Зміст

[Перелік умовних позначень, символів, одиниць, скорочень і термінів 12](#_Toc358976600)

[Вступ 13](#_Toc358976601)

[1 Кубічні атаки 15](#_Toc358976602)

[1.1 Опис атаки 16](#_Toc358976603)

[1.2 Математичне підґрунтя та основні поняття 18](#_Toc358976604)

[1.3 Стадії атаки 20](#_Toc358976605)

[1.3.1 Попередні обчислення 20](#_Toc358976606)

[1.3.2 Відновлення бітів ключа 24](#_Toc358976607)

[1.4 Підходи до підсилення атаки 25](#_Toc358976608)

[Висновки до розділу 1 28](#_Toc358976609)

[2 Часова складність кубічної атаки 29](#_Toc358976610)

[2.1 Часова складність на прикладі простого потокового шифру 29](#_Toc358976611)

[2.2 Формула для часової складності кубічної атаки 33](#_Toc358976612)

[Висновки до розділу 2 36](#_Toc358976613)

[3 Дослідження кількості булевих функцій з макстермами малого степеня 37](#_Toc358976614)

[3.1 Кількість сприятливих функцій для кубічної атаки з макстермами першого степеня 37](#_Toc358976615)

[3.2 Програмна перевірка теоретичних співвідношень 42](#_Toc358976616)

[Висновки до розділу 3 45](#_Toc358976617)

[4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 47](#_Toc358976618)

[4.1 Аналіз та відповідність нормам робочого простору для користувачів комп’ютерів 47](#_Toc358976619)

[4.1.1 Робоче приміщення 47](#_Toc358976620)

[4.1.2 Повітря робочої зони 49](#_Toc358976621)

[4.1.3 Виробниче освітлення 51](#_Toc358976622)

[4.1.4 Виробничий шум 54](#_Toc358976623)

[4.1.5 Випромінювання 54](#_Toc358976624)

[4.2 Електробезпека 55](#_Toc358976625)

[4.3 Безпека в надзвичайних ситуаціях 56](#_Toc358976626)

[Висновки до розділу 4 57](#_Toc358976627)

[Висновки 58](#_Toc358976628)

[Перелік посилань 60](#_Toc358976629)

[Додаток А. Код алгоритмів кубічної атаки 63](#_Toc358976630)

[Додаток Б. Код генератора потенційно сприятливих булевих функцій 70](#_Toc358976631)

[Додаток В. Код програми для підрахунку кількості сприятливих функцій 82](#_Toc358976632)

[Додаток Г. Код допоміжних програмних модулів 86](#_Toc358976633)

# Перелік умовних позначень, символів, одиниць, скорочень і термінів

|  |  |
| --- | --- |
|  | поле Галуа порядку p |
|  | поле Галуа характеристики p зі степенем розширення m |
| ***K*** | множина секретних змінних шифру |
| ***IV*** | множина відкритих змінних шифру |
| ***макстерм*** | терм вигляду v1v2…vd, де vi – відкрита змінна (біт вектора ініціалізації), d – степінь макстерма |
| ***суперполіном*** | лінійний многочлен від секретних змінних (бітів ключа), результат сумування виходу функції по всім значенням змінних, що входять у відповідний макстерм |

# Вступ

В умовах сучасних масштабів електронного документообігу важливим питанням є захист інформаційних ресурсів. Для забезпечення надійного захисту необхідне поєднання організаційних, технічних і криптографічних методів. Зокрема, важливу роль у криптографічному захисті інформації відіграють потокові криптосистеми (або, більш загально, криптосистеми із відкритими змінними). Оскільки вони є широко розповсюдженими, то задача розробки, дослідження, удосконалення методів криптоаналізу таких схем шифрування задля забезпечення їх стійкості є важливою в практичному і теоретичному аспекті. Інструментом для аналізу в цьому випадку можуть виступати різні види криптоатак, одним з яких є новий перспективний вид атаки – кубічні атаки, вперше описані у статті «Cube Attacks On Tweakable Black Box Polynomials» [1], опублікованій дослідниками Дінуром та Шаміром (Itai Dinur, Adi Shamir) у 2008 році й застосовані до учасника конкурсу eStream потокових шифрів під назвою Trivium. Пізніше, у 2009 році, світ побачила оновлена і доповнена версія цієї статті [2], вже видана у складі збірника лекцій EUROCRYPT.

Дослідження особливостей застосування кубічної атаки дозволяє виявити класи шифрів, що є вразливими до цієї атаки, а отже, потребують удосконалення, а також умови стійкості криптосистем до цього виду атаки. Таким чином, аналіз кубічних атак є *актуальною* проблемою, що може вплинути як на удосконалення потокових криптосистем, так і на розвиток методів їх криптоаналізу.

**Об’єктом дослідження** роботи є саме *кубічні атаки* на криптосистеми з наявними відкритими змінними, зокрема, на потокові криптосистеми. У роботі розглядається питання застосовності кубічних атак за умови існування макстермів малого степеня.

**Завдання роботи**. Отримати математичний вираз для залежності кількості сприятливих функцій від кількості відкритих і секретних змінних, а також степеня функції за умови наявності макстермів лише малого степеня (зокрема, першого степеня) і перевірити теоретичні результати за допомогою програмних розрахунків. Проаналізувати імовірність успіху атаки для функцій з різними кількостями змінних.

**Висновки до роботи**.

Отримано математичне співвідношення для кількості сприятливих функцій довільного степеня від кількості відкритих та секретних змінних для випадку існування макстермів лише першого степеня. Теоретичні результати співпадають з даними, отриманими завдяки програмній реалізації алгоритмів кубічної атаки, генератора потенційно сприятливих булевих функцій і програми для підрахунку кількості сприятливих функцій за заданим степенем функції і кількістю змінних.

Було виявлено, що при збільшенні різниці між кількістю відкритих і секретних змінних імовірність успіху атаки прямує до 1. Якщо ж дана різниця є мінімальною (зокрема, нульовою), то із збільшенням кількості змінних, за умови наявності лише макстермів першого степеня, ймовірність успішної атаки є близькою до 0. Тому доцільніше використовувати кубічну атаку з більшими степенями макстермів, допускаючи існування у функції макстермів різних степенів.

# ****1 Кубічні атаки****

Кубічні атаки – це новий перспективний інструмент криптоаналізу. Одним із перших видів атаки подібного типу була алгебраїчна диференціальна атака на вектор ініціалізації (Algebraic IV Differential Attack, AIDA), описана Вільхабером у [1] і пізніше вдосконалена в [2]. Вільхабер застосував її для спрощеного варіанту шифру Trivium (зі зменшеним числом раундів ініціалізації, 576 замість 1152). У результаті було отримано лінійні рівняння для бітів ключа в термінах сум вихідних бітів шифру, які можуть бути застосовані для атаки шифру. Але Вільхабер не запропонував жодного систематичного методу отримання таких рівнянь. Дану процедуру вперше описали саме Ітаі Дінур та Аді Шамір у статті, опублікованій у криптографічному архіві ePrint [3], а потім на конференції Eurocrypt 2009 [4], де вже були наведені лінійні рівняння для варіантів Trivium з 672 і 767 раундами ініціалізації.

Після успішного застосування до шифру Trivium атака набула подальшого розвитку, що відображено у статтях [5-10]. При чому, окрім шифру Trivium, був проведений криптоаналіз й інших крипто примітивів, зокрема, послабленої версії алгоритму MD6 [8], а також поточних шифрів Courtois Toy Cipher [9] і GRAIN-128 [10]. Як бачимо, атака дійсно претендує на універсальність.

У наступних розділах розглядаються основні поняття кубічної атаки, її детальний опис та деякі методи для підсилення ефективності базового варіанту атаки.

## ****1.1 Опис атаки****

Майже кожен криптографічний алгоритм може бути описаний як деякий поліном над полем GF(2), що містить як секретні (біти ключа), так відкриті змінні (біти відкритого тексту або вектора ініціалізації). Криптоаналітик може підбирати значення відкритих змінних таким чином, щоб отримати систему рівнянь певного виду, що залежать від секретних змінних, і знайти її розв'язок. Таким чином, необхідний опис підходу, що визначає, як само треба змінювати відкриті змінні, щоб мати змогу побудувати розв'язувану систему рівнянь. Розглянемо це питання з точки зору потокових криптосистем.

**Кубічна атака є однією з різновидів алгебраїчних атак, що можуть застосовуватися до потокових криптосистем. Кожен біт вихідної послідовності потокового шифру інтерпретується як значення деякої булевої функції (поліному), що залежить від бітів вектору ключа та вектора ініціалізації. Таким чином, нехай К – множина бітів ключа, а IV – бітів вектора ініціалізації. Тоді вищевказана функція має вигляд:** .

Однією з суттєвих переваг кубічної атаки є можливість взаємодіяти з криптосистемою як з «чорною скринею», тобто, навіть не маючи жодних відомостей про алгоритм шифрування та введення ключа, атаку можна застосувати. Це усуває необхідність технічного втручання у шифруючі пристрої та математичних досліджень у разі прихованості алгоритму шифрування, що дозволяє значно спростити та пришвидшити процес атаки на криптосистему.

Кубічна атака дуже схожа на *інтегральну атаку (integral attack)* та *диференціальну атаку вищого порядку (high order differential attack),* в яких проводиться сумування виходів криптографічного алгоритму по деяким підмножинам вхідних змінних. Але така схожість – це, переважно, наслідок особливостей поля GF(2), в якому додавання і віднімання – це одна і та ж операція.

**Щоб наочно продемонструвати потужність кубічної атаки, розглянемо дещо екстремальний приклад [4]. Нехай криптосистема побудована на ЛРЗ довжиною 10000 бітів із секретним характеристичним поліномом, регістр фільтрується S-блоками, кількість яких дорівнює 1000. Кожний S-блок є секретним відображенням 8 біт ЛРЗ у вихідний біт, всі S-блоки різні і структура зв’язку між ЛРЗ та S-блоками є прихованою. На кожному такті роботи шифр видає лише один біт виходу, що є булевою сумою виходів усіх S-блоків. Кожен біт ЛРЗ ініціалізується окремим секретним квадратичним поліномом, що залежить від 10000 бітів ключа та вектора ініціалізації. ЛРЗ працює велику і невідому кількість тактів без генерування виходу і лише один біт виходу для кожного вектора ініціалізації є доступним для криптоаналітика. Схема такого шифру зображена на рисунку 1.1.**

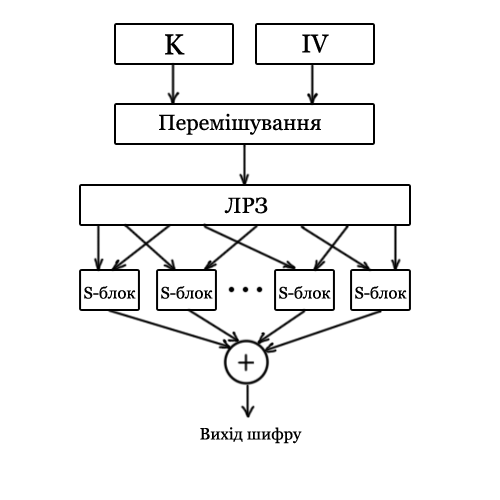


Рисунок 1.1 – Схема гіпотетичного шифру

**В умовах прихованості елементів структури шифру всі його особливості, що можуть бути використаними для кубічної атаки, – це той факт, що вихід кожного S-блоку є поліномом випадкового вигляду 16 степеня (отримано заміною квадратичних виразів для кожної з 8 вхідних змінних S-блоку), булева сума цих виходів також є поліномом 16 степеня (залежить від 10000 секретних та відкритих змінних), а також існує можливість достатньо вільно маніпулювати першим бітом виходу шифру (за рахунок зміни значень ключа та вектора ініціалізації). Атака використовує лише**  бітів виходу (один біт для кожного значення вектора ініціалізації), які сумуються по 10000 15-вимірних кубів, що частково перекриваються (відмітимо, що ). Виходячи з цього, крипто аналітик може отримати 10000 лінійних рівнянь від 10000 змінних, систему яких він може просто розв’язати за рахунок знаходження оберненої матриці коефіцієнтів цієї системи. Потоковий шифр в такому випадку може бути зламаним за менше, ніж за  бітових операцій, навіть незважаючи на той факт, що він не міг бути атакованим будь-якою із раніше розроблених технік. Тож, як бачимо, у деяких випадках кубічні атаки можуть проявити дуже вражаючу ефективність або ж бути застосованими до шифрів із надскладною будовою, які важко піддаються математичному аналізу.

## ****1.2 Математичне підґрунтя та основні поняття****

Нехай ­­– булева функція від n змінних.

– деяка підмножина множини змінних, .

Тоді функцію можна представити у вигляді:

 (1.1),

де ;  – деякий многочлен від змінних ; поліном  (назвемо його *поліномом-залишком*) такий, що будь-яка кон'юнкція, що входить у цей поліном, не містить хоча б однієї змінної з множини *I* [4].

Означення 1.1 Поліном  називається ***макстермом***(англ. *maxterm*), якщо степінь полінома  дорівнює 1 (тобто,  – лінійна булева функція, що тотожно не дорівнює константі). При цьому  називається ***суперполіномом***(англ. *superpoly*), а множина ***I*** називається ***кубом***.

*Властивість 1.1*: 

*Доведення*. При підсумовуванні значень функції *f* по всіх можливих значеннях змінних, що входять у макстерм, залишиться лише той доданок, в якому значення всіх змінних кубу дорівнюють 1, оскільки у решті випадків  (наявність принаймні одного нуля у добутку всіх змінних кубу робить добуток рівним нулю), при цьому поліномвходить як доданок у результуючу суму парну кількість разів (так як не містить хоча б однієї змінної, по якій проходить сумування), тому маємо:

,

Де m – деяке натуральне число, що залежить від n та k.

Приклад 1.1



У цьому випадку:

 ­– куб;

 – макстерм;

 – суперполіном;

 – поліном-залишок;





Як бачимо, у цьому прикладі ми змогли отримати із функції 3 степеня лінійний поліном від підмножини змінних функції. Отримання таких виразів і є основною метою кубічної атаки.

## ****1.3 Стадії атаки****

**Атака має кілька стадій обчислень [11]:**

1. **попередні обчислення (англ. *preprocessing*) –** пошук макстермів та суперполіномів**;**
2. **відновлення бітів ключа (англ. *key bit recovery*) ­–** отримання та розв’язання системи лінійних рівнянь від бітів секретного ключа**.**

**Розглянемо ці стадії докладніше.**

### ****1.3.1 Попередні обчислення****

**На стадії *попередніх обчислень* виконується пошук кубів, тобто макстермів та відповідних їм суперполіномів. При цьому передбачається, що криптоаналітик може змінювати як секретні ключі криптосистеми, так і вектори ініціалізації, що в ній використовуються.**

**Розглянемо процес пошуку кубів більш детально. Для знаходження кубу за властивістю 1 підраховується сума (у полі GF(2)) всіх значень досліджуваної булевої функції по всіх значеннях змінних, що входять у деякий можливий макстерм (при цьому множина змінних макстерма *I*** є підмножиною ***IV*). При цьому змінні, що не входять у макстерм, є фіксованими (наприклад, їх можна проініціалізувати нулями). Після цього перевіряється, чи є результат підсумовування лінійним поліномом від бітів секретного ключа, тобто суперполіномом. Якщо суперполіном знайдено, то коефіцієнти в ньому визначаються за допомогою встановлення відповідних значень бітів ключа рівними 1, а решту бітів ключа покладають рівною 0. Вільний член у суперполіномі можна знайти, якщо всі біти ключа дорівнюють 0.**

**На практиці зазвичай виконують пошук макстермів і суперполіномів наступного виду:** , де

,, *с* – константа, .

Варто зазначити, що кожному такту роботи потокового шифру відповідає своя булева функція виходу. Тому пошук кубів можна поширити на багато тактів роботи, щоб отримати достатню кількість пар макстерм-суперполіном для наступної стадії атаки.

**Стадія попередніх обчислень є ресурсоємною, але виконується лише один раз для кожної конкретної криптосистеми. Основними проблемами цього етапу є ефективний перебір можливих кубів та алгоритм перевірки лінійності.**

***Перебір макстермів* (кубів) для перевірки відповідних суперполіномів на лінійність є тривалою операцією і тому потребує ефективних підходів для її здійснення. Одним із таких підходів є обмеження степеня макстермів, що беруться до розгляду. Таке обмеження можна робити як знизу, так і зверху (наприклад, перебирати лише макстерми, степінь яких більший за 2, але менший за 5). Джерелом обмежень може виступати просто уявлення про можливий степінь функції, що описує вихід шифру (наприклад, у схемі нелінійної фільтрації за умови відкритості алгоритму степінь функції шифрування є наперед заданим і постійним, тобто, не залежить від такту роботи шифру), або, при більш строгому підході, знаходження степеня нелінійної функції за допомогою відповідних тестів, які, однак, є досить складними і важкими для реалізації.**

**Якщо степінь функції не вдалося визначити, то можливий наступний варіант перебору макстермів: криптоаналітик випадковим чином знаходить число *m,***  і підмножину *І* множини відкритих змінних, що містить ***m* елементів. Потім він знаходить суперполіном підсумовуванням виходів шифруючої функції по всіх значеннях змінних, що входять в *І.* Якщо потужність *І* занадто велика, то сумування призведе до отримання постійного значення (0 або 1 в залежності від вибору значень секретних змінних), і в цьому випадку криптоаналітику необхідно вилучити з множини *І* одну змінну й після цього повторити процес. Якщо ж потужність *І* є занадто малою, то з великою імовірністю кандидат у суперполіноми виявиться нелінійною функцією, тому в цьому випадку треба до множини *І* додати нову відкриту змінну й знову повторити процес. Правильний вибір множини *І*  є компромісом між цими двома випадками, і якщо вищевказані дії не призвели до отримання кубу, то процес необхідно повторити для нового випадкового варіанту множини *І.* Роботу цього алгоритму можна також пришвидшити, якщо запам’ятовувати певні проміжні результати (наприклад, можназапам’ятати результат сумування значень шифруючої функції по деякій підмножині *І* і використовувати його багаторазово при додаванні або вилученні змінних поза цією підмножиною).**

**Існує декілька алгоритмів *перевірки лінійності* функції, наприклад, *BLR*-тест (Blum-Luby-Rubinfeld) [12]. Це простий імовірнісний тест, що для заданого поліному *f*(x) перевіряє рівність *f*(*x*)+*f*(*y*)+*f*(0)=*f*(*x*+*y*) для випадкових вхідних аргументів *х* та *у*. Якщо поліном *f*(x) є лінійним, то тест гарантовано видає позитивний результат; якщо ж *f*(x) є сильно нелінійним, то тест «провалюється» з великою імовірністю. Цей тест повторюється для кожного *f*(x) до тих пір, поки криптоаналітик не впевниться в тому, що *f*(x) дійсно дуже близький до лінійного поліному (наприклад, *f*(x) є лінійним, окрім кількох термів високого степеня, що майже завжди дорівнюють нулю).**

**Більш зручний та швидкий алгоритм був запропонований у [13] (“**Term-by-Term Linearity Testing”**). Він є імовірнісним, але простим у реалізації та швидким: кількість обчислень алгоритму становить** , де *d* – кількість лінійних змінних, а *С* – константа, маніпулюючи значенням якої можна збільшувати або зменшувати імовірність правильного результату тесту. Цей алгоритм наведений нижче:

S – множина лінійних змінних, що мають пройти перевірку

S ← порожня множина

for i = 1 to n do

, де тільки *і*-тий біт дорівнює 1

if f(x) + f(0) = 1 then

Внести i до множини S

end if

end for

for each j у множині S do

for c = 1 to C do

випадковим чином вибрати значення y



and 

if f(y) = f(z) then

Відхилити функцію як нелінійну та завершити алгоритм

end if

end for

end for

Нехай у результаті стадії попередніх обчислень маємо **N пар макстерм-суперполіном. Тоді можна скласти матрицю** **розмірності** , рядками якої будуть виступати коефіцієнти знайдених суперполіномів [4]. Якщо матриця не є виродженою, то можна знайти  з метою пониження складності алгебраїчних обчислень на наступній стадії з  до (складність перемноження двох матриць).

Так як  є матрицею, в якій кожен елемент обирається з імовірністю 1/2 , то дуже легко підрахувати ймовірність того, що  буде не виродженою:

*Лема 1.1* Імовірність того, що матриця розмірності  з випадкових елементів, що належать полю GF(2), має обернену, становить .

*Доведення*. Доведення леми базується на простій індукції по рядках матриці.

Ця імовірність є постійною, але може бути зроблена близькою до 1 під час стадії попередніх обчислень за допомогою отримання більшої кількості макстермів.

### ****1.3.2 Відновлення бітів ключа****

**Стадія відновлення бітів ключа – це етап кубічної атаки, на якому розв’язується система лінійних рівнянь, отриманих на основі знайдених на попередній стадії кубів та заданого фіксованого секретного ключа (при цьому криптоаналітик може довільно змінювати вектор ініціалізації). Кожне рівняння системи отримується як сума виходів функції по всім значенням змінних, що входять у відповідний знайдений макстерм. Оскільки ключ фіксований і заданий, то таким чином ми дізнаємось, чому дорівнює значення** відповідного суперполінома. Система має наступний вигляд:

, (1.2)

**де N – кількість рівнянь,**  – *і*-тий суперполіном, а  – сума значень функції шифрування по всім значенням змінних, що входять в *і*-тий макстерм, .

**Система лінійних рівнянь з двійковими коефіцієнтами може бути розв’язана, наприклад, методом Гауса (якщо на попередній стадії не проводився пошук оберненої матриці системи, оскільки в цьому разі достатньо просто перемножити цю матрицю на стовбець** **). При цьому необхідно проводити відбір лінійно незалежних рівнянь (тому потенційно кількість кубів, що отримуються під час попередніх обчислень, має бути набагато більшою, аніж потужність K).**

## ****1.4 Підходи до підсилення атаки****

В залежності від особливостей кожної потокової криптосистеми можна виробити багато підходів до підсилення та узагальнення кубічної атаки, крім того, для деяких шифрів знаходження макстермів невеликого степеня виявляється дуже складною чи навіть неможливою задачею, тому модифікації базового варіанту кубічної атаки в таких випадках є конче необхідними. Розглянемо кілька з найбільш загальних таких підходів та зауважень.

1. У деяких потокових криптосистемах з великою кількістю раундів ініціалізації дуже складно знайти необхідні для атаки макстерми низького степеня. В таких випадках, якщо внутрішня структура шифру є відомою, можна використати наступний підхід: криптоаналітик представляє змінні стану регістра зсуву в точності як деякі поліноми від бітів ключа та вектора ініціалізації на деякому проміжному раунді ініціалізації шифру. Маючи таке точне представлення, криптоаналітик проводить лінеаризацію змінних ключа, замінюючи їх на нову множину секретних змінних, тим самим зменшуючи степені поліномів, що описують змінні стану ЛРЗ. Значення нових секретних змінних можуть бути знайдені з використанням базових технік кубічної атаки. Після того, як ці значення стали відомими, можна знайти і значення початкових секретних змінних виходячи з рівнянь, отриманих при лінеаризації. Якщо стан шифру є оборотним, або ж близьким до цього, то криптоаналітик може запустити шифр у зворотному порядку для знаходження значень бітів ключа замість того, щоб розв’язувати систему рівнянь. Відмітимо, що подібний прийом може бути застосований і до блочних шифрів, застосувавши який, можна знайти точне представлення поліномів від секретних та відкритих змінних на певному проміжному раунді шифрування.
2. Криптоаналітик може використати будь-який нелінійний суперполіном, який він може знайти, представивши його у компактному вигляді за допомогою вгадування деяких значень секретних змінних в ньому і спрощуючи результат. У частковому випадку, вгадування значення N-1 секретних змінних завжди призводить до отримання із суперполіному легкого для розв’язання лінійного рівняння від змінної, що залишилася, і, як наслідок, одержимо атаку, що ефективніша за повний перебір, якщо припустити, що визначення суперполіному не є дуже тривалою операцією.
3. Криптоаналітик може спробувати розв’язати систему рівнянь, отриманих при застосуванні атаки, навіть якщо вони не є лінійними, але за умови, що вони є невеликого степеня. Якщо кількість відкритих змінних є достатньо великою, то можна виконати сумування по багатьох можливих підмножинах потужності *d-*1 (*d* – степінь рівняння) множини відкритих змінних і отримати велику систему з нелінійних рівнянь від бітів ключа, яку можна розв’язати лінеаризацією або будь-якими іншими прийомами.
4. Окрім знаходження лінійних суперполіномів, криптоаналітик може достатньо просто отримувати і квадратичні. Для цього можна використати узагальнений BLR-тест: для заданого поліному *f*(*x*) випадковим чином вибираються вхідні аргументи  і перевіряється рівність . Як і в звичайному BLR-тесті лінійності, не квадратичні функції з високою імовірністю будуть виявлені вже після кількох застосувань тесту. Подібний тест може бути в подальшому узагальненим для кубічних функцій і для поліномів вищих степенів із експоненційно зростаючою (в залежності від степеня поліному) кількістю обчислень значень функції *f*(*x*). Обчислення коефіцієнтів для поліномів вищих степенів може також бути узагальнене.
5. Отримані в результаті атаки поліноми від секретних змінних на різних тактах роботи шифру можна використовувати як окремо (тобто, в якості різних рівнянь від бітів ключа), так і в будь-якій комбінації, що може призвести до отримання нових рівнянь. Це є особливо корисним у разі нелінійності цих поліномів, так як певні їх комбінації можуть призвести до отримання не тільки просто нових рівнянь, а ще й, можливо, лінійних виразів, що може значно спростити розв’язування результуючої системи рівнянь.
6. Для більшості потокових шифрів, в яких |*K*| = |*IV*| і біти ключа й вектора ініціалізації сумуються в полі GF(2) під час ініціалізації шифру, максимальний степінь функції, яка описує вихід шифру, становить |*K*|, а не |*K*| + |*IV*|, як у загальному випадку.
7. У випадку, коли криптосистема містить недостатню кількість відкритих змінних (або не містить їх взагалі), можна замінити деякі секретні змінні на вирази вигляду  (де  – *і*-тий біт ключа, а  – штучно введена відкрита змінна), тим самим отримуючи можливість застосувати кубічну атаку і виявити лінійні суперполіноми, які раніше не існували.
   1. Підсумовуючи, слід відмітити, що окрім вищевказаних прийомів, можна розробити ще багато інших узагальнень чи посилень атаки, в тому числі, специфічних для кожного конкретного шифру. Тому в разі застосування кубічної атаки на певний обраний криптоаналітиком шифр корисним є попередньо провести аналіз можливості спростити або доповнити базовий алгоритм атаки.

## Висновки до розділу 1

Кубічні атаки є ефективним та перспективним методом криптоаналізу симетричних шифрів (зокрема, потокових). У даному розділі розглядалися базові поняття і теоретичне підґрунтя кубічної атаки, повний опис різних її стадій. На стадії попередніх обчислені здійснюється пошук кубів, тобто, макстермів і відповідних їм суперполіномів. Кожен суперполіном є елементом системи рівнянь, яка отримується і розв’язується на стадії відновлення бітів ключа.

Окрім базового алгоритму кубічної атаки, існує кілька її варіацій та підходів до підсилення, наприклад, знаходження нелінійних суперполіномів, які за рахунок вгадування бітів ключа можна звести до лінійних, або ж включати в систему рівнянь рівняння квадратичного чи кубічного вигляду, тим самим збільшуючи складність стадії відновлення бітів ключа, але отримуючи можливість набрати більшу кількість рівнянь. Варто зазначити, що у цій роботі розглядатиметься лише базовий алгоритм атаки, де суперполіноми є лінійними, а відкриті змінні, що не входять у макстерм, встановлюються рівними нулеві.

# 2 Часова складність кубічної атаки

При аналізі кубічної атаки важливим є питання оцінки часової складності її реалізації. Так як атака має дві стадії, що виконуються послідовно, то при оцінці загальної часової складності атаки складності її стадій підсумовуються. Але, оскільки складність стадії відновлення бітів ключа набагато нижча, ніж складність попередніх обчислень, то нею можна знехтувати при оцінці загальної складності. Отже, більш суттєвим є розгляд складності першої стадії кубічної атаки. Як вже було сказано раніше, основними факторами, які впливають на швидкість попередніх обчислень, є ефективний перебір макстермів та алгоритм перевірки лінійності відповідних кандидатів у суперполіноми.

**Для пришвидшення пошуку макстермів зазвичай доцільніше проводити його на кількох тактах роботи шифру, що дозволяє знайти достатню кількість кубів навіть за умови розгляду невеликих степенів макстермів. Більш наглядно часову складність кубічної атаки можна представити на прикладі простого шифру.**

## ****2.1 Часова складність на прикладі простого потокового шифру****

**Розглянемо примітивний** потоковий шифр на основі схеми нелінійної фільтрації, побудований на лінійному регістрі зсуву довжиною 32 біти. Параметри шифру за замовченням:

1. *К* – 16 біт;
2. *IV* – 16 біт;
3. примітивний поліном регістра зсуву (за замовченням):

* у шістнадцятирічному записі - 0xF5000000;
* алгебраїчний вигляд – x32+x7+x5+x3+x2+x+1;

1. фільтруюча нелінійна функція (за замовченням):
2. ;
3. робота шифру без ініціалізації:

початкове заповнення регістру (двійковий вектор стану регістру має вигляд (s0,s1,…,s30,s31)) встановлюється рівним наступному вектора: (k0,k1,…,k15,v0,v1,…,v15), де ki – і-та компонента вектора ключа, а vj – j-та компонента вектора ініціалізації.Далі кожен біт виходу шифру – це значення фільтруючої функції на поточному стані регістру (після кожного обчислення значення фільтруючої функції регістр переходить у наступний стан за звичайною схемою регістра з лінійним зворотним зв’язком);

1. ініціалізація:
   1. (s0,s1,…,s30,s31) ← (0,0,…,0, k0,k1,…,k15);
   2. далі впродовж 16 ітерацій до зворотного зв'язку регістру додається по одному біту IV починаючи з нульового. Після цього регістр ще 32 рази працює у звичайному режимі (при чому, під час ініціалізації на вихід шифру нічого не подається). На цьому ініціалізація завершується і шифр готовий генерувати вихідну послідовність (за схемою, описаною в попередньому пункті).

**Графіки залежності часової складності атаки від максимального степеня макстерма (при фіксованій кількості тактів роботи шифру перебираються всі макстерми, степінь яких не перевищує задану максимальну) та від кількості тактів роботи шифру (при фіксованому максимальному степені макстерма) наведені на рисунку 2.1 та рисунку 2.2. Результати отримані на основі програмної реалізації шифру, що розглядається, та алгоритму кубічної атаки, наведеному в додатку А.**

**Рисунок 2.1 – Графік залежності часової складності пошуку кубів від максимального степеня макстерма**

**Рисунок 2.2 – Графік залежності часової складності пошуку кубів від кількості тактів роботи шифру**

**Такий вигляд графіків можна пояснити за допомогою простих міркувань щодо складності перебору макстермів та тактів роботи шифру. На одному такті складність попередніх обчислень атаки прямо пропорційно залежить переважно від складності перебору макстермів певного степеня, що не більше попередньо заданого значення. Оскільки пошук макстермів на кожному такті роботи шифру виконується окремо і незалежно від інших тактів (так як кожному такту відповідає свій поліном, що описує вихід шифру), то часова складність стадії попередніх обчислень атаки прямо пропорційно залежить і від кількості тактів. Подальші міркування з цього приводу наведені у розділі 2.2.**

## ****2.2 Формула для часової складності кубічної атаки****

**Позначимо складність перебору макстермів (що залежить від максимального степеня макстерма** **) як** , а кількість тактів як . Формула для має враховувати складність підсумовування по всіх значеннях змінних, що входять у макстерм (що багаторазово виконується для перевірки лінійності відповідного суперполіному). Ця складність становить  для степеня макстерма *і* (наприклад, для макстерма першого степеня необхідна лише одна операція додавання, для другого – 3 і т.д.). З урахуванням цього факту отримуємо наступний вираз:

 (2.1)

Отже, загальна формула залежності часової складності стадії попередніх обчислень кубічної атаки  від та :

 (2.2)

**Щоб оцінити, як співвідносяться експериментальні дані з отриманим виразом для складності, зведемо числові дані у таблиці (таблиця 2.1 та таблиця 2.2). Експериментальні значення в таблиці 2.1 отримані при фіксованій кількості тактів роботи шифру (**), а в таблиці 2.2 – при фіксованому максимальному степені макстерма (). **Значення констант *с* знайдені на основі експериментальних даних при** ,  (при с=1 теоретичне значення і для узгодження з експериментальним значенням покладаємо с=**454928/27031190≈0,01683**) і при ,  (при с=1 теоретичне значення , тому покладаємо с=**4922/1263840≈0,003894**) відповідно.

**Таблиця 2.1 – Залежність часової складності кубічної атаки від максимального степеня макстерма**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | (, ), мс | **Експериментальне значення**  **(рис 2.1)** |
| 1 | 2 | 3 | **4** |
| **1** | **16** | **1** | **20** |
| **2** | **376** | **32** | **76** |
| **3** | **4296** | **362** | **557** |
| **4** | **31596** | **2659** | **2960** |
| **5** | **167004** | **14053** | **14523** |
| **6** | **671508** | **56507** | **57043** |
| **7** | **2124388** | **178767** | **179611** |
| **8** | **5406238** | **454935** | **454928** |

**Таблиця 2.2 – Залежність часової складності кубічної атаки від кількості тактів роботи шифру**

|  |  |  |
| --- | --- | --- |
|  | (,), мс | **Експериментальне значення (рис 2.2)** |
| 1 | 2 | **3** |
| **1** | **615** | **585** |
| **2** | **1230** | **1170** |

**Продовження таблиці 2.2.**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **3** | **1846** | **1747** |
| **4** | **2461** | **2340** |
| **5** | **3076** | **2960** |
| **6** | **3691** | **3604** |
| **7** | **4306** | **4258** |
| **8** | **4921** | **4922** |

**Як бачимо із наведених таблиць, теоретичні наближені оцінки часової складності реалізації кубічної атаки відповідають експериментальним результатам, але значення константи *с* залежить від значення відповідного фіксованого параметра кубічної атаки (кількості тактів роботи шифру або максимального степеня макстерма), від особливостей і потужності обчислювальної системи (комп’ютера), на якій проводиться атака, а крім того, що є найбільш суттєвим, від вибраного алгоритму перевірки лінійності, тому на вибір цього алгоритму необхідно звернути особливу увагу.**

**Якщо криптоаналітику доступна багатопроцесорна обчислювальна система, то процес пошуку кубів можна досить ефективно розпаралелити за рахунок проведення всіх необхідних обчислень для кожного такту роботи шифру на окремому процесорі (або в окремому потоці). Це можливо через незалежність пошуку кубів на кожному з тактів роботи шифру (у загальному випадку). Таким чином, маючи** процесорів, час пошуку кубів можна зменшити у разів.

## Висновки до розділу 2

Найбільш ресурсоємним етапом кубічної атаки є пошук кубів. Складність даної стадії атаки базується на переборі (послідовному або випадковому) можливих макстермів заданого степеня на різних тактах роботи шифру та тестуванні лінійності поліному, отриманому в результаті сумування вихідної функції виходу шифру по всім змінним, що входять у вибраний макстерм. При цьому, із зростанням кількості тактів роботи шифру складність зростає лінійно, в той час як при збільшенні степеня макстерма – експоненційно. Тому більш раціональним є підхід пошуку кубів малого степеня для різних тактів роботи шифру, даний спосіб можна також пришвидшити за рахунок розпаралелювання алгоритму кубічної атаки, що можливе завдяки незалежності пошуку кубів на різних тактах роботи шифру.

# ****3 Дослідження кількості булевих функцій з макстермами малого степеня****

## 3.1 Кількість сприятливих функцій для кубічної атаки з макстермами першого степеня

Позначимо  кількість «гарних» функцій  степеня , тобто функцій, у яких за допомогою макстермів степенів, що не перевищують , можна знайти щонайменше  лінійно незалежних суперполіномів, враховуючи, що всі відкриті змінні, що не входять до кубу, покладаються нулями. У даній роботі виводяться формули лише для  при будь-яких , тобто , та будь-якому співвідношенні  та . У результаті знаходиться ймовірність успішної кубічної атаки з використанням макстермів тільки першого степеня  - ймовірність набрати таким чином систему з  лінійно незалежних рівнянь відносно секретних змінних.

Зрозуміло, що при  . Отже, в подальшому вважаємо, що . У загальному випадку, тобто, для більших степенів макстерма *d*, вимагається виконання умови .

Будемо виводити формули для , поступово збільшуючи степінь функції  від 1 до .

Очевидно, що  при будь-яких , .

Розглянемо . Функція 2-го степеня від змінних  містить хоча б один терм 2-го степеня виду ,  або , , або ,  та лінійну частину (можливо нульову), яку загалом позначимо . Терми, які не впливають на наявність або відсутність макстермів і, відповідно, суперполіномів, будемо називати *нейтральними*, терми, які утворюють макстерми 1-го степеня, будемо називати *сприятливими* і терми, які унеможливлюють існування відповідних макстермів – *несприятливими*. Так, наприклад, всі терми 0-го та 1-го степенів – нейтральні, від їх наявності або відсутності кількість суперполіномів не залежить. Таким чином, уся лінійна частина є нейтральною. Нейтральними є також терми виду та , а терми виду  - сприятливі.

Кількість можливих варіантів для  становить  - кількість лінійних функцій від  змінних.

Кількість можливих лінійних комбінацій з нейтральних термів 2-го степеня дорівнює (таких термів може й взагалі не бути, бо у «гарній» функції повинен бути хоча б один сприятливий терм, що гарантує 2-й степінь функції).

Позначимо кількість всіх можливих лінійних комбінацій нейтральних термів у функції 2-го степеня від  змінних через . Тоді 

Порахуємо кількість можливих лінійних комбінацій сприятливих термів, при яких гарантується наявність  лінійно незалежних суперполіномів. Для початку, розглянемо випадок, коли , тобто, маємо лише одну секретну змінну. Тоді кількість можливих варіантів побудови пар макстерм-суперполіном становить . Це є число усіх можливих лінійних комбінацій відкритих змінних, окрім нульової.

При  для першої секретної змінної число можливих комбінацій відкритих дорівнює , а для другої – вже , оскільки використану для першої змінної комбінацію відкритих змінних ми відкидаємо (одна і та сама комбінація не додає нових макстермів, а отже, і нових рівнянь у систему). При  для третьої секретної змінної кількість комбінацій відкритих становить , цього разу не враховуються лінійні комбінацій термів, використаних для попередніх секретних змінних. Аналогічно, при збільшенні величини  для кожної наступної змінної кількість комбінацій відкритих змінних становить , де *і* – індекс, що відповідає секретній змінній., . Отже, кількість варіантів побудови системи незалежних рівнянь дорівнює . Позначимо даний вираз як  (від англ. kernel - ядро).

Спробуємо представити  в іншому вигляді. Для цього розпишемо  наступним чином:



У даному представлені легко бачити рекурентну залежність, а саме:

. (3.1)

Повне визначення  має наступний вигляд:

 (3.2)

Підсумовуючи вищесказане, приходимо до висновку, що

. (3.3)

Формула для  є базовою. Терми степенів вище за 2, не можуть бути спрятливими при , вони можуть бути лише нейтральними або несприятливими, і нам залишається врахувати їх вплив для функцій степенів .

Розглянемо . У функції 3-го степеня додатково до термів 0, 1, 2-го степенів з’являється хоча б один доданок виду:

1) , 2) , 3) , 4) .

Терми виду 1), 2), 4) – нейтральні, виду 3) – несприятливі. Кількість можливих нейтральних термів дорівнює , звідки кількість можливих варіантів для нейтральних термів 3-го степеня дорівнює

. (3.4)

Якщо у функції 3-го степеня немає несприятливих доданків, то кількість таких «гарних» функцій дорівнює

.

Якщо у функції 3-го степеня є несприятливі доданки виду 3) тільки з одним макстермом, наприклад, виду , то кількість таких функцій дорівнює

.

Множник  виникає з тої причини, що в цьому випадку всі доданки виду  та  стають нейтральними, і доданок  також нейтральний. При підрахунку  на них можна не зважати; решта термів виду ,...,  залишаються сприятливими. Враховуючи, що «гарна» функція може мати терми виду 3), де індекс  пробігає будь-яку підмножину множини індексів {1,…,*t*}, в тому числі і пусту (але крім усієї множини, бо в цьому випадку функція не буде «гарною»), отримуємо формулу для :



. (3.5)

Таким чином, загальна кількість «гарних» функцій степеня, не більшого за 3, дорівнює:



Аналогічно можна показати, що кількість «гарних» функцій степеня  дорівнює:



 (3.6)

Тут , де

, (3.7)

так як всі терми -го степеня, за виключенням термів виду , є нейтральними.

При 

, (3.8)

так як всі терми -го степеня в цьому випадку є нейтральними і хоча б один зних має бути присутнім. , але при 

 (3.9)

а при 

. (3.10)

Отже, ймовірність успішної атаки з використанням макстермів тільки першого степеня становить

, (3.11)

де величини  визначаються за формулами (3.2)-(3.10).

## 3.2 Програмна перевірка теоретичних співвідношень

Задля перевірки отриманих теоретичних співвідношень була розроблена програмна реалізація алгоритму пошуку сприятливих функцій заданого степеня і заданої кількості секретних та відкритих змінних. Вона включає в себе реалізацію генератора потенційно сприятливих булевих функцій (код надається у додатку Б), базових алгоритмів кубічної атаки (додаток А) та програми для підрахунку кількості шуканих функцій (додаток В). Схематично дана реалізація представлена на рисунку 3.1.

*f*(*x*)

Ранг

системи = s?

Система рівнянь

Алгоритми кубічної атаки

Генератор потенційно сприятливих булевих функцій

*r*, *t*, *s*

Додати *f*(*x*) до списку сприятливих функцій

Відкинути *f*(*x*) як несприятливу функцію

Рисунок 3.1 – Схема програмної реалізації алгоритму перебору і підрахунку кількості сприятливих функцій

У загальному випадку, необхідно здійснювати перебір усіх можливих булевих функцій заданого степеня, що є вкрай ресурсоємним. За прийнятний час у такому варіанті розрахунку можна здійснити лише перебір функцій із сумарною кількістю змінних до 5. Саме складністю перебору функцій обґрунтоване використання генератора функцій спеціального вигляду, зокрема функцій, що не містять (або містять мінімальну кількість) нейтральних термів. Таким чином уникається генерування, наприклад, лінійних і константних функцій, що вже пришвидшує перебір у  рази.

Проте, з усіма оптимізаціями та відкиданням нейтральних термів, зважаючи на стрімке зростання кількості функцій, програмні розрахунки вдалося провести для досить обмеженої величини суми змінних функції. Зокрема, при  – для суми змінних , а при  – для .

Отримані програмою результати відповідають виведеним теоретичним співвідношенням і наведені у таблицях 3.1-3.2. Для всіх випадків функцій степінь макстерма незмінна і дорівнює одиниці (*d* = 1).

Таблиця 3.1 – Кількість сприятливих функцій для випадку .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| 1 / 1 | 1 | 3 | 4 / 4 | 20160 | 21 |
| 2 / 1 | 3 | 5 | 8 / 1 | 255 | 38 |
| 2 / 2 | 6 | 7 | 7 / 2 | 16002 | 32 |
| 3 / 1 | 7 | 8 | 6 / 3 | 234360 | 28 |
| 3 / 2 | 42 | 10 | 5 / 4 | 624960 | 26 |
| 4 / 1 | 15 | 12 | 9 / 1 | 511 | 47 |
| 5 / 1 | 31 | 17 | 8 / 2 | 64770 | 40 |
| 4 / 2 | 210 | 14 | 7 / 3 | 1984248 | 35 |
| 3 / 3 | 168 | 13 | 6 / 4 | 13124160 | 32 |
| 6 / 1 | 63 | 23 | 5 / 5 | 9999360 | 31 |
| 5 / 2 | 930 | 19 | 10 / 1 | 1023 | 57 |
| 4 / 3 | 2520 | 17 | 9 / 2 | 260610 | 49 |
| 7 / 1 | 127 | 30 | 8 / 3 | 16322040 | 43 |
| 6 / 2 | 3906 | 25 | 7 / 4 | 238109760 | 39 |
| 5 / 3 | 26040 | 22 | 6 / 5 | 629959680 | 37 |

Таблиця 3.2 – Кількість сприятливих функцій для випадку .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 1 | 2 | 3 | 4 | 5 |
| 1 / 1 | 0 | 0 | 0 | 0 |
| 2 / 1 | 96 | 0 | 0 | 0 |
| 2 / 2 | 2304 | 3072 | 0 | 0 |
| 3 / 1 | 26880 | 28672 | 0 | 0 |
| 3 / 2 | 14899200 | 463208448 | 478150656 | 478150656 |
| 4 / 1 | 62853120 | 1950351360 | 2013265920 | 2013265920 |
| 5 / 1 | 4260603494400 | 13960732784092700 | 879553007225457000 | 8795530072254570000 |

Продовження таблиці 3.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 4 / 2 | 2183987429376 | 71562828837814200 | 4508595808207110000 | 4508595808207110000 |
| 3 / 3 | 2817196032 | 11542053519360 | 727326941773824 | 727326941773824 |

За даними таблиць 3.1 і 3.2 можна порахувати сумарну імовірність успіху атаки за формулою (3.11). Отримані значення ймовірностей представлені у таблиці 3.3.

Таблиця 3.3 – Імовірність успіху атаки з використанням макстермів лише першого степеня.

|  |  |  |
| --- | --- | --- |
|  | Загальна кількість функцій: |  |
| 1 / 1 | 16 | 0,5 |
| 2 / 1 | 256 | 0,75 |
| 2 / 2 | 65536 | 0,09375 |
| 3 / 1 | 65536 | 0,875 |
| 3 / 2 | 4294967296 | 0,22265625 |
| 4 / 1 | 4294967296 | 0,9375 |
| 5 / 1 | 1,84467E+19 | 0,961181641 |
| 4 / 2 | 1,84467E+19 | 0,492702484 |
| 3 / 3 | 1,84467E+19 | 7,94828E-05 |

Як бачимо, чим більша різниця між *t* та *s*, тим більшою є імовірність успіху. Але у реальних криптоалгоритмах зазвичай виконується умова , тому більш цікавим є випадок (мінімально допустиме значення *t*, за якого атака ще може бути успішною). А із зростанням суми кількостей секретних і відкритих змінних за такої умови імовірність має тенденцію до стрімкого спаду. Можна стверджувати, що при для достатньо великих значень (наприклад, ), імовірність успіху атаки при використанні лише макстермів першого степеня є близькою до 0. Отже, для забезпечення більш прийнятого рівня імовірності успіху, необхідний розгляд макстермів вищих степенів.

Маючи універсальну програмну реалізацію алгоритму підрахунку кількості сприятливих функцій для довільного степеня макстерма, вдалося отримати результати для випадку *d* = 2, . Відповідні дані представлені у таблиці 3.4.

Таблиця 3.4 – Кількість сприятливих функцій для випадку .

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 2 / 1 | 128 | 0 | 0 |
| 2 / 2 | 0 | 0 | 0 |
| 3 / 1 | 28672 | 28672 | 0 |
| 3 / 2 | 44040192 | 434110464 | 478150656 |
| 4 / 1 | 66060288 | 2047868928 | 2113929216 |

Як бачимо, порівнюючи із значеннями з таблиці 3.2, у даному випадку кількість сприятливих функцій дещо зросла (але при  кількість сприятливих функцій тут дорівнює нулеві, оскільки не виконується умова ). Якщо ж брати до розгляду функції, в яких є макстерми як першого, так і другого степеня, то кількість сприятливих функцій ще суттєво зросте. Оцінка кількості таких функцій є перспективною задачею, а отримані для випадку *d* = 1 математичні співвідношення можуть бути основою для подальших досліджень.

## Висновки до розділу 3

У результаті аналізу залежності кількості сприятливих функцій, що містять макстерми лише першого степеня, від степеня функції та кількості відкритих і секретних змінних, отримано математичний вираз для згаданої величини, а розраховані за ним значення перевірено методом порівняння із результатами програмного розрахунку. Функція визнається сприятливою, якщо ранг отриманої системи лінійних рівнянь дорівнює кількості секретних змінних. Ідея програмної реалізації полягає у переборі потенційно сприятливих функцій та застосуванні до них алгоритмів кубічної атаки. Оскільки перебір всіх функцій є надто ресурсоємною задачею, у програмі було використано генератор потенційно сприятливих функцій, що дозволив значно прискорити та оптимізувати розрахунки.

Наведені таблиці числових значень кількостей сприятливих функцій для різних параметрів: степеня функції, кількості секретних та відкритих змінних. Усі табличні значення відповідають результатам розрахунків за отриманими математичними співвідношеннями.

Також були розраховані імовірності успіху кубічної атаки з макстермами тільки першого степеня для різних комбінацій кількостей секретних і відкритих змінних (*t* і *s* відповідно). Було виявлено, що при зростанні різниці між *t* і *s*, імовірність успіху прямує до 1. Але на практиці зазвичай виконується нерівність . В найкращому випадку, коли  (інакше, тобто, коли , кількість сприятливих функцій дорівнює 0), спостерігається стрімкий спад імовірності успіху атаки, і для досить великих значень кількості змінних ця імовірність є близькою до 0. Отже, за таких умов макстермів лише першого степеня недостатньо.

Програмно були отримані результати також для випадку *d* = 2 (другий степінь макстерма). Порівняно із випадком *d* = 1, для аналогічних значень степеня функції і кількості змінних, кількість сприятливих функцій із макстермами другого степеня є дещо більшою (при ). Якщо ж комбінувати макстерми різних степенів, то можна отримати ще кращі результати. Таким чином, перспективною є задача аналізу кубічних атак із більшими степенями макстермів.

# 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

У зв’язку з тим, що виконання даної роботи вимагає програмної реалізації алгоритмів кубічної атаки, а для розрахунків потрібні обчислювальні можливості комп’ютерної техніки, доцільно розглянути особливості роботи за ЕОМ з точки зору охорони праці користувачів комп’ютерів.

## 4.1 Аналіз та відповідність нормам робочого простору для користувачів комп’ютерів

### 4.1.1 Робоче приміщення

Основні вимоги до виробничого приміщення для експлуатації ВДТ: площа заскління повинна складати 25-50% від загальної площі; приміщення повинні мати природне і штучне освітлення; не допускається розташування робочих місць ПЕОМ в підвальних приміщеннях і цокольних поверхах; робочі місця з ВДТ і ПЕОМ під час виконання творчої роботи, яка потребує значної розумової напруги чи великої концентрації уваги, слід ізолювати одне від одного перегородкою висотою 1,5-2,0 м; неприпустимим є розташування приміщень категорій А і Б, а також виробництв з мокрими технологічними процесами поряд з приміщеннями, де розташовуються ЕОМ, виконується їх обслуговування, налагодження і ремонт, а також над такими приміщеннями або під ними; виробничі приміщення, в яких розташовані ЕОМ, не повинні межувати з приміщеннями, де рівні шуму та вібрації перевищують; площа, на якій розташовується одне робоче місце з ПЕОМ або ВДТ, повинна становити не менше як 6,0 м2, об’єм приміщення – не менше як 20 м3; поверхня підлоги має бути рівною, без вибоїн, неслизькою, зручною для очищення та вологого прибирання, мати антистатичні властивості; при розміщенні робочих місць необхідно виключити можливість прямого засвічування екрана джерелом природного освітлення; для оздоблення приміщень з ВДТ повинні використовуватися дифузно-відбиваючі матеріали з коефіцієнтами відбиття: стелі – 0,7-0,8; стін – 0,4-0,5; підлоги – 0,2-0,3; забороняється застосовувати для оздоблення інтер’єру полімерні матеріали, що виділяють у повітря шкідливі хімічні речовини.

Висота робочої поверхні столу має бути 680-800 мм, ширина 600-1400 мм, глибина 800-1000 мм. Стіл повинен мати простір для ніг висотою від 600 мм, шириною від 500 мм, глибиною на рівні колін від 450 мм, на рівні витягнутої ноги – від 650 мм. Стілець повинен мати сидіння, спинку, підлокітники. Ширина та глибина сидіння — не менше 400 мм. Висота спинки сидіння має становити 300±20 мм,ширина - не менше 380 мм, радіус кривизни в горизонтальній площині - 400 мм. Необхідно застосовувати підлокітники довжиною не менше 250 мм, шириною - 50-70 мм, що регулюються по висоті над сидінням у межах 230±30 мм та по відстані між підлокітниками в межах 350-500 мм. Відстань від екрана до ока працівника повинна складати 700 - 800 мм при розмірі екрану по діагоналі 43 см (17", екран ноутбука). Очі працівника знаходяться на рівні верхнього краю монітору, нахил становить 20–30О.Клавіатуру слід розміщувати на відстані 100 - 300 мм від краю, ближчого до працівника. Кут нахилу клавіатури має бути в межах 5 - 15 град.

Приміщення знаходиться на другому поверсі. Розмір приміщення становить 5,9х3,8 м, висота – 2,8 м (рисунок 4.1). Відповідно, площа становить 22,42 м2, а об’єм приміщення – 62,78 м3.

В даному приміщені працюють три особи. Таким чином, на кожного припадає площа 7,47 м2 та об’єм – 20,92 м3, що відповідає нормам.

Поверхня підлоги без нерівностей і вибоїн, не слизька.

При розміщенні робочих місць виключена можливість прямого засвічування екрана джерелом природного освітлення.

Кожне робоче місце обладнане столом, кріслом та відповідним обладнанням, що відповідають нормам.

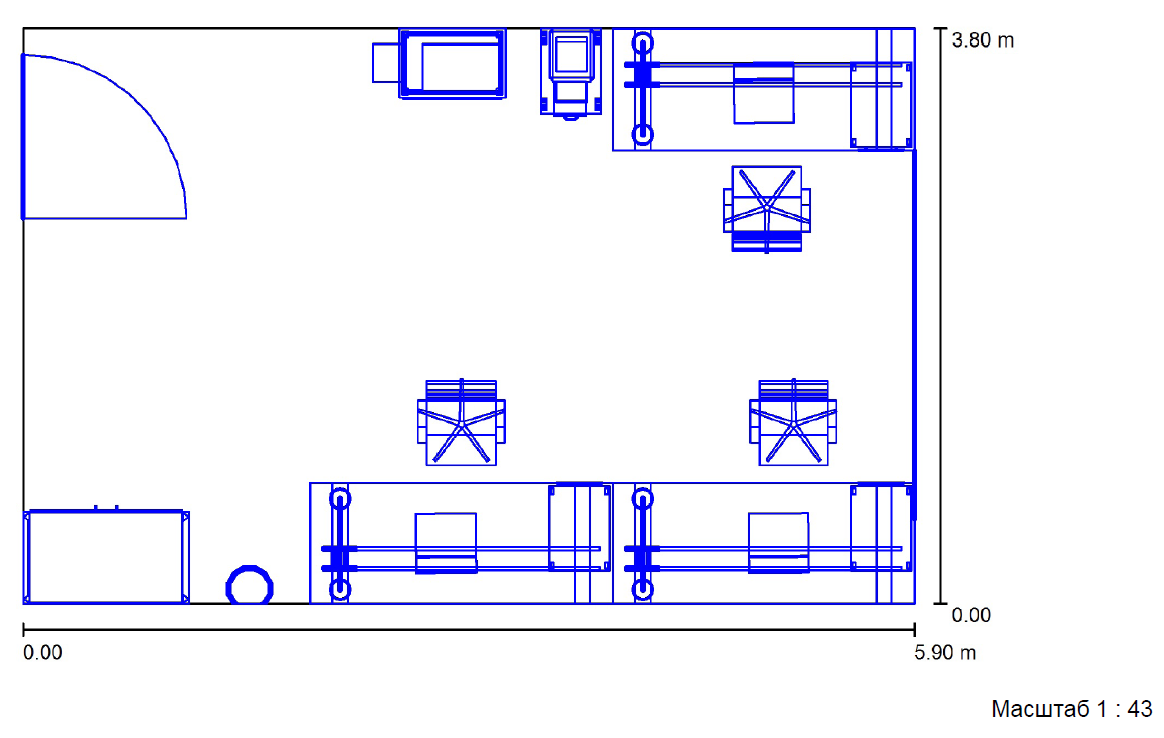
****

Рисунок 4.1 – Загальний план приміщення та розташування обладнання.

### 4.1.2 Повітря робочої зони

Згідно з ДСН 3.3.6.042-99 для категорії тяжкості робіт легка 1б (переважно сидяча) передбачено оптимальні параметри мікроклімату що представлені в табл. 1.

Таблиця 4.1 – Оптимальні параметри мікроклімату для категорії тяжкості робіт - легка 1б.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Температура  повітря | Відносна вологість повітря | Швидкість вітру |
| Холодний період року | 21-23˚С | 40-60% | не більше 0,1 м/с |
| Теплий період року | 22-24˚С | 40-60% | не більше 0,2 м/с |

Регламентовані рівні іонізації повітря приміщень при роботі за ВДТ та ПК наведені в таблиці 4.2.

Таблиця 4.2 – Рівні іонізації повітря приміщень при роботі за ВДТ та ПК

|  |  |  |
| --- | --- | --- |
| Рівні | Кількість іонів в 1 см3 повітря | |
| позитивні | негативні |
| Мінімально-необхідний  Оптимальний  Максимально-допустимий | 400  1500-3000  50000 | 600  3000-5000  50000 |

Як в холодний, так і теплий періоди року параметри мікроклімату в приміщені відповідають нормам.

Вміст забруднюючих речовин у повітрі і рівень іонізації приміщення також відповідають нормам.

Для забезпечення оптимальних мікрокліматичних умов в будь-який період року приміщення, в яких розташовані комп’ютеризовані робочі місця, передбачені системи опалення. Однак проектом передбачене краще вирішення цього питання – встановлення кондиціонерів, які автоматично підтримують задані параметри мікроклімату.

Вміст озону у повітрі робочої зони повинен бути мінімальним. Основними джерелами озону на комп’ютеризованих місцях є ВДТ та лазерні принтери. З огляду на це, передбачено виключати ВДТ у випадках, коли він не використовується, а лазерний принтер – розташувати подалі від робочого місця оператора. Для запобігання несприятливого впливу озону та інших шкідливих речовин на здоров’я операторів передбачається функціонування припливно-витяжної вентиляції.

### 4.1.3 Виробниче освітлення

Норми освітлення виробничих приміщень вказані в ДБН В.2.5.-28-2006. Освітлення у приміщеннях з ЕОМ має бути суміщеним, при якому недостатнє за нормами природне освітлення доповнене штучним. Природне освітлення має бути бічним, бажано одностороннім. Для зменшення засліплюючої дії сонячних променів світлові отвори мають бути зорієнтовані на північ чи північний схід. Коефіцієнт природної освітленості (КПО) має становити не менше 1,5 %. Нормований рівень освітленості на робочому столі - 300-500 лк.

Для приміщень до 6 м глибини розраховується для відстані 1 м від стіни, на рівні робочої поверхні. В нашому випадку це 5 м від вікна.

Освітлення у приміщеннях є суміщеним.

Природне освітлення – бічне, одностороннє. Для регулювання інтенсивності природнього освітлення на вікнах встановлені непрозорі жалюзі з можливістю варіювання ступенем закритості.

Значення освітленості на відстані 5 м – 100-110 лк. Максимальна (біля вікна) – 1350 лк (рисунок 4.3). Для цього розрахунку значення зовнішньої горизонтальної освітленості, яка створюється світлом повністю відкритого небосхилу 50127 лк. КПО для 5 м становить 0,22%, що менше норми. Нормативне значення 1,35% виконується для відстаней до 1,8-1,9 м від вікна.

Штучне освітлення у приміщеннях обладнане у вигляді загальної системи рівномірного освітлення. Світильники розміщені зверху, тому не світять прямо в око чи в екран монітору. Газорозрядні лампи прямого світла П.

На більшій частині приміщення штучне освітлення складає 350-420 лк, що відповідає нормам (рисунок 4.4).

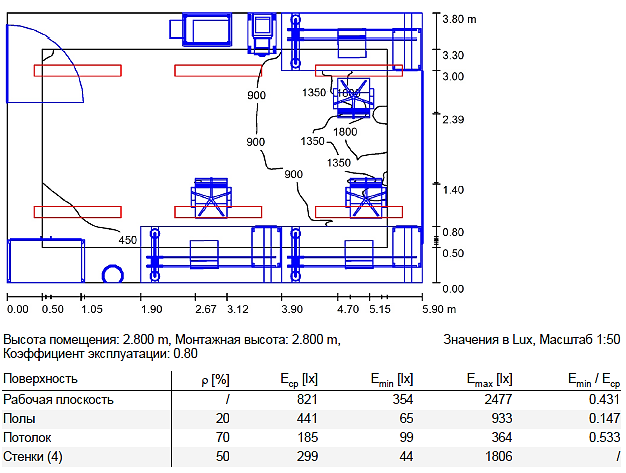


Рисунок 4.2 – Розподіл ізоліній при комбінованому освітленні

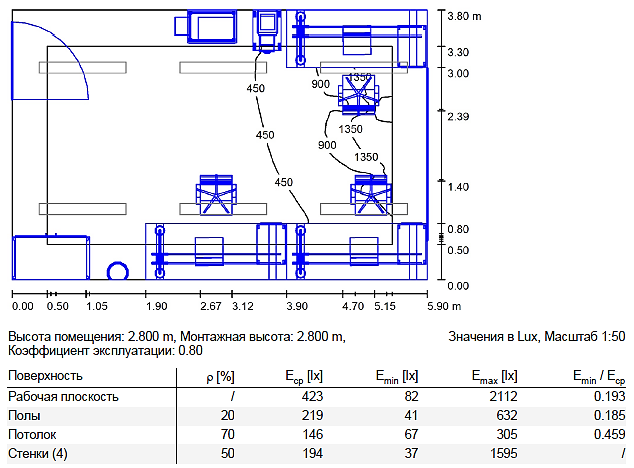


Рисунок 4.3 – Розподіл ізоліній при природньому освітленні

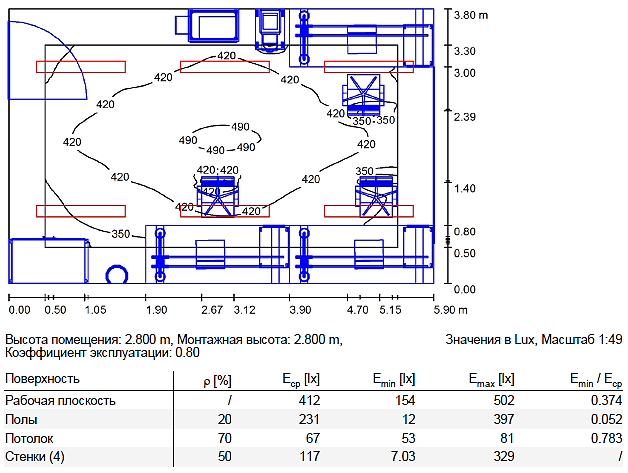


Рисунок 4.4 – Розподіл ізоліній при штучному освітленні

Для створення безпечної зорової роботи на ВДТ передбачено: створювати на робочих місцях освітленість, що відповідає гігієнічним нормам; забезпечувати рівномірність і постійність рівня освітленості; не створювати на робочому місці різких і глибоких тіней; обмежувати пульсацію світлового потоку; не зменшувати необхідний контраст фону та об'єктів, зображених на екрані ВДТ; застосувати на екрані ЕОМ найкращі за видимістю поєднання кольорів, а також чергувати фони.

### 4.1.4 Виробничий шум

Згідно з ДСН 3.3.6.037-99 максимальне значення звукового тиску для програмістів ЕОМ складає 50 дБА.

Основними джерелами шуму в приміщені є вентилятори системи охолодження ноутбука, накопичувач на жорстких магнітних дисках, принтер та копіювальний апарат.

У приміщенні присутні 3 комп'ютери з рівнем шуму 30 дБА, принтер з рівнем шуму 42 дБА та копіювальний апарат з рівнем шуму 48 дБА.

Загальний шум: LΣ = 10 lg(3·103+104,2+104,8) = 49,1 дБА, що відповідає нормам, але є близьким до порогового значення. Тому слід вжити заходів щодо зменшення рівня шуму, зокрема, від периферійних пристроїв.

Основними заходами та засобами боротьби з шумом є: зниження рівнів шуму в джерелі його утворення (застосовується, як правило, в процесі проектування); використання звукопоглинаючих та звукоізолюючих засобів; раціональне планування виробничих приміщень та робочих місць, вибір менш шумних елементів обладнання.

Задля зменшення рівня шуму проектом передбачено замінити копіювальний апарат на більш сучасну і менш шумну модель.

### 4.1.5 Випромінювання

Допустимі рівні напруженості електромагнітного поля радіочастотного діапазону відповідно до ГОСТ 12.1.006-84 "ССБТ. Электромагнитное поле радиочастот. Допустимые уровни на рабочих местах и требования к проведенню контроля" наведені у таблиці 4.5.

Таблиця 4.5 – Допустимі рівці напруженості електромагнітного поля радіочастотного діапазону

|  |  |  |  |
| --- | --- | --- | --- |
| Діапазон частот, Гц | Допустимі рівні напруженості електромагнітного поля | | Допустима поверхнева щільність потоку енергії, Вт/м2 |
|  | за електричною складовою (Е), В/м | за магнітною складовою (Н), А/м |  |
| 60кГц до 3МГц | 50 | 5 | - |
| 3МГц до 30МГц | 20 | - | - |
| 30 МГц до 50МГц | 10 | 0,3 | - |
| 50 МГц до 300 МГц | 5 | - | - |
| 300 МГц до 300 ГГц | - | - | 10 |

Рівні напруженості електромагнітного поля у робочому приміщенні відповідають вказаним нормам.

## 4.2 Електробезпека

Виробниче приміщення обладнане стандартною однофазною мережею змінного струму потужністю 220В. Наявні 2 електророзетки, що розраховані на напругу електромережі, а також 1 вимикач світла.

Серед використовуваних електричних приладів є три персональні комп’ютери (ноутбуки), лазерний принтер, копіювальний апарат, 6 ламп загального освітлення. ЕОМ та периферійні пристрої підключаються до електромережі за допомогою двох мережевих концентраторів (на 4 входи кожен). Всі прилади мають захисну електроізоляцію та запобіжні пристрої. Дотримуються правила безпеки при роботі з увімкненими ЕОМ.

Всі елементи електромережі заізольовані. Використовуються лише справні розетки та перемикачі заводського виконання.

Серед недоліків потрібно відмітити малу кількість електророзеток. Через це доводиться використовувати трійники і збільшувати навантаження на мережу.

Для забезпечення нормальної роботи електромережі планується збільшити кількість електророзеток до 4-6.

## 4.3 Безпека в надзвичайних ситуаціях

В даному приміщенні до надзвичайної ситуації може призвести пожежа.

Виробниче приміщення знаходить в цегляній споруді з товщиною стін близько 50 см. Будинок відноситься до другої категорії вогнестійкості.

Виробниче приміщення відноситься до класу П-IIа, тобто в ньому знаходяться тверді горючі речовини. Схема розташування електроприладів та розміщення провідників електромережі не створює пожежонебезпечних ситуацій. Під час роботи не використовуються прилади, які можуть стати джерелами займання.

Приміщення оснащене пожежним краном, який знаходиться біля виходу з кімнати в загальному коридорі. Наявна пожежна сигналізація та один вогнегасник ОУ-2. Працівники ознайомлені з правилами пожежної безпеки, способами гасіння пожежі та планом евакуації (зображений на рисунку 4.5).

Для поліпшення протипожежного стану приміщення передбачене встановлення в приміщенні ще одного вогнегаснику типу ОУ-2 (площа кімнати становить 22,42 м2, вогнегасник розрахований на 20 м2, тому бажано мати резерв ресурсів пожежогасіння).

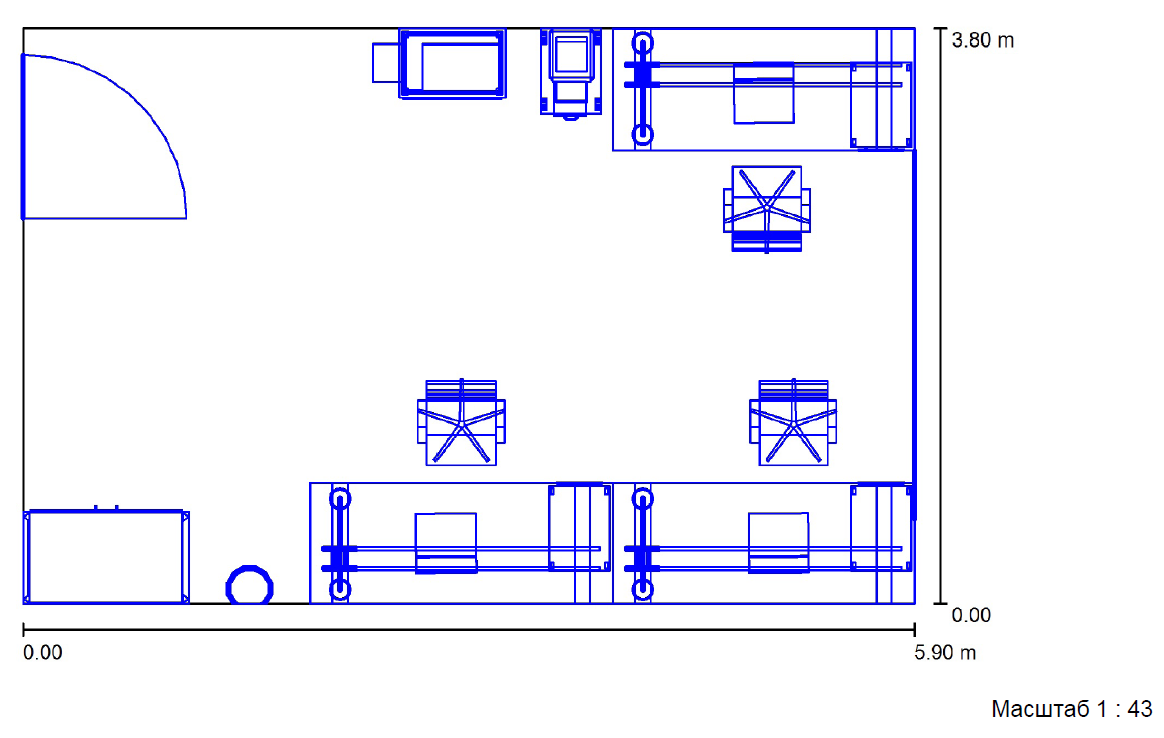
****

Рисунок 4.5 – План евакуації з приміщення

## Висновки до розділу 4

Даний розділ було присвячено охороні праці та безпеки у надзвичайних ситуаціях для користувачів комп’ютерів та програмістів ЕОМ. Було встановлено, що виробниче приміщення, в якому виконувалася робота, відповідає нормам НПАОП 0.00-1.28-10 “Правила охорони праці на час експлуатації ЕОМ” , а також ДСанПіН 3.3.2-007-98 “Державні санітарні правила й норми роботи з візуальними дисплейними терміналами ЕОМ”. Але для покращення умов роботи та збільшення рівня безпеки проектом передбачено встановлення кондиціонерів, заміна копіювального пристрою на менш шумну модель, збільшення кількості розеток для зменшення навантаження на електромережу, а також встановлення ще одного вогнегаснику типу ОУ-2, оскільки наявного вогнегаснику може не вистачити для покриття всього об’єму приміщення.

# Висновки

У цій роботі було проведено аналіз перспективного інструменту криптоаналізу – кубічних атак, ключовими поняттями для яких є поняття мак терму та супеполіному. Макстерм – така підмножина множини відкритих змінних шифруючої функції, що просумувавши виходи даної функції по всім значенням змінних, які входять у макстерм, можна отримати лінійний поліном від секретних змінних. Цей поліном в такому випадку називається суперполіномом.

Було показано, що складність атаки експоненційно зростає при збільшенні степеня макстерма. Тому ефективно можна застосовувати атаку лише при досить невеликих степенях макстерма. У роботі було проведено аналіз випадку, коли у функції наявні макстерми лише першого степеня.

У результаті аналізу залежності кількості сприятливих функцій, що містять макстерми лише першого степеня, від степеня функції та кількості відкритих і секретних змінних, отримано математичний вираз для згаданої величини, а розраховані за ним значення перевірено методом порівняння із результатами програмного розрахунку. Функція визнається сприятливою, якщо ранг отриманої на основі знайдених пар макстерм-суперполіном системи лінійних рівнянь дорівнює кількості секретних змінних.

Ідея програмної реалізації полягає у переборі потенційно сприятливих функцій та застосуванні до них алгоритмів кубічної атаки. Оскільки перебір всіх функцій є надто ресурсоємною задачею, у програмі було використано генератор потенційно сприятливих функцій, що дозволив значно прискорити розрахунки.

Також були розраховані імовірності успіху кубічної атаки з макстермами тільки першого степеня для різних комбінацій кількостей секретних і відкритих змінних (*t* і *s* відповідно). Було виявлено, що при зростанні різниці між *t* і *s*, імовірність успіху прямує до 1. Але на практиці зазвичай виконується нерівність . В найкращому випадку, коли  (інакше, тобто, коли , кількість сприятливих функцій дорівнює 0), спостерігається стрімкий спад імовірності успіху атаки, і для досить великих значень кількості змінних ця імовірність є близькою до 0. Отже, за таких умов макстермів лише першого степеня недостатньо.

Програмно були отримані результати також для випадку *d* = 2 (другий степінь макстерма). Порівняно із випадком *d* = 1, для аналогічних значень степеня функції і кількості змінних, кількість сприятливих функцій із макстермами другого степеня є дещо більшою (при ). Якщо ж комбінувати макстерми різних степенів, то можна отримати ще кращі результати. Таким чином, доцільною і перспективною є задача аналізу кубічних атак із більшими степенями макстермів.

Отримані результати є підґрунтям для подальших досліджень за даною тематикою, зокрема, математичні співвідношення та результати програмних розрахунків можуть стати основою для аналізу кубічних атак із більшими степенями макстермів.

# ****Перелік посилань****

1. Vielhaber M. Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack / Vielhaber M. – IACR Cryptology ePrint Archive, 2007/413. [Online] Available at: <http://eprint.iacr.org/2007/413.pdf>.
2. Vielhaber M. AIDA Breaks (BIVIUM A and B) in 1 Minute Dual Core CPU Time / Vielhaber M. – IACR Cryptology ePrint Archive, 2009/402. [Online] Available at: http://eprint.iacr.org/2009/402.pdf.
3. Dinur I. Cube attacks on tweakable black box polynomials / Dinur I., Shamir A. – Cryptology ePrint Archive, 2008/385. [Online] Available at: http://eprint.iacr.org/2008/385.pdf.
4. Dinur I. Cube attacks on tweakable black box polynomials / Dinur I., Shamir A. // EUROCRYPT, vol. 5479 of Lecture Notes in Computer Science ­– Springer, 2009. – P. 278-299.
5. Bedi S.S. Cube Attacks on Trivium / Bedi S.S., Pillai R. – IACR Cryptology ePrint Archive, 2009/15. [Online] Available at: http://eprint.iacr.org/2009/015.pdf.
6. Mroczkowski P. The Cube Attack on Stream Cipher Trivium and Quadraticity Tests / Mroczkowski P., Szmidt J. – IACR Cryptology ePrint Archive, 2010/580. [Online] Available at: <http://eprint.iacr.org/2010/580.pdf>.
7. Zhang A. Extensions of the Cube Attack Based on Low Degree Annihilators / Zhang A., Lim C.-W., Khoo K., Wei L., Pieprzyk J. – CANS '09 Proceedings of the 8th International Conference on Cryptology and Network Security – Springer, 2009. – P. 87-102.
8. Aumasson J-P. Cube Testers and Key Recovery Attacks on Reduced Round MD6 and Trivium / Aumasson J-P., Meier W., Dinur I., Shamir A. // Fast Software Encryption 2009, LNCS, vol 5665 – Springer, 2009. – P. 1-22.
9. Mroczkowski P. The Cube Attack on Courtois Toy Cipher / Mroczkowski P., Szmidt J. – IACR Cryptology ePrint Archive, 2009/497. [Online] Available at: <http://eprint.iacr.org/2009/497.pdf>.
10. Aumasson J.-P. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128 / Aumasson J-P., Dinur I., Henzen L., Meier W., Shamir A. – IACR Cryptology ePrint Archive, 2009/218. [Online] Available at: http://eprint.iacr.org/2009/218.pdf.
11. Meier W. Cube Testers and Key Recovery in Symmetric Cryptography / Meier W. – 2009. [Online] Available at: <http://indocrypt09.inria.fr/slides_cube_ind09.pdf>.
12. Blum M. Self-testing/correcting with applications to numerical problems / Blum M., Luby M., Rubinfeld R. – Journal of Computer and System Sciences, 47 – Elsevier, 1993. – P. 549-595.
13. Zhu B. A Practical Platform for Cube-Attack-like Cryptanalyses / Zhu B., Yu W., Wang T. – IACR Cryptology ePrint Archive, 2010/664. [Online] Available at: http://eprint.iacr.org/2010/664.pdf.

**Додатки**

# ****Додаток А. Код алгоритмів кубічної атаки****

**Файл «CubeAttack.h»**

#ifndef \_\_CUBE\_ATTACK\_H\_\_

#define \_\_CUBE\_ATTACK\_H\_\_

#include <iostream>

#include <set>

#include <vector>

#include <BoolFunction.h>

using namespace std;

#define MTDEG\_NOTSPECIFIED 0

struct Cube

{

uint32\_t maxterm;

uint32\_t superpoly;

};

class CubeAttack32

{

public:

CubeAttack32();

//finds proper cubes

void find\_cubes(uint32\_t maxtermDegree = MTDEG\_NOTSPECIFIED);

//finds the superpoly for given maxterm

uint32\_t getSuperpoly(uint32\_t maxterm,int iteration);

//finds the constant in superpoly for given maxterm

uint32\_t getConstant(uint32\_t maxterm,int iteration);

//checks if maxterm has linear multiplier

bool is\_linear(uint16\_t\* mt\_indices,uint32\_t deg,uint32\_t maxterm,int iteration);

//prints the system of linear equations

//finds the sum of cipher outputs over maxterm variables at some iteration

uint32\_t maxtermSum(uint16\_t\* mt\_indices,uint32\_t deg,

uint32\_t maxterm,uint32\_t key,

int iteration);

//outputs the system of linear equations for key recovery

void getEquationsSystem(uint32\_t key);

//solves the system of linear equations and gets the fixed\_key

void solveSystem();

//resets all data

void reset();

void getIndices(uint32\_t maxterm,uint16\_t\* mt\_indices);

// creates matrix of linearly independent superpolies

void rebuildSPMatrix();

// returns superpoly matrix rank

uint32\_t getSPMatrixRank() const { return superpolies.size(); };

// returns true if superpoly matrix rank is equal to s

bool isSuccessfull() const;

private:

set<uint32\_t> superpolies;

vector<Cube> cubes;

//returns cipher stream bit on given iteration and given key

uint32\_t getCipherBit(uint32\_t key, int iteration);

};

#endif /\* \_\_CUBE\_ATTACK\_H\_\_ \*/

**Файл «CubeAttack.cpp»**

#include "CubeAttack.h"

#include <stdlib.h>

#include <support.h>

using namespace std;

//---------------------------------------------------------------------------

uint32\_t max\_deg = 2; //max degree of maxterm

uint32\_t max\_iter = 1; //max number of cipher output bits

uint32\_t max\_sysDim = 20; //size of the buffer for linear equations

bool init = false; //disable/enable cipher initialization

int cipherNum = 0; //number that indicates what cipher is in use (for example, 0 for VeryBadCipher)

int keylen = 16, ivlen = 16; //key and IV lengths

uint32\_t mt\_counter[8] = {0,0,0,0,0,0,0,0};

// value of constant C in linearity test

#define LIN\_TEST\_REPEAT (0x20UL << keylen) /\*0x10UL\*/

//---------------------------------------------------------------------------

BoolFunction E0; //arbitrary Boolean function

//---------------------------------------------------------------------------

CubeAttack32::CubeAttack32()

{

srand(0);

cubes.reserve(max\_sysDim);

}

void CubeAttack32::reset()

{

cubes.clear();

superpolies.clear();

srand(0);

}

//returns cipher stream bit on given iteration

uint32\_t CubeAttack32::getCipherBit(uint32\_t key, int iteration)

{

uint32\_t res = 0;

res = E0.getOutput(key);

return res;

}

//find proper cubes

void CubeAttack32::find\_cubes(uint32\_t maxtermDegree)

{

uint32\_t maxterm = 1;

uint32\_t deg;

uint16\_t ind[16] = {0};

uint32\_t lim = 0xFFFFFFFFUL >> (32-ivlen);

Cube cube = {0, 0};

cubes.clear();

superpolies.clear();

if (maxtermDegree == MTDEG\_NOTSPECIFIED)

{

for (; maxterm <= lim; maxterm++)

{

deg = sumOnes(maxterm);

if (deg > max\_deg) continue;

getIndices(maxterm, ind);

if (is\_linear(ind, deg, maxterm, 1))

{

mt\_counter[deg-1]++;

cube.maxterm = maxterm;

cube.superpoly = getSuperpoly(maxterm, 0);

cubes.push\_back(cube);

superpolies.insert(cube.superpoly);

}

}

}

else

{

if (maxtermDegree > ivlen) return;

if (maxtermDegree == 1)

{

deg = 1;

uint32\_t indice = 0;

for (; maxterm <= lim; maxterm <<= 1)

{

ind[0] = indice++;

if (is\_linear(ind, deg, maxterm, 1))

{

mt\_counter[deg-1]++;

cube.maxterm = maxterm;

cube.superpoly = getSuperpoly(maxterm, 0);

cubes.push\_back(cube);

superpolies.insert(cube.superpoly);

}

}

}

else

{

for (; maxterm <= lim; maxterm++)

{

deg = sumOnes(maxterm);

if (deg != maxtermDegree) continue;

getIndices(maxterm, ind);

if (is\_linear(ind, deg, maxterm, 1))

{

mt\_counter[deg-1]++;

cube.maxterm = maxterm;

cube.superpoly = getSuperpoly(maxterm, 0);

cubes.push\_back(cube);

superpolies.insert(cube.superpoly);

}

}

}

}

}

void CubeAttack32::getIndices(uint32\_t maxterm,uint16\_t\* ind)

{

int k = 0;

for (int i = 0; i < ivlen; i++)

{

if ((maxterm >> i) & 0x1)

{

ind[k] = i;

k++;

}

}

return;

}

uint32\_t CubeAttack32::maxtermSum(uint16\_t\* mt\_indices,uint32\_t deg,

uint32\_t maxterm,uint32\_t key,

int iteration)

{

uint32\_t sum = 0, one;

uint32\_t keyOnes = key | maxterm, keyZeros = key & (~maxterm);

sum ^= getCipherBit(keyZeros, iteration);

sum ^= getCipherBit(keyOnes, iteration);

switch (deg)

{

case 1:

return sum;

case 2:

one = 1 << mt\_indices[0];

sum ^= getCipherBit(keyZeros | one, iteration);

sum ^= getCipherBit(keyOnes & (~one), iteration);

return sum;

case 3:

for(uint32\_t i = 0; i < deg; i++)

{

one = 1 << mt\_indices[i];

sum ^= getCipherBit(keyZeros | one, iteration);

sum ^= getCipherBit(keyOnes & (~one), iteration);

}

return sum;

default:

{

uint32\_t limit = 0xFFFFFFFF << deg, temp = 0;

limit =~ limit;

for(uint32\_t i = 1; i < limit; i++)

{

for(uint32\_t j = 0; j < deg; j++)

{

one = (i >> j) & 0x1;

one = one << mt\_indices[j];

temp ^= one;

}

sum ^= getCipherBit(keyZeros | temp, iteration);

temp = 0;

}

return sum;

}

}

}

//finds the superpoly for given maxterm

uint32\_t CubeAttack32::getSuperpoly(uint32\_t maxterm,int iteration)

{

uint32\_t superpoly = 0, constant;

constant = getConstant(maxterm, iteration);

uint32\_t deg = sumOnes(maxterm);

uint16\_t ind[16];

getIndices(maxterm, ind);

uint32\_t one, temp;

for(int i = 0; i < keylen; i++)

{

one = 1 << (i+ivlen);

temp = maxtermSum(ind, deg, maxterm, one, iteration) ^ constant;

if (temp) superpoly ^= one;

}

return superpoly;

}

//returns constant in superpoly

uint32\_t CubeAttack32::getConstant(uint32\_t maxterm,int iteration)

{

uint32\_t constant;

uint32\_t deg = sumOnes(maxterm);

uint16\_t ind[16];

getIndices(maxterm, ind);

constant = maxtermSum(ind, deg, maxterm, 0, iteration);

return constant;

}

//Algorithm 3 Term-by-Term Linearity Testing (Bo Zhu, Wenye Yu and Tao Wang)

bool CubeAttack32::is\_linear(uint16\_t\* ind,uint32\_t deg,uint32\_t maxterm,int iteration)

{

uint32\_t temp, t, superpoly=0, constant, y, z, fy, fz;

constant=getConstant(maxterm,iteration);

for(int i = 0; i < keylen; i++)

{

t = 1 << (i+ivlen);

temp = maxtermSum(ind, deg, maxterm, t, iteration) ^ constant;

if (temp)

{

superpoly ^= t;

}

}

if(!superpoly) return 0;

uint32\_t keymask = 0xFFFFFFFF >> (32-ivlen-keylen);

keymask &= 0xFFFFFFFF << ivlen;

for(int j = 0; j < keylen; j++)

{

t = 1 << (j+ivlen);

if ((superpoly >> (j+ivlen)) & 0x1)

{

for (uint32\_t c = 0; c < LIN\_TEST\_REPEAT; c++)

{

y = (uint32\_t)rand() << ivlen;

y &= keymask;

z = y;

y &= ~t;

z = z | t;

fy = maxtermSum(ind, deg, maxterm, y, iteration);

fz = maxtermSum(ind, deg, maxterm, z, iteration);

if (fy == fz) return 0;

}

}

}

return 1;

}

void CubeAttack32::rebuildSPMatrix()

{

vector<uint32\_t> eq;

if (!superpolies.size() || (superpolies.size() == 1)) return;

// copy superpolies to temporary vector

copy(superpolies.begin(), superpolies.end(), back\_inserter(eq));

//straight move of Gauss' method

for (uint32\_t i = 0; i < eq.size() - 1; i++)

{

int index = 0;

for (int k = keylen + ivlen - 1; k >= ivlen; k--)

{

if ((eq[i] >> k) & 0x1)

{

index = k;

break;

}

}

for (uint32\_t j = i + 1; j < eq.size(); j++)

{

if ((eq[j] >> index) & 0x1)

{

eq[j] ^= eq[i];

}

}

}

superpolies.clear();

for (vector<uint32\_t>::const\_iterator it = eq.begin(); it != eq.end(); it++)

{

if (\*it) superpolies.insert(\*it);

}

}

// returns true if superpoly matrix rank is equal to s

bool CubeAttack32::isSuccessfull() const

{

return getSPMatrixRank() >= keylen;

}

# ****Додаток Б. Код генератора потенційно сприятливих булевих функцій****

**Файл «GoodFuncGen.h»**

#ifndef \_\_GOOD\_FUNCTION\_GENERATOR\_H\_\_

#define \_\_GOOD\_FUNCTION\_GENERATOR\_H\_\_

#include <stdint.h>

#include <BoolFunction.h>

#include <vector>

using namespace std;

typedef std::vector<uint32\_t> vector\_uint32;

typedef std::vector<uint64\_t> vector\_uint64;

///

/// @class GoodFunctionGenerator

/// @brief Generator of boolean functions 'good' for CA

///

class GoodFunctionGenerator

{

public:

explicit GoodFunctionGenerator(

uint32\_t ivlen, uint32\_t keylen,

uint32\_t deg, uint32\_t mtDeg = 1);

bool nextGoodFunction(BoolFunction& F);

uint32\_t getTotalSummandCount() const

{

return totalSummandCount;

}

uint32\_t getHighDegSummandCount() const

{

if (summands.size()) return summands.back().size();

else return 0;

}

uint32\_t getLowDegSummandCount() const

{

return totalSummandCount - getHighDegSummandCount();

}

uint32\_t getSummandCount(uint32\_t deg) const

{

if (summands.size() < deg + 1) return 0;

else return summands[deg].size();

}

uint32\_t getFunctionDegree() const

{

return summands.size() ? summands.size() - 1 : 0;

}

private:

uint32\_t \_ivlen, \_keylen;

vector<vector\_uint32> summands;

uint32\_t totalSummandCount;

uint64\_t index\_highDeg;

uint64\_t index\_lowDeg;

bool hasNext;

vector\_uint64 indexes;

GoodFunctionGenerator() {}

};

#endif /\* \_\_GOOD\_FUNCTION\_GENERATOR\_H\_\_ \*/

**Файл «GoodFuncGen.cpp»**

#include <GoodFuncGen.h>

#include <assert.h>

#include <support.h>

#include <iostream>

#include <set>

#include <vector>

GoodFunctionGenerator::GoodFunctionGenerator(

uint32\_t ivlen, uint32\_t keylen,

uint32\_t deg, uint32\_t mtDeg)

{

uint32\_t dim = ivlen + keylen;

\_ivlen = ivlen;

\_keylen = keylen;

\_d = mtDeg;

totalSummandCount = 0;

index\_lowDeg = 0;

index\_highDeg = 1;

hasNext = false;

assert(((deg > 1) && (deg <= dim)) &&

"Function degree should be in range [2, ivlen + keylen]");

assert((ivlen >= keylen) &&

"Number of public variables should be greater than number of private variables");

if ((deg > 1) && (deg <= dim))

{

uint32\_t summand, var1, var2;

vector\_uint32 summandsForDeg;

// init summands

summands.resize(deg + 1);

summands[0] = summandsForDeg;

summands[1] = summandsForDeg;

// init indexes

indexes.resize(deg + 1);

indexes[0] = 0;

indexes[1] = 0;

// reserve space for summands

summandsForDeg.reserve(0x1UL << dim);

// for summand degree == 2

if (deg == 2)

{

// we take all summands of kind v\*x,

// where v is a public variable and

// x is a secret variable

for (uint32\_t i = 0; i < ivlen; i++)

{

var1 = 0x1UL << i;

for (uint32\_t j = ivlen; j < dim; j++)

{

var2 = 0x1UL << j;

summand = var1 ^ var2;

summandsForDeg.push\_back(summand);

}

}

summands[2] = summandsForDeg;

indexes[2] = 0;

totalSummandCount = summandsForDeg.size();

summandsForDeg.clear();

}

else

{

// we take all summands of kind v\*x,

// where v is a public variable and

// x is a secret variable

for (uint32\_t i = 0; i < ivlen; i++)

{

var1 = 0x1UL << i;

for (uint32\_t j = ivlen; j < dim; j++)

{

var2 = 0x1UL << j;

summand = var1 ^ var2;

summandsForDeg.push\_back(summand);

}

}

summands[2] = summandsForDeg;

indexes[2] = 0;

totalSummandCount = summandsForDeg.size();

summandsForDeg.clear();

var1 = 0x1UL << ivlen;

for (uint32\_t i = 3; i < deg; i++)

{

for (uint32\_t k = 7; k < (0x1UL << dim); k++)

{

if ((sumOnes(k) == i) && (k >= var1))

{

summandsForDeg.push\_back(k);

}

}

summands[i] = summandsForDeg;

indexes[i] = 0;

totalSummandCount += summandsForDeg.size();

summandsForDeg.clear();

}

for (uint32\_t k = 7; k < (0x1UL << dim); k++)

{

if (sumOnes(k) == deg)

{

summandsForDeg.push\_back(k);

}

}

summands[deg] = summandsForDeg;

indexes[deg] = 0;

totalSummandCount += summandsForDeg.size();

summandsForDeg.clear();

}

}

if (totalSummandCount) hasNext = true;

}

bool GoodFunctionGenerator::nextGoodFunction(BoolFunction& F)

{

uint32\_t \*arrayOfSummands = 0, k = 0;

vector\_uint32 tempVec;

set<uint32\_t> badIVBits;

tempVec.reserve(256);

vector\_uint32 bitCounters(\_ivlen + \_keylen, 0);

BEGIN:

if (!hasNext) return false;

// generate new function

if ((indexes.back() == 0) || (getLowDegSummandCount() == 0))

{

indexes.back()++;

}

else if (indexes.back())

{

uint32\_t allZero = 0;

for (uint32\_t i = 2; i < indexes.size() - 1; i++)

{

indexes[i]++;

if (indexes[i] == (0x1ULL << getSummandCount(i)))

{

indexes[i] = 0;

allZero += getSummandCount(i);

}

else break;

}

if (allZero == getLowDegSummandCount()) indexes.back()++;

}

if (indexes.back() == (0x1ULL << summands.back().size()))

{

indexes.back() = 0;

hasNext = false;

return hasNext;

}

// select summands according to indexes

uint32\_t ivMask = ~(0xFFFFFFFFUL << \_ivlen);

uint32\_t keyMask = 0xFFFFFFFFUL << \_ivlen;

for (uint32\_t i = 2; i < indexes.size(); i++)

{

for (uint32\_t j = 0; j < getSummandCount(i); j++)

if ((indexes[i] >> j) & 0x1UL)

{

tempVec.push\_back(summands[i][j]);

if ((\_d == 1) && (sumOnes(tempVec.back() & keyMask) > 1) &&

(sumOnes(tempVec.back() & ivMask) == 1))

{

uint32\_t k = 0, tmp = tempVec.back() & ivMask;

while (tmp >>= 1) k++;

badIVBits.insert(k);

}

for (uint32\_t k = 0; k < bitCounters.size(); k++)

{

bitCounters[k] += (tempVec.back() >> k) & 0x1UL;

}

}

}

// reset bad public variable counters

for (set<uint32\_t>::iterator it = badIVBits.begin(); it != badIVBits.end(); it++)

{

bitCounters[\*it] = 0;

}

// calculate how many public and secret vars we really have

uint32\_t keyVarCount = 0, pubVarCount = 0;

for (uint32\_t k = 0; k < \_ivlen; k++)

pubVarCount += bitCounters[k] > 0;

for (uint32\_t k = \_ivlen; k < \_ivlen + \_keylen; k++)

keyVarCount += bitCounters[k] > 0;

// don't generate functions that have number of summands < key length

// or don't have all the key bits present

if ((tempVec.size() < \_keylen) ||

(pubVarCount < keyVarCount) ||

(keyVarCount < \_keylen))

{

tempVec.clear();

badIVBits.clear();

goto BEGIN;

}

if (tempVec.size()) arrayOfSummands = new uint32\_t[tempVec.size()];

else return hasNext;

for (uint32\_t i = 0; i < tempVec.size(); i++)

{

arrayOfSummands[i] = tempVec[i];

}

F.setFunction(\_ivlen + \_keylen, arrayOfSummands, tempVec.size());

return hasNext;

}

**Файл «BoolFunction.h»**

#ifndef \_\_BOOL\_FUNCTION\_H\_\_

#define \_\_BOOL\_FUNCTION\_H\_\_

#include <stdint.h>

///

/// @class BoolFunction

/// @brief Arbitrary boolean function representation

///

/// Boolean function is created regarding to the given

/// function index value and number of variables.

/// Each bit of index corresponds to some function summand.

/// For example:

/// for number of variables = 2:

/// index = 1011 ==> f(x) = x1\*x2 + x1 + 1

/// index = 0111 ==> f(x) = x2 + x1 + 1

/// index = 1100 ==> f(x) = x1\*x2 + x2 + 0

///

class BoolFunction

{

public:

BoolFunction();

BoolFunction(uint32\_t var, uint64\_t ind);

~BoolFunction();

uint32\_t getNumVariables() const { return dim; }

uint32\_t getNumSummands() const { return size; }

uint32\_t getOutput(uint32\_t input) const;

void setFunction(uint32\_t var, uint64\_t ind, uint32\_t skippedBits = 0);

void setFunction(uint32\_t var, uint32\_t\* pSummands, uint32\_t numSummands);

void print();

bool isConstOrLinear() const;

bool isConstOrHasLinSummands() const;

bool hasVarMul(uint32\_t startVarID,

uint32\_t endVarID, uint32\_t deg = 0); // variable ID belong to set {1;dim}

uint32\_t getDegree() const;

uint32\_t getMinSummandDegree() const;

bool isNotGoodForCA(uint32\_t ivlen, uint32\_t keylen, uint32\_t deg);

// returns single summand of specified degree

// (if there is no (or more than 1) such summand, then result is -1)

int getSingleSummandOfDeg(uint32\_t deg);

private:

uint32\_t dim;

uint64\_t index;

uint32\_t size;

uint32\_t constant;

uint32\_t \* summands;

uint32\_t skipBits;

bool hasVarMulInSummand(uint32\_t startVarID,

uint32\_t endVarID, uint32\_t summand, uint32\_t deg = 0);

};

#endif /\* \_\_BOOL\_FUNCTION\_H\_\_ \*/

**Файл «BoolFunction.cpp»**

#include <iostream>

#include <math.h>

#include <BoolFunction.h>

#include <support.h>

#include <Combinatorics.h>

#include <stdio.h>

#include <assert.h>

using namespace std;

BoolFunction::BoolFunction()

{

dim = 1;

index = 0;

size = 1;

summands = new uint32\_t[1];

summands[0] = 0;

constant = 0;

skipBits = 0;

}

BoolFunction::~BoolFunction()

{

delete [] summands;

}

BoolFunction::BoolFunction(uint32\_t var, uint64\_t ind)

{

BoolFunction();

this->setFunction(var, ind);

}

uint32\_t BoolFunction::getOutput(uint32\_t input) const

{

uint32\_t mul = 1, sum = constant;

for (uint32\_t i = 0; i < size; i++)

{

sum ^= summands[i] == (summands[i] & input);

}

return sum;

}

void BoolFunction::setFunction(uint32\_t var, uint64\_t ind, uint32\_t skippedBits)

{

uint32\_t tmp;

uint32\_t j = 1 - (skippedBits > 0);

uint64\_t max = (uint64\_t)pow(2, var);

dim = var;

index = ind;

constant = 0;

skipBits = skippedBits;

tmp = sumOnes(ind);

if (tmp == 0)

{

size = 0;

return;

}

if (!skipBits && (index & 0x1))

{

constant = 1;

tmp--;

if (tmp == 0)

{

size = 0;

return;

}

}

if (size != tmp)

{

size = tmp;

delete [] summands;

summands = new uint32\_t[size];

}

for (uint32\_t i = 0; i < size; i++)

{

summands[i] = 0;

for (; j < max; j++)

{

if ((ind >> j) & 0x1)

{

summands[i] = j + skipBits;

j++;

break;

}

}

}

}

void BoolFunction::print()

{

if (!size)

{

cout << "f(x) = 0";

return;

}

// sort summands array before output

sortdec(summands, size);

cout << "f(x) = ";

for (uint32\_t i = 0; i < size; i++)

{

printExpr(summands[i],'\*');

cout << " + ";

}

cout << constant;

}

bool BoolFunction::isConstOrLinear() const

{

uint32\_t a = 0;

if (!size) return true;

for (uint32\_t i = 0; i < size; i++)

{

a += sumOnes(summands[i]);

}

if (a == size) return true;

return false;

}

bool BoolFunction::isConstOrHasLinSummands() const

{

uint32\_t a;

if (!size || constant) return true;

for (uint32\_t i = 0; i < size; i++)

{

a = sumOnes(summands[i]);

if (a <= 1) return true;

}

return false;

}

uint32\_t BoolFunction::getDegree() const

{

uint32\_t a, deg;

if (!size) return 0;

deg = sumOnes(summands[0]);

for (uint32\_t i = 1; i < size; i++)

{

a = sumOnes(summands[i]);

if (a > deg) deg = a;

}

return deg;

}

uint32\_t BoolFunction::getMinSummandDegree() const

{

uint32\_t a, deg;

if (!size) return 0;

deg = sumOnes(summands[0]);

for (uint32\_t i = 1; i < size; i++)

{

a = sumOnes(summands[i]);

if (a < deg) deg = a;

}

return deg;

}

bool BoolFunction::hasVarMulInSummand(uint32\_t startVarID,

uint32\_t endVarID, uint32\_t summand, uint32\_t deg)

{

uint32\_t bit, vars = 0;

if (!startVarID || (startVarID >= endVarID)) return false;

for (uint32\_t i = startVarID - 1; i < endVarID; i++)

{

bit = (summand >> i) & 0x1UL;

vars += bit;

summand ^= bit << i;

}

if (!deg) return (vars > 1) && (!summand);

else return (vars > 1) && (vars < deg) && (!summand);

}

bool BoolFunction::hasVarMul(uint32\_t startVarID,

uint32\_t endVarID, uint32\_t deg)

{

if (!startVarID || (startVarID >= endVarID)) return false;

for (uint32\_t i = 0; i < size; i++)

{

if (hasVarMulInSummand(startVarID, endVarID, summands[i], deg))

return true;

}

return false;

}

bool BoolFunction::isNotGoodForCA(uint32\_t ivlen, uint32\_t keylen, uint32\_t deg)

{

uint32\_t a, maxDeg, minDeg;

if (!size || constant) return true;

// skip constant and linear functions (or functions with linear tails)

// skip functions with min summand degree > ivlen+1 and max summand degree != deg

maxDeg = sumOnes(summands[0]);

if (maxDeg <= 1) return true;

minDeg = maxDeg;

for (uint32\_t i = 1; i < size; i++)

{

a = sumOnes(summands[i]);

if (a <= 1) return true;

if (a > maxDeg) maxDeg = a;

if (a < minDeg) minDeg = a;

}

// skip functions with min summand degree > ivlen+1

if ((minDeg > ivlen + 1) || (maxDeg != deg)) return true;

// skip functions with summands that are multiplications of public variables

if (hasVarMul(1, ivlen, deg)) return true;

// skip functions with summands that are multiplications of private variables

if (hasVarMul(ivlen + 1, ivlen + keylen, deg)) return true;

return false;

}

// returns single summand of specified degree (if there is no (or more than 1) such summand, then result is -1)

int BoolFunction::getSingleSummandOfDeg(uint32\_t deg)

{

int ind = -1;

uint32\_t a;

for (uint32\_t i = 0; i < size; i++)

{

a = sumOnes(summands[i]);

if (a == deg)

{

if (ind == -1) ind = i;

else return -1;

}

}

return ind;

}

void BoolFunction::setFunction(

uint32\_t var, uint32\_t\* pSummands, uint32\_t numSummands)

{

assert (pSummands && "Error, NULL pointer");

if (!pSummands) return;

dim = var;

index = 0;

size = numSummands;

constant = 0;

skipBits = 0;

if (summands) delete [] summands;

summands = pSummands;

}

# Додаток В. Код програми для підрахунку кількості сприятливих функцій

**Файл «GoodFuncCount.cpp»**

#include <stdlib.h>

#include <iostream>

#include <math.h>

#include <CubeAttack.h>

#include <BoolFunction.h>

#include <ctime>

#include <Combinatorics.h>

#include <support.h>

#include <GoodFuncGen.h>

using namespace std;

using namespace combinatorics;

//---------------------------------------------------------------------------

// Defines and used functions

#define OK 0

#define ERROR 1

#define PRINT\_FUNCTIONS 0

#if PRINT\_FUNCTIONS

#define LOG(m) cout << m;

#else

#define LOG(m)

#endif

#define MIN(a,b) ( ((a) < (b)) ? (a) : (b) )

static void GoodFuncCount(uint32\_t m, uint32\_t d, int s, int t);

//---------------------------------------------------------------------------

// Cube Attack - related globals

extern unsigned long max\_deg; //max degree of maxterm

extern unsigned long max\_iter; //max number of cipher output bits

extern unsigned long max\_sysDim; //size of the buffer for linear equations

extern bool init; //disable/enable cipher initialization

extern int cipherNum; //number that indicates what cipher is in use (for example, 0 for VeryBadCipher)

extern int keylen,ivlen; //key and IV lengths

extern uint32\_t mt\_counter[8];

extern BoolFunction E0; //arbitrary bool function

uint64\_t total;

//---------------------------------------------------------------------------

// Entry point

int main(int argc, char\* argv[])

{

uint32\_t s = 1;

uint32\_t t = 1;

uint32\_t d = 1;

uint32\_t m = 2;

if (argc == 5)

{

m = atoi(argv[1]);

d = atoi(argv[2]);

s = atoi(argv[3]);

t = atoi(argv[4]);

}

if (!m)

{

cerr << "Error, function degree m should be > 0" << endl;

return ERROR;

}

cout << "Input arguments:" << endl;

cout << "m = " << m << "; d = " << d << "; s = " << s << "; t = " << t << endl;

GoodFuncCount(m, d, s, t);

return OK;

}

//---------------------------------------------------------------------------

// Used functions implementation

void GoodFuncCount(uint32\_t m, uint32\_t d, int s, int t)

{

long double time1, time2;

CubeAttack32 CA;

max\_iter = 1;

keylen = s;

ivlen = t;

max\_deg = MIN(ivlen, m-1);

total = (uint64\_t)pow(2, ivlen + keylen);

uint64\_t linFuncCount = (uint64\_t)pow(2, ivlen + keylen + 1);

uint64\_t goodFunctions = 0;

uint64\_t tooGoodFunctions = 0;

uint64\_t\* mt\_func\_counter = new uint64\_t[ivlen];

for(int i = 0; i < ivlen; i++) mt\_func\_counter[i] = 0;

time1 = clock();

// initialize good function generator

GoodFunctionGenerator goodFuncGen(t, s, m);

uint64\_t neutral = BinomCoeffSum(t + s, 0, m) - goodFuncGen.getTotalSummandCount();

neutral = 0x1ULL << neutral;

// Cube Attack on any bool function of specified dimension

while (goodFuncGen.nextGoodFunction(E0))

{

CA.reset();

// skip 'bad' functions

if (E0.isNotGoodForCA(t, s, m)) continue;

// find maxterms using Cube Attack algorithms

CA.find\_cubes(d);

// rebuild superpoly matrix to make all superpolies linearly independent

if (CA.getSPMatrixRank() >= keylen) CA.rebuildSPMatrix();

bool success = CA.isSuccessfull();

if (success)

{

goodFunctions++;

if (mt\_counter[d-1] > s)

{

tooGoodFunctions++;

LOG("[MEGA SUCCESS!] ");

}

else

{

LOG("[SUCCESS!] ");

}

}

#if PRINT\_FUNCTIONS

E0.print();

#endif

LOG(" | Maxterms: ");

for(int i = 0; i < d; i++)

{

if(mt\_counter[i]) mt\_func\_counter[i]++;

LOG("deg = " << i + 1 << ": " << mt\_counter[i] << "; ");

mt\_counter[i] = 0;

}

LOG(endl);

}

time2 = clock();

goodFunctions \*= neutral;

tooGoodFunctions \*= neutral;

cout << "Total number of functions: 2^" << total << endl;

cout << "Number of linear functions: " << linFuncCount << endl;

cout << "Number of neutral functions: " << neutral << endl;

cout << "Number of 'good' functions: " << goodFunctions << endl;

cout << "Number of 'too good' functions (number of maxterms > s): " << tooGoodFunctions << endl;

long double millis = ((time2 - time1) \* (long double)1000.0) / (long double)CLOCKS\_PER\_SEC;

long hours = millis / 3600000.0; millis -= hours \* 3600000.0;

long minutes = millis / 60000.0; millis -= minutes \* 60000.0;

long seconds = millis / 1000.0; millis -= seconds \* 1000.0;

cout << "Calculation time: ";

if (hours) cout << hours << " hours; ";

if (minutes) cout << minutes << " minutes; ";

if (seconds) cout << seconds << " seconds; ";

cout << millis << " milliseconds" << endl;

delete [] mt\_func\_counter;

}

# Додаток Г. Код допоміжних програмних модулів

**Файл «support.h»**

#ifndef \_SUPPORT\_H\_

#define \_SUPPORT\_H\_

#include <stdint.h>

// calculates sum of ones in bit representation of a

uint32\_t sumOnes(uint32\_t a);

uint32\_t sumOnes(uint64\_t a);

// prints bit representation of uint32\_t a

void printBits(uint32\_t a, int startFrom = 31);

// prints bit representation of uint64\_t a

void printBits(uint64\_t a, int startFrom = 63);

// prints an uint32\_t a as a math expression with separators s

void printExpr(uint32\_t a, char s);

// sort array in increasing order

void sortinc(uint32\_t\* array, uint32\_t size);

// sort array in decreasing order

void sortdec(uint32\_t\* array, uint32\_t size);

#endif // \_SUPPORT\_H\_

**Файл «support.cpp»**

#include <support.h>

#include <iostream>

#include <stdlib.h>

using namespace std;

uint32\_t sumOnes(uint32\_t a)

{

uint32\_t result = 0;

for(int i = 0; i < 32; i++)

{

result += (a >> i) & 0x1;

}

return result;

}

uint32\_t sumOnes(uint64\_t a)

{

uint32\_t result = 0;

for(int i = 0; i < 64; i++)

{

result += (a >> i) & 0x1;

}

return result;

}

//prints bit representation of uint32\_t a

void printBits(uint32\_t a, int startFrom)

{

if (startFrom > 31) startFrom = 31;

for(int i = startFrom; i >= 0; i--)

{

cout << ((a >> i) & 0x1);

}

}

//prints bit representation of uint64\_t a

void printBits(uint64\_t a, int startFrom)

{

if (startFrom > 63) startFrom = 63;

for(int i = startFrom; i >= 0; i--)

{

cout << ((a >> i) & 0x1);

}

}

//prints an uint32\_t a as a math expression with separators s

void printExpr(uint32\_t a, char s)

{

int firstBitIndx = 0;

for(int i = 0; i < 32; i++)

{

if ((a >> i) & 0x1)

{

firstBitIndx = i;

break;

}

}

for(int i = 31; i > firstBitIndx; i--)

{

if((a >> i) & 0x1)

{

cout << "x" << (i + 1) << s;

}

}

if ((a >> firstBitIndx) & 0x1) cout << "x" << (firstBitIndx + 1);

else cout << 0;

}

static int compare\_inc(const void \* a, const void \* b)

{

return ( \*(int\*)a - \*(int\*)b );

}

static int compare\_dec(const void \* a, const void \* b)

{

return ( \*(int\*)b - \*(int\*)a );

}

// sort array in increasing order

void sortinc(uint32\_t\* array, uint32\_t size)

{

qsort(array, size, sizeof(uint32\_t), compare\_inc);

}

// sort array in decreasing order

void sortdec(uint32\_t\* array, uint32\_t size)

{

qsort(array, size, sizeof(uint32\_t), compare\_dec);

}

**Файл «Combinatorics.h»**

#ifndef \_COMBINATORICS\_H\_

#define \_COMBINATORICS\_H\_

#include <stdint.h>

namespace combinatorics

{

// calculates x!

uint32\_t Fact(uint32\_t x);

// calculates binomial coefficient C{n,k}

uint32\_t BinomCoeff(uint32\_t n, uint32\_t k);

// calculates sum of binomial coefficients

// C{n,start} + C{n,start+1} + ... + C{n,end}

uint32\_t BinomCoeffSum(uint32\_t n, uint32\_t start, uint32\_t end);

} // namespace combinatorics

#endif // \_COMBINATORICS\_H\_

**Файл «Combinatorics.cpp»**

#include "Combinatorics.h"

namespace combinatorics

{

// calculates x!

uint32\_t Fact(uint32\_t x)

{

uint32\_t result = x;

if (x <= 1) return 1;

while (x != 1) result \*= --x;

return result;

}

// calculates binomial coefficient C{n,k}

uint32\_t BinomCoeff(uint32\_t n, uint32\_t k)

{

uint32\_t C;

if (k > n) return 0;

if (k == n) return 1;

C = Fact(n) / (Fact(k) \* Fact(n-k));

return C;

}

// calculates sum of binomial coefficients

// C{n,start} + C{n,start+1} + ... + C{n,end}

uint32\_t BinomCoeffSum(uint32\_t n, uint32\_t start, uint32\_t end)

{

uint32\_t sum = 0;

if (!n) return 0;

for (uint32\_t i = start; i <= end; i++)

{

sum += BinomCoeff(n, i);

}

return sum;

}

} // namespace combinatorics