НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____2022 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: Методи диференціального обчислювального криптоаналізу вбудованих систем

Виконав (-ла): здобувач вищої освіти **IV** курсу, групи ФБ-83
(шифр групи)

Костецька Аліна Ярославівна
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н., доцент Ільїн Микола Іванович
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент к.ф-м.н., доцент Терещенко Іван Миколайович
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2022 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Костецька Аліна Ярославівна
(прізвище, ім'я, по батькові)

1. Тема роботи:
Методи диференціального обчислювального криптоаналізу вбудованих систем, керівник роботи Ільїн Микола Іванович к.т.н., доцент, затверджені наказом по університету від « » _____ 2022 р. №

2. Термін подання здобувачем вищої освіти роботи ___ червня 2022 р.

3. Вихідні дані до роботи: література по криптографії білої скриньки та атаці диференціального обчислювального аналізу, існуючі методи диференціального обчислювального криптоаналізу та інструменти для їх проведення.

4. Зміст роботи: огляд існуючих методів проведення диференціального обчислювального аналізу з необхідними інструментами, удосконалення одного з методів для можливості проведення атаки на архітектурах AMD64, ARM, MIPS вбудованих систем.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): Методи диференціального обчислювального криптоаналізу вбудованих систем - презентація

6. Дата видачі завдання 22.10.2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Отримання завдання	22.10.2021	виконано
2	Вивчення літератури по темі	22.10.2021-22.12.2021	виконано
3	Огляд існуючих методів	22.12.2021-15.01.2022	виконано
4	Налаштування та запуск інструментів для проведення атаки	15.01.2022-20.03.2022	виконано
5	Пошук white-box криптоалгоритму для аналізу	20.03.2022-01.04.2022	виконано
6	Додавання нової функціональності до інструментів (можливість емуляції на архітектурах AMD64, ARM, MIPS)	01.04.2022-10.05.2022	виконано
7	Здійснення атаки з використанням створених рішень	10.05.2022-20.05.2022	виконано
8	Аналіз результатів	20.05.2022-30.05.2022	виконано
9	Проходження переддипломної практики	02.05.2022-29.05.2022	виконано
10	Написання дипломної роботи	10.04.2022-08.06.2022	виконано
11	Передзахист дипломної роботи	13.06.2022	виконано
12	Захист дипломної роботи	20.06.2022	

Здобувач вищої освіти _____ Аліна КОСТЕЦЬКА
(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

Керівник роботи _____ Микола ІЛЬЇН
(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

РЕФЕРАТ

Обсяг роботи 61 сторінка, 23 ілюстрації, 4 таблиці, 6 додатків, 24 джерела літератури.

У роботі визначено існуючі методи проведення диференціального обчислювального аналізу: трасування програми та трасування емулятора. Доповнено існуючі інструменти для виконання другого методу, створено програму з допомогою Qiling Framework, що здатна емулювати файли на архітектурах AMD64, ARM, MIPS, та знімати траси. У трасах логується інформація про запити програми до пам’яті (на читання чи запис), які потім фільтруються та перетворюються у потрібний формат для проведення кореляційного аналізу. Існуючий інструмент для проведення кореляційного аналізу був модифікований для можливості візуалізації результатів.

З допомогою вдосконалених інструментів було проведено атаку диференціального обчислювального аналізу, емульовано роботу криптоалгоритму білої скриньки у незахищеному середовищі виконання та доведено успішність такого методу. Зроблено висновок, що успішність атаки диференціального обчислювального аналізу залежить виключно від імплементації криптоалгоритму, та може бути проведена на будь-якій платформі чи архітектурі.

Отримані результати стануть корисними при дослідженнях атак диференціального обчислювального аналізу у вбудованих системах, що використовують криптографію білої скриньки. Створені інструменти можуть допомогти у аналізі імплементацій криптоалгоритмів на вразливість до даної атаки.

Ключові слова: криптографія білої скриньки, диференціальний обчислювальний аналіз, вбудовані системи, емуляція, трасування.

ABSTRACT

The volume of work is 61 pages, 23 illustrations, 4 tables, 6 appendices, and 24 literature sources.

The paper recalls two existing methods of differential computation analysis: instrumenting the binary and instrumenting an emulator that is in charge of the binary execution. Existing tools have been modified to make them suitable for the second method, including a Qiling Framework program that can emulate files on AMD64, ARM, and MIPS architectures, and record traces. The traces include the information about the addresses in memory that are accessed, the type of the access (read, write, execute), and the value the addresses contain. Then they are filtered and formatted for the differential power analysis tools. The existing tool for conducting differential correlation analysis was modified to allow results visualization.

The improved tools allowed to conduct the differential computation analysis attack, emulate a white-box algorithm in an untrusted execution environment, and demonstrate the method's success. After the attack, it has been concluded that the success of the differential computational analysis depends solely on the implementation of the algorithm, and can be carried out on any platform or architecture.

The results obtained may be useful in studies of differential computational analysis attacks in embedded systems that use white-box cryptography. Improved tools can help analyze whether some white-box implementation is vulnerable to the attack or not.

Keywords: white-box cryptography, differential computation analysis, embedded systems, emulation, tracing.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AES – Advanced Encryption Standard

CFB – Cipher FeedBack (режим AES)

CTR – Counter (режим AES)

DCA – диференціальний обчислювальний аналіз

DES – Data Encryption Standard

DFA – диференціальний аналіз помилок

DPA – диференціальний аналіз енергоспоживання

DRM – технології керування цифровими правами

ECB – Electronic Code Book (режим AES)

OFB – Output FeedBack (режим AES)

WBA – криптоалгоритм білої скриньки (white-box криптоалгоритм)

ВСТУП

З стрімко зростаючою кількістю вбудованих систем гостро постає питання захисту даних, що на них зберігаються. Оскільки абсолютна безпека не можлива, то деяка система може бути скомпрометована, або ж узагалі не передбачати належного захисту. В такому випадку важливо говорити про модель атак, коли зловмисник перебуває безпосередньо в системі та може спостерігати за процесами і контролювати їх.

Криптоалгоритми білої скриньки теоретично дозволяють забезпечити захист у такій моделі. Це відбувається за рахунок того, що ключ вбудовується усередину коду та ніколи не з’являється в пам’яті у відкритому чи зашифрованому вигляді. Найпростішою до застосування атакою на такі криптоалгоритми є атака диференціального обчислювального аналізу. Вона не вимагає ні знань про внутрішню будову криптоалгоритму, ні досвіду у зворотній розробці. Основним кроком атаки є зняття трас під час виконання програми. Методів зняти траси є два: інструментування програми та інструментування емулятора, у якому програма буде виконуватись.

У той час як існуючі інструменти в основному фокусуються на першому методі, у роботі буде також досліджено другий, оскільки він є зручнішим для спрощення проведення атаки і оцінки захищеності вбудованих системах.

Актуальність роботи обумовлюється актуальністю питання захисту конфіденційних даних у недовіреному середовищі виконання у вбудованих системах.

Мета роботи – удосконалення методів диференціального обчислювального криптоаналізу для можливості проведення атаки на архітектурах AMD64, ARM, MIPS вбудованих систем.

Завдання роботи:

- аналіз існуючих методів проведення диференціального обчислювального аналізу,

- вдосконалення програмної моделі диференціального обчислювального аналізу з метою розширення підтримуваних архітектур,
- експериментальне дослідження вдосконаленого методу для застосунків архітектур AMD64, ARM, MIPS.

Об’єкт дослідження – диференціальний обчислювальний криптоаналіз в вбудованих системах.

Предмет дослідження – методи проведення атаки диференціального обчислювального криптоаналізу вбудованих систем.

Методи дослідження – системний аналіз, методи об’єктно-орієнтованого програмування.

Наукова новизна – вдосконалення методу диференціального обчислювального аналізу для архітектур AMD64, ARM, MIPS вбудованих систем.

Практичне застосування – створена програмна реалізація, що дозволяє проводити аналіз прошивок для пристроїв архітектур AMD64, ARM, MIPS.

1 ЗАСТОСУВАННЯ АТАКИ ДИФЕРЕНЦІАЛЬНОГО ОБЧИСЛЮВАЛЬНОГО АНАЛІЗУ У КРИПТОГРАФІЇ БІЛОЇ СКРИНЬКИ

1.1 Криптографія білої скриньки та сфери її застосування

Стандартна модель атаки на систему шифрування і досі часто розглядається як чорна скринька - тобто зловмисник має доступ до входів, виходів системи, але не володіє жодними деталями та знаннями про її внутрішню роботу. Ще одна існуюча модель - сірої скриньки, у ній атакуючий частково володіє деталями про роботу системи чи може спостерігати за її характеристиками під час роботи (такими як енергоспоживання). Атаки в межах цієї моделі ще називають атаками по сторонніх каналах.

У роботі 2002 року [1] зазначено, що лише цих двох варіантів недостатньо, адже у сучасному світі часто середовища, у яких відбуваються криптографічні операції, можуть бути незахищеними або скомпрометованими. В такому випадку зловмисник може не тільки знати використовуваний криптоалгоритм, а й мати доступ до внутрішніх станів системи. Це означає: якщо ключ у той чи інший момент часу з’явиться у відкритому вигляді у пам’яті - зловмисник може його прочитати, і навіть найскладніша криптографічна система у такому випадку стає безсильною. Саме тому автори вводять поняття криптографії білої скриньки (white-box cryptography), яка мала б забезпечити безпеку навіть у незахищених середовищах.

Криптографія білої скриньки передбачає, що секретний ключ ніколи не буде знаходитись у відкритому вигляді у пам’яті. Для цього автори пропонують вбудовувати секретний ключ усередину криптоалгоритму та як приклад пропонують та пояснюють можливий підхід до побудови white-box імплементації AES (у тому ж 2002 році вони також запропонували алгоритм для побудови DES [2]). Ідея полягає у заміні ключових обчислень криптоалгоритму на операції отримання потрібних байтів з таблиць, які генеруються одноразово перед

використанням програми та залежать від секретного ключа. Важливо, що таблиці повинні генеруватись не на цільовій системі, де будуть використовуватись, оскільки при генерації ключ перебуватиме у відкритому вигляді у пам’яті, що фактично руйнує усю ідею криптографії білої скриньки.

Використовувати криптографію білої скриньки варто не завжди. Часто зловмиснику достатньо можливості шифрувати і розшифровувати дані, і відновлення секретного ключа не є основною метою. Тому white-box криптоалгоритми мають свої основні сфери застосування [3]:

- технології керування цифровими правами (digital rights management),
- системи мобільних платежів.

Технології керування цифровими правами (DRM) призначені для контролю доступу до контенту на пристрої користувача. Зазвичай зашифрований контент разом із програмою та ліцензією надсилається на пристрій. Для отримання доступу до даних користувач надає запит програмі, яка опрацьовує ліцензію і робить висновок про те, чи дозволено операцію. Якщо так, контент розшифровується самою програмою. При використанні звичайного криптографічного алгоритму, користувач зможе відновити ключ та за бажання поділитись ним із іншими. Завдяки криптографії білої скриньки ключ залишається захищеним, і доступ до контенту отримує лише власник пристрою.

У системах мобільних платежів усі комунікації та конфіденційні дані шифруються. Щоб не допустити викриття ключа та зловживання ним, може використовуватись криптографія білої скриньки. Тоді навіть перебуваючи у системі, де відбувається шифрування, (наприклад, у формі шкідливого програмного забезпечення) зловмисник не зможе відновити секретний ключ.

У обох зазначених вище системах при використанні криптографії білої скриньки є спільна проблема - якщо програму повністю перенести на іншу систему, вона продовжить свою роботу і видаватиметься легітимною, адже для перенесення атакуючому не потрібен секретний ключ. Тому у [3] автори зазначають: “white-box програма повинна забезпечувати не лише неможливість

відновлення ключа, ключовою властивістю повинна бути також протидія атакам перенесення коду”.

Автори [4] вважають, що одним з рішень, яке може забезпечити захист від атаки перенесення коду, є використання зовнішнього кодування (external encoding). Розглядаючи криптографічну систему (E) як частину деякої більшої системи, вони пропонують обчислювати:

$$E' = G \circ E \circ F^{-1}$$

де F та G – вхідне та вихідне кодування відповідно. Під кодуванням розуміється деяка випадково обрана бієкція, що не дозволяє зловмиснику відновити E з E`.

У статті [5] навпаки зазначається, що варто одразу розглядати найгірший випадок, а саме - атакуючий завжди зможе перемістити програму на іншу систему, хоч це і може вимагати значних зусиль, тому варто зосередити увагу на інших властивостях криптоалгоритмів при їх побудові: односторонність (one-wayness), неможливість стиснення (incompressibility), відстежуваність (traceability). Це може значно обмежити дії атакуючого навіть у випадку успішної атаки перенесення коду.

Односторонність передбачає використання програми лише для шифрування або розшифрування. Неможливість стиснення повинна забезпечити менш зручне зберігання та обмін програмою. Для забезпечення відстежуваності алгоритм генерації імплементації криптоалгоритму повинен створювати декілька різних варіацій того ж шифрування (чи розшифрування), щоб розповсюдити їх між користувачами. В випадку перенесення програми на іншу систему можна буде визначити її джерело.

Відсутність однозначного підходу можна пояснити відносно невеликою кількістю досліджень у сфері криптографії білої скриньки, а також тим, що поки не вдалось розробити алгоритм безпечної імплементації шифру. На поточний момент, якщо криптографія білої скриньки десь і використовується, то вона працює через принцип “безпека через неясність” (security through obscurity). Використовуючи криптоалгоритм білої скриньки у комерційних проєктах,

компанії не розголошують деталі імплементації, а також періодично змінюють секретний ключ, щоб зменшити наслідки від можливої атаки.

1.2 Огляд атаки диференціального обчислювального аналізу

Головним кандидатом на створення white-box імплементацій очікувано є американський стандарт симетричного шифрування Advanced Encryption Standard [6]. Перша імплементація була запропонована одночасно із введенням терміну “криптографія білої скриньки”, але уже через два роки було доведено, що секретний ключ можна відновити з часовою складністю 2^{30} (у найгіршому випадку) [7]. Поки що рано стверджувати, що будь-які з наступних запропонованих імплементацій забезпечують достатній рівень безпеки, оскільки більшість із них або уже були зламані, або є відносно новими і недостатньо дослідженими.

Усі атаки на white-box криптографію можна поділити на три категорії [8]:

- статистичні атаки,
- атаки, що спираються на перетворення Фур’є,
- атаки, що спираються на атаки по сторонніх каналах.

Атака диференціального обчислювального аналізу (differential computation analysis – DCA) відноситься до третьої категорії та є програмним аналогом атаки диференціального аналізу енергоспоживання (differential power analysis – DPA) [9]. На відміну від DPA, DCA має більшу ефективність, оскільки отримує дані без шуму у вимірюваннях.

DCA дозволяє без зворотної розробки чи особливих знань про використовувані у імплементації таблиці відновити (частково чи повністю) секретний ключ. Вперше атака була представлена у 2015 році у [4].

Для успішної виконання атаки зловмисник повинен мати змогу неодноразово виконувати програму та мати доступ до відкритих та (або) зашифрованих текстів. Далі атака передбачає такі кроки:

1. Визначення криптоалгоритму. У [4] спершу пропонується зняти траси під час виконання програми або її частини для одного вибраного (чи випадкового) відкритого тексту. Цю дію можна виконати двома шляхами: інструментуючи виконуваний файл або інструментуючи емулятор, що виконує програму. В той час як у статті розглядається перший варіант, у роботі буде також застосований другий. Суть інструментування у додаванні деяких інструкцій під час виконання, щоб відстежити певні події, якими у даному випадку є: адреси пам’яті, до яких програма робить запити (для читання, запису чи виконання) і їх вміст. Далі автори пропонують візуалізувати траси, щоб визначити використовуваний шифр, його конфігурацію, кількість раундів.
2. Зняття (деякої кількості) трас під час виконання програми чи її частини. Коли було визначено криптоалгоритм, потрібно зняти деяку кількість трас для відновлення ключа. Ця кількість буде відрізнятись в залежності від криптоалгоритму. Чим складніша імплементація, тим більше трас може знадобитись.
3. Перетворення трас у формат, який очікують DPA утиліти (тобто у вектори з “0” і “1”). Перевагою такого алгоритму є те, що відсутня необхідність розробляти нові програми для аналізу та відновлення ключа, оскільки неважкі перетворення дозволяють звести траси у необхідний формат для уже існуючих DPA утиліт.
4. Використання DPA утиліт для відновлення ключа. У результаті ключ може бути відновлено не повністю, це залежить від імплементації, кількості трас. Якщо після атаки не вистачає декількох байт, то їх можна спробувати відновити перебором.

Використання DPA утиліт для відновлення ключа (останній крок у DCA) передбачає виконання техніки кореляційного аналізу, що дозволяє відновити ключ за відібраними даними. Одним з основних кроків даного аналізу є обчислення коефіцієнта кореляції. Хоча існують і інші коефіцієнти кореляції, найчастіше у

атаках обчислюють коефіцієнт Пірсона (він використовується також у інструменті, який буде використано у роботі).

Хоча запропоновані у [4] інструменти теоретично підтримують широкий набір архітектур для аналізу, повністю протестовані були лише x86 та x86-64, що піднімає питання про можливість здійснення атаки на інших архітектурах. З великою кількістю вбудованих систем та ймовірним застосуванням white-box криптоалгоритмів на них, стає актуальним дослідження атаки у даному середовищі.

1.3 Огляд методів проведення диференціального обчислювального аналізу

У другому розділі буде розглянуто інструменти для проведення DCA у порядку, якому вони мають використовуватись. Ці інструменти передбачають:

1) інструментування програми, що дозволить логувати потрібну інформацію: про виклики функцій, потоки, інструкції, що пишуть чи читають з пам’яті та інше,
2) зняття трас,
3) застосування кореляційного аналізу для відновлення ключа,
4) (у разі потреби) перебір байтів ключа, що не були відновлені.

Останні два пункти повністю повторюють кроки атаки DPA, що дозволяє використовувати уже написані утиліти для DPA і спрощує проведення атаки DCA. Перші два кроки відрізняються: при виконанні атаки DPA зловмисник збирає траси енергоспоживання пристрою за допомогою фізичних маніпуляцій, а при DCA збирається інформація за допомогою додавання коду у програму чи емулятор, тому на виході немає зайвого шуму і затримок.

Для двох методів проведення DCA (інструментування програми та емулятора) останні два пункти теж є незмінними, але перші два виконуються по-різному. Один метод передбачає, що всередину програми спеціальні утиліти

додають інструкції, що перехоплюють та логують деякі визначені події. Інший метод, на якому буде зосереджена робота, полягає у інструментуванні емулятора програми.

Емуляція програм здійснюватиметься з допомогою Qiling Framework – багатоплатформного та багатоархітектурного фреймворку [10], написаного на основі емулятора процесора Unicorn [11]. Його особливості:

- підтримка платформ Linux, Windows, MacOS, BSD, UEFI, DOS,
- підтримка архітектур Arm, Arm64, X86, X86_64, MIPS, 8086,
- підтримка форматів: PE, ELF, MachO, COM,
- забезпечення високорівневого інтерфейсу для налаштувань пісочниці,
- можливість перехоплення інструкцій, адрес, базових блоків, переривань (interrupts), запитів на читання чи запис у пам’ять,
- можливість зміни коду під час виконання програми.

Фреймворк об’єднує ідею емулятора та пісочниці. Як пісочниця Qiling будує середовище без будь-якої необхідності завантаження додаткових файлів. Виконувані файли можуть взаємодіяти з зовнішнім світом через системні виклики чи Windows Api (в залежності від типу файлу). Фреймворк відповідає за усі комунікації під час емуляції, що дозволяє спостерігати за процесом виконання та контролювати його.

Перевагою Qiling є його гнучкість, будь-який системний виклик чи адресу можна власноруч перехопити, за бажанням обробити, інструкції програми та значення регістрів можна змінити на ходу,

Незважаючи на це, Qiling відносно новий фреймворк, тому вимагає доопрацювання. Значна кількість системних викликів поки не імплементована, що ускладнює емуляцію складних програм, але він добре працює на невеликих з 1-2 функціональностями, тому підходить для аналізу обраної white-box імплементації в межах цієї роботи.

1.4 Методи захисту від атаки диференціального обчислювального аналізу

Для захисту від атак DPA часто допомагає використання маскування (masking) чи перемішування (shuffling), що забезпечується деяким зовнішнім генератором випадкових чисел. У даному вигляді для захисту від DCA такі методи використовувати ризиковано, адже зловмисник за замовчуванням має доступ до системи і може відключити/замінити генератор, щоб він подавав лише потрібні значення.

Щоб зробити відключення генератора складнішим, автори [12] пропонують замінити зовнішній генератор випадкових чисел на внутрішній псевдогенератор (усередині криптографічної системи), та додати до нього декілька шарів обфускації. Тоді зловмиснику знадобиться набагато більше сил і досвід у зворотній розробці, щоб обійти ці заходи протидії. Детально описані методи для збільшення рівня захищеності white-box криптоалгоритму та протидії атаці DCA, що походять від методів протидії DPA, описані у [12], [13].

Оскільки DCA призначений саме для простого та легкого відновлення ключа без знань криптоалгоритму та додаткових зусиль, то такі методи можуть значно ускладнити атаку та зменшити інтерес зловмисника. За рахунок використання техніки “безпека через неясність”, періодичної зміни ключа та згаданих вище методів, криптографія відкритої скриньки може відносно безпечно використовуватись у комерційних проєктах.

Висновки до розділу 1

У першому розділі було розглянуто поняття криптографії білої скриньки, її сфери застосування, атаку DCA та кроки, необхідні для її проведення. Проаналізовано два методи здійснення атаки (трасування програми та трасування емулятора) та їх відмінності. Окрім того, було згадано методи протидії даній атаці.

За результатами першого розділу визначено, що незважаючи на потенціал використання криптографії білої скриньки у вбудованих системах, не існує практичної реалізації методів досліджень, що б дозволяли здійснювати атаку на архітектурах, окрім x86 та x86-64. Це зумовлює актуальність обраної теми дипломної роботи.

2 УДОСКОНАЛЕННЯ МЕТОДУ ДИФЕРЕНЦІАЛЬНОГО ОБЧИСЛЮВАЛЬНОГО КРИПТОАНАЛІЗУ ДЛЯ ВБУДОВАНИХ СИСТЕМ

2.1 Плагіни для зняття трас

Для здійснення атаки диференціального обчислювального аналізу, описаної в [4], автори розробили декілька платформ з плагінами та утилітами. Перша з них - Tracer [14]. Вона складається з трьох частин - TracerPIN, TracerGrind та TracerGraph.

TracerPIN. Плагін на основі Intel PIN [15], дозволяє генерувати траси виконання програми на архітектурах, що підтримуються Intel PIN (але був протестований лише на x86 та x86-64), може трасувати всю програму чи лише обраний діапазон адрес (корисно, якщо програма велика і шифрування є лише однією з функціональностей). Може логувати:

- [C] – виклики функцій,
- [B] – базові блоки,
- [I] – інструкції,
- [R] – читання з пам’яті,
- [W] – запис в пам’ять,
- [T] – операції з потоками,
- [!] – інформацію про відфільтровані елементи,
- [-] – інформацію про бібліотеки,
- [*] – аргументи.

На Рисунку 2.1 зображено приклад частини трас, згенерованих командою “Tracer -o ls.log -- ls” (з файлу ls.log). При зберіганні інформації про виклики функцій, зберігається також їх адреса в пам’яті, із запитами до пам’яті зберігається адреса інструкції, що виконала запит, адреса у пам’яті, кількість байт, значення, що має бути записане/прочитане.

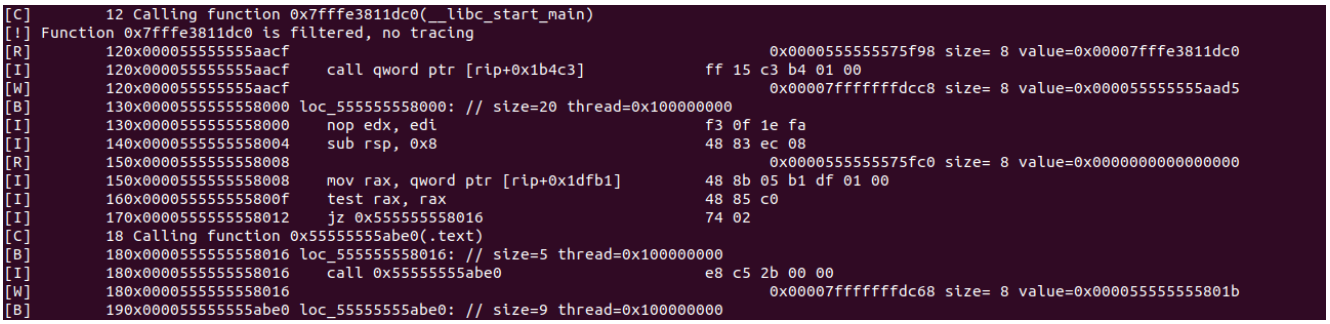


Рисунок 2.1 – Частина виводу програми TracerPIN, запущеної для трасування програми ls (що виконує перегляд вмісту каталогу)

TracerGrind. Плагін на основі Valgrind [16], протестовані архітектури x86, x86-64, ARM, але теоретично підтримує усі ті, що підтримуються Valgrind, також може трасувати усю програму чи її частину, логує:

- [T] – інформацію про потоки,
- [M] – доступи до пам’яті,
- [L] – завантаження бібліотек у пам’ять,
- [I] – інструкції,
- [B] – базові блоки,
- [!] – додаткову інформацію.

Приклад виводу на Рисунку 2.2 теж показує, що зберігається основна інформація про запити, як адреса інструкції, адреса у пам’яті, кількість байт, значення. Саме ці дані будуть використані для обчислення кореляцій.

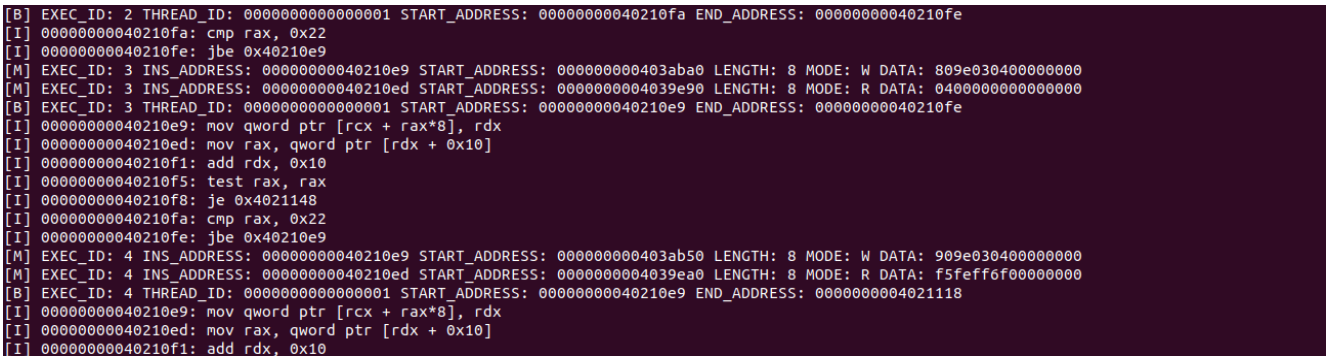


Рисунок 2.2 – Частина виводу програми TracerGrind, запущеної для трасування програми ls (що виконує перегляд вмісту каталогу)

TracerGraph. Утиліта для візуалізації трас, отриманих з допомогою TracerPIN та TracerGrind. Вона використовує SQLite базу даних, згенеровану цими плагінами, і відкриває вікно з графіком (Рисунок 2.3), на якому позначено:

- чорні блоки - виконані інструкції,
- зелені блоки - читання з пам’яті,
- червоні блоки - запис в пам’ять,
- оранжеві блоки - запис і читання.

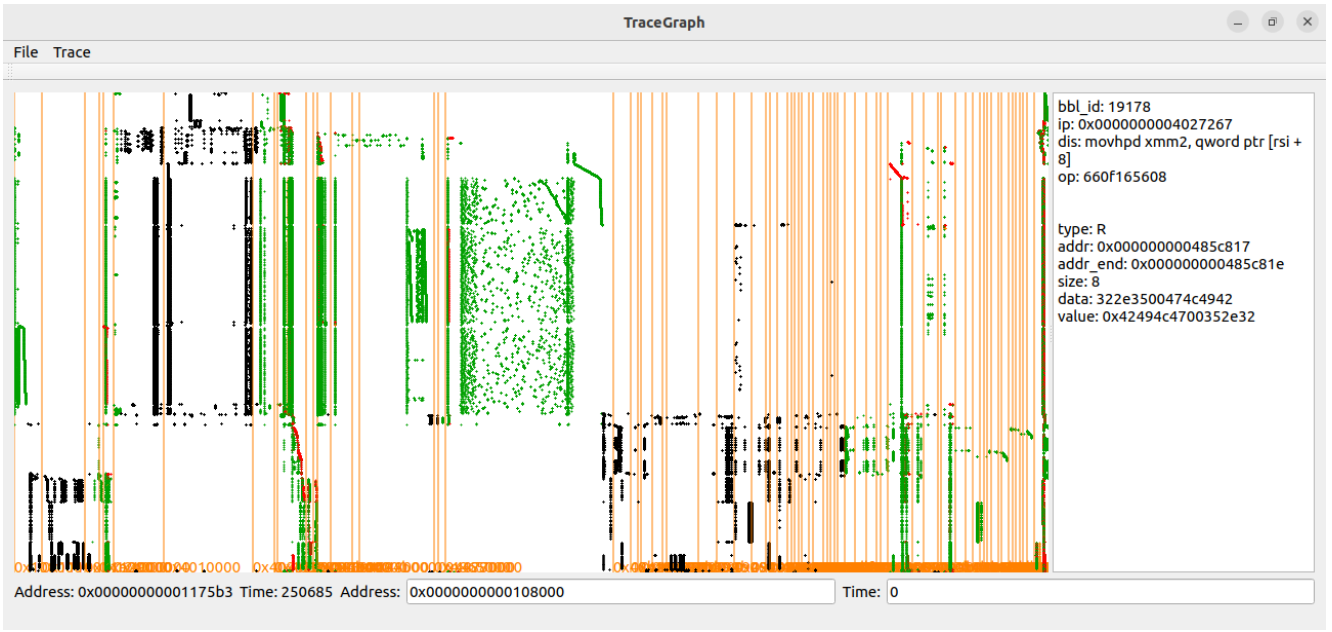


Рисунок 2.3 – Візуалізація трас програми ls, отриманих з допомогою TracerGrind

Вісь ординат відповідає за час (початок виконання - зверху), вісь абсцис – за пам’ять (менші адреси лівіше). Є можливість переходу за заданою адресою та певною міткою часу. У правій частині вікна видно інформацію про вибраний блок (для вибору потрібно натиснути лівою кнопкою миші на один з блоків будь-якого кольору), таку як адреса інструкції, адреса у пам’яті для прочитання байт (якщо зелений блок) чи для запису (червоний), довжина даних, значення та ін.

Задля виконання завдання дипломної роботи до платформи Tracer, окрім TracerPIN та TracerGrind, було написано та додано ще один трасер – *TracerQiling* (Додаток А), який, на відміну від двох інших, інструментує не виконувану програму, а емулятор, який її виконує. Даний трасер є скриптом на Python, що використовує Qiling Framework для емуляції заданих виконуваних програм. Окрім іншого, фреймворк може перехоплювати адреси, системні виклики, запити до пам’яті, що дозволяє логувати події, ідентичні Tracer PIN та TracerGrind.

Емуляція програм відбувається за допомогою створення об’єкта типу Qiling, аргументами є шлях до виконуваної програми, шлях до кореневої файлової

системи (який залежить від потрібної архітектури та операційної системи) та інші необов'язкові опції, як рівень деталізації (verbose) та багатопотоковість. За допомогою перехоплення запитів до пам'яті на запис та (або) читання відбувається логування потрібних даних.

Перехоплення запитів до пам'яті відбувається з допомогою функцій ql.hook_mem_write та ql.hook_mem_read (фреймворк також може обробляти запити некоректні запити до пам'яті з допомогою ql.hook_mem_read_invalid та ql.hook_mem_write_invalid, але у даному випадку вони не використовуються). Єдиним аргументом функцій є деяка інша функція, що буде обробляти ці запити. На Рисунку 2.4 зображено приклад такої функції (яка справді використовується у TracerQiling) для обробки ql.hook_mem_read.

Спершу у функції перевіряється, чи справді був запит на читання з пам'яті. Оскільки перехоплення відбувається перед самим запитом, то у аргументі value пусто, а щоб отримати доступ до прочитаних даних, їх можна прочитати вручну з допомогою ql.mem.read. Додатково перевіряється порядок байтів у даній архітектурі, щоб коректно прочитати байти. Остання дія у функції – додавання події у загальний масив у правильному форматі.

```
12  def mem_read(ql: Qiling, access: int, address: int, size: int, value: int) -> None:
13      # only read accesses are expected here
14      assert access == UC_MEM_READ
15
16      if ql.archendian==QL_ENDIAN.EL:
17          value = int.from_bytes(ql.mem.read(address, size), "little")
18      else:
19          value = int.from_bytes(ql.mem.read(address, size), "big")
20
21      traces.append(f'[R] {address:016x} size= {size:d} value={value:016x}\n')
```

Рисунок 2.4 – Код функції mem_read, що обробляє обробляє перехоплений з допомогою ql.hook_mem_read запит та читання байтів з пам'яті

TracerQiling теоретично підтримує архітектури та платформи, які підтримує Qiling Framework, був протестований на AMD64, ARM, MIPS, може трасувати усю програму чи її частину. Запуск скрипта відбувається з командного рядка з такими аргументами:

- “-t” – шлях до програми, яка повинна емулюватись (щоб уникнути помилок, рекомендується вводити повний шлях, починаючи з кореневої папки),
- “-i” – вхідний аргумент для емульованої програми (наприклад, програма приймає відкритий текст, тоді тут його потрібно вказати тут),
- “-of” – файл для трас,
- “-f” – діапазон адрес для трасування,
- “-r” – шлях до кореневої системи (зазвичай знаходиться у папці rootfs/)

Обов’язковими аргументами є шлях до програми, що буде емулюватись, файл для зберігання трас та шлях до кореневої системи.

Траси зберігаються у заданий при запуску файл, їх формат схожий на вивід TracerPIN (Рисунок 2.5) і складається з:

- [W] або [R] для позначення запиту на запис/читання з пам’яті,
- адреси, до якої здійснено запит,
- розміру байт для запису/прочитання,
- прочитаних/записаних байтів.

```
[W] 000000007ff3cc12 size= 1 value=000000000000005f
[W] 000000007ff3cc13 size= 1 value=0000000000000023
[R] 000000007ff3cbc4 size= 4 value=0000000000000000
[R] 000000007ff3cc04 size= 1 value=0000000000000076
[W] 000000007ff3cccc size= 1 value=0000000000000076
[R] 000000007ff3cbc8 size= 4 value=0000000000000005
[R] 000000007ff3cc09 size= 1 value=00000000000000ee
[W] 000000007ff3cccd size= 1 value=00000000000000ee
[R] 000000007ff3cbcc size= 4 value=000000000000000a
[R] 000000007ff3cc0e size= 1 value=00000000000000ad
[W] 000000007ff3ccce size= 1 value=00000000000000ad
[R] 000000007ff3cbd0 size= 4 value=000000000000000f
[R] 000000007ff3cc13 size= 1 value=0000000000000023
```

Рисунок 2.5 – Частина виводу програми TracerQiling

Поки що до TracerQiling не додано можливість вносити траси у базу даних, щоб у подальшому візуалізувати їх з TracerGraph.

2.2 Інструмент для автоматизації збору трас

Deadpool [17] - платформа, що дозволяє автоматизувати атаки диференціального обчислювального аналізу та диференціального аналізу помилок (differential fault analysis – DFA). Друга атака не входить в межі обраного дослідження, тому далі буде розглядатись лише DCA.

Deadpool дозволяє автоматизувати процес збору трас з допомогою раніше згаданих трасерів, та перетворити траси у необхідний формат для відновлення ключа. По суті платформа є обгорткою до вищезгаданої платформи Tracer, оскільки автоматизує виклики трасерів, дозволяє за один запуск зняти потрібну кількість трас і обробити їх для подальшого аналізу.

У основі програми - клас Tracer, та його нащадки TracerPIN та TracerGrind. Їх конструктори приймають шлях до виконуваного файлу, обгортки для виводу і вводу для коректної роботи з програмою шифрування, та інші необов’язкові конфігурації, як діапазон адрес для трасування, архітектуру та фільтри.

Під фільтрами розуміються об’єкти класу Filter, що зберігають інформацію для фільтрації знятих трас. За замовчуванням створено 7 фільтрів. Їхні назви та описи наведені у Таблиці 2.1.

Наприклад, застосувавши фільтр stack_w1, з кожного трасу буде відібрано лише рядки, що логували інформацію про запис у пам’ять, у межах стеку, і розмір записаних байт був 1. У результаті фільтрації для подальшого аналізу будуть відібрані записані дані (у розмірі 1 байт). Саме вони будуть використовуватись для знаходження кореляцій і відновлення ключа.

Необхідний фільтр можна визначити безпосередньо протестувавши його на обраній імплементації. В той час як один фільтр може відновити 1-2 байти ключа, інший відновить усі 16. Тому слід уважно розглянути усі варіанти.

Таблиця 2.1 – Фільтри створені у Deadpool за замовчуванням та їх описи

Назва	Запити до пам'яті, що логуються (R - читання, W - запис)	К-сть прочитаних/ записаних байт	Операція в межах стеку	Які дані відбирає для аналізу
stack_w1	W	1	так	записані байти
stack_w4	W	4	так	записані байти
mem_addr1_rw1	R, W	1	ні	останній байт адреси
mem_addr1_rw4	R, W	4	ні	останній байт адреси
mem_addr2_rw1	R, W	1	ні	останній 2 байти адреси
mem_data_rw1	R, W	1	ні	прочитані/записані байти
mem_data_rw4	R, W	4	ні	прочитані/записані байти

За аналогією до класів-нащадків TracerPIN та TracerGrind для досліджень в межах дипломної роботи було додано клас TracerQiling (Додаток Б). Він використовує написаний раніше TracerQiling (програму-емулятор для трасування) та не сильно відрізняється від інших класів-нащадків Tracer. Його перевага - можливість роботи з архітектурами AMD64, ARM та MIPS.

Найважливішими аргументами конструктору класу є:

- target – ціль для емуляції,
- processinput – функція, що має коректно обробляти дані для подачі в ціль, вони будуть приєднані до команди-виклику трасера (наприклад, якщо ціль приймає на вхід рядок з 16 байт, то функція повинна повертати 16 байт, якщо ціль приймає назву файлу після зазначення опції "-i", то processinput повинен повернути масив ["-i", filename])

- processoutput – функція для обробки виводу з цілі, очікуване значення для повернення - ціле число: int(key, 16),
- arch – цільова архітектура,
- blocksize – розмір блоку для шифрування,
- addr_range – діапазон адрес, траси будуть зібрані тільки в цьому діапазоні,
- filters – застосовані фільтри (один або декілька), можуть бути вибрані з готових чи створені нові.

Клас TracerQiling коректно працює з усіма фільтрами, що задані за замовчуванням. З допомогою регулярного виразу *r'\[(.)\] *([0-9a-fA-F]+) *size= *([0-9]+) *value= *(.*)'* він відбирає необхідні дані з трас для подальшої обробки.

Окрім згаданих класів, платформа реалізує важливу функція bin2daredevil, що збирає усі зібрані траси, їхні відкриті тексти і шифротексти в три окремі файли (Рисунок 2.6). Потім функція створює конфігураційний файл для запуску Daredevil, який уже відновлює ключ чи його частину за допомогою атаки кореляційного аналізу.

```
with open('%s_%i_%i.trace' % (keyword, ntraces, nsamples), 'wb') as filetrace,\
     open('%s_%i_%i.input' % (keyword, ntraces, nsamples), 'wb') as fileinput,\
     open('%s_%i_%i.output' % (keyword, ntraces, nsamples), 'wb') as fileoutput:
    for filename, (iblock, oblock) in traces_meta.iteritems():
        if iblock_available:
            fileinput.write(iblock.decode('hex'))
        if oblock_available:
            fileoutput.write(oblock.decode('hex'))
        with open(filename, 'rb') as trace:
            filetrace.write(serializechars(trace.read(min_size)))
        if delete_bin:
            os.remove(filename)
```

Рисунок 2.6 – Частина коду функції bin2daredevil.

Daredevil - інструмент з відкритим кодом, який буде використовуватись у даній роботі. Окрім нього, існує також Riscure Inspector [18] - комерційний проєкт. Якщо є бажання його використати, Deadpool може перетворювати траси у потрібний формат з допомогою функції bin2trs.

На Рисунку 2.7 показано найпростіший приклад використання класу TracerQiling. У результаті буде запущено трасер з виконуваних файлом,

скомпільованим під архітектуру AMD64, для збиранням 200 трас та перетворення їх у правильний формат для Daredevil. До трас буде застосовано фільтр mem_data_rw4 (будуть обрані лише запити з запитами на читання/запис в пам'ять розміром 4 байти, для обрахування кореляції будуть використовуватись записані/прочитані байти) та діапазон адрес для обмеження загального розміру трас до тих, які залежать від секретного ключа.

```
t = TracerQiling(path_to_executable, qprocessinput, \
    processoutput, ARCH.amd64, 16, addr_range="0x401AD5-0x4028A7",\
    filters=[DefaultFilters.mem_data_rw4])

t.run(200)

bin2daredevil(keywords=[DefaultFilters.mem_data_rw4],
              configs={'attack_sbox':   {'algorithm':'AES', 'position':'L
                       'attack_multinv':{'algorithm':'AES', 'position':'L
```

Рисунок 2.7 – Найпростіший приклад використання класу TracerQiling для автоматизації збору трас та перетворення їх у правильний формат

В результаті після виконання програми у директорії з'являться нові файли:

- *.input - з усіма відкритими текстами,
- *.output - з шифротекстами,
- *.trace - з трасами, зібраними в один файл,
- *.config - по одному файлу конфігурації на кожну атаку.

У Deadpool є приклади застосування DCA та DFA на доволі великій кількості різних white-box імплементацій AES. За рахунок схожості використання класів трасерів, приклади неважко перенести на TracerQiling.

2.3 Інструмент для аналізу трас

Daredevil [19] – утиліта для здійснення атаки кореляційного аналізу. Вона дозволяє обчислити кореляцію між заданими трасами та усіма ймовірними варіантами ключа.

Як аргумент даний інструмент приймає конфігураційний файл. Серед іншого у ньому зазначаються (всього налаштувань дуже багато, їх можна переглянути у [19]):

- конфігурації трас:
- files: кількість файлів, що містять траси,
- trace_type: тип трас, один з 32 bit float/64 bit float/8 bit integer/unsigned 8 bit integer,
- nsamples: кількість точок (даних) у кожному трасі,
- trace: шлях до файлу, кількість рядків і стовпців,
- конфігурації для створення припущень щодо байтів ключа:
- files: кількість файлів, що містять відкриті тексти (якщо атакуватись буде перший раунд) чи шифротексти (якщо атакується останній раунд),
- guess_type: тип припущень (стандартно - байти - unsigned 8 bit integer),
- guess: шлях до файлу, кількість рядків і стовпців,
- загальні конфігурації:
- threads: кількість потоків для обчислень,
- order: порядок атаки,
- return_type: тип, який буде повернено (наприклад, double),
- algorithm: криптографічний алгоритм (AES, DES),
- position: позиція, на яку буде проведено атаку,
- correct_key: правильний ключ - зазначати не обов'язково.

Позиція для атаки повинна бути вказана як шлях до відповідної таблиці, наприклад:

- LUT/AES_AFTER_SBOX - атака першого раунду шифрування (у guess – шлях до відкритих текстів)
- LUT/AES_AFTER_SBOXINV - атака останнього раунду шифрування (у guess – шлях до шифротекстів). В результаті цієї атаки буде знайдено раундовий ключ, тому доведеться з нього відновлювати ключ AES.

Приклад конфігурації наведено на Рисунку 2.8.

```
[Traces]
files=1
trace_type=i
transpose=true
index=0
nsamples=9216
trace=mem_data_rw4_200_9216.trace 200 9216

[Guesses]
files=1
guess_type=u
transpose=true
guess=mem_data_rw4_200_9216.input 200 16

[General]
threads=8
order=1
return_type=double
algorithm=AES
position=LUT/AES_AFTER_SBOX
round=0
bitnum=all
bytenum=all
correct_key=0x0BE85AB47311B2E4AC18BC78607F76AA
```

Рисунок 2.8 – Приклад конфігурації для атаки першого раунду AES

У випадку проведення атаки DCA на платформі Deadpool, конфігураційний файл створюється автоматично під час виконання функції bin2daredevil.

Найпростіше запустити Daredevil командою “daredevil -c <шлях до конфігураційного файлу>”. В залежності від того, чи було вказано секретний ключ, вивід буде відрізнятись. Якщо ключ було вказано, для кожного байту буде вираховуватись порядковий номер (Рисунок 2.9, а), в іншому випадку рядок “Best bit…” буде відсутній, зате в кінці аналізу буде виведено найкращих кандидатів у секретний ключ за кожною характеристикою (Рисунок 2.9, б). Оцінюються дві характеристики: сума по абсолютних значеннях кореляцій усіх бітів байта, найвище абсолютне значення кореляції біта.

```
Best 10 candidates for key byte #4 according to sum(abs(bit_correlations)):
 1: 0x73  sum: 3.0374    <==
 2: 0x67  sum: 2.5074
 3: 0xca  sum: 2.41357
 4: 0x48  sum: 2.41012
 5: 0x17  sum: 2.39799
 6: 0x77  sum: 2.39354
 7: 0x8b  sum: 2.39239
 8: 0xf9  sum: 2.39215
 9: 0x7d  sum: 2.3919
10: 0xad  sum: 2.38878

Best 10 candidates for key byte #4 according to highest abs(bit_correlations):
 1: 0x73  peak: 1         <==
 2: 0x77  peak: 0.40008
 3: 0x80  peak: 0.378314
 4: 0x40  peak: 0.375423
 5: 0xe2  peak: 0.372639
 6: 0x55  peak: 0.369536
 7: 0x1b  peak: 0.36732
 8: 0x12  peak: 0.35952
 9: 0xc0  peak: 0.359322
10: 0xe4  peak: 0.353951

[INFO] Attack of byte number 4 done in 4.587134 seconds.
Best bit: 0 rank: 0.               1    0x73     256
```

(а)

```
Most probable key sum(abs):
1: 50.5668: ba141fb473a1b2e4ae18bc7860fc76aa
2: 50.5605: ba141fb4730fb2e4ae18bc7860fc76aa
3: 50.5532: ba141fb473a1b2e4ae18bc78606576aa
4: 50.5526: ba141fb473a1b2e4ae18bc78605b76aa
5: 50.5485: ba141fb473a1b2e4ae18bc7860a676aa
6: 50.5468: ba141fb4730fb2e4ae18bc78606576aa
7: 50.5463: ba141fb4730fb2e4ae18bc78605b76aa
8: 50.5421: ba141fb4730fb2e4ae18bc7860a676aa
9: 50.5355: ba141fb473a1b2e4ae18bc78603776aa
10: 50.5353: 93141fb473a1b2e4ae18bc7860fc76aa

Most probable key max(abs):
1: 12.3847: ba46f0b473f7b2e49718bc7860fc76aa
2: 12.3842: ba46f0b4730fb2e49718bc7860fc76aa
3: 12.3815: ba46f0b473f7b2e4ae18bc7860fc76aa
4: 12.3809: ba46f0b4730fb2e4ae18bc7860fc76aa
5: 12.3795: ba46f0b473f7b2e4a618bc7860fc76aa
6: 12.3789: ba46f0b4730fb2e4a618bc7860fc76aa
7: 12.3786: ba46f0b473f7b2e40c18bc7860fc76aa
8: 12.378: ba46f0b4730fb2e40c18bc7860fc76aa
9: 12.3754: ba4611b473f7b2e49718bc7860fc76aa
10: 12.3748: ba4611b4730fb2e49718bc7860fc76aa
```

(б)

Рисунок 2.9 – Приклад виводу утиліти Daredevil (а) – з відомим секретним ключем, (б) – без відомого секретного ключа

2.4 Візуалізація обчислених значень кореляцій

Для візуалізації значень, обчислених для знаходження найкращих кандидатів для байтів ключа, було внесені невеликі зміни в код Daredevil. Окрім

виводу на екран десяти найкращих значень ключа по кожній з характеристик, усі 256 значень будуть також виведені у файл (Рисунок 2.10).

```
for (int i = 1; i <= nbest; i++) {
  cout << setfill(' ') << setw(2) << i << ": 0x" << setfill('0') << setw(2) << hex << sum_bit_
  cout << setfill(' ') << dec << "  sum: " << setw(8) << left << sum_bit_corels.back()[n_keys-
  if (sum_bit_corels.back()[n_keys-i].key == correct_key)
    cout << "  <==";
  cout << endl;
}
for (int i = 1; i <= filebest; i++) {
  sumabs << setfill('0') << setw(2) << hex << sum_bit_corels.back()[n_keys-i].key;
  sumabs << ": " << setw(8) << left << sum_bit_corels.back()[n_keys-i].corr << right<<"\n";
}
```

Рисунок 2.10 – Доданий код (нижній цикл) у код Daredevil

Тоді у файл будуть записані значення у форматі <байт>:<обчислене значення>. З допомогою написаної на Python програми (Додаток В) можна візуалізувати обчислені кореляції. Невеликий цикл, що читає усі файли, де записані байти та sum(abs(bit_correlation)) зображено на Рисунку 2.11.

```
for bn in range(16):
    y = [0 for i in range(256)]
    with open("sum_abs_"+str(bn)+".txt") as f:
        te = f.readlines()
    for line in te:
        byte = int(line[:line.find(":")], 16)
        y[byte] = float(line[line.find(":")+2:])
    print(y)
    plt.plot(x, y, label = "byte_"+str(bn))
```

Рисунок 2.11 – Цикл для прочитання обчислених значень кореляцій та їх візуалізації на графіку

Бібліотека matplotlib дозволяє зображати декілька залежностей на одній системі координат. З допомогою plt.plot буде додано новий графік для кожного байта з 16 (кількість можна змінити). Змінна “x” у даному випадку містить масив з цілих значень від 0 до 255. Після виклику функції plt.show можна отримати графік, схожий на Рисунок 2.12.

У даному випадку помітно, що, наприклад, перший байт ключа має значення біля 110 (по графіку важко точно визначити), десятий байт десь у межах 250-255, а для другого байту не було знайдено хорошого кандидата.

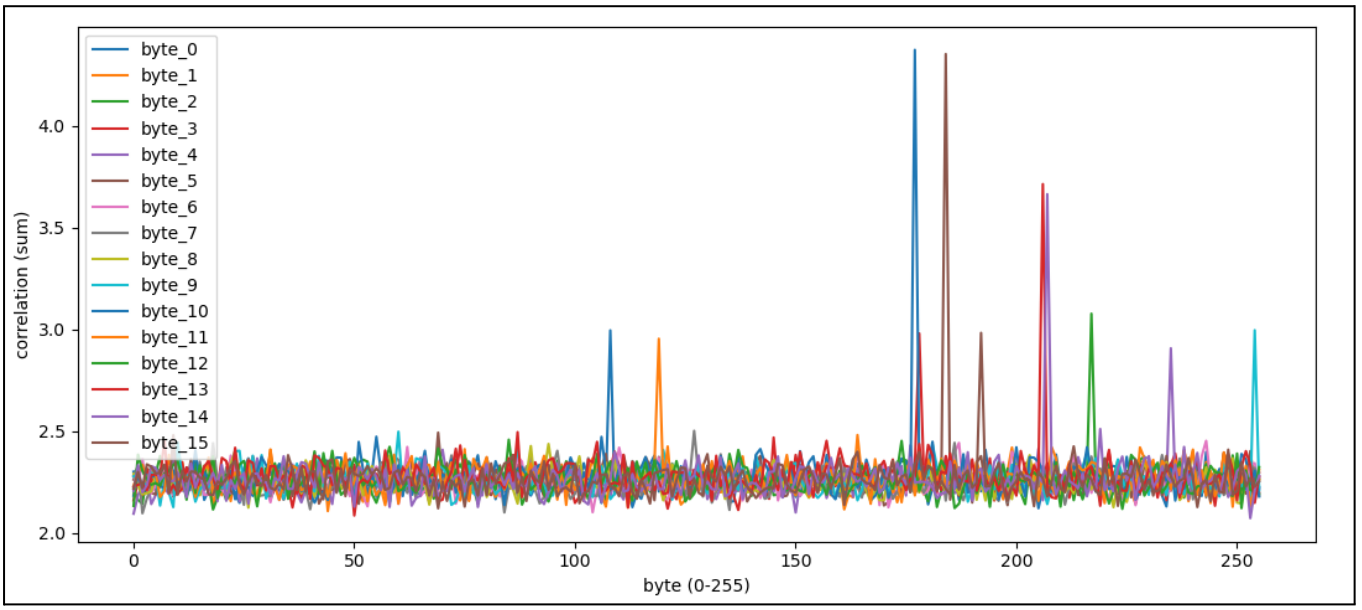
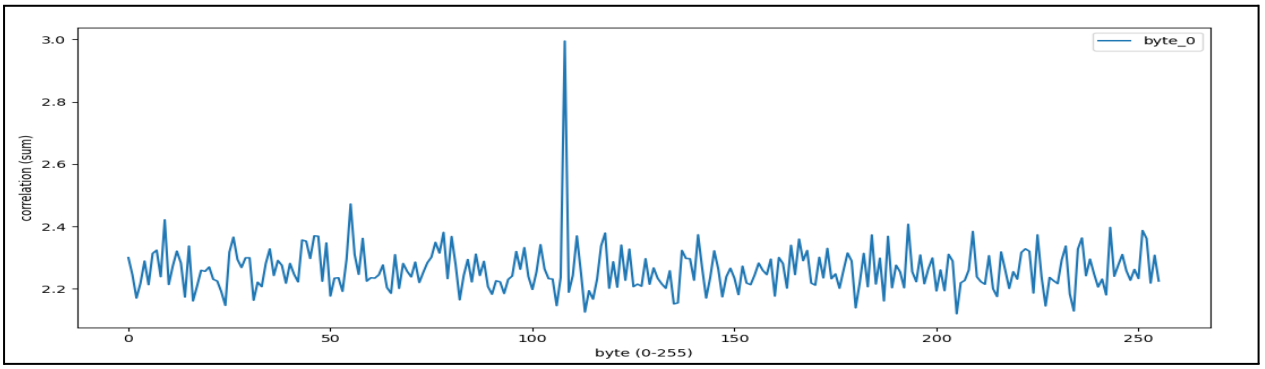
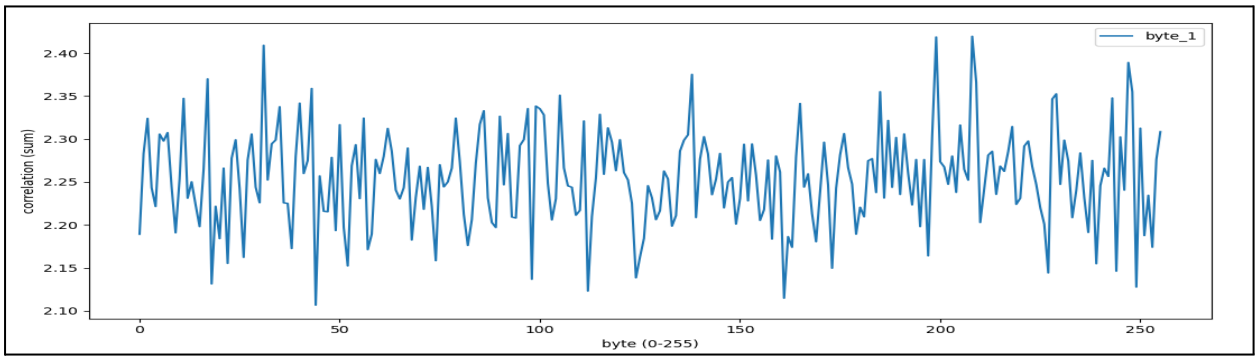


Рисунок 2.12 – Візуалізація обчислених значень кореляції між заданими даними та байтами ключа



(а)



(б)

Рисунок 2.13 – Побудовані значення кореляції для випадків з знайденим значенням байта (а) та без чітких кандидатів (б)

На Рисунку 2.12 деякі значення перетинаються, тому у випадку, якщо потрібна чітка картина, можна робити окремий графік для кожного байту ключа (Рисунок 2.13, (а) - байт вдалось відновити, (б) - кореляція слабка).

Як уже було зазначено, по графіку важко однозначно визначити значення потрібного байта, але він може бути корисним для огляду загальної картини та оцінки кількості відновлених байт. З допомогою нього можна визначити, чи була атака успішною, чи варто змінити конфігурацію (чи фільтр, чи імплементацію).

2.5 Інструмент для повного відновлення ключа у випадку декількох відсутніх байтів

Якщо у деяких байт ключа кореляції занадто слабкі та точно визначити кандидата неможливо, то в нагоді стане платформа Hulk [20]. Аргументи у виконуваного файлу прості: буква “e” чи “d” для позначення дії шифрування чи розшифрування відповідно, далі відкритий текст, шифротекст, ключ (невідомі байти позначені через “??”). З допомогою розширення системи команд AES автори домоглися швидкого відновлення байтів ключа (обчислює до 6 невідомих байтів).

Приклад виводу програми можна побачити на Рисунку 2.14. Спершу програма перелічує відомі значення входу, виходу, порядкові номери байт для відновлення, вказує значення, серед яких перебиратиме варіанти (00000000-FFFFFFFF), тоді запускає атаку. У даному випадку 4 байти ключа було відновлено і вивід програми співпадав з заданим.

```
Hulk v1.2 (Peter Garba 2018)
[*] AES-NI is supported by this CPU!
[*] Mode            : Decryption
[*] Key             : 3525AC10BB391D8D5F3914FE????????
[*] Input           : 00000000000000000000000000000000
[*] Expected        : ac589cfe8a7ae5dd875d9786e5832400
[!] Bruteforce      : 4 missing bytes
[*] Byte 0 Index 12
[*] Byte 1 Index 13
[*] Byte 2 Index 14
[*] Byte 3 Index 15
[*] AES-NI Units    : 1
[*] Range           : 00000000 - FFFFFFFF
[*] Step            : FFFFFFFF
[*] T00 Range       : 00000000 - FFFFFFFF
[!] T00 Key found   : 3525ac10bb391d8d5f3914fe94341985
[*] Output          : ac589cfe8a7ae5dd875d9786e5832400
[!] Valid key!
```

Рисунок 2.14 – Приклад виводу Hulk, запущеного для відновлення байт ключа “3525ac10bb391d8d5f3914fe????????”

Висновки до розділу 2

У другому розділі було запропоновано удосконалення інструментів для покрокового проведення атаки диференціального обчислювального аналізу шляхом інструментації емулятора: трасер TracerQiling (у межах платформи Tracer) з емуляцією програм на архітектурах AMD64, ARM, MIPS; клас TracerQiling (для платформи Deadpool) для автоматизації збору трас за аналогією з класами TracerPIN та TracerGrind.

Також у розділі пропонується модифікація платформи Daredevil з подальшою візуалізацією обчислених кореляцій для кожного байту. Вона може бути корисною для отримання загальних даних, як кількість відновлених байт та їх приблизне значення.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1 Дослідження ефективності створеного трасера

Для спрощеного аналізу і порівняння трасерів та пізніше - виконання шифрування на різних архітектурах - було створено скрипт, що через командний рядок може прийняти усі необхідні аргументи (Додаток Г):

- фільтр (діапазон адрес),
- трасер (TracerPIN, TracerGrind або TracerQiling),
- архітектуру,
- виконуваний файл,
- кількість трас для зняття та аналізу.

З допомогою нього можна порівняти існуюче рішення для збирання трас (у вигляді TracerPIN та TracerGrind) та створене (TracerQiling). У першому випадку відбувається трасування виконуваної програми, у другому - емулятора, тому можливі невеликі відмінності у кількості зібраних даних та (або) часі виконання. За замовчуванням зібрано 200 трас для кожного трасера та обрано лише один фільтр – mem_data_rw4 (тобто прочитані або записані в пам’яті дані розміром 4 байти).

Таблиця 3.1 – Час збирання трас, відновлення ключа, кількість відновлених байт для окремого одного випадкового ключа

	TracerPIN	TracerGrind	TracerQiling
час збирання 200 трас	138 секунд	91 секунд	236 секунд
час відновлення ключа по зібраних трасах	117 секунд	630 секунд	135 секунд
к-сть відновлених байтів з Daredevil	14	14	14

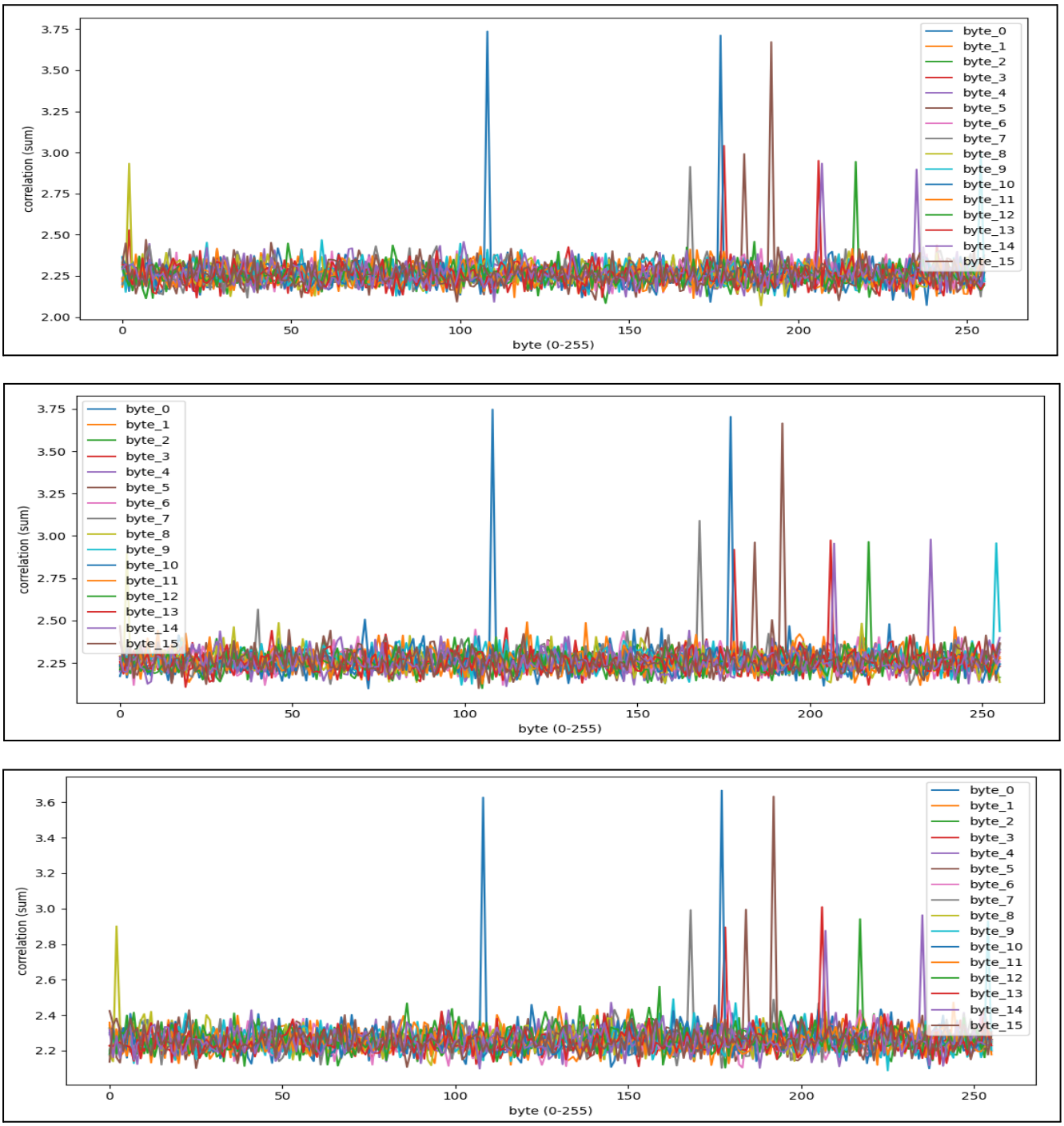


Рисунок 3.1 – Візуалізація значень кореляцій для байтів ключа, обчислених за допомогою знятих трас з (а) TraceGrind, (б) TracerPIN, (в) TracerQiling

За результатами збирання трас (Таблиця 3.1): TracerQiling збирає траси більш як вдвічі повільніше ніж TracerGrind та в 1,5 рази повільніше ніж TracerPIN, це можна пояснити тим, що емульована програма, в якій відбувається перехоплення певних адрес та інструкцій, очевидно буде працювати довше за інструментовану програму, що виконується в межах робочої операційної системи.

Час аналізу трас навпаки менший для TracerQiling та TracerPIN і більший для TracerGrind через кількість зібраних даних.

На Рисунку 3.1 зображено три різні графіки кореляцій для знятих трас при використанні вибраної white-box імплементації AES з одним випадковим вбудованим ключем трасерами TracerGrind, TracerPIN, TracerQiling відповідно. Хоча існують невеликі розбіжності у обчислених значеннях, кількість відновлених байтів, їх порядковий номер та співвідношення у обчислених кореляціях зберігаються.

Час збирання та аналізу не має впливу на кількість відновлених байтів секретного ключа, по результатах виконання Daredevil з отриманими у кожному випадку трасерами було відновлено однакову кількість байтів. Можна зробити висновок, що імплементоване рішення трасера з допомогою Qiling Framework працює на рівні з іншими трасерами.

3.2 Підготовка обраної імплементації до емуляції на AMD64, ARM, MIPS

Для аналізу white-box криптоалгоритму у вбудованих системах було обрано імплементацію AES [21], що спирається на роботи [1] та [22].

Стандартний алгоритм AES передбачає використання 4 функцій: AddRoundKey (побітове додавання ключа), SubBytes (підстановка байтів), ShiftRows (зсув рядків), MixColumns (перемішування стовпців). White-box AES [22] замінює усі операції, окрім ShiftRows, на підстановку з таблиць, згенерованих на основі секретного ключа.

Обрана імплементація дозволяє шифрувати AES-128, AES-192, AES-256 (надалі буде використано лише AES-128) у режимах CFB, OFB та CTR. Для дослідження буде використовуватись режим ECB. Автор не передбачив шифрування у цьому режимі, тому потрібно вдатись до певної маніпуляції. Вважаючи, що шифрування відбувається з white-box AES (позначимо як Ek),

застосуємо режим CFB для шифрування одного блоку, подавши відкритий текст як вектор ініціалізації, а рядок з нулів як відкритий текст (Рисунок 3.2). Тоді результат буде аналогічним режиму ECB, оскільки (при побітовому додаванні):

$$a \oplus 0 = a$$

Дана маніпуляція дозволяє не змінювати значну частину коду і лише використовувати надані функції.

Оскільки даний код повинен запускатись на декількох архітектурах, важливо, щоб у нього було небагато залежностей. Дана імплементація відповідає даному критерію, адже використовує лише стандартну бібліотеку мови Сі. Для генерування таблиць також використовується бібліотека NTL [23], але цей крок можна виконати на хостовій системі, а емулювати програму уже з готовими таблицями.

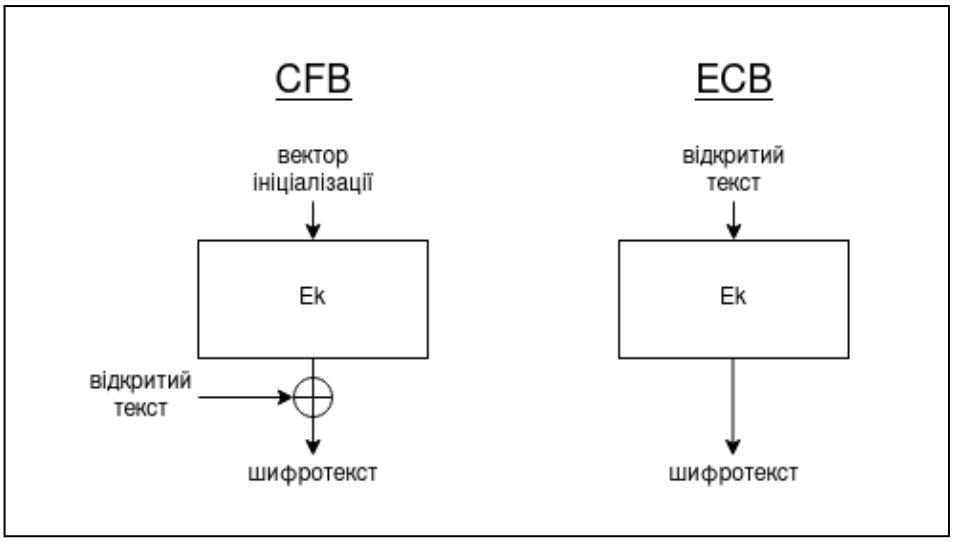


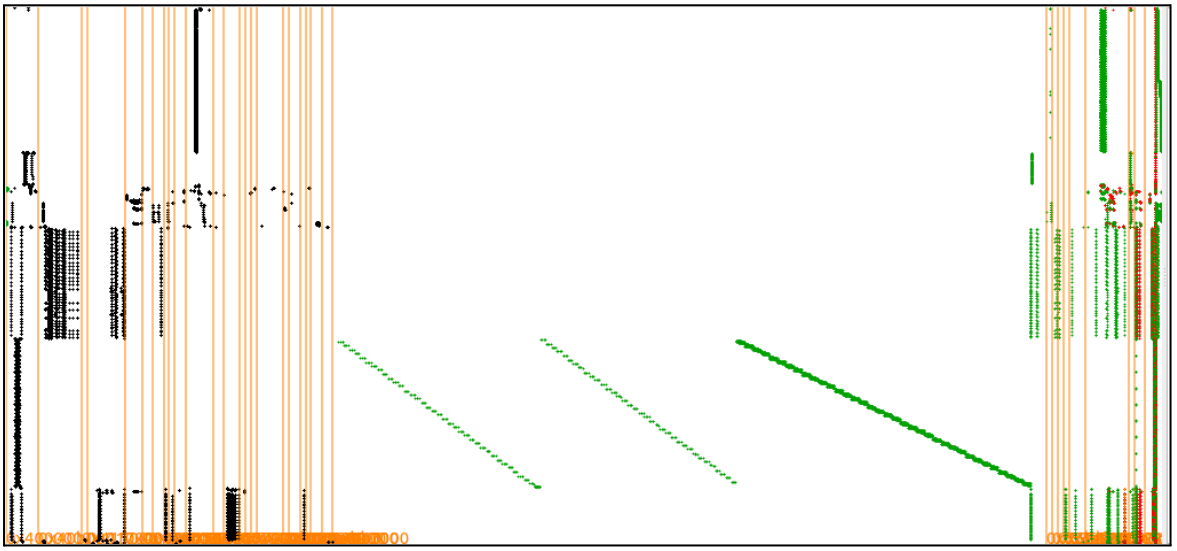
Рисунок 3.2 – Візуалізація шифрування одного блоку за допомогою white-box криптоалгоритму у режимах CFB та ECB

Для компіляції коду для архітектур AMD64, ARM, MIPS були використані команди з пакетів g++-x86-64-linux-gnu, g++-arm-linux-gnueabi, g++-mips-linux-gnu відповідно (Рисунок 3.3). У результаті було отримано три виконувані файли.

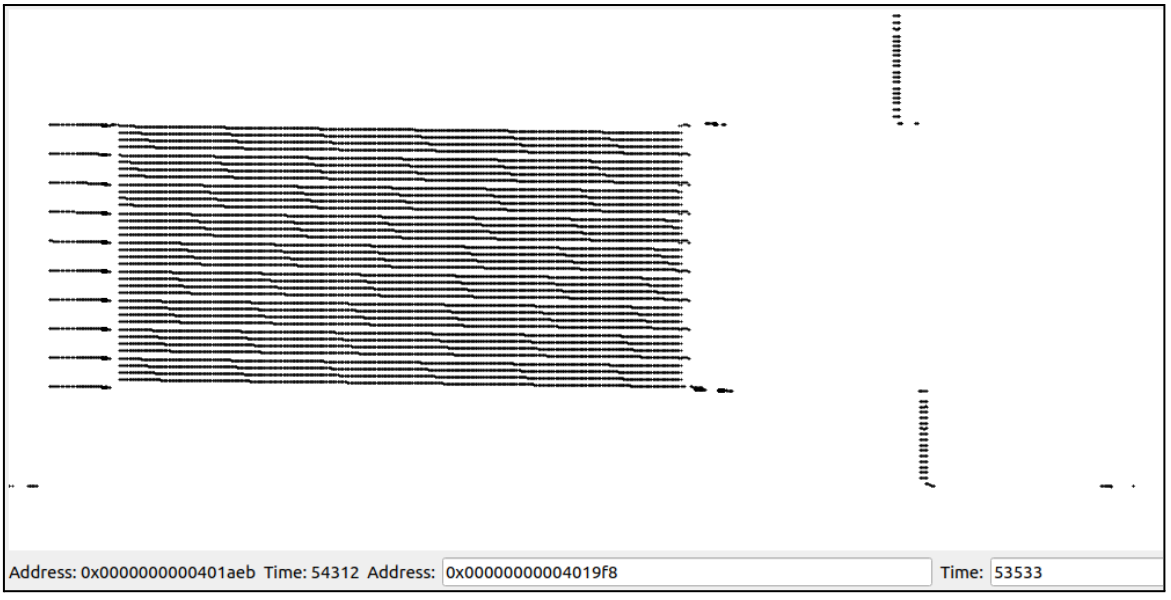
```
aes128-mips: aes_whitebox_compiler prepare
    mips-linux-gnu-g++ $(CXXFLAGS) -c aes-whitebox/aes_whitebox.cc -o aes-whitebox/aes_whitebox.o
    mips-linux-gnu-g++ $(CFLAGS) -c aes128.c -o aes128.o
    mips-linux-gnu-gcc $(LDFLAGS) -static aes-whitebox/aes_whitebox.o aes128.o -o $@
```

Рисунок 3.3 – Частина Makefile призначена для компіляції програми під архітектуру MIPS

Для зменшення розміру трас було вирішено обрати діапазон адрес, між якими здійснюватиметься трасування. Використовуючи TracerGrind для запуску виконуваного файлу, скомпільованого раніше для архітектури AMD64, було отримано базу даних трас.



(а)



(б)

Рисунок 3.4 – Траси вибраної white-box імплементації AES, скомпільованої для AMD64: (а) - повністю, (б) - частина з шифруванням

З допомогою TraceGraph траси було візуалізовано (Рисунок 3.4, а) та знайдено, що інструкції раундів шифрування знаходяться між адресами 0x401AD5 та 0x4028A7 (Рисунок 3.4, б). На Рисунку 3.4, (б) також видно 10 раундів, що підтверджує використання AES-128 у програмі.

Хоча DCA не передбачає жодних зусиль зворотної розробки програми та розбору низькорівневого коду, для архітектур ARM та MIPS візуалізація трас поки не імплементована, тому адреси для трасування було визначено з допомогою Ghidra [24].

Кожна з функцій, що імплементує певний режим AES всередині викликає функцію Cipher (Рисунок 3.5), що виконує шифрування одного блоку даних (16 байт). Всередині функції Cipher є виклики ShiftRows, тому визначаючи діапазон адрес, у якому виконується шифрування вибраного тексту, достатньо визначити адреси, де знаходяться Cipher та ShiftRows.

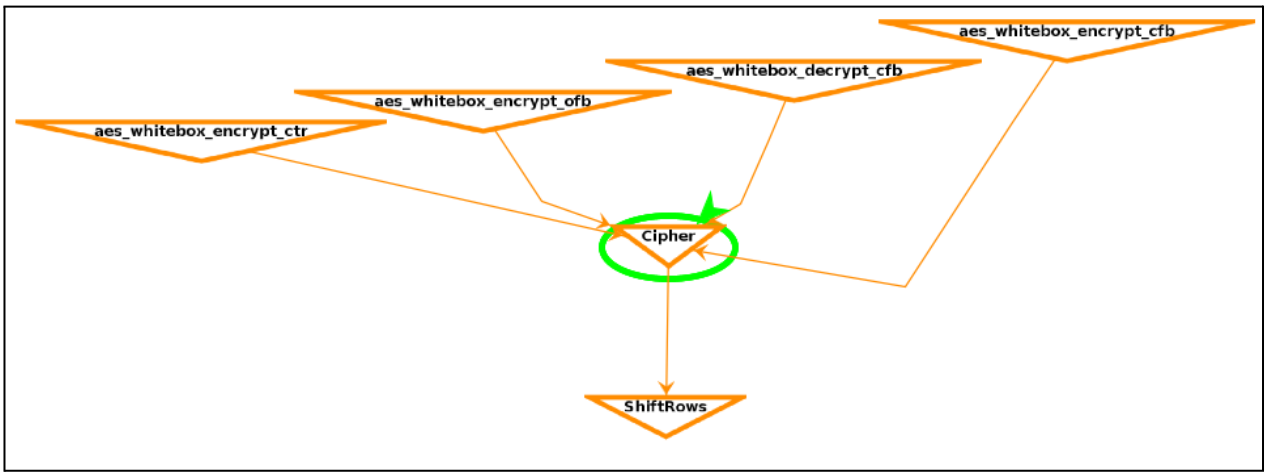


Рисунок 3.5 – Граф викликів функцій вибраної AES імплементації

```
     undefined4        Stack[-0x68]:4 local_68                                XRE
                     _ZN12_GLOBAL__N_19ShiftRowsEPh                    XREF[3]:
                     anon_unknown.conflictc9::ShiftRows
004007a0 27 bd ff 88     addiu       sp,sp,-0x78
     assume gp = <UNKNOWN>
004007a4 af bf 00 74     sw          ra,local_4(sp)
```

(а)

```
0040128c 8f b0 00 58     lw          s0,local_28(sp)
00401290 27 bd 00 80     addiu       sp,sp,0x80
00401294 03 e0 00 08     jr          ra
```

(б)

Рисунок 3.6 – Початок функції ShiftRows (а), закінчення функції Cipher (б) у виконуваному файлі для архітектури MIPS

У пам’яті Cipher підвантажується одразу після ShiftRows, тому діапазоном буде адреса початку ShiftRows та закінчення Cipher. Для MIPS знайдений діапазон 0x4007a0-0x401294 (Рисунок 3.6). Для ARM – 0x01055c-0x010de4 (знайдено аналогічним способом).

3.3 Вибір конфігурацій для здійснення атаки диференціального криптоаналізу

Для проведення атаки важливо вибрати правильний фільтр (описаний в Deadpool) та кількість трас, щоб отримати максимальну кореляцію між даними та секретним ключем. Для цього з допомогою TracerQiling для програми, скомпільованої для архітектури AMD64, було знято 20, 50, 100, 200, 500, 1000 трас та застосовано кожен існуючий фільтр. Результати наведені у Таблиці 3.2.

Таблиця 3.2 – Кількість відновлених байт при застосованому фільтрі та заданій кількості трас

к-сть трас ——— фільтр	20	50	100	200	500	1000
stack_w1	0	0	2	2	0	0
stack_w4	1	4	4	4	3	3
mem_addr1_rw1	2	14	15	15	15	15
mem_addr1_rw4	0	1	1	2	0	1
mem_addr2_rw1	1	14	15	15	15	15
mem_data_rw1	0	0	0	2	1	0
mem_data_rw4	5	14	15	15	14	14

Непостійність у кількості відновлених байт пов’язана з тим, що рахувались лише байти, що були на першому місці по кореляції, інколи з збільшенням чи зменшенням кількості трас кореляція могла стати слабшою, а навіть якщо правильний байт був на другому місці, він уже не враховувався.

За результатами визначено, що для відновлення максимальної кількості байт ключа достатньо 100-200 трас. При більшій кількості ситуація не змінюється, тому збирати більше трас не має сенсу. Серед фільтрів найкращі результати показали mem_addr1_rw1, mem_addr2_rw1 та mem_data_rw4. Надалі буде застосовано mem_data_rw4.

3.4 Здійснення атаки та аналіз результатів

За результатами попередніх підрозділів отримано: скомпільовані програми, що виконують шифрування AES-128 у режимі ECB, для виконання на архітектурах AMD64, ARM, MIPS; визначену кількість трас (200) та фільтр (mem_data_rw4) для успішної атаки; діапазон адрес для трасування (щоб відкинути дані, що не залежать від ключа) (AMD64 - 0x401AD5-0x4028A7, ARM - 0x01055c-0x010de4, MIPS - 0x4007a0-0x401294).

З допомогою невеликої програми Python (Додаток Д) було автоматизовано процес збору та аналізу трас. Скрипт складається з двох частин:

- збирання заданої кількості трас для кожної архітектури (Рисунок 3.7), також скрипт вираховує час збору та записує його в окремий файл (для подальшого аналізу),
- запуск daredevil для аналізу трас (Рисунок 3.8), також скрипт обчислює час аналізу та записує його в кінець виводу daredevil.

```
for a in range(len(archs)):
    print("[+] Generating traces for", archs[a])
    for n in tr_numb:
        comm = ["../DCA/trace_it.py", "-f", adranges[a], "-t", "2", "-x", "/home/gg/Deadpool/
        with open(path+archs[a]+"/"+key+"/"+str(n)+".time", "w") as of:
            start_time = time.time()
            run(comm, stdout=of)
            of.write("[TIME] " + format(time.time()-start_time, ".3f") + " seconds")
```

Рисунок 3.7 – Частина програми для автоматизації, що відповідає за збір трас

```
for a in archs:
    print("[+] Running daredevil for arch", a)
    os.chdir(path+"/"+a+"/"+key)
    for filename in glob.glob("*.attack_multinv.config"):
        with open(filename+".daredevil", "w") as of:
            start_time = time.time()
            run(["daredevil", "-c", filename], stdout=of)
            of.write("[TIME] " + format(time.time()-start_time, ".3f") + " seconds")
    for filename in glob.glob("*.attack_sbox.config"):
        with open(filename+".daredevil", "w") as of:
            start_time = time.time()
            run(["daredevil", "-c", filename], stdout=of)
            of.write("[TIME] " + format(time.time()-start_time, ".3f") + " seconds")
```

Рисунок 3.8 – Частина програми для автоматизації, що відповідає за запуск daredevil

Для підрахунку результатів створено ще одну програму (Додаток Е), за результатами якої створено Таблицю 3.3. Усього було знято по 200 трас для 150 ключів на кожній з архітектур та знайдено середнє значення кожної характеристики.

Час зняття трас трохи вищий для емуляції на ARM, час аналізу трас перевищує час аналізу на AMD64 та MIPS у більш як 5 разів. Це пов’язано зі специфікою імплементації та кількістю запитів на доступ до пам’яті. У кожному випадку у середньому було відновлено 14 байт, мінімальна та максимальна кількість відновлених байт 10 та 16 відповідно (для MIPS мінімальна кількість 9).

Таблиця 3.3 – Результати запуску імплементації AES на кожній з вибраних архітектур, усереднені значення

архітектура _____	AMD64	ARM	MIPS
середній час зняття 200 трас	202 секунд	225 секунд	192 секунд
середній час аналізу трас	102 секунд	756 секунд	146 секунд
середня к-сть відновлених байт	14 байт	14 байт	14 байт
мінім. та максим. к-сть відновлених байт	min = 10 max = 16	min = 10 max = 16	min = 9 max = 16

Обравши випадково один ключ серед 150, можна візуалізувати знайдені кореляції з допомогою програми у Додатку 7 та доповненим інструментом Daredevil.

Нехай обрано імплементацію з вбудованим ключем “88EFAE0AD28B4538A39346CD1FE18117”, графіки з кореляціями для кожного байту на кожній з архітектур зображені на Рисунку 3.9.

Незважаючи на загальну успішність атаки і однаковий ключ, жодних зв’язків між обчисленими значеннями кореляції не зберігається, відновлюється різна кількість байт, при відновлення першого байта на одній архітектурі, він не обов’язково буде відновлений на іншій. З цього можна зробити висновок: успішність атаки DCA залежить від white-box імплементації, правильно підібраної конфігурації та не залежить від характеристик середовища.

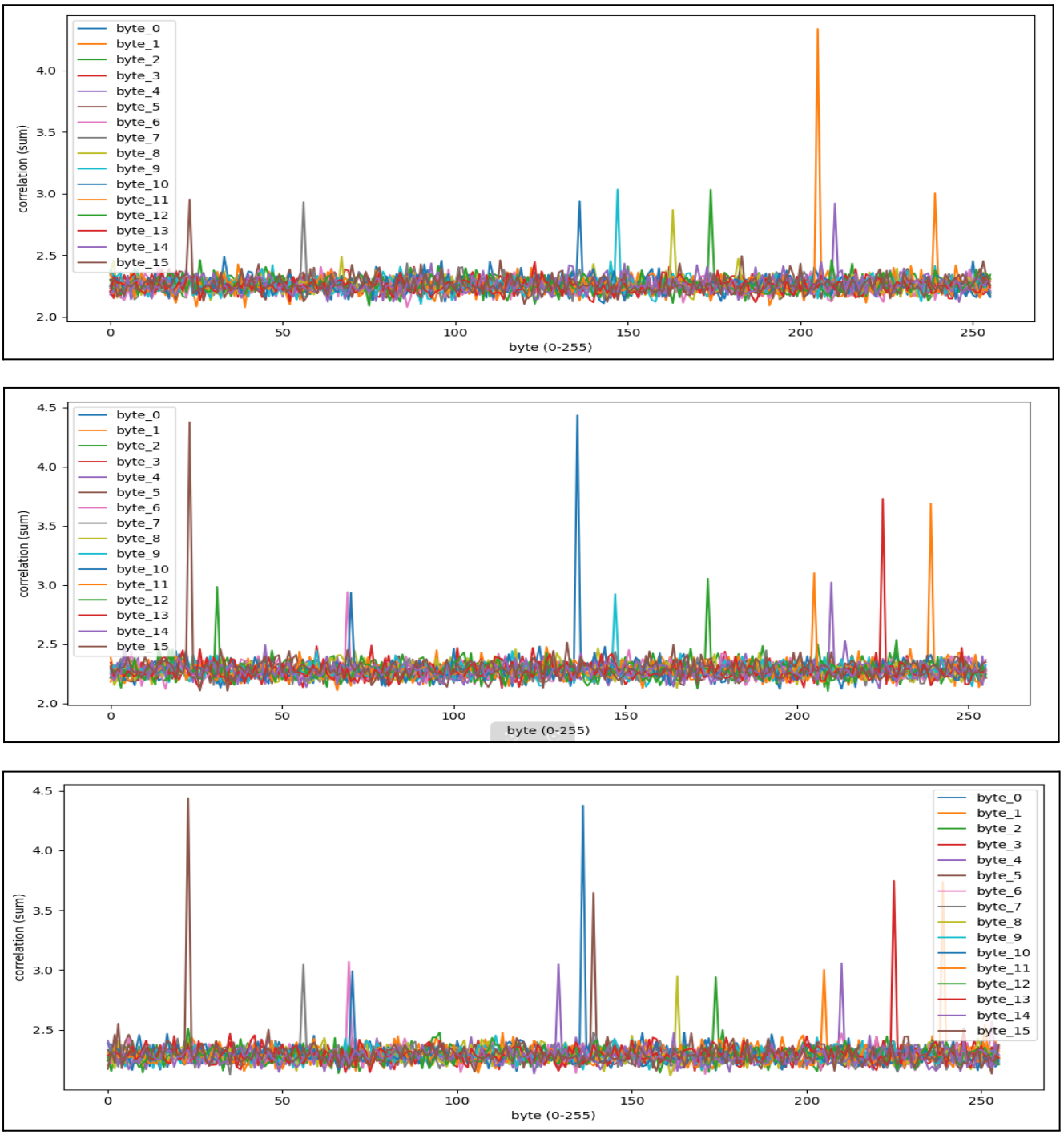


Рисунок 3.9 – Візуалізація значень кореляції для байтів одно ключа, обчислена за трасами, знятими на архітектурі (а) – AMD64, (б) – ARM, (в) – MIPS

Висновки до розділу 3

У третьому розділі було здійснено аналіз роботи створеного трасера (TracerQiling) шляхом порівняння його з існуючими (TracerPIN та TracerGrind) за часом роботи та відновлення ключа, визначено кількість відновлених байт (14). Час роботи TracerQiling більший приблизно вдвічі. Інші характеристики (такі як кількість відновлених байт і їх порядкові номери) для трасерів співпадають, що означає ефективність TracerQiling на рівні з іншими трасерами.

З допомогою TracerQiling було визначено оптимальний фільтр (mem_data_rw4) і кількість трас (200) для подальшої успішної атаки. Окрім того, було визначено та аргументовано вибір цілі для перевірки удосконалених інструментів, обрано діапазони адрес для трасування для заданої імплементації з метою зменшення розміру трас та пришвидшення атаки (AMD64 – 0x401AD5-0x4028A7, ARM – 0x01055c-0x010de4, MIPS – 0x4007a0-0x401294).

На архітектурах AMD64, ARM та MIPS успішно проведено атаку методом інструментування емулятора на 150 різних ключах з максимальною кількістю відновлених байт - 16, середньою - 14. Зроблено висновок про ефективність атаки у вбудованих системах: атака буде успішною, якщо імплементація вразлива до DCA.

ВИСНОВКИ

У роботі розглянуто короткі відомості про криптографію білої скриньки та атаку диференціального аналізу. Було наведено два методи для здійснення атаки: з інструментуванням програми та емулятора. Зроблено огляд інструментів для першого методу: Tracer, Deadpool, Daredevil, Hulk.

Для проведення атаки другим методом обрані інструменти було удосконалено та створено програмну модель, що включає: новий трасер у межах платформи Tracer, клас у Deadpool, удосконалення та візуалізацію результатів Daredevil.

Було обрано імплементацію, що легко переноситься на інші архітектури, окрім AMD64, скомпільовано її під ARM та MIPS, обрано діапазон адрес для трасування з метою зменшення розміру трас.

Було проведено аналіз роботи TracerQiling у порівнянні з TracerPIN та TracerGring: хоча імплементований трасер приблизно вдвічі повільніший, це не впливає на кінцевий результат. Також обрано конфігурації для ефективної атаки диференціального аналізу на обрану імплементацію у вбудованих системах: фільтр mem_data_rw4 та кількість трас – 200. Було проведено атаку на 150 різних ключів з використанням удосконалених інструментів з відновленням 10-16 байтів ключа у кожному випадку (у середньому 14). Середній час зняття трас на кожній архітектурі приблизно однаковий. Час аналізу відрізняється за рахунок особливостей імплементації.

Отримана програмна реалізація методу проведення диференціального обчислювального криптоаналізу для архітектур AMD64, ARM та MIPS може використовуватись при аналізі прошивок вбудованих систем на вразливість до даної атаки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Chow S. White-box cryptography and an AES implementation [Текст] / S. Chow, P. Eisen, H. Johnson and P. C. van Oorschot // 9th Annual Workshop on Selected Areas in Cryptography (SAC 2002), Aug. 15-16, pp. 1-18, 2002.
2. Chow S. A white-box DES implementation for DRM applications [Текст] / S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot // Proceedings of the 2nd ACM Workshop on Digital Rights Management, vol. 2696 of Lecture Notes in Computer Science, pp. 1-15, Washington, DC, USA, November 2002.
3. Alpirez Bock E. On the security goals of white-box cryptography [Текст] / E. Alpirez Bock, A. Amadori, C. Brzuska, and W. Michiels // IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2020, no. 2, pp. 327–357, 2020.
4. Bos J.W. Differential computation analysis: Hiding your white-box designs is not enough. [Текст] / J.W. Bos, C. Hubain, W. Michiels, P. Teuwen // In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, Santa Barbara, CA, USA, 17–19 August 2016; Springer: Berlin/Heidelberg, Germany, pp. 215–236, 2016.
5. Delerablée C. White-box security notions for symmetric encryption schemes [Текст] / C. Delerablée, T. Lepoint, P. Paillier, and M. Rivain // International Conference on Selected Areas in Cryptography, vol.8282, pp.247-264, Springer, 2013.
6. Announcing the ADVANCED ENCRYPTION STANDARD (AES) [Текст] / Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.
7. Billet O. Cryptanalysis of a white-box AES implementation [Текст] / O. Billet, H. Gilbert, C. Ech-Chatbi // In Selected Areas in Cryptography(SAC), pp. 227-240, 2004.
8. Carlet C. Structural Attack (and Repair) of Diffused-Input-Blocked-Output White-Box Cryptography [Текст] / C. Carlet, S. Guilley, S. Mesnager // IACR

Transactions on Cryptographic Hardware and Embedded SystemsISSN 2569-2925, Vol. 2021, No. 4, pp. 57–87, 2021.

9. Kocher P. Differential Power Analysis [Текст] / P. Kocher, J. Jaffe, and B. Jun // Advances in Cryptology - CRYPTO ’99, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.
10. Qiling Framework Documentation [Електронний ресурс] – Режим доступу: https://docs.qiling.io/en/latest/.
11. Unicorn Engine [Електронний ресурс] – Режим доступу: https://www.unicorn-engine.org/.
12. Bogdanov A. Higher-order DCA against standard side-channel countermeasures [Текст] / A. Bogdanov, M. Rivain, P. S. Vejre, J. Wang // International Workshop on Constructive Side-Channel Analysis and Secure Design, pp. 118–141. Springer, 2019.
13. Biryukov A. Attacks and countermeasures for white-box designs [Текст] / A. Biryukov, A. Udovenko // In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, 8–12 December 2019; Springer: Cham, Switzerland, pp. 373–402, 2019.
14. Tracer Project [Електронний ресурс] – Режим доступу: https://github.com/SideChannelMarvels/Tracer.
15. Pin – A Dynamic Binary Instrumentation Tool [Електронний ресурс] – Режим доступу: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.
16. Valgrind [Електронний ресурс] – Режим доступу: https://valgrind.org/.
17. Deadpool Project [Електронний ресурс] – Режим доступу: https://github.com/SideChannelMarvels/Deadpool.
18. Inspector Side Channel Analysis [Електронний ресурс] – Режим доступу: https://www.riscure.com/security-tools/inspector-sca.
19. Daredevil Project [Електронний ресурс] – Режим доступу: https://github.com/SideChannelMarvels/Daredevil.

20. Hulk Project [Електронний ресурс] – Режим доступу: https://github.com/pgarba/Hulk.
21. AES Whitebox Project [Електронний ресурс] – Режим доступу: https://github.com/balena/aes-whitebox/.
22. Muir J. A. A Tutorial on White-box AES [Текст] / J. A. Muir // Cryptology ePrint Archive, 2013.
23. NTL: A Library for doing Number Theory [Електронний ресурс] – Режим доступу: https://libntl.org/.
24. Ghidra [Електронний ресурс] – Режим доступу: https://ghidra-sre.org/.

ДОДАТОК А

ІНСТРУМЕНТОВАНА ПРОГРАМА-ЕМУЛЯТОР

```
#!/usr/bin/env python3

from capstone import Cs
from qiling import Qiling
from qiling.const import QL_VERBOSE, QL_ENDIAN
import argparse

from unicorn.unicorn_const import UC_MEM_WRITE, UC_MEM_READ

traces = []

def mem_read(ql: Qiling, access: int, address: int, size: int, value: int) ->
None:
   # only read accesses are expected here
   assert access == UC_MEM_READ

   if ql.archendian==QL_ENDIAN.EL:
       value = int.from_bytes(ql.mem.read(address, size), "little")
   else:
       value = int.from_bytes(ql.mem.read(address, size), "big")

   traces.append(f'[R] {address:016x} size= {size:d} value={value:016x}\n')

def mem_write(ql: Qiling, access: int, address: int, size: int, value: int) ->
None:
   # only write accesses are expected here
   assert access == UC_MEM_WRITE

   traces.append(f'[W] {address:016x} size= {size:d} value={value:016x}\n')

def start_hooks(ql, *args, **kw):
   if len(ql._hook)<2:
       ql.hook_mem_write(mem_write)
       ql.hook_mem_read(mem_read)

def stop_hooks(ql, *args, **kw):
   del ql._hook[1024]
   del ql._hook[2048]
   # this would delete all the hooks, which causes an error in mips arch
   # ql._hook = {}


if __name__ == "__main__":
   parser = argparse.ArgumentParser(description='Qiling Tracer')
   # make sure you set full path to the target, otherwise it may crash
   parser.add_argument("-t", dest="target", required=True)
   parser.add_argument("-i", dest="input", required=False)
   parser.add_argument("-of", dest="outputfile", required=True)
   parser.add_argument("-f", dest="filters", required=False)
```

```
    parser.add_argument("-r", dest="rootfs", required=True)


    args = parser.parse_args()

    argv = [args.target, args.input]
    ql = Qiling(argv, args.rootfs, verbose=QL_VERBOSE.DEFAULT,
multithread=True)


    if args.filters:
        filters = [int(i, 16) for i in args.filters.split('-')]
        ql.hook_address(start_hooks, filters[0])
        ql.hook_address(stop_hooks, filters[1])
    else:
        # captures every read and write; not recommended
        ql.hook_mem_write(mem_write)
        ql.hook_mem_read(mem_read)

    ql.run()

    with open(args.outputfile, 'w') as of:
        of.writelines(traces)
```

ДОДАТОК Б

КЛАС ДЛЯ ВИКОРИСТАННЯ СТВОРЕНОГО ТРАСЕРА

```
class TracerQiling(Tracer):
   def __init__(self, target,
                processinput=processinput,
                processoutput=processoutput,
                arch=ARCH.amd64,
                blocksize=16,
                tmptracefile='default',
                addr_range='default',
                stack_range='default',
                filters='default',
                tolerate_error=False,
                shell=False,
                debug=False,
                record_info=True):
       super(TracerQiling, self).__init__(target, processinput, processoutput,
arch, blocksize, tmptracefile, addr_range, stack_range, filters,
tolerate_error, shell, debug)
       if self.addr_range=='default':
           self.addr_range=None
       if stack_range == 'default':
           if self.arch==ARCH.i386 or self.arch==ARCH.mips or
self.arch==ARCH.arm:
               self.stack_range =(0x7ff0d000, 0x7ff3d000)
           elif self.arch==ARCH.amd64:
               self.stack_range =(0x7ffffffde000, 0x80000000e000)
       if record_info:
           for f in self.filters:
               f.record_info=False
   def get_trace(self, n, iblock):
       processed_input=self.processinput(iblock, self.blocksize)
       input_stdin, input_args = processed_input
       if input_stdin is None:
           input_stdin=b''
       if input_args is None:
           input_args=[]
       if self.addr_range:
           cmd_list=[qiling_exec, '-f', str(self.addr_range), '-of',
self.tmptracefile, '-t'] + self.target + input_args + ["-r", self.arch]
       else:
           cmd_list=[qiling_exec, '-of', self.tmptracefile, '-t'] +
self.target + input_args + ["-r", self.arch]
       output=self._exec(cmd_list, input_stdin)
       oblock=self.processoutput(output, self.blocksize)
       self._trace_init(n, iblock, oblock)
       with open(self.tmptracefile, 'r') as trace:
           for line in iter(trace.readline, ''):
               if len(line) > 2 and (line[1]=='R' or line[1]=='W'):
                   m=re.search(r'\[(.)\] *([0-9a-fA-F]+) *size= *([0-9]+)
*value= *(.*)', line)
```

```
                    assert m is not None
                    mem_mode=m.group(1)
                    mem_addr=int(m.group(2), 16)
                    mem_size=int(m.group(3), 16)
                    mem_data=int(m.group(4).replace(" ",""), 16)
                    for f in self.filters:
                        if mem_mode in f.modes and
f.condition(self.stack_range, mem_addr, mem_size, mem_data):
                            if f.record_info:
                                self._trace_info[f.keyword].append((mem_mode,
item, ins_addr, mem_addr, mem_size, mem_data))

self._trace_data[f.keyword].append(f.extract(mem_addr, mem_size, mem_data))
        self._trace_dump()
        if not self.debug:
            os.remove(self.tmptracefile)
        return oblock
```

ДОДАТОК В

ВІЗУАЛІЗАТОР РЕЗУЛЬТАТІВ DAREDEVIL

```
#!/bin/env python3

import matplotlib.pyplot as plt
x = [i for i in range(256)]

for bn in range(16):
   y = [0 for i in range(256)]
   with open("sum_abs_"+str(bn)+".txt") as f:
       te = f.readlines()
   for line in te:
       byte = int(line[:line.find(":")], 16)
       y[byte] = float(line[line.find(":")+2:])
   print(y)
   plt.plot(x, y, label = "byte_"+str(bn))


plt.xlabel('byte (0-255)')
plt.ylabel('correlation (sum)')
plt.legend()
plt.show()
```

ДОДАТОК Г

ДОПОМІЖНА ПРОГРАМА З МОЖЛИВІСТЮ ВИБОРУ ТРАСЕРА ДЛЯ АНАЛІЗУ

```
#!/usr/bin/env python

import sys, glob, os
sys.path.insert(0, '../../')
from deadpool_dca import *
import argparse

def processinput(iblock, blocksize):
   return (None, ['%0*x' % (2*blocksize, iblock)])

def qprocessinput(iblock, blocksize):
   return (None, ['-i', '%0*x' % (2*blocksize, iblock)])

def processoutput(output, blocksize):
   try:
       return int(''.join([x for x in output.split('\n') if
len(x)==32][0]), 16)
   except:
       print("There must have been some error, could not return
anything")
       return 0


parser = argparse.ArgumentParser(prog='./trace_it.py',
description='Capture and prepare application traces for the daredevil.')
parser.add_argument('N', type=int, help='number of traces to capture')
parser.add_argument('-f', '--filter', metavar='address_range',
dest='range', required=False, default=None, help='filter traces to
capture only this address range')
parser.add_argument('-t', '--tracer', metavar='tracer_index',
dest='tracer', default=0, type=int, choices=[0, 1, 2], help='tracer to
use (default=0): TracerGrind=0, TracerPIN=1, TracerQiling=2')
parser.add_argument('-a', '--arch', metavar='architecture', dest='arch',
default="amd64", required=False, choices=["i386", "amd64", "arm",
"arm64", "mips"], help='set architecture: "i386", "amd64", "arm", "arm64"
or "mips"')
parser.add_argument('-x', '--exec', metavar='executable',
dest='executable', required=True, help='file to be executed (set full
path)')

args = parser.parse_args()

tracers=[TracerGrind, TracerPIN, TracerQiling]

# DefaultFilters.mem_addr1_rw1 is so far one of the best
filters=[DefaultFilters.mem_data_rw4]
```

```
pi = processinput
arch = ARCH.amd64

if args.arch == "i386":
   arch = ARCH.i386
elif args.arch == "arm":
   arch = ARCH.arm
elif args.arch == "mips":
   arch = ARCH.mips


if args.tracer==2:
   pi=qprocessinput


if args.range:
   T = tracers[args.tracer](args.executable, pi, processoutput, arch, 16,
addr_range=args.range, filters=filters, shell=True, debug=True,
record_info=False)
else:
   T = tracers[args.tracer](args.executable, pi, processoutput, arch, 16,
filters=filters, shell=True, debug=True, record_info=False)


T.run(args.N)

# an error sometimes arises when using Qiling Tracer and mips as
architecture
# so make sure to clean those traces
if args.tracer==2:
   for f in glob.glob("*00000000000000000000000000000000.bin"):
       os.remove(f)

bin2daredevil(keywords=filters,
             configs={'attack_sbox':   {'algorithm':'AES',
'position':'LUT/AES_AFTER_SBOX',
'correct_key':'0x6C99D9B2EBB869A802FEB177BFCECFC0'},
                      'attack_multinv':{'algorithm':'AES',
'position':'LUT/AES_AFTER_MULTINV',
'correct_key':'0x6C99D9B2EBB869A802FEB177BFCECFC0'}})
```

ДОДАТОК Д

ПРОГРАМА ДЛЯ АВТОМАТИЗАЦІЇ ЗБОРУ ТА АНАЛІЗУ ТРАС

```
#!/bin/env python3

from codecs import ignore_errors
from subprocess import run
import glob, os, shutil
import time
import sys

path = sys.argv[1]
key  = sys.argv[2]
print(path)
archs = ["amd64", "arm", "mips"]
adranges = ["0x401AD5-0x4028A7", "0x01055c-0x010de4", "0x4007a0-0x401294"]
tr_numb = [200]
for a in archs:
   os.mkdir(path+a+"/"+key)

for a in range(len(archs)):
   print("[+] Generating traces for", archs[a])
   for n in tr_numb:
       comm = ["../DCA/trace_it.py", "-f", adranges[a], "-t", "2", "-x",
"/home/gg/Deadpool/wbs_aes_chow_fork_balena/target/aes128-"+archs[a], "-a",
archs[a], str(n)]
       with open(path+archs[a]+"/"+key+"/"+str(n)+".time", "w") as of:
           start_time = time.time()
           run(comm, stdout=of)
           of.write("[TIME] " + format(time.time()-start_time, ".3f") + "
seconds")
       for filename in glob.glob("mem_*"):
           os.rename(filename, path+archs[a]+"/"+key+"/"+filename)
       for filename in glob.glob("stack_*"):
           os.rename(filename, path+archs[a]+"/"+key+"/"+filename)

for a in archs:
   print("[+] Running daredevil for arch", a)
   os.chdir(path+"/"+a+"/"+key)
   for filename in glob.glob("*.attack_multinv.config"):
       with open(filename+".daredevil", "w") as of:
           start_time = time.time()
           run(["daredevil", "-c", filename], stdout=of)
           of.write("[TIME] " + format(time.time()-start_time, ".3f") + "
seconds")
   for filename in glob.glob("*.attack_sbox.config"):
       with open(filename+".daredevil", "w") as of:
           start_time = time.time()
           run(["daredevil", "-c", filename], stdout=of)
           of.write("[TIME] " + format(time.time()-start_time, ".3f") + "
seconds")
```

ДОДАТОК Е

ПРОГРАМА ДЛЯ ОБЧИСЛЕННЯ І УСЕРЕДНЕННЯ РЕЗУЛЬТАТІВ АНАЛІЗУ

```
#!/bin/env python3

from subprocess import run
import glob, os, shutil
import time

def recovered(files):
   rec = set()
   cur_ind = 0
   for filename in files:
       with open(filename, "r") as f:
           te = f.read().split('\n')
       for i in te:
           if 'Attack of byte number' in i:
               ind_pos = i.find("Attack of byte number ")+len("Attack of byte
number ")
               cur_ind = int(i[ind_pos: ind_pos+2])
           if 'rank: 0' in i:
               rec.add(cur_ind)
   return (len(rec))

def get_time(files):
   res = 0
   for filename in files:
       with open(filename, "r") as f:
           te = f.read()
       ind_pos = te.find("[TIME] ")+len("[TIME] ")
       res += float(te[ind_pos:te.find(" seconds", ind_pos)])
   return (res)



pwd = os.getcwd()
archs = ["amd64", "arm", "mips"]

keys = []
for a in glob.glob(archs[0]+"/*"):
   keys.append(a[a.find("/")+1:])

recoveredarchs = {a:[] for a in archs}
timesarchs = {a:[] for a in archs}
timesanalysis = {a:[] for a in archs}

for k in keys:
   for a in archs:
       found = {}
       #print("__________"+a+"__________"+k+"__________")
```

```
        os.chdir(pwd+"/"+a+"/"+k)

        files = sorted(glob.glob("*.daredevil"))
        if len(files)==2:
            for i in range(len(files)//2):
                r = recovered(files[i*2:i*2+2])
                found[files[i*2][:files[i*2].find(".attack_")]] = r
            sort = sorted(found.items(), reverse=True, key=lambda x: x[1])
            recoveredarchs[a].append(sort[0][1])
            timesarchs[a].append(get_time(["200.time"]))
            timesanalysis[a].append(get_time(files))



for a in archs:
   print("Architecture:                  ", a)
   print("Number of traces:              ", len(recoveredarchs[a]))
   print("Mean number of recovered bytes:",
sum(recoveredarchs[a])/len(recoveredarchs[a]))
   print("Mean time of capturing:        ",
sum(timesarchs[a])/len(timesarchs[a]))
   print("Mean time of analysis:         ",
sum(timesanalysis[a])/len(timesanalysis[a]))
   print("Minimum recovered bytes:       ", min(recoveredarchs[a]))
   print("Maximum recovered bytes:       ", max(recoveredarchs[a]))
   print()
```