UNIVERZITA KOMENSKÉHO V BRATISLAVE

Fakulta Matematiky, fyziky a informatiky

b97dd65d-30d0-4268-9207-97ec2e493473

IMPLEMENTÁCIA KRYPTOGRAFICKÝCH ÚTOKOV V PARALELNOM VÝPOČTOVOM PROSTREDÍ GRAFICKÝCH KARIET

2011 Martin Višňovec

UNIVERZITA KOMENSKÉHO V BRATISLAVE

Fakulta matematiky, fyziky a informatiky

Implementácia kryptografických útokov v paralelnom výpočtovom prostredí grafických kariet

Bakalárska práca

Študijný program: Informatika

Študijný odbor: 9.2.1 Informatika

Školiace pracovisko: Katedra Informatiky

Školiteľ: Mgr. Michal Rjaško

2011 Martin Višňovec

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedených zdrojov.

V Bratislave, 1. 6. 2011 .....

Ďakujem vedúcemu bakalárskej práce Mgr. Michalovi Rjaškovi za odborné vedenie práce, cenné rady, ochotu a čas venovaný konzultáciám, bez ktorých by práca nedospela do finálnej podoby.

Abstrakt

Martin Višňovec: Implementácia kryptografických útokov v paralelnom výpočtovom prostredí grafických kariet

Univerzita Komenského v Bratislave, Fakulta matematiky, fyziky a informatiky, Katedra informatiky, Bakalárska práca, 40, 2011

Cube attack je typ rýchleho algebraického útoku proti prúdovým šifrám. Skladá sa z prípravnej a aktívnej fázy. Podrobne popíšeme tento útoku ako aj hlavné princípy prúdových šifier. Potom implementujeme popísaný útok na zvolenú prúdovú šifru. Použitím Cuda technológie na grafických kartách nVidia využijeme výhodu rýchleho paralelného spracovanie veľkého počtu vlákien a porovnáme efektivitu oproti našej pôvodnej implementácii.

Kľúčové slová: prúdové šifry, algebraické útoky, Cube attack, Toyocrypt, Trivium, Cuda

Abstract

Martin Višňovec: Implementation of cryptonalytic attacks in parallel computing environment of graphics cards

Comenius University in Bratislava, Faculty of mathematics, physics and informatics, Department of informatics, Bachelor work, 40, 2011

Cube attack is a type of fast algebraic attack on stream ciphers. It consists of preprocessing and active phase. We will precisely describe this attack as well as main principles of stream ciphers. Then we implement described attack on chosen stream ciphers. Using Cuda technology of nVidia graphic cards, we use its advantage of fast parallel processing great number of threads and compare effectivity with our original implementation.

Key words: stream ciphers, algebraic attacks, Cube attack, Toyocrypt, Trivium, Cuda

Obsah

Úvod

Informácie a dáta sú v dnešnej dobe jednou z najcennejších vecí. Okrem ich spracovania je dôležitou oblasťou aj zabezpečenie ich autentickosti a dôvernosti. Stále aktuálnou otázkou je tiež šifrovanie dát, ktorého cieľom je zabrániť tomu, aby sa dostali k neželaným osobám. Vedný odbor - kryptológia - skúma práve rôzne formy zabezpečenia informácií. Delí sa na kryptografiu, ktorej cieľom je návrh algoritmov a protokolov spĺňajúcich požadované bezpečnostné vlastnosti. Výsledný algoritmus by nemal byť prelomiteľný pri výpočtových schopnostiach počítačov v súčastnosti ako aj v priebehu najbližších rokov. Druhou časťou kryptológie je kryptoanalýza, ktorá skúma tieto konštrukcie za účelom analýzy potenciálnych útokov. Touto spätnou väzbou prispieva k zlepšovaniu samotných kryptografických konštrukcií a odhaľuje slabé miesta v ich návrhu.

Cieľom tejto práce je implementovať algebraický typ útoku nazývaný 'cube attack', umožňujúci prelomenie niektorých typov prúdových šifier s doposiaľ najnižšou časovou zložitosťou. Výpočet algoritmu bude prebiehať na grafickej karte, ktorá umožňuje mnohonásobne rýchlejšie riešenie paralelných typov problémov, využívajúc naplno výhody mnohojadrového typu procesora. Vďaka tomu je možné zvýšenie počtu naraz vykonaných operácii a zníženie celkového času potrebného na prelomenie šifry.

Štruktúra práce

Táto práca je rozdelená do štyroch základných častí. Cieľom prvej časti je úvod do problematiky kryptológie, základný prehľad o konštrukcii prúdových šifier a popis dvoch vybraných šifier. Hlavnú pozornosť budeme venovať šifrám Toyocrypt a Trivium. Ďalej uvedieme možné typy útokov, pričom sa zameriame na algebraické útoky. Poskytneme tiež súhrn ostatných známych metód útokov na tento typ šifier.

V druhej časti sa podrobne venujeme algoritmu Cube attack. Popíšeme jeho prípravnú fázu, ktorú je potrebné vypočítať len raz pre danú šifru. Tiež popíšeme aktívnu fázu, ktorá prebieha počas útoku na konkrétny použitý kľúč. Vysvetlíme algebraické princípy, ktoré algoritmus útoku využíva.

Tretia časť práce je venovaná popisu CUDA výpočtového prostredia, úvodu do práce s ním a základnej syntaxi jazyka. Ukážeme výhody výpočtu na grafickej karte a porovnáme výpočtovú silu v prípade riešenia paralelných úloh. Vzhľadom na to, že našim cieľom je predovšetkým algebraická kryptoanalýza, venujeme tejto časti najmenší rozsah.

Nakoniec posledná, štvrtá, časť má za úlohu ukázať konkrétny útok na prúdovú šifru, za použitia paralelných výpočtov na grafickej karte pomocou technológie CUDA. Nami vybraný útok je Cube attack, respektíve jeho mierne pozmenená varianta. Cieľom útoku bude šifra Toyocrypt.

Kapitola 1

Prúdové šifry

1.1 Zatriedenie a použitie

Prúdové šifry sú typ symetrických kryptografických šifier, ktoré kombinujú otvorený text so pseudonáhodne generovaným prúdom, obvykle pomocou XOR operácie. Vstupný text je šifrovaný postupne, pričom transformácia na šifrové bity sa pre jednotlivé bity mení a závisí od vnútorného stavu šifry.

Symetrické šifry používajú pri zašifrovaní aj odšifrovaní jednotný tajný kľúč. Druhá veľká skupina šifier, asymetrické, používajú dvojicu kľúčov. Verejný, ktorý sa použije pri tvorbe šifrového textu, a súkromný, pomocou ktorého sa spätne získa otvorený text. Prúdové šifry spracúvajú otvorený text ako postupnosť bitov, narozdiel od blokových šifier, ktoré pracujú s blokmi otvoreného textu určitej veľkosti (napr. 128 bitov).

Vo všeobecnosti je výhodou prúdových šifier vysoká rýchlosť šifrovania otvorených dát. Dôležitou otázkou je ich bezpečnosť. Hlavnou úlohou vhodne navrhnutej prúdovej šifry je práve bezpečnosť voči všetkých známym typom útokov za súčasnej jednoduchosti implementácie a zachovania vysokej rýchlosti šifrovania dát. V súčastnosti sú často využívané, ako príklad možno uviesť šifru RC4, široko používanú v bezpečnostných protokoloch vrátane SSL. Prípadne protokol Bluetooth, ktorý používa ako jeden zo štyroch svojich zabezpečovacích algoritmov prúdovú šifru E0, obsahujúcu štyri LFSR s celkovou dĺžkou 128 bitov.

1.2 Linear feedback shift register

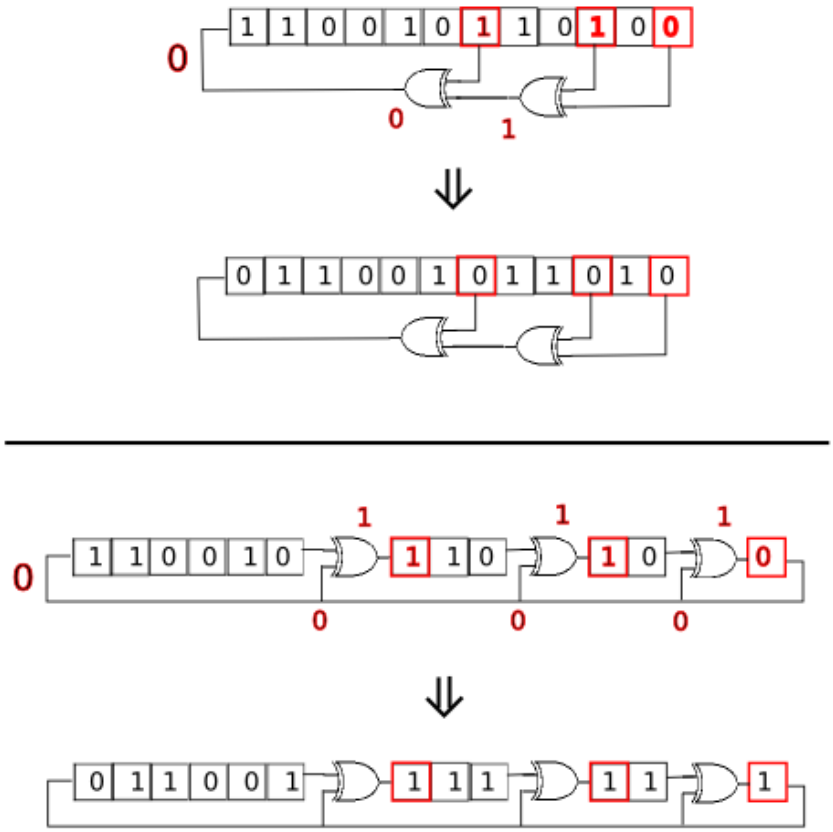
V tejto sekcii opíšeme princíp špecifického typu posuvných registrov, ktoré tvoria základný prvok prúdových šifier. Zjednodušene možno povedať, že postupnosť, ktorú registre vytvárajú kombinujeme s otvoreným textom do šifrového textu. Pre zvýšenie bezpečnosti šifry sa používa množstvo mechanizmov, ktoré tento proces rozširujú a modifikujú, ale postupnosť vytváraná posuvnými registrami zostáva vždy súčasťou a tvorí základ algoritmu.

PRBS (pseudorandom binary sequence) je binárna deterministická sekvencia, ktorá sa po istom čase začne opakovať. Jednotlivé bity sekvencie sa javia nezávislé od predchádzajúcich, čím vytvárajú pseudonáhodnú sekvenciu bitov. Avšak táto sekvencia vôbec náhodnou nie je. Aktuálny stav a funkcia výpočtu v každom momente jednoznačne určujú nasledujúci stav. Vlastnosti PRBS používajú niektoré kryptografické konštrukcie, napríklad LFSR.

LFSR (linear feedback shift register) je posuvný register, používajúci lineárnu funkciu na výpočet nasledujúceho bitu z aktuálneho stavu. Pri prechode na ďalší stav sa na niekoľko určených prvkov registra uplatní operácia XOR, výsledkom čoho získame nový bit. Je možné použiť na tento výpočet aj funkciu XNOR. Pozície bitov, ovplyvňujúce výpočet nasledujúceho stavu registra sa nazývajú tap-y. Následne po získaní bitu funkciou nad tap pozíciami sa celý obsah registra posunie o jeden bit a na prázdne miesto sa dosadí vypočítaný nový bit. Postupnosť generovaných bitov takéhoto registra predstavuje už spomínanú PRBS. Takto pracujúci LFSR sa zvykne nazývať Fibonacciho.

Cyklus registra dĺžky n môže pri vhodne zvolenej funkcii obsahovať maximálne $2^n - 1$ rôznych stavov, po ktorých sa začne výstup LFSR opakovať. Tento maximálny počet je zároveň aj počet všetkých možných stavov. Jediný stav, ktorý takýto cyklus neobsahuje, je register so všetkými hodnotami rovnými nule. V tomto stave je register statický, pretože vždy vráti len nulu, a jeho stav obsahuje stále nuly na každej pozícii. Aby funkcia umožňovala prejsť maximum stavov bez ich opakovania, musia byť vhodne zvolené tapy. Základnou podmienkou je nesúdeliteľnosť poradia tapov. Počet zvolených prvkov taktiež musí byť párny. Funkcia pre maximálnu dĺžku cyklu nie je pre daný register jedinečná, a môže ich byť vysoký počet.

Výpočet sekvencie generovanej LFSR je závislý iba od výpočtovej funkcie a jeho



Obr. 1.1: Grafická schéma porovnávajúca priebeh výpočtu Fibonacciho(hore) a Galoisovho(dolu) typu LFSR, červeno sú označené tap pozície a priebežné výpočty znázorňujúce postup výpočtu.

stavu. Pokiaľ je výpočtová funkcia nasledujúceho bitu známa stačí poznať úsek postupnosti bitov o dĺžke LFSR, aby sme vedeli získať počiatočný stav. Aj v prípade, že funkcia známa nie je, stačí nám iba úsek dĺžky dvojnásobku LFSR a je možné ďalej generovať postupnosť. Práve kvôli tejto jeho vlastnosti LFSR nie je vhodný na použitie v šifrovaní samostatne, ale využíva sa väčšinou v kombinácii s nelineárnou funkciou na jeho aktuálnom stave, prípadne sa iným spôsobom zabezpečuje nelineárnosť.

Modulárne LFSR, nazývané aj Galoisovo, je alternatívnou konštrukciou LFSR. Takýto register môže generovať rovnakú sekvenciu ako tradičné LFSR. Pri výpočte nasledujúceho stavu sa všetky bity registra posunú s nezmenenou hodnotou, okrem tap-ov. Na tie je uplatnená booleanovská funkcia XOR s výstupným bitom v danom kroku. Výstupný bit sa zároveň stane aj novo vypočítaním bitom. To znamená, že

ak je výstupný bit rovný 1, všetky hodnoty tap-ov sú invertované na opačné ešte pred posunom registra. V opačnom prípade sa hodnota nemení a dochádza iba k posunu. Implementácia modulárneho LFSR je hardware-ovo aj software-ovo výpočtovo niekoľkonásobne rýchlejšia vďaka možnému spracovanie operácii XOR paralelne.

1.3 Konštrukcia prúdových šifier

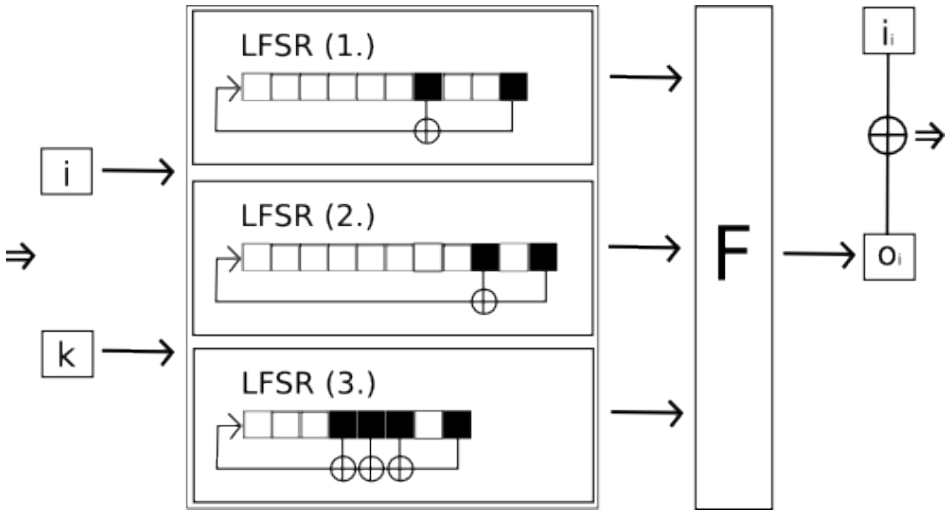
Základná konštrukcia prúdových šifier väčšinou pozostáva z jedného, alebo niekoľkých LFSR, predstavujúcich vnútorný stav šifry. Počiatočný stav predstavujú bity $p_1...p_n$, tvoriace počiatočný vektor P. Ďalší stav LFSR je vypočítaný z predchádzajúceho lineárnou funkciou S. Otvorený text je tvorený postupnosťou bitov $i_1...i_n$ a im prislúchajúci šifrový text tvoria bity $o_1...o_n$. Funkcia F vypočítava po každom kroku LFSR bit pseudonáhodného prúdu na základe aktuálneho stavu LFSR. Na rozdiel of funkcie S, funkcia F nijako neovplyvňuje obsah LFSR a teda ani vnútorný stav šifry. Základnou požiadavkou na funkciu F je jej vysoká nelineárnosť. To znamená, že nie je dobre aproximovateľná lineárnou funkciou. Pseudonáhodne generovaný prúd šifry je nakoniec binárnou operáciou XOR prepojený s prúdom otvoreného textu, čím získame zašifrovanú verziu tohto textu.

V takejto jednoduchej konštrukcii môžeme popísať výpočet nasledovne:

$$o_k = F(S^k(P)) \oplus i_k$$

kde S^k predstavuje stav LFSR po vykonaní k krokov.

Alternatívnou konštrukciou je použitie trojice LFSR. Výstup prvého z nich svojou hodnotou určuje, ktoré LFSR zo zvyšných dvoch bude použité pri výpočte nasledujúceho šifrového bitu. Používa sa tiež systém dvoch LFSR, pričom prvý "krokovací" je normálne krokovaný a svojim výstupom určuje krokovanie druhého LFSR. Ten je použitý na generovanie bitov pseudonáhodného prúdu, preto sa tiež nazýva "dátový". Princíp takéhoto nepravidelného krokovania bol využitý napríklad v šifre LILI-II. Jej krokovacie LFSR v dvoch po sebe idúcich krokoch vypočíta dvojbitovú postupnosť, ktorá svojou hodnotou určí počet krokov (v rozsahu 1-4) dátového LFSR pred ďalšími výpočtami z jeho stavu. Presnejší popis a analýza LILI-II je v [11].



Obr. 1.2: Grafická schéma jednoduchej prúdovej šifry obsahujúcej niekoľko LFSR, ktoré sú ďalej spracované nelineárnou funkciou F. Nakoniec je prúd vstupných dát a náhodne generovaný prúd spojené operáciou XOR do výsledného prúdu zašifrovaných dát.

1.4 Typy útokov

Možný útok na prúdovú šifru predstavuje vyriešenie rovníc so šifrovými bitmi a im zodpovedajúcich polynomiálnych rovníc. Riešenie takéhoto systému je NP-úplnej zložitosti, a to už pri polynómoch kvadratického stupňa, čím tvorí základ bezpečnosti šifrovacieho systému. Napriek tomu je mnoho šifier zraniteľných algebraickými útokmi, ktoré rôznymi metódami riešia ich systémy rovníc polynómov.

Cieľom útoku je získanie kľúča alebo otvoreného textu. Často je predpokladom znalosť šifrovacie algoritmu. Útok na šifru s neznámym spôsobom výpočtu je oveľa ťažší, a preto sa niekedy algoritmus šifry drží v utajení. Avšak ani prípadná znalosť výpočtu by nemala umožňovať prelomenie šifry. Navyše je väčšinou len otázkou času, kým sa stane používaný algoritmus verejne známy alebo môže byť potenciálne získaný reverzným inžinierstvom. Preto by bezpečnosť šifry mala závisieť od utajenia použitého kľúča. Tento princíp sa nazýva Kerckoffov zákon [14].

Triedenie útokov podľa typu dát, ktoré sú dostupné:

1. Útok len so šifrovým textom (Ciphertext-only attack): útočník má k dispozícii iba šifrový text. Cieľom je získať hodnotu kľúča alebo aspoň zodpovedajúceho otvoreného textu. Aby bol takýto útok efektívny, musí byť v texte redundancia. To znamená, že musí byť zložený z viac bitov, ako je nevyhnutné pre dané dáta. Musí obsahovať

aj 'nadbytočnú' informáciu. Ak text nie je redundantný, je potrebné vyskúšať všetky možné hodnoty kľúča, od ktorého dĺžky bude závisieť zložitosť.

2. Útok s otvoreným textom (Know-plaintext attack): útočník má k dispozícii dvojice bitov (i_k, o_k) predstavujúce bity otvoreného textu a im zodpovedajúci šifrový text. Cieľom je zistenie hodnoty kľúča alebo zvyšného neznámeho otvoreného textu v prípade, že nie je celý známy.

3. Útok so zvoleným otvoreným textom (Chosen-plaintext attack): útočník má k dispozícii šifrovací systém. Môže si zvoliť niekoľko otvorených textov a získať im zodpovedajúce šifrové texty. Cieľom útoku je získanie hodnoty použitého kľúča.

4. Útok so zvoleným šifrovým textom (Chosen-ciphertext attack): podobne ako pri útoku so zvoleným otvoreným textom. Útočník si môže zvoliť niekoľko šifrových textov a získať im zodpovedajúce otvorené texty. Cieľom útoku je získať hodnotu použitého kľúča.

Útoky sú očíslované podľa vzrastajúcej obtiažnosti ich realizácie a zároveň rovnako vzrastajúcej sile potencionálneho útoku. Od používaných šifier sa vyžaduje odolnosť voči všetkým vyššie spomenutým typom útokov.

1.5 Algebraické útoky

Tento typ útoku sa na prúdové šifry pozerá ako na algebraické rovnice. Riešením systému týchto rovníc pre šifrové bity teoreticky môžeme prúdovú šifru prelomiť a získať použitý kľúč. Toto je možné ak je nelineárna funkcia F dostatočne nízkeho stupňa. Ak je funkcia pre systém algebraických rovníc vysokého stupňa, ale je možné ju aproximovať funkciou dostatočne nízkeho stupňa s pravdepodobnosťou veľmi blízkou nule, dokážeme šifru riešiť výpočtom rovníc za použitia aproximačnej funkcie.

V prípade, že tento postup nie je možný je možné využiť takzvané annihilátory. Annihilátorom označujeme vhodne zvolenú funkciu G nižšieho stupňa ako F. Funkciu F, ktorej niektoré prvky sú vysokého stupňa, vynásobíme funkciou G, pričom výsledná funkcia $F \cdot G$ bude nízkeho stupňa.

Upravená rovnica výpočtu s použitím funkcie G je:

$$o_k \cdot G = F(S^k(P)) \cdot G,$$

čo je vlastne:

$$o_k \cdot G(S^k(P)) = F(S^k(P)) \cdot G(S^k(P))$$

Ak získame takúto rovnicu pre dostatočne veľa šifrových bitov, dostaneme systém rovníc nízkeho stupňa, ktorý môžeme ďalej riešiť. Teda šifru dokážeme vyriešiť aj v prípade, že F je vysokého stupňa a zároveň nie je aproximovateľná funkciou nízkeho stupňa, za predpokladu existencie vhodného G.

Pri útoku s neznámym otvoreným textom sa často využíva predpoklad, že najvyšší bit v ASCII kódovaní bude nula (platí to pre znaky a-z, A-Z, čísla aj štandardnú interpunkciu), prípadne znalosť hodnoty fixnej hlavičky správy. V útoku potom využívame kombináciu takto určených otvorených bitov a im prislúchajúcich šifrových bitov. Je to v podstate útok len so šifrovým textom, ale môžeme využiť výhody ako pri útoku aj s otvoreným textom [1].

Algebraické útoky sú pomerne novou metódou analýzy šifrovacích systémov. Ich výhodou je potreba znalosti menšieho počtu dvojíc bitov oproti niektorým iným útokom. Nevýhodou môže byť častá potreba resynchronizácie, čo znamená, že bity nemôžu byť získané z jedného výstupného prúdu. Musia byť vypočítané v niekoľkých kolách, medzi ktorými dochádza k opätovnej inicializácii šifry za použitia rovnakého kľúča. Často sa tiež využíva prípravná fáza pred samotným útokom, ktorá znižuje časovú náročnosť výpočtu počas tohto útoku.

Spôsoby riešenia systému algebraických rovníc nízkeho stupňa:

- Buchbergerov algoritmus, pracuje výpočtom Groebnerových báz, má exponenciálnu časovú zložitosť

- linearizácia: jednotlivé členy polynómov sú nahradené premennými, a rieši sa systém rovníc s týmito novými premennými. Pre hľadanie riešenia je však nepraktická, pretože vyžaduje pri výpočte veľké množstvo rovníc, rádovo až n^d pre daný počet premenných n a stupeň rovnice d.

- XL(eXtended Linearization) metóda: každú rovnicu samostatne násobíme jednočlennými polynómami do stupňa nanajvýš d. Členy týchto rovníc následne nahradíme premennými. Ďalej už riešime systém rovníc o týchto nových premenných [13]. Výpočet vyžaduje menší počet rovníc ako obyčajná linearizácia. Požaduje

predefinovaný systém m rovníc $f_m(X_1, ..., X_n) = 0$. Systém je predefinovaný, ak je dimenzia vektorového priestoru generovaného rovnicami väčšia ako n.

Základný postup si názorne ukážeme na príklade, máme 3 rovnice:

$$R_1 : x_1^2 + x_2x_3 = 0$$
$$R_2 : x_2^2 + x_1x_3 = 0$$
$$R_3 : x_3^2 + x_1x_3 + x_1x_2 = 0$$

každú rovnicu vynásobíme mononómami do stupňa $d = 3$, dostaneme:

$$R_1 : x_1^3 + x_1x_2x_3 = 0$$
$$R_1 : x_1^2x_2 + x_2^2x_3 = 0$$
$$R_1 : x_1^2x_3 + x_2x_3^2 = 0$$
$$R_2 : x_2^2x_1 + x_1^2x_3 = 0$$
$$R_2 : x_2^3 + x_1x_3x_2 = 0$$
$$R_2 : x_2^2x_3 + x_1x_3^2 = 0$$
$$R_3 : x_3^2x_1 + x_1^2x_3 + x_1^2x_2 = 0$$
$$R_3 : x_3^2x_2 + x_1x_3x_2 + x_1x_2^2 = 0$$
$$R_3 : x_3^3 + x_1x_3^2 + x_1x_2x_3 = 0$$

nakoniec linearizujeme:

$$y_1 + y_2 = 0$$
$$y_3 + y_4 = 0$$
$$y_5 + y_6 = 0$$
$$y_7 + y_5 = 0$$
$$y_8 + y_2 = 0$$
$$y_4 + y_9 = 0$$
$$y_9 + y_5 + y_3 = 0$$
$$y_6 + y_2 + y_7 = 0$$
$$y_{10} + y_9 + y_2 = 0$$

a riešime sústavu lineárnych rovníc.

- XSL(eXtended Sparse Linearization) metóda, zdokonalenie XL metódy, rovnice sa násobia nie všetkými, ale len niektorými vhodne zvolenými mononómami - jednočlennými polynómami
- MutantXL, popísaný v [10], a rôzne iné varianty linearizačných algoritmov

1.6 Iné útoky

1.6.1 Útok hrubou silou (Brute-force attack/Exhaustive key search)

Najprimitívnejší spôsob získania kľúča. Tento útok môže byť použitý proti akejkoľvek prúdovej šifre. Prebieha skúšaním všetkých možných hodnôt kľúča, až kým sa nenájde správny kľúč. Pre dĺžku kľúča n bitov je v najhoršom prípade potrebné vyskúšať 2^n možností, respektíve 2^{n-1} možností v priemernom prípade. Útok s vyššou výpočtovou zložitosťou ako tento nie je už ani považovaný za útok.

1.6.2 Časovo-pamäťový kompromis (Time-memory trade-off)

Útok využívajúci veľké množstvo predspracovaných dát na zníženie výpočtovej zložitosti. Vychádza z poznatku, že často existuje kompromis medzi množstvom pamäte, potrebným na uloženie dát, a množstvom času, potrebného na výpočet. To znamená, že požiadadavky na pamäť môžu byť znížené na úkor zložitosti výpočtu a naopak výpočet môže byť zrýchlený za použitia väčšieho množstva pamäte.

Pre vyhnutie sa tomuto typu útokov by počet možností vnútorného stavu šifry mal byť aspoň 2-násobkom počtu možností kľúča a dĺžka otvoreného textu by mala byť prinajmenšom rovná dĺžke kľúča. V prípade nesplnenia týchto podmienok bude zložitosť nižšia ako zložitosť útoku hrubou silou.

1.6.3 Korelačné útoky (Correlation attacks)

Aby bol možný tento typ útoku, musí výstupný prúd $o_1, o_2, o_3, ...$ korelovať s prúdom $c_1, c_2, c_3, ...$ Prúd bitov c je výstup generovaný jednoduchším princípom, ako napríklad

LFSR. Tieto dva prúdy dát korelujú v prípade, že platí $o_k = c_k$ s pravdepodobnosťou odlišnou od 50 percent. V prípade korelácie je potenciálne možné získať počiatočný stav generujúceho LFSR.

Predpokladajme šifru obsahujúcu niekoľko LFSR. Ak koreluje výstup jedného z týchto LFSR s výstupom šifry, je možné určiť jeho pôvodný stav. Použitím princípu rozdeľ a panuj, rozdelenia problému na niekoľko menších podproblémov, môžeme takto jednotlivo získať stavy všetkých LFSR. Najskôr nájdeme koreláciu jedného LFSR a obnovíme jeho stav. So znalosťou jeho stavu hľadáme postupne koreláciu pri ďalšom LFSR až kým týmto postupom nezískame stavy všetkých používaných LFSR. Zložitosť útoku bude výrazne nižšia, ako by bola v prípade skúšania všetkých možností. Napríklad pre tri LFSR dĺžky 16 bitov je celková zložitosť $2^{16}+2^{16}+2^{16}$ oproti zložitosti $2^{3 \cdot 16} = 2^{48}$.

1.6.4 Rozlišovacie útoky (Distinguishing attacks)

Výstupný prúd generovaný šifrovacím algoritmom nesmie byť odlíšiteľný od náhodnej binárnej postupnosti. V prípade, že tomu tak nie je, môže byť na šifru použitý rozlišovací typ útokov, ktorý využíva týchto odchýlok. Často je vyžadovaná znalosť veľmi veľkého množstva zašifrovaného textu, aby mohli byť využité rozdiely oproti náhodnému prúdu. Získaniu dostatočného množstva dát môže zabrániť nutnosť použitia nového kľúča po zašifrovaní určitého množstva otvoreného textu. Útok prebieha nastavením jedného alebo niekoľkých vstupov šifry a následným porovnávaním generovaného výstupu s čisto náhodne generovanými dátami.

1.7 Toyocrypt

Táto šifra bola jednou zo šifier podaných v rámci japonského národného kryptografického programu Cryptec. V čase jej návrhu nebol voči nej známy žiaden účinný útok. Návrh šifry je jednoduchý a nevyžaduje veľa pamäte, ale rýchlosť šifrovania je pomerne nízka. Pracuje s jedným 128-bitovým modulárnym LFSR [7]. Pri spracovaní každého bitu otvoreného prúdu dochádza k výpočtu jedného kroku v LFSR šifry a následne uplatneniu funkcie F na jeho stav pre výpočet bitu šifrového prúdu.

Jeho výpočtová funkcia F je nasledujúca:

$$f(s_0, .., s_{127}) = s_{127} + \sum_{i=0}^{62} s_i s_{\alpha_i} + s_{10}s_{23}s_{32}s_{42} +$$

$$s_1 s_2 s_9 s_{12} s_{18} s_{20} s_{23} s_{25} s_{26} s_{28} s_{33} s_{38} s_{41} s_{42} s_{51} s_{53} s_{59} + \prod_{i=0}^{62} s_i$$

Množina prvkov $\{ \alpha_0, ..., \alpha_{62} \}$ v kvadratických členoch je nejakou permutáciou množiny prvkov $\{ s_{63}, ..., s_{125} \}$

Funkcia F sa skladá z 67 členov, ktorých dĺžky sú 1, 2, 4, 17 a 63. Počet členov každej dĺžky je jedna, okrem kvadratických, ktorých je 63. Využíva vo svojom výpočte všetky bity aktuálneho stavu LFSR okrem s_{126}. Vďaka kvadratickým členom je funkcia dostatočne nelineárna. Hodnoty takýchto členov však nie sú vyvážené. To znamená, že pre náhodný vstup nie je pravdepodobnosť výstupu rovnaká pre hodnotu 0 aj 1. Tým, že člen s_{127} je rádu jedna, táto funkcia spĺňa aj dôležitú požiadavku na vyváženosť. Každý člen polynómu stupňa aspoň 4 obsahuje prvky s_{23} a s_{42}.

Polynómy vysokého rádu, veľkosti 17 a 63, sú skoro vždy rovné nule. Upravená funkcia bez týchto polynómov bude mať stupeň iba 4. Aby bol polynóm rádu 17 nenulový, musia mať všetky jeho prvky hodnotu 1. Pravdepodobnosť, že upravená funkcia nevypočíta správny výsledok, a teda odobraný polynóm sa nerovná nule, je v tomto prípade 2^{-17}. Druhý polynóm je nenulový s pravdepodobnosťou 2^{-63}. Nižšie číslo môžeme zanedbať, a teda celková pravdepodobnosť rovnosti upravenej funkcie s pôvodnou funkciou F je $1 - 2^{17}$ [2].

Kľúč je tvorený 256 bitmi. Z toho 128 bitov pripadá na počiatočný stav hodnôt v LFSR. Možné sú všetky hodnoty, okrem samých núl, teda $2^{128} - 1$. Ďalších 128 bitov tvorí fixný kľúč generujúci primitívny polynóm, ktorý určuje tap pozície v LFSR. Proces vytvorenia fixného kľúča môže byť pomerne časovo náročný, pretože treba vyskúšať veľký počet jeho možných hodnôt, kým sa nájde taký, ktorého polynóm generuje LFSR s maximálnym cyklom. Preto môže byť namiesto generovania niekedy dopredu určený v prípade, že je prvoradým cieľom vyššia rýchlosť. Počet vhodných polynómov je približne 2^{120}. Celkovo má teda množina možných kľúčov veľkosť 2^{248}, oproti teoretickému maximu 2^{256}.

1.8 Trivium

Tento šifrovací algoritmus nemá príliš komplikovanú konštrukciu a jeho rýchlosť spracovania dát je vysoká. Napriek jeho jednoduchosti voči Triviu doteraz nie je známy žiadny efektívny útok.

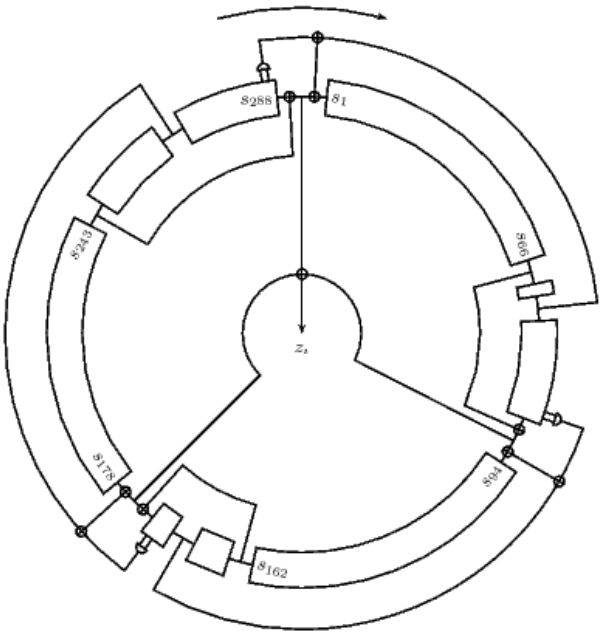
Používa tri LFSR, ktorých dĺžky sú 93, 84 a 111 bitov, s celkovou spoločnou dĺžkou 288 bitov. Pre obsah registrov $L_1 = (s_0, ..., s_{92}), L_2 = (s_{93}, ..., s_{176}), L_3 = (s_{177}, ..., s_{287})$ je postup výpočtu nasledovný: [9]

$$
\begin{aligned}
t_1 &= s_{65} + s_{92} \\
t_2 &= s_{161} s_{176} \\
t_3 &= s_{242} s_{287} \\
o &= t_1 + t_2 + t_3 \\
t_1 &= t_1 + s_{90} \cdot s_{91} + s_{170} \\
t_2 &= t_2 + s_{174} \cdot s_{175} + s_{263} \\
t_3 &= t_3 + s_{285} \cdot s_{286} + s_{68} \\
L_1 &= (t_3, s_0, ..., s_{91}) \\
L_2 &= (t_1, s_{93}, ..., s_{175}) \\
L_3 &= (t_2, s_{177}, ..., s_{286})
\end{aligned}
$$

pričom o je vygenerovaný bit výstupného prúdu. L_i označuje všetky bity i-teho LFSR.

Na inicializáciu obsahu LFSR sa používa 80 bitov kľúča a 80 bitov otvoreného textu, zvyšných 128 bitov sú na začiatku nuly, okrem troch posledných bitov tretieho LFSR, ktoré sú nastavené na jednotku. Bity kľúča sú priradené prvým 80 bitom prvého LFSR a bity otvoreného textu prvým 80 bitom druhého LFSR. Následne prebehne 4x288(=1152) inicializačných krokov, počas ktorých nie sú generované bity súčasťou výstupného prúdu.

Keďže sa zatiaľ nepodarilo prelomiť takýto algoritmus, boli skúmané možnosti útoku voči takémuto typu šifier aj vytvorením variant zjednodušeného Trivia. Pre použitie iba dvoch LFSR namiesto troch sa nazývajú Bivium A a Bivium B. Ich LFSR



Obr. 1.3: Grafická schéma konštrukcie šifry Trivium, zdroj [15].

majú dĺžky 93 a 84 bitov. Funkcia výpočtu pre variant Bivium A je nasledujúca:

$$t_1 = s_{65} + s_{92}$$
$$t_2 = s_{161}s_{176}$$
$$o = t_2$$
$$t_1 = t_1 + s_{90} \cdot s_{91} + s_{170}$$
$$t_2 = t_2 + s_{174} \cdot s_{175} + s_{68}$$
$$L_1 = (t_2, s_0, ..., s_{91})$$
$$L_2 = (t_1, s_{93}, ..., s_{175})$$

Výpočet verzie Bivium B je úplne totožný s variantom A, okrem výpočtu šifrového bitu, ktorý je nasledovný:

$$o = t_1 + t_2$$

Počet ich inicializačných krokov je 4x177, teda nižší ako v pôvodnom algoritme. Proti týmto zjednodušeným variantom Trivia sú už známe efektívne útoky, pričom Bivium A je výrazne jednoduchšie prelomiteľné. Ich uplatnenie na plné Trivium však

nie je dostatočne účinné. Taktiež boli skúmané útoky proti Triviu s nižším počtom inicializačných krokov, pričom sú aj známe účinné útoky ak je ich počet dostatočne nízky. Napríklad Cube attack dokázal efektívny útok voči Triviu, ktorý ich mal až 735, z celkového pôvodného počtu 1152.

Kapitola 2

Cube attack

2.1 Princíp útoku

Z algebraického hľadiska je výstup prúdovej šifry závislý od bitov kľúča a bitov otvoreného textu, ktoré spolu tvoria vstupné dáta. Tieto vstupné dáta sú ďalej spracované algoritmom šifry až je výsledkom procesu vypočítaný šifrový text. Takýto systém teda môžeme popísať ako systém algebraických rovníc, kde ľubovoľná jedna rovnica definuje výpočet určitého bitu z prúdu zašifrovaných dát. Každá rovnica je tvorená priradeným jedného šifrového bitu k polynómu, ktorý je určený bitmy kľúča a otvoreného textu.

Tieto algebraické rovnice pre konkrétny zašifrovaný bit však väčšinou nie sú nezávislé, ale sú to varianty odvodené od jediného *hlavného polynómu*, nastavením jeho meniteľných bitov na akúkoľvek hodnotu [3]. Meniteľnými bitmi polynómu sú jeho vstupy: tajné bity tvoriace kľúč a bity otvoreného prúdu, určené na zašifrovanie. Zmenou týchto otvorených bitov a kľúčov dostaneme odvodené polynomiálne rovnice, ktoré sú vzájomne súvisiace.

Celý útok pozostáva z dvoch fáz, prípravnej a aktívnej. Počas prípravnej fázy je umožnené nastavenie všetkých premenných, vstupných aj kľúča, a výpočet zodpovedajúceho výsledného bitu. Počas aktívnej fázy je hodnota kľúča fixná a nie je známa. Predpokladá sa znalosť dostatočného počtu otvorených bitov zodpovedajúcich prislúchajúcim zašifrovaným hodnotám bitov. Následne je s použitím výsledkov z prípravnej fázy vypočítaný kľúč, ktorý bol pri šifrovaní použitý.

Systém algebraických rovníc je určený počtom premenných n, ktoré obsahuje, a svojim maximálnym stupňom d . Dôležitou časťou útoku je správne zvoliť hodnotu d,

určujúcu stupeň hlavného polynómu. Znalosť tejto hodnoty je potrebná počas útoku a je hlavným faktorom určujúcim celkovú výpočtovú zložitosť ako aj minimálny počet požadovaných dvojíc otvoreného a šifrového prúdu na jeho prevedenie.

Cube attack nepožaduje znalosť vnútorného fungovania prúdovej šifry. Pracuje s ňou ako skrinkou s neznámym mechanizmom (blackbox-om), ktorej odovzdá ako vstup otvorený prúd údajov a následne dostane výstupný prúd spracovaných šifrovaných údajov.

Základom útoku je, že môžeme nahradiť n^d všetkých možností nelineárnych členov iba $2^d \cdot n$ nastavenými variantmi. Následne vyriešime predspracovanú verziu n lineárnych rovníc s n neznámymi použitím iba n^2 bitových operácii. Z hľadiska časovej náročnosti je potrebné, aby bol hlavný polynóm dostatočne nízkeho stupňa.

Cube attack dostal svoje meno podľa toho, že pri svojom priebehu využíva výpočet súčtových "algebraických kociek". Prebieha to tak, že k voliteľných premenných, určujúcich kocku, je nastavených na všetky možné kombinácie hodnôt pričom všetkým ostatným premenným sa hodnota nezmení. Ukážeme si to na nasledovnom polynóme s piatimi premennými x_1, x_2, x_3, x_4, x_5:

$$x_1x_3 + x_1x_2x_3x_4 + x_2x_4 + x_5$$

Zvolené nastaviteľné premenné budú x_1 a x_2. Kombináciou možností ich hodnôt dostaneme tieto štyri polynómy:

$$x_1 = 0, x_2 = 0 : x_5$$
$$x_1 = 0, x_2 = 1 : x_3 + x_5$$
$$x_1 = 1, x_2 = 0 : x_4 + x_5$$
$$x_1 = 1, x_2 = 1 : x_3 + x_3x_4 + x_4 + x_5$$

Algebraická kocka C_I je teda definovaná množinou premenných I s veľkosťou k a je tvorená 2^k vektormi nad týmito premennými. Nakoniec je vypočítaný súčet všetkých polynómov určených vektormi v kocke.

2.2 Maximálne termy a superpolynómy

Uvažujme prúdovú šifru s typickou konštrukciou pozostávajúcou z LFSR určeného lineárnou krokovaciu funkciou a následným spracovaním jeho obsahu nelineárnou funkciou F. V tomto prípade stupeň d pre rovnice určené šifrou bude totožný so stupňom funkcie F. Navyše, jej stupeň nebude s výpočtom nasledujúcich bitov vzrastať, ako by tomu bolo pri LFSR určenom nelineárnym spôsobom.

V binárnej sústave platí, že $x^k = x$, a teda nezáleží na hodnote prípadnej mocniny premennej, pretože výsledná hodnota výrazu bude aj tak rovnaká. Vďaka tomu môžeme pre zjednodušenie zápisu každý člen polynómu zapísať ako t_I, kde I je množinou premenných v ňom obsiahnutých.

Najskôr popíšeme princíp hlavných polynómov a ich variant, ktorý je využívaný pri Cube attack-u. Majme polynóm p a podmnožinu premenných jeho prvkov I. Polynóm môžeme upraviť na tvar s členmi, ktoré sú nadmnožinou premenných $x_1, ..., x_k$ z I a sumu ostatných členov, ktoré nie sú:

$$p(x_1, ..., x_n) = t_I \cdot p_{S(I)} + q(x_1, ..., x_n)$$

$S(I)$ označuje nejakú podmnožinu z $n - k$ premenných okrem k premenných tvoriacich člen t_I . Polynóm $p_{S(I)}$ nazývame *superpolynómom* z I v polynóme p. Takýto polynóm neobsahuje žiadnu spoločnú premennú s členom t_I a každému členu v $q(x_1, ..., x_n)$ chýba aspoň jedna premenná obsiahnutá v t_I. *Maximálny term* pre p je taký člen t_I, pre ktorý platí, že stupeň jemu prislúchajúceho superpolynómu $p_{S(I)}$ je rovný jedna. To znamená, že daný superpolynóm je lineárny ale, zároveň polynóm nie je konštantný.

Nové pojmy si názorne ukážeme na nasledujúcom príklade polynómu s 5 premennými, ktorý je stupňa 3:

$$x_1x_2x_3 + x_1x_3 + x_1x_4 + x_2x_3 + x_3x_4x_5$$

upravíme na tvar:

$$x_1x_3(x_2 + 1) + x_1x_4 + x_2x_3 + x_3x_4x_5$$

kde x_1x_3 predstavuje maximálny term a $x_2 + 1$ je príslušným superpolynómom.

Každá podmnožina prvkov I obsahujúca k premenných polynómu definuje booleanovskú kocku C_I o dimenzii k, tvorenú 2^k vektormi. Vektory sú tvorené priradením všetkým možných kombinácii hodnôt (0/1) premenným v množine I. Každý vektor kocky C_I určuje nový odvodený polynóm s $n-k$ premennými. Sčítaním všetkých 2^k odvodených polynómov v kocke dostaneme nový polynóm p_I, ktorého vlastnosti sa ukážu ako veľmi užitočné.

Zamerajme sa najprv na polynóm $q(x_1, ..., x_n)$. Nech J je množinou premenných tvoriacich člen t_J tohto polynómu. Každému t_J musí určite chýbať aspoň jedna nejaká premenná z množiny I. Z toho vyplýva, že za zachovania všetkých hodnôt ostatných premenných zostávajú vždy dve možné hodnoty chýbajúcej premennej, ktorá hodnotu termu neovplyvňuje. Čiže v celkovom súčte sa bude člen t_J vyskytovať párny počet krát. V binárnej sústave sa preto z tohto súčtu s každým párom vynuluje a jeho hodnotu neovplyvňuje.

Člen t_I sa vyhodnotí ako jedna práve vtedy, keď všetky premenné z množiny I budú nastavené na hodnotu 1. To znamená, že $p_{S(I)}$ bude v súčte práve jeden krát. Dôsledkom je, že pre každý polynóm p a podmnožinu jeho premenných I platí, že $p_I \equiv p_{S(I)}$ $(mod\ 2)$, ich zvyšok po delení 2 je taký istý. V binárnej sústave sa teda ich hodnoty rovnajú.

To znamená, že suma 2^k polynómov odvodených z pôvodného polynómu p dosadeným všetkých možných kombinácii hodnôt k premenných množiny I, bude obsahovať iba členy zo superpolynómu $p_{S(I)}$. Stupeň výsledného polynómu bude teda nižší aspoň o k. Ak je člen t maximálnym termom, dostaneme lineárnu rovnicu superpolynómu. Aby sme toho docielili, je potrebné zvoliť množinu I obsahujúcu presne $d-1$ premenných, čiže $k = (d-1)$.

2.3 Náhodné polynómy

Náhodný polynóm stupňa d je taký polynóm, v ktorom sa každý jeho možný člen vyskytuje s pravdepodobnosťou 0,5. Teda pre ľubovoľne vybraný člen je rovnaká pravdepodobnosť, že sa v polynóme vyskytovať bude ako tá, že sa v ňom vyskytovať nebude. Pre *d-náhodný polynóm* platí, že každý člen stupňa d s jednou tajnou a $d-1$ verejnými

premennými je jeho členom s pravdepodobnosťou 0,5. Všetky ostatné členy môžu byť obsiahnuté ľubovoľne.

V náhodnom aj d-náhodnom polynóme každý člen t zložený z $d-1$ verejných premenných má extrémne vysokú pravdepodobnosť byť maximálnym termom. Jemu prislúchajúci superpolynóm je maximálne stupňa 1, pričom konštantný je iba ak neobsahuje ani jednu premennú x_i. Pravdepodobnosť nastania takejto situácie je pri d-náhodnom polynóme iba 2^{-n}.

Pre vygenerovanie n maximálnych termov nepotrebujeme nezávislé podmnožiny premenných I, takže nám stačí $d + log_d n$ nastaviteľných verejných premenných pre vytvorenie n rôznych maximálnych termov, keďže platí:

$$\binom{d + log_d n}{d} = \binom{d + log_d n}{log_d n} \approx d^{log_d n} = n$$

Po zvolení n náhodných maximálnych termov definujeme maticu s rozmermi $n \times n$, ktorá bude obsahovať im zodpovedajúce superpolynómy. Jednotlivými riadkami matice budú superpolynómy, a každý stĺpec určuje hodnotou 0/1 prítomnosť i-tej premennej z kľúča v superpolynóme. Ak je matica regulárna vypočítame a uložíme jej invertovanú maticu A^{-1}. Ak je matica singulárna, teda jej hodnosť nie je rovná počtu jej riadkov, neexistuje k nej inverzná matica. A pre d-náhodný polynómy je náhodná matica, ktorej každý prvok je rovný hodnote 0/1 s rovnakou pravdepodobnosťou 0,5. Pravdepodobnosť, že takáto matica nebude singulárna ale regulárna je:

$$\prod_{i=1}^{n}(1 - 2^{-i}) \approx 0,28879$$

Výpočtom niekoľkých ďalších rezervných maximálnych termov môžme túto pravdepodobnosť priblížiť skoro až na rovnú 1.

2.4 Prípravná fáza

V prvej, prípravnej, fáze je cieľom nájdenie maximálnych termov t_I a im prislúchajúcich superpolynómov $p_{S(I)}$ pre daný hlavný polynóm. Konštantný člen superpolynómu vieme vypočítať sčítaním všetkých hodnôt polynómu p (modulo 2) na premenných kľúča a vs-

tupných bitoch rovných nule okrem d-1 premenných v sumačnej kocke C_I. Koeficient premennej x_i v superpolynóme vypočítame sčítaním dvoch hodnôt. Prvou je suma všetkých hodnôt polynómu p na vstupných bitoch, ktoré sú rovné nule všade okrem sumačnej kocky C_I. Druhú hodnotu určíme ako sumu všetkých hodnôt polynómu p na vstupných bitoch, ktoré sú nula všade okrem sumačnej kocky a premennej x_i nastavenej na jedna.

Najprv musíme určiť veľkosť k množiny premenných I. Náhodne skúšame čísla k, a vypočítame hodnotu superpolynómu sčítaním nad kockou C_I. Suma bude konštantná, ak je množina I príliš veľká. Ak je množina I naopak príliš malá, superpolynóm bude pravdepodobne nelineárny.

V prípravnej fáze nájdeme koeficienty pre n rôznych superpolynómov. Každý superpolynóm má, vrátane konštantného člena, $n+1$ členov, na ich výpočet potrebujeme spraviť sumu booleanovskej kocky pre $n+1$ kľúčov. Je potrebné $n \cdot (n+1) * 2^{d-1}$ výpočtov funkcie F. Vypočítame tiež inverznú maticu zo získaných lineárnych vzťahov, čo vyžaduje nanajvýš n^3 operácii.

2.5 Aktívna fáza

V aktívnej fáze vypočítame sumu booleanovskej kocky pre n lineárnych vzťahov superpolynómov. Každá suma si vyžaduje 2^{d-1} vyhodnotení funkcie F. Potrebujeme ešte vynásobiť výsledný vektor za použitia už invertovanej matice, na čo budeme potrebovať ďalších n^2 operácii.

Celý útok požaduje len správne určenie stupňa d funkcie F, a možnosť výpočtu tejto funkcie. Nie je vyžadovaná znalosť tejto funkcie ani jej koeficientov. Takisto priebeh útoku nijako neovplyvňuje, či sa používa jedno, alebo niekoľko rôznych LFSR pre výpočet.

Pre prúdové šifry využívajúce viac LFSR, ktorých výstupy sú spojené binárnou operáciou XOR je Cube attack rovnako jednoduchý, ako keby bolo použité iba jedno LFSR (na rozdiel od mnohých iných útokov). Ak sú totiž použité LFSR nízkeho stupňa, tak aj ich výsledok po XOR-e bude nízkeho stupňa, a je možné použiť ten istý postup. Princípy Cube attack-u je možné dokonca použiť aj proti blokovým šifrám.

Celková výpočtová zložitosť útoku je v 1. fáze: $n \cdot (n+1) \cdot 2^{d-1} + n^3$ a v 2. fáze:

$n \cdot 2^{d-1} + n^2$. Reálna zložitosť Cube attack-u je napríklad pri šifre Trivium s počtom úvodných krokov 672 iba 2^{19} bitových operácii a pri počte úvodných krokov 735 je to 2^{30} operácii. Pre druhú popísanú funkciu, Toyocrypt, je zložitosť útoku bez ďalších vylepšení oveľa väčšia. V prípravnej fáze je rovná 2^{76} a v aktívnej fáze pribudne ďalších 2^{69}, taktiež počet vyžadovaných dvojíc bitov pred a po zašifrovaní je značne vysoký, $2^{62} \cdot 128$.

Kapitola 3

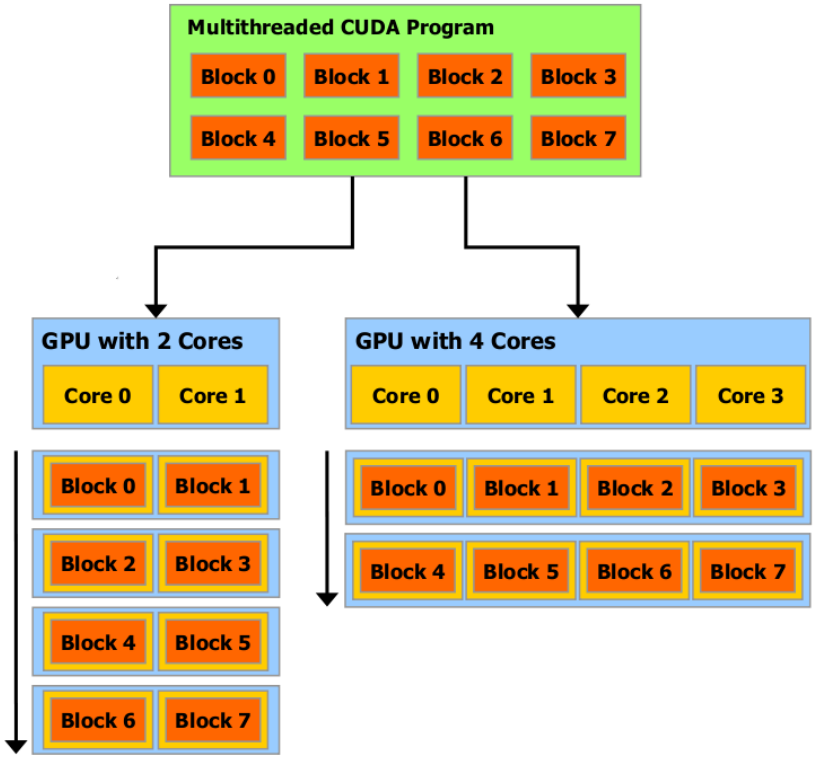
Cuda

3.1 Úvod do Cudy

Cuda je architektúra pre paralelné výpočty na kompatibilných grafických kartách. Procesor grafickej karty (GPU) má v porovnaní s CPU počítača väčšie množstvo tranzistorov na výpočtové účely a menej na riadenie toku programu. GPU predstavuje mnohojadrový procesor umožňujúci paralelné výpočty veľkému množstvu vlákien. Zároveň má vysokú dátovú priepustnosť. Preto je pri paralelnom výpočtovom spracovaní veľkého množstva dát niekoľkonásobne výkonnejšie oproti CPU.

Pre Cudu existuje programovacie prostredie, umožňujúce používať jazyk C. Časť priamo riadiaca spracovanie dát na grafickej karte je písaná v jazyku Cuda C, čo je vlastne tradičné C s niekoľkými novými formami zápisu. Pre pochopenie fungovania Cuda kódu sú dôležité tri kľúčové abstrakcie: hierarchia skupín vlákien, zdieľanej pamäte a synchronizácie [6].

Celkový problém je rozdelený na niekoľko podproblémov, ktoré môžu byť riešené paralelne a nezávisle od seba v blokoch vlákien. Jednotlivé podproblémy môžu byť riešené paralelne a kooperatívne vrámci svojho bloku vlákien. Tento prístup umožňuje škálovateľnosť problému a zároveň aj kooperáciu vlákien pri riešení podproblému. Každý blok môže byť na ľubovoľnom jadre procesora, nezávisle od poradia ostatných, čo umožňuje plné využitie výpočtovej kapacity pri všetkých počtoch jadier.



Obr. 3.1: Spracovanie blokov podľa použitého GPU

3.2 Kernel

Cuda C rozširuje jazyk C o definíciu funkcií, nazývaných kernely. Kernel sa pri zavolaní vykoná paralelne n-krát ako n vlákien. Kernel sa definuje špecifikátorom __global__. Konkrétny počet vlákien, ktoré sa majú vykonať, sa pri volaní kernelu určí syntaxou <<< ... >>>. Každé vykonávané vlákno kernelu má unikátne identifikačné číslo, ktoré je prístupné cez premennú threadIdx. Všetky vlákna vykonávajú ten istý kód s rozličnými dátami, ktoré sú určené pomocou hodnoty threadIdx. Skupiny vlákien sú usporiadavané do blokov vlákien. Dimenziu bloku obsahuje vektorová premenná blockDim.

Tieto nové konštrukcie v jazyku Cuda C oproti štandardnému C znázorňuje nasledovný príklad výpočtu súčtu dvoch vektorov, kde úlohou každého vlákna je súčet n-tej dvojice prvkov dvoch vektorov:

```
// definicia kernelu
__global__ void VecAdd(float* A, float* B, float* C)
{
```

```
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // volanie kodu kernelu
    VecAdd<<<1, N>>>(A, B, C);
}
```

Pri volaní kernelu sa zadáva počet blokov a počet vlákien tvoriacich jeden blok. Toto znázorňuje nasledovný pseudokód:

$$KernelVolanie <<< PocBlokov, PocVlakienNaBlok >>> (Premenne...);$$

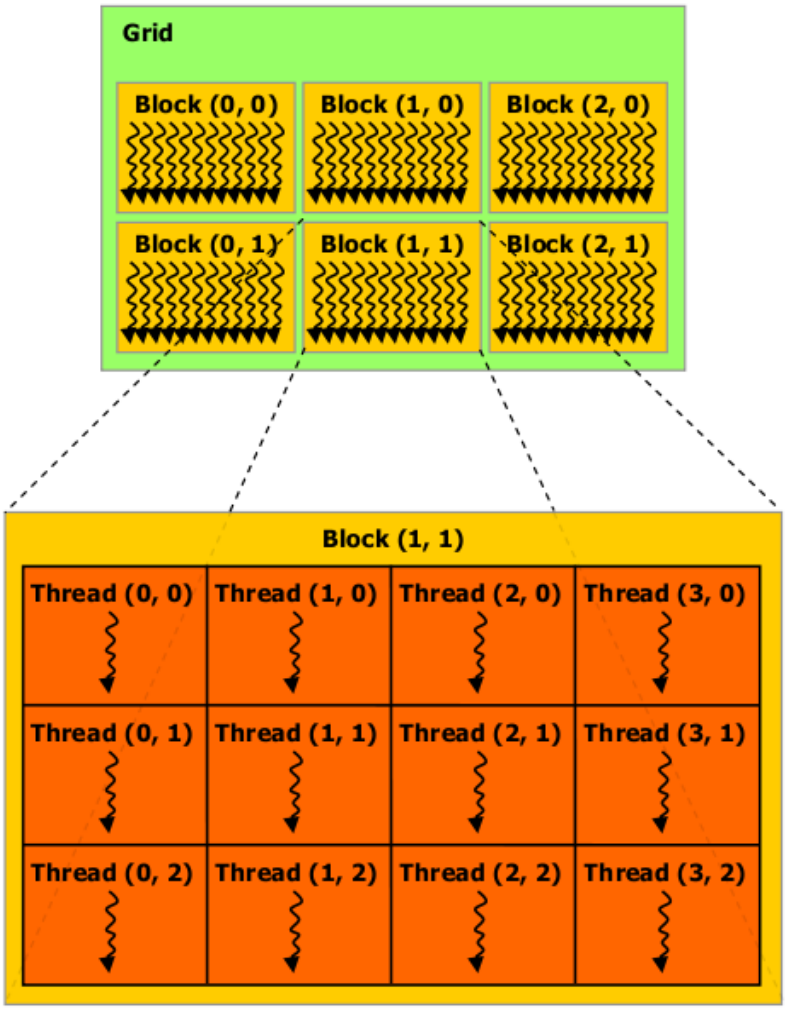
CUDA programovací model predpokladá, že bloky vlákien kernelu sa fyzicky vykonávajú na inom zariadení, ako je procesor, na ktorom beží C program, volajúci kernel. Ďalším predpokladom je, že hlavný procesor a CUDA zariadenie udržujú každý svoj vlastný pamäťový priestor. Program využívajúci CUDA prostredie potom pamäť prislúchajúcu kernelu spravuje CUDA volaniami.

3.3 Vlákna

Blok vlákien môže byť 1, 2 alebo 3-rozmerný. Premenná určujúca index vlákna je vektor s tromi hodnotami. Toto umožňuje prirodzený spôsob výpočtu pri práci s matematickými objektami ako je napríklad matica. Prepočet 2- a 3-rozmerného indexu na ID vlákna prebieha prirodzeným spôsobom podľa nasledujúceho vzorca:

$$ID = (x + y \cdot Dx + z \cdot Dx \cdot Dy),$$

kde Dx a Dy určujú veľkosť príslušných rozmerov bloku. Limit na počet vlákien v rámci jedného bloku je maximálny počet 512 vlákien. Kernel môže však byť vykonaný



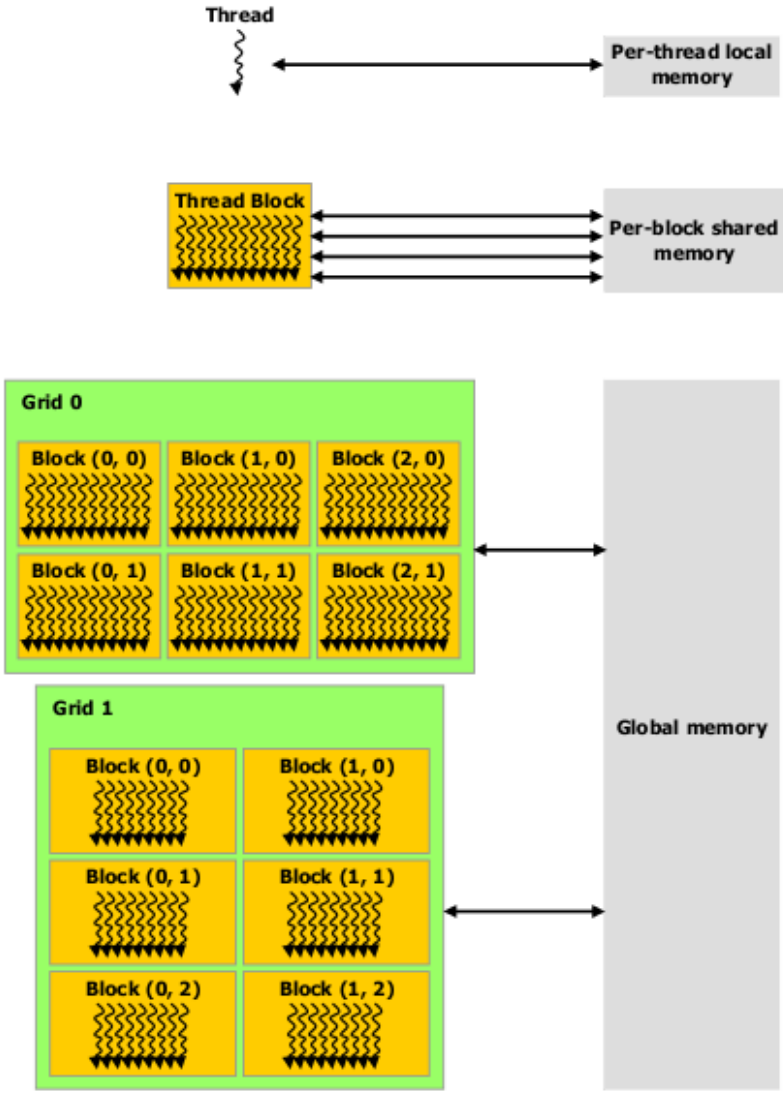
Obr. 3.2: Štruktúra blokov v gride

aj pomocou viacerých blokov. Bloky sú organizované v 1- alebo 2- rozmernom gride.

Každý blok v gride je jednoznačne určený pomocou premennej blockIdx, slúžiacej na podobný účel ako threadIdx pre vlákno. Dimenziu gridu blokov obsahuje vektorová premenná gridDim.

Dôležité je, aby nebol výpočet blokov závislý na poradí ich vykonávania, pretože môžu byť vykonané v ľubovoľnom poradí. Tiež nemôžu byť závislé od počtu jadier, na ktorých výpočet prebieha. V opačnom prípade výpočet nemusí prebehnúť podľa očakávaní.

V rámci jedného bloku vlákien je vymedzená zdieľaná pamäť, ktorá je prístupná všetkými vláknami tvoriacimi daný bloku. Vlákna v bloku je možné vzájomne synchronizovať použitím funkcie $\_\_syncthreads()$. Táto funkcia slúži ako bod programu, ktorého vykonanie je možné až po jeho dosiahnutí všetkými vláknami v bloku. Bloky



Obr. 3.3: Úrovne pamäte vrámci Cudy a prístup k nim

vlákien medzi sebou synchronizovať možné nie je.

3.4 Hierarchia pamäte

Každé vlákno v CUDA programe má prístup k niekoľkým rôznym typom pamäti:

1. vlastná privátna pamäť vlákna
2. pamäť zdieľaná s ostatnými vláknami rovnakého bloku
3. globálna pamäť

Ku globálnej pamäti majú prístup všetky bloky kernelu ako aj rôzne kernely. Neexistuje teda úroveň pamäte, zdieľaná na úrovni kernelu. Na kopírovanie údajov do a z pamäte CUDA zariadenia slúži funkcia $cudaMemcpy()$. Táto funkcia má štyri parametre, prvými tromi sú: odkaz na cieľovú premennú, odkaz na zdrojovú premennú, veľkosť kopírovanej pamäte. Smer presunu, z hlavnej pamäte alebo do nej určuje posledný parameter volania – $cudaMemcpyHostToDevice/$ $cudaMemcpyDeviceToHost$. Správu pamäte v rámci zariadenia zabezpečuje dvojica funkcií na jej pridelenie a uvoľnenie, $cudaMalloc()$ a $cudaFree()$.

Pre deklaráciu globálnej pamäte na zariadení je vyhradený špeciálny špecifikátor $\_\_device\_\_$, ktorý sa píše pred samotnú deklaráciu premennej. Pre deklaráciu premennej zdieľanej medzi vláknami vrámci bloku slúži obdobný špecifikátor $\_\_shared\_\_$.

3.5 Programovacie prostredie

Momentálne sú podporované dve prostredia pre písanie programov v CUDE. Prvým je CUDA C a tým druhým je CUDA DRIVER API. CUDA C predstavuje jazyk C s čo možno minimom nových rozšírení k nemu. Kompilátor C pre kód s týmito CUDA rozšíreniami je nvcc. Ním skompilovaný zdrojový kód je potom vykonateľný na zariadený podporujúcom CUDA technológiu. Zdrojové súbory obsahujúce kód kernelu majú príponu '.cu'.

Nvcc kompiluje kód kernelu z jazyka C do sady inštrukcii PTX architektúry. Zvyšný kód, ktorý je určený pre hlavný procesor, je skompilovaný štandardným C kompilátorom. Kernel môže byť aj priamo písaný s použitím architektúry PTX, je to však náročnejší proces. Všetok PTX kód programu je skompilovaný driverom CUDA zariadenia až po spustení a jeho nahraní. Tento spôsob umožňuje programu schopnosť bežať na všetkých CUDA zariadeniach ako aj využiť prípadné výhody a vylepšenia novších verzii driveru. Kvôli potrebe kompilácie je však časovo náročnejší na dobu spustenia programu.

S nvcc je možné kompilovať aj v emulačnom móde pomocou voľby -deviceemu. Takýto kód bude bežať výhradne na CPU, pričom jednotlivé vlákna budú emulované a spracované ako vlákna CPU. Veľkou výhodou tohto módu je, že umožňuje jednoduché debugovanie kódu určeného pre CUDA zariadenie.

3.6 Inštalácia a nastavenia

Základnými požiadavkami na úspešné spustenie Cuda kódu je grafická karta, podporujúca túto technológiu a príslušný driver pre dané zariadenie. Ďalej je potrebné Microsoft Visual Studio ako prostredie pre vývoj v jazyku C s Cuda rozšíreniami. Toto prostredie podporuje aj režim kompatibility, v ktorom kód určený na paralelné spracovanie na GPU je namiesto toho sériovo spracovaný na CPU. Slúži to ako na účely jednoduchšieho debugovania tak aj na skúšobné spustenie v prípade absencie vhodnej grafickej karty. Podporované GPU sú najmä GeForce série 8, 9, 200, 400.

Prvým krokom procesu inštalácie je driver grafickej karty. Ten je možné, tak ako aj všetky ďalšie veci, ktoré spomenieme, stiahnuť na oficiálnych stránkach spoločnosti NVIDIA venovaných Cuda technológii. V čase písania tejto práce bola funkčná nasledujúca stránka: http://developer.nvidia.com/cuda-downloads/. Voľba konkrétneho súboru závisí od operačného systému a procesora.

V prípade absencie Visual Studia pokračujeme jeho stiahnutím a inštaláciou. Nachádza sa na stránkach spoločnosti Microsoft, vhodná aktuálna verzia je C++ 2010 Expres, ktorá je poskytovaná zadarmo, s potrebou registrovať produkt v priebehu prvých 30 dní. Prevziať potrebné súbory je možné na stránke: http://www.microsoft.com/express/Downloads/.

Nasleduje inštalácia Cuda Toolkit balíčka, obsahujúceho nástroje potrebné pre kompiláciu a spúšťanie Cuda aplikácii. Dôležité upozornenie je, že nová verzia Cuda Toolkit balíku môže vyžadovať taktiež novú verziu driveru, takže treba dať pozor na použitie jeho vhodnej verzie pre používaný balík. Nakoniec nainštalujeme GPU Computing SDK, obsahujúce aj ukážkové projekty pripravené pre prostredie Visual Studia.

Kapitola 4

Implementácia útoku

4.1 Základný princíp

Zameriame sa na šifru Toyocrypt. Útok, ktorý použijeme, bude pozmenenou variantou už popísaného Cube attack-u. Tento útok budeme implementovať v jazyku C a vhodné vybrané časti v jazyku Cuda C, pre rýchlejšie spracovanie na GPU oproti CPU. Pred štandardným Cube attackom najprv pozmeníme funkciu F tak, že bude iba rádu 4, teda zanedbáme členy vyššieho rádu. Tieto členy sú skoro vždy nulové a teda s vysokou pravdepodobnosťou výpočet neovplyvnia.

Ďalším možným postupom je nájdenie vhodnej funkcie G takej, že nová funkcia $H = F \cdot G$ bude nižšieho stupňa. Vďaka tomu sa výrazne zníži výpočtová náročnosť počas zvyšných častí útoku.

Môžeme využiť toho, že vo funkcii F sú vo všetkých členoch stupňa vyššieho ako 4 obsiahnuté dve premenné, s_{23} a s_{42}. Pre každý výstupný bit o_k jemu zodpovedajúcu rovnicu vynásobíme funkciou G, ktorá zníži stupeň rovnice na 3. Takéto funkcie vieme nájsť dve, jednu pre každú premennú. Sú to $(s_{23} - 1)$ a $(s_{42} - 1)$ [8]. Dostaneme teda dvojnásobný počet rovníc nízkeho stupňa, ktoré budú vždy pravdivé. Členy, ktoré boli vyššieho stupňa, sa všetky vykrátili, pretože obsahovali totožné premenné.

4.2 Simulácia Toyocryptu

Pre implementáciu útoku musíme najprv mať šifru, na ktorú budeme útočiť. Našim cieľom bude Toyocrypt, ktorý sme opísali sekcii 1.7 v prvej kapitole. Jeho algorit-

mus budeme simulovať niekoľkými funkciami a dátovými štruktúrami. Počiatočný stav LFSR inicializujeme lineárnou kombináciou kľúča a vektora otvoreného textu.

Pre zjednodušenie vynecháme úvodnú časť algoritmu, ktorou je hľadanie vhodnej maximálnej funkcie LFSR a namiesto toho použijeme fixnú hodnotu premenných, generujúcu takúto funkciu. Jednou z podmienok na šifrovací systém je, aby neexistovali "slabé" kľúče, čo znamená že rovnaká bezpečnosť by mala byť zachovaná pre všetky možné generované kľúče. Teda náš prístup by nemal nijako ovplyvniť výpočtovú zložitosť útoku, a nami použitá situácia môže reálne nastať, keďže použijeme platné hodnoty, ktoré môžu byť výsledkom hľadania fixného kľúča.

Používaný LFSR register dĺžky 128 bitov implementujeme poľom s rovnakou dĺžkou v počte prvkov, nazvaným $Lfsr[]$. Tap pozície budú uchovávané pomocou poľa $Taps[]$. Hodnota i-teho prvku poľa bude určovať či je daná pozícia tapovacou pozíciou. Pri krokovaní prepočet nasledujúceho bitu a posunutie obsahu celého registra zabezpečí funkcia $StepLfsr()$. Výpočet ďalšieho bitu pseudonáhodného prúdu bude počítať funkcia $ComputeLfsr$. Prúd otvoreného textu a samotný kľúč bude v uložený v poliach $IV[]$ a $Key[]$.

Dátové štruktúry:

```
unsigned short Lfsr[128];
unsigned short Taps[128];
unsigned short* IV;
unsigned short* Key;
```

Kód funkcie $StepLfsr()$**:**

```
/*vypocet dalsieho kroku LFSR*/
unsigned short StepLfsr() {
  /*posun celeho registra*/
  unsigned short NewBit = Lfsr[127];
  for (int i=127; i>=1; i--)
    Lfsr[i] = Lfsr[i-1];
  Lfsr[0] = NewBit;
```

```
  /*XOR tap pozicii s novym bitom*/
  if (NewBit == 1) {
    for (int i=0; i<TapsNum; i++)
      Lfsr[Taps[i]] = (Lfsr[Taps[i]] + 1) % 2;
  }

  return NewBit;
}
```

Výpočet i-teho bitu, ktorý budeme využívať počas prípravnej aj aktívnej fázy, bude spracovávať funkcia *CipherBit*(). Jej vstupmi sú vektor kľúča, inicializačný vektor a poradie bitu v prúde otvoreného textu, ktorý chceme získať. Najprv inicializujeme dátové štruktúry Toyocryptu, potom krokujeme LFSR potrebný počet krát a nakoniec uplatníme na jeho obsah nelineárnu funkciu.

Kód funkcie *CipherBit*()**:**

```
unsigned short CipherBit
(unsigned short* Key, unsigned short* IV, int Order) {
        InitLfsr(Key, IV);
        InitTaps();
        for (int i=0; i<=Order; i++)
                StepLfsr();
        return (ComputeLfsr() + IV[Order]) % 2;
}
```

4.3 Prípravná fáza

V tejto časti popíšeme implementáciu prípravnej fázy Cube attack-u. Túto fázu zabezpečuje funkcia *PreProcess*, ktorá sa skladá z cyklu hľadajúceho maximálne termy a superpolynómy, a z inverzie výslednej matice, ktorú týmto procesom získame. Hľadanie maximálneho člena zabezpečuje funkcia *NextTerm* a výpočet prislúchajúceho superpolynómu má na starosti funkcia *GetSuperPoly*. Obe funkcie ako parameter volania

dostanú poradové číslo hľadaného maximálneho termu/superpolynómu.

Dátové štruktúry

```
int Degree;
int KeyLength = 128;
int** Maxterms;
unsigned short** SuperPolys;
int* UsedTerms;
unsigned short** KeyMatrix;
```

Kód funkcie *PreProcess*()

```
/*pripravna faza*/
void PreProcess() {
  DegFind();
  //cyklicke hladanie maximalnych temov
  for (int i=0; i<128; i++) {
    InitTerm(i);
    GetSuperPoly(i);
    while (!NonZero(i)) {
      NextTerm(i);
      GetSuperPoly(i);
    }
  }
  InvertMatrix();
}
```

Kód funkcie *GetSuperPoly*()

```
/*vypocet superpolynomu prislusneho termu*/
void GetSuperPoly(int order) {
  unsigned short Vector[Degree-1];
  unsigned short Key[KeyLength];
```

```
    unsigned short IV[KeyLength];
    for (int i=0; i<(int)pow(2.0,Degree-1); i++) {
      int meta = i;
      for (int j=Degree-2; j>=0; j--) {
        Vector[j] = meta / (int)pow(2.0,j);
        meta = meta % (int)pow(2.0,j);
      }
      for (int j=0; j<KeyLength; j++) {
        for (int k=0; k<KeyLength; k++) {
          Key[k] = 0;
          IV[k] = 0;
        }
        for (int k=0; k<Degree-1; k++)
          IV[Maxterms[order][k]] = Vector[k];
        SuperPolys[order][j] =
          (SuperPolys[order][j]+CipherBit(Key,IV,order)) % 2;
        Key[j] = 1;
        SuperPolys[order][j] =
          (SuperPolys[order][j]+CipherBit(Key,IV,order)) % 2;
      }
    }
}
```

4.4 Aktívna fáza

Na základe predspracovanej matice z prípravnej fázy môžeme následne v tejto fáze výpočtom vektora zloženého zo súm algebraických kociek získať tajný kľúč, ktorý bol použitý pri ich šifrovaní. Táto fáza prebieha vo funkcii *Active*(), ktorej jediným parametrom je práve tajný kľúč. Testovaný kľúč je náhodne vygenerovaný. Sumy kociek s pevným kľúčom prebiehajú obdobne ako v prípravnej fáze. Po tom, čo je vypočítaná sumačná matica je vynásobená s maticou z predchádzajúcej fázy, pričom ako ich výsledok získame hľadaný kľúč. Pre overenie správnosti ho môžeme následne

porovnať s použitým kľúčom. Vypočítaný kľúč je uložený v poli $SecretKey[]$.

Dátové štruktúry

```
unsigned short* Computed;
unsigned short* SecretKey;
```

Kód funkcie *Active()*

```
/*aktivna faza*/
void Active(unsigned short* Key) {
        /*vypocet vektora*/
        ComputeTerms(Key);
        /*vynasobenie matic*/
        Multiply();
}
```

4.5 Výsledky

Počas prípravnej fázy je základnou časťou výpočtu hľadanie členov, ktorých superpolynómy budú lineárne. Polynómy šifry Toyocrypt pre jednotlivé šifrové bity nie sú ani zďaleka náhodné či d-náhodné. Preto je potrebné vyskúšať rádovo niekoľko násobné množstvo členov, kým je úspešne nájdený maximálny term. Vzhľadom na to zaberá táto časť väčšinu času z prípravnej fázy, až vyše 99%. Ďalšie výpočty, ktoré v tejto fáze prebehnú, ako napríklad invertovanie matice, prebiehajú konštantný a zanedbateľne krátky čas. Optimalizáciu je očividne potrebné zamerať práve na hľadanie maximálnych termov a ich superpolynómov. Vďaka tomu, že výpočet jednotlivých bitov superpolynómu pre zvolený člen je úplne nezávislý, je ho možné spracovať paralelne. Môžeme teda pri ňom využiť výhod Cuda technológie a presunúť túto operáciu na grafickú kartu. Realizáciou tohto prístupu sme naozaj dosiahli zlepšenie výkonu, ako ukazujú nasledovné časy:

výpočet superpolynómov na CPU: 39,0s
výpočet superpolynómov na GPU: 18,7s

porovnávajúce dĺžku výpočtu spriemerovanú z 10 meraní pre eliminovanie možnej nepresnosti. Maximálnou odchýlkou nameraných hodnôt od uvedeného priemeru bolo 0,2 sekundy. Naozaj teda došlo k zrýchleniu približne o 20 sekúnd, t.j. vyše 50% z celkovej doby útoku. Výpočty prebehli na dvojjadrovom procesore Intel Pentium 1.5 GHz a grafickej karte nVidia 7800GT.

Pri aktívnej fáze útoku nemá využitie grafickej karty na výpočet pozitívny efekt na jeho rýchlosť. Násobenie matíc je síce priam ukážkový príklad efektívne paralelizovateľného algoritmu, ale výsledkom je len jednozmerná matica o dĺžke kľúča, t.j. 128 bitov. To je priveľmi nízky počet vlákien (128), pretože pre reálne zrýchlenie by sme potrebovali rádovo tisíce paralelných vlákien. Pri čísle, ktoré máme, sa výhoda počtu stratí v nižšej výpočtovej kapacite jednotlivých jadier GPU. Nezanedbateľný čas stojí aj samotné prenášanie spracovaných dát pred a po výpočte. Celá fáza však vďaka malému množstvu potrebných operácii prebieha len niekoľko desiatok milisekúnd, čo je prijateľne krátky čas a potreba optimalizácie vlastne ani nenastáva.

Veríme, že je možné dosiahnúť ešte niekoľkonásobne vyššie zrýchlenie implementovaného útoku. Napríklad eliminovaným drahých prenosov dát pri využitý Cuda technológie napríklad prípadne prenosom celej prípravnej fázy na grafickú karty. Pomôcť môže tiež lepšie paralelizovanie algoritmu, predovšetkým hľadania maximálnych termov, kde sa zlepšenie najviac ukáže v celkovom výslednom čase. Taktiež je možné optimalizovať samotný výpočet, dosiahneme tým však zmenu v spracovaní ako na GPU tak aj na CPU a pomer sa teda nezmení v plnej miere úprav.

Záver

Popísali sme základné princípy konštrukcie prúdových šifier. Podrobnejšie sme sa venovali vybraným šifrám. Vysvetlili sme algoritmus Cube attack a spomenuli sme aj iné metódy útokov na takýto typ šifier. Potom sme sa zamerali na technológiu Cuda umožňujúcu použitie grafickej karty na výpočty, pričom výhodou je veľký počet jadier, na ktorých môže byť vypočet paralelne spracovaný. Napriek slabšej výpočtovej sile jednotlivých jadier GPU môžeme pri ich maximálnom využití dosiadnuť výrazného zrýchlenia oproti spracovaniu na CPU.

Cube attack sme úspešne implementovali na zvolenú šifru a následne pre vybrané časti, v ktorých je možné využiť paralelizmu, sme použili pre výpočet grafickú kartu. Porovnali sme oba spôsoby výpočtu a algoritmus sme paralelizovali dostatočne na to, aby prebiehal v paralelnom prostredí rýchlejšie napriek potrebe časovo drahého prenášania dát z a do hlavnej pamäte. Nami dosiahnuté zlepšenie dosahovalo okolo 50% a navrhli sme aj spôsoby možného ďalšieho vylepšenia.

Literatúra

[1] Nicolas T. Courtois and Willi Meier, *Algebraic Attacks on Stream Ciphers with Linear Feedback*, ISBN:3-540-14039-5, 2003, available at. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.8379&rep=rep1&type=pdf`.

[2] Nicolas T. Courtois, *Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt* , ISBN:3-540-00716-4, 2002, available at. `http://eprint.iacr.org/2002/087`.

[3] Itai Dinur and Adi Shamir, *Cube Attacks on Tweakable Black Box Polynomials* , ISBN: 978-3-642-01000-2, 2009, available at. `http://eprint.iacr.org/2008/385.pdf`.

[4] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir , *Cube Testers and Key Recovery Attacks On Reduced-Round MD6 and Trivium*, in Proc. FSE, 2009, available at. `http://www.131002.net/data/papers/ADMS09.pdf`.

[5] Jean-Philippe Aumasson, *Cube Testers: Theory and Practice* , available at. `http://eprint.iacr.org/2009/015.pdf`.

[6] NVIDIA, *NVIDIA CUDA C Programming Guide version 3.2* , available at. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[7] Information Security Research Centre, Queensland University of Technology, *Evaluation of TOYOCRYPT-HR1* , available at. `www.ipa.go.jp/security/.../1090_Analysis_of_TOYOCRYPT-HRI.pdf`

[8] Aileen Zhang , Chu-Wee Lim , Khoongming Khoo , Lei Wei , and Josef Pieprzyk, *Extensions of the Cube Attack based on Low Degree Annihilators* , available at. eprint.iacr.org/2009/049.pdf

[9] Håvard Raddum, *Cryptanalytic Results on Trivium* , available at. www.ecrypt.eu.org/stream/papersdir/2006/039.ps

[10] Johannes Buchmann, Jintai Ding, Mohamed Saied Emam Mohamed, and Wael Said Abd Elmageed Mohamed, *MutantXL: Solving Multivariate Polynomial Equations for Cryptanalysis* , available at. drops.dagstuhl.de/opus/volltexte/2009/1945/

[11] A. Clark, E. Dawson, J. Fuller, J. Golić, H-J. Lee, W. Millan, S-J. Moon, L. Simpson, *The LILI-128 Keystream Generator* , available at. kowon.dongseo.ac.kr/~hjlee/LILI-II.pdf

[12] NVIDIA, *NVIDIA CUDA C GETTING STARTED GUIDE FOR MICROSOFT WINDOWS* , available at. http://developer.download.nvidia.com/compute/cuda/3_2_prod/docs/Getting_Started_Windows.pdf

[13] Claus Diem, *The XL-Algorithm and a Conjecture from Commutative Algebra* , available at. www.math.uni-leipzig.de/~diem/preprints/xl.ps

[14] John Mattsson, *Stream Cipher Design* , available at. http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2006/rapporter06/mattsson_john_06111.pdf

[15] Christophe De Cannière, *Trivium, A Stream Cipher Construction Inspired by Block Cipher Design Principles* , eSTREAM, ECRYPT Stream Cipher, 2005, available at. www.cosic.esat.kuleuven.be/publications/article-850.ps

[16] Cyril Zeller, *Tutorial CUDA* , available at. people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf