5: Nahoře: část záznamu sledovaného běhu programu PuTTy. Uprostřed: záznam síťové komunikace, požadavek klienta a odpověď serveru. Obsahy obou paketů jsme dešifrovali díky informacím získaným naším nástrojem. Dole: Obsah dešifrovaného paketu. Klient obdržel od serveru jediný znak – malé písmeno *s*.

6.2.3 Invoke-WebRequest

TLS je protokol umožňující zabezpečenou komunikaci po internetu a jeho nejznámější použití je v rámci protokolu HTTPS. Protokol TLS obsahuje množinu symetrických šifer, ze kterých je možné při komunikaci vybírat. Jednou z těchto šifer je i AES [47]. Rozhodli jsme se odhalit šifrovaná data přenášená v rámci HTTPS komunikace. Jako klientský program jsme vybrali Windows PowerShell verze 5.1 s jeho příkazem Invoke-WebRequest[19]. Pomocí následujícího příkazu jsme navázali HTTPS spojení a stáhli obsah webové stránky `example.com`:

```
powershell.exe -Command "Invoke-WebRequest https://example.com"
```

Příkaz jsme vykonali ve virtuálním stoji s 64bitovými Windows 10 (Build 17134) a skrytým rozšířením AES-NI. Při vykonání příkazu bylo vytvořeno spojení využívající protokol TLS verze 1.0. Šifrovanou komunikaci jsme opět zachytávali programem Wireshark. Obě komunikující strany se dohodly na použití šifry AES-128 v režimu řetězení bloků (CBC). Odhalili jsme použité klíče a šifrovaná i dešifrovaná data jsme nalezli v zachycené síťové komunikaci (obrázek 6.6).

[19] `https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest`

```
Recovered key: 634AD937C97D056EB8F64E67D73E5778
First access: 2019−04−09 19:41:58.494
Last access: 2019−04−09 19:41:58.566

Recovered encryption data:
Input data: ECA8A511B8F8E1AAE86736D562942F52
Output data: 193074CDA28EC615FFB6BE2B0215ED95
Used key: 634AD937C97D056EB8F64E67D73E5778
First access: 2019−04−09 19:41:58.598
Last access: 2019−04−09 19:41:59.835

Recovered encryption data:
Input data: 7FEFE6FC0C84CC1FF5BCB421081FE79F
Output data: DF30A08D1A882B06F3171572C2339AD0
Used key: 634AD937C97D056EB8F64E67D73E5778
First access: 2019−04−09 19:41:59.845
Last access: 2019−04−09 19:42:01.020
...
```

```
Time                       Protocol Info
2019-04-09 19:42:14.402359 TLSv1    Application Data, ...
Transport Layer Security
    TLSv1 Record Layer: Application Data Protocol: http-over-tls
        Content Type: Application Data (23)
        Version: TLS 1.0 (0x0301)
        Length: 32
        Encrypted Application Data:
            193074cda28ec615ffb6be2b0215ed95
            df30a08d1a882b06f3171572c2339ad0
    TLSv1 Record Layer: Application Data Protocol: http-over-tls
        ...
```

Obrázek 6.6: Nahoře: část záznamu sledování vykonávání příkazu Invoke-WebRequest v programu Windows PowerShell. Dole: záznam HTTPS komunikace, zašifrované bloky dat jsou obsaženy přímo v těle paketu.

6.3 Vliv na výkon

Abychom zjistili, k jak výraznému zpomalení sledovaného programu dojde při použití našeho nástroje, měřili jsme dobu běhu programu 7-Zip v různých konfiguracích. Zvolili jsme stejnou verzi programu jako při zkoušce odhalení v sekci 6.2.1. Využili jsme stejný virtuální stroj a prováděli jsme stejnou akci – uložení nekomprimovaných dat do ZIP archivu šifrovaného pomocí AES-256. Vytvářeli jsme archivy ze souborů o velikostech 256 B, 4 KiB a 64 KiB. Měřili jsme dobu běhu nesledovaného programu a dobu běhu programu sledovaného naším nástrojem. Daný program jsme sledovali při čtyřech různých nastaveních našeho nástroje – při detekci všech typů AES, bez detekce AES-192, pouze s detekcí AES-256 a bez jakékoliv detekce. Těchto nastavení jsme dosáhli kombinací přepínačů `--no-aes-128`, `--no-aes-192` a `--no-aes-265`, přidaných do našeho programu pro testovací účely. Každé měření jsme provedli třikrát a jako výsledek jsme použili aritmetický průměr naměřených hodnot.

Velikost šifrovaného souboru [B]	256	4096	65536
Doba běhu programu [s]			
Bez sledování	0,060416	0,063542	0,151042
Sledování bez detekce	148,827083	150,959375	166,932292
Sledování a detekce AES-256	150,065243	182,255805	329,562500
Sledování bez detekce AES-192	159,518952	204,534746	347,505208
Sledování a plná detekce	199,973958	662,807813	8138,843750

Obrázek 6.7: Vliv sledování běhu programu a detekce šifry AES na dobu šifrování ZIP archivu programem 7-Zip.

Z tabulky 6.7 vidíme, že při použití našeho nástroje běžel sledovaný program tisíckrát až desetitisíckrát déle. Jednou z příčin pozorovaného zpomalení je vysoká režie použitého ladicího rozhraní. Každý přístup do sledované stránky způsobí několik přepnutí kontextu mezi sledovaným procesem, operačním systémem a naším nástrojem. Drobnými změnami ve sledovaném programu můžeme dosáhnout velmi rozdílných dob běhu. Například pokud je ve sledovaném programu umístěna tabulka konstant nebo virtuálních metod[20] příliš blízko k S-Boxu AES, může dojít k obsažení do společné paměťové stránky. Každý přístup do takové tabulky konstant nebo virtuálních metod způsobí nechtěné zastavení sledovaného programu a prodlouží celkovou dobu běhu programu.

Dále pozorujeme výrazný vliv detekce AES-192 na dobu běhu. Důvodem k tomuto zpomalení je detekce 192bitového klíče, kterou spouštíme pro každých 32 zaznamenaných přístupů. Vazby použité při rekonstrukci klíče pracují

[20] Anglicky *Virtual Method Table*, tabulka ukazatelů na funkce.

s pěticemi sloupců a pro každý sloupec hledáme takovou permutaci, pro kterou je daná vazba splněna. Během řešení každé vazby prozkoumáme až $4!^5$ možných uspořádání bajtů do sloupců. Při rekonstrukci 128bitových a 256bitových klíčů nedochází k tak výraznému zpomalení, protože použité vazby pracují pouze s trojicemi sloupců.

Pro účely porovnání námi zvoleného přístupu s plným trasováním programu jsme přidali přepínač `--trace`, který způsobí provádění sledovaného programu po jednotlivých instrukcích. Tímto přepínačem také deaktivujeme hledání S-Boxů a detekci šifry. Opět jsme provedli tři měření šifrování ZIP archivu. Archivovali jsme pouze 256bajtový soubor, doby běhu pro větší soubory jsme neměřili. Aritmetický průměr naměřených časů byl 38964,265625 sekund. Trasování daného programu trvalo řádově stokrát déle než sledování naším nástrojem.

Závěr

Prozkoumali jsme šifru AES, její návrh (šifra Rijndael) a různé způsoby implementace. Dále jsme se zaměřili na rozličné přístupy k detekci této šifry a postupy odhalení použitých klíčů a dat. Jako klíčovou komponentu každé implementace AES jsme určili substituční tabulku S-Box. Na základě těchto poznatků jsme navrhli vlastní algoritmus detekující tuto šifru. Sledováním programu a jeho za běhu prováděných přístupů do S-Boxu odhalíme klíče a data použitá při šifrování. Tento algoritmus jsme implementovali v jazyce `C++` jako aplikaci pro systém Microsoft Windows a architekturu Intel x86.

Výsledný nástroj jsme otestovali na sadě programů používajících čtyři známé šifrovací knihovny – OpenSSL, CryptoPP, Botan a šifrovací rozhraní poskytované operačním systémem. Dále jsme zkoušeli odhalit šifrování při vytváření ZIP archivu programem 7-Zip, při SSH komunikaci pomocí programu PuTTy a při HTTPS komunikaci vyvolané skrz Windows PowerShell.

Ve většině případů se nám podařilo odhalit použité klíče i data. V některých případech (CryptoPP, Botan) jsme S-Box nenalezli dostatečně rychle a první prováděné šifrování jsme nestihli odhalit. Ve složitějších programech byl větší časový rozestup mezi inicializací a samotným šifrováním, což nám poskytlo potřebný čas na nalezení S-Boxu.

Programy sledované naším nástrojem běžely řádově rychleji než při trasování. Dalšího zrychlení bychom mohli dosáhnout například využitím instrukcí AES-NI při rekonstrukci klíče a dat nebo asynchronním zpracováváním přístupů v odděleném vlákně.

Nástroj jsme navrhli tak, aby byl lehce rozšiřitelný o další funkcionalitu. Základem je platformově nezávislá knihovna, která obsahuje implementaci našeho algoritmu. Tuto knihovnu by například bylo možné ve formě zásuvného modulu začlenit do nějakého jiného nástroje pro dynamickou analýzu programů. Pokud je naším cílem zrychlit samotné odhalení, mohli bychom knihovnu učinit součástí ovladače a zavést ji přímo do operačního systému. Také se můžeme zaměřit na přenositelnost a vytvořit jinou platformově závislou část kódu, která by umožnila běh na jiných operačních systémech a architekturách.

Literatura

[1] Daemen, J.; Rijmen, V.: AES submission document on Rijndael, Version 2. 1999. Dostupné z: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf

[2] National Institute of Standards and Technology: AES Development. [online], 2016, [cit. 6. 3. 2019]. Dostupné z: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development

[3] Daemen, J.; Rijmen, V.: *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002, ISBN 3-540-42580-2.

[4] National Institute of Standards and Technology: Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[5] Wikipedia: Rijndael key schedule. [online], 2019, [cit. 9. 3. 2019]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Rijndael%20key%20schedule&oldid=885147903

[6] Welschenbach, M.: *Cryptography in C and C++*. Apress, 2005, ISBN 1-59059-502-5.

[7] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. 2019. Dostupné z: https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf

[8] Gladman, B.: Byte Oriented AES. [online], [cit. 11. 3. 2019]. Dostupné z: https://web.archive.org/web/20070724062438/http://fp.gladman.plus.com/AES/aes-byte-22-11-06.zip

[9] Malbrain, K.: A byte oriented C version of AES. [online], [cit. 11. 3. 2019]. Dostupné z: https://www.geocities.ws/malbrain/aestable_c.html

[10] Rijmen, V.; Bosselaers, A.; Barreto, P.: Optimised ANSI C code for the Rijndael cipher. [online], 2000, [cit. 12. 3. 2019]. Dostupné z: https://github.com/openssl/openssl/blob/master/crypto/aes/aes_x86core.c

[11] Käsper, E.; Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. *Cryptographic Hardware and Embedded Systems – CHES 2009*, 2009: s. 1–17. Dostupné z: https://cryptojedi.org/papers/aesbs-20090616.pdf

[12] Matsui, M.: How Far Can We Go on the x64 Processors? *Fast Software Encryption*, 2006: s. 341–358. Dostupné z: https://www.iacr.org/archive/fse2006/40470344/40470344.pdf

[13] Matsui, M.; Nakajima, J.: On the Power of Bitslice Implementation on Intel Core2 Processor. *Cryptographic Hardware and Embedded Systems – CHES 2007*, 2007: s. 121–134. Dostupné z: https://link.springer.com/content/pdf/10.1007/978-3-540-74735-2_9.pdf

[14] Basu, A.; Bhargav-Spantzel, A.: Intel AES-NI Performance Testing on Linux/Java Stack. Technická zpráva, Intel Corporation, 2012, [cit. 14. 3. 2019]. Dostupné z: https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack

[15] Gueron, S.: Intel Advanced Encryption Standard (AES) New Instructions Set. Technická zpráva, Intel Corporation, 2010. Dostupné z: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[16] Wikipedia: AES implementations. [online], 2019, [cit. 15. 3. 2019]. Dostupné z: https://en.wikipedia.org/w/index.php?title=AES%20implementations&oldid=871666483

[17] Microsoft Docs: *PE Format*. Microsoft Corporation, 2018, [cit. 15. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format#import-directory-table

[18] Linux Programmer's Manual: *elf – format of Executable and Linking Format (ELF) files*. The Linux Kernel Organization. Dostupné z: http://man7.org/linux/man-pages/man5/elf.5.html

[19] Microsoft Docs: *Dynamic-Link Library Security*. Microsoft Corporation, 2018, [cit. 15. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-security

[20] Linux Programmer's Manual: *ld.so, ld-linux.so* – dynamic linker/loader*. The Linux Kernel Organization. Dostupné z: http://man7.org/linux/man-pages/man8/ld.so.8.html

[21] Nouh, L.; Rahimian, A.; Mouheb, D.; aj.: BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables. *ICT Systems Security and Privacy Protection*, 2017: s. 341–355.

[22] Hex-Rays SA: IDA F.L.I.R.T. Technology: In-Depth. [online], 2015, [cit. 15. 3. 2019]. Dostupné z: https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

[23] Jacobson, E. R.; Rosenblum, N.; Miller, B. P.: Labeling Library Functions in Stripped Binaries. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2011: s. 1–8.

[24] Guilfanov, I.: FindCrypt2. [online], 2006, [cit. 18. 3. 2019]. Dostupné z: https://www.hexblog.com/?p=28

[25] Wikipedia: Executable compression. [online], 2019, [cit. 18. 3. 2019]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Executable%20compression&oldid=869640883

[26] Takehisa, T.; Nogawa, H.; Morii, M.: AES Flow Interception: Key Snooping Method on Virtual Machine – Exception Handling Attack for AES-NI. *IACR Cryptology ePrint Archive*, 2011: str. 428. Dostupné z: https://eprint.iacr.org/2011/428.pdf

[27] BSDaemon; Pirata: Extending crypto-related backdoors to other scenarios. *PoC ||GTFO*, 2015: s. 42–48. Dostupné z: https://archive.org/download/pocorgtfo07/pocorgtfo07.pdf

[28] Sharkey, J.: Breaking Hardware-Enforced Security with Hypervisors. 2016. Dostupné z: https://www.blackhat.com/docs/us-16/materials/us-16-Sharkey-Breaking-Hardware-Enforced-Security-With-Hypervisors.pdf

[29] Gröbert, F.; Willems, C.; Holz, T.: Automated Identification of Cryptographic Primitives in Binary Programs. *Recent Advances in Intrusion Detection*, 2011: s. 41–60.

[30] Calvet, J.; Fernandez, J. M.; Marions, J.-Y.: Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. *ACM Conference on Computer and Communications Security*, 2012: s. 169–182.

[31] Xu, D.; Ming, J.; Wu, D.: Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. *2017 IEEE Symposium on Security and Privacy (SP)*, 2017: s. 921–937.

[32] Lestringant, P.; Guihéry, F.; Fouque, P.-A.: Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. *ACM Symposium on Information, Computer and Communications Security - ASIA CCS*, 2015: s. 203–214.

[33] Whitham, J.: x86 single step mode – how slow is it? [online], 2016, [cit. 19. 3. 2019]. Dostupné z: https://www.jwhitham.org/2016/01/x86-single-step-mode-how-slow-is-it.html

[34] VMware, Inc.: *Change CPU Identification Mask Settings*. 2018, [cit. 19. 3. 2019]. Dostupné z: https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.vm_admin.doc/GUID-E64FC69D-E845-4562-BB8E-CCAC39E78CEA.html

[35] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. 2019. Dostupné z: https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf

[36] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 1*. 2019. Dostupné z: https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf

[37] Microsoft Docs: *Large-Page Support*. Microsoft Corporation, 2018, [cit. 19. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/memory/large-page-support

[38] Linux kernel: arch/x86/include/asm/page_types.h. [online], 2018, [cit. 19. 3. 2019]. Dostupné z: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/page_types.h

[39] Microsoft Docs: *processthreadsapi.h header*. Microsoft Corporation, 2019, [cit. 29. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/

[40] Microsoft Docs: *debugapi.h header*. Microsoft Corporation, 2019, [cit. 29. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/api/debugapi/

[41] Microsoft Docs: *memoryapi.h header*. Microsoft Corporation, 2019, [cit. 29. 3. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/

[42] Microsoft Docs: *Creating Guard Pages*. Microsoft Corporation, 2019, [cit. 7. 4. 2019]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/memory/creating-guard-pages

[43] Polyakov, A.: crypto/aes/asm/aes-586.pl. [online], 2016, [cit. 29. 3. 2019]. Dostupné z: https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aes-586.pl

[44] zyantific team: zydis: The ultimate X86 & X86-64 disassembler library. [online], 2018, [cit. 29. 3. 2019]. Dostupné z: https://zydis.re/

[45] Corel Corporation: WinZip – AES Encryption Information. [online], 2009, [cit. 8. 4. 2019]. Dostupné z: https://www.winzip.com/win/en/aes_info.html#encrypted-data

[46] Bellare, M.; Kohno, T.; Namprempre, C.: The Secure Shell (SSH) Transport Layer Encryption Modes. RFC 4344, Thammasat University, 2006. Dostupné z: https://www.rfc-editor.org/rfc/rfc4344.txt

[47] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Mozilla Corporation, 2018. Dostupné z: https://www.rfc-editor.org/rfc/rfc8446.txt

Příloha A

Obsah přiloženého CD

- `/readme.txt` .... popis obsahu CD a pokyny pro překlad programu
- `/text.pdf` .... text práce ve formátu PDF
- `/bin/` .... přeložené verze programu
- `/src/` .... zdrojové kódy programu
- `/test/` .... programy a soubory použité pro testování
- `/text/` .... práce ve formátu LaTeX a příslušné soubory